OpenQL

QuTech, TU Delft

Jul 15, 2021

USER MANUAL

OpenQL is a framework for high-level quantum programming in C++/Python. The framework provides a compiler for compiling and optimizing quantum code. Compared to competing frameworks, such as Qiskit, OpenQL's focus lies more on retargetability and compiling all the way down to assembly code for the various control (micro)architectures used by QuTech, and less on high-level constructs such as circuit conjugation: in general, the input you provide is a complete circuit and a platform description, and the output is an equivalent circuit that complies to platform constraints and/or machine code for running that circuit on a real quantum computer.

CHAPTER

ONE

HOW TO READ THE DOCUMENTATION

The documentation is roughly split into three main parts:

- the user manual;
- the user reference; and
- the developer documentation.

The user manual portion is intended to be read like a book, to give new users an overview of how to use OpenQL and build intuition for what does what. It culminates in a few tutorials that take you from a basic algorithm all the way to simulation of the compiled algorithm. The reference may then be used for more exhaustive information about particular topics of interest to you, such as particular API functions, passes, architectures, and so on. Note that most of the contents of the reference section are also available from within Python using the various `dump_*()` functions; this document only provides a more pleasingly laid-out version of the same information.

The developer documentation is only intended for (new) contributors. That is to say: unless you're intending to understand or change OpenQL's internal C++ implementation, the information here is not relevant to you. Rather, the section provides an overview of the codebase and the conventions used, and includes internal interface documentation as generated by Doxygen. Note however, that the intention is that the code is "self-documenting," in the sense that the relevant documentation is placed inside the code as long comment blocks, to incentivize keeping the code and documentation synchronized. Ideally this would all be generated into the Doxygen documentation, but not everything has been converted to Doxygen-recognized docstrings yet.

1.1 Concepts

This section goes over some key concepts that you should understand before doing anything with OpenQL.

1.1.1 OpenQL versus other compilers

A key difference between the OpenQL compiler and traditional compilers is that OpenQL is a library rather than an application. That means that you can't just invoke OpenQL on the command line given some input file. Instead, your input file is (usually) a Python script that imports the OpenQL module, builds a representation of the algorithm that OpenQL understands as it runs, and eventually tells OpenQL to compile that algorithm representation somehow. The output of the compilation process is then usually written to output files, though the behavior of the compiler depends entirely on its configuration.

Also different compared to most compilers is that OpenQL is inherently retargetable. Whereas with for instance `gcc` the target architecture is built right into it, with OpenQL you can compile code for many different kinds of control architectures and quantum devices. This target architecture is described via the platform configuration structure.

1.1.2 Platform configuration

In OpenQL, the *platform configuration* is what determines what quantum device and control architecture will be compiled for, also known as the compilation *target*. It ultimately defines the subset of describable quantum circuits that can actually be executed on the target by way of a set of constraints and reduction rules. Here are some examples of things described in the structure.

- The primitive instruction set, along with decomposition rules for common gates that cannot be directly represented.
- The number of usable qubits within the device and their connectivity.
- Control and instrument constraints on available gate parallelism.

The goal for the compiler is to take the user-specified algorithm and convert it to a behaviorally equivalent circuit within this set, preferably the most optimal one it can find.

As of version 0.9, OpenQL has *a bunch of default target descriptions built into it*. You can use them directly if they're good enough for your use case, or you can use them as a baseline for making your own. The complete configuration structure is defined *here*.

1.1.3 Quantum algorithm representation

Note: This is *not* a description of the current implementation of the intermediate representation of the compiler, but rather an overview of what it behaves like from a user perspective.

OpenQL models a quantum algorithm as follows:

- a complete algorithm is referred to as a *program*;
- a program consists of one or more *kernels*; and
- each kernel consists of one or more "statically-scheduled" *gates* (a.k.a. instructions) without control-flow within the kernel.

Typically, most of the Python or C++ program using the OpenQL compiler consists of building an algorithm using this model, although it's also possible to build it from a cQASM file using the cQASM reader pass.

Depending on your background, "static," "scheduled," and "control-flow" may require further explanation.

- "Static" just means "known by the compiler," or equivalently, "not dependent on information only known at runtime." In the world of quantum computing, this typically just means "not dependent on measurement results."
- The "schedule" is what defines when a gate is applied, in this context relative to the start of the kernel. So, "statically-scheduled" means that (if a gate is applied) that gate must always be applied at the same time with respect to the start of the kernel.
- "Control-flow" is *almost* anything to do with conditional statements (like `if`) and loops. More formally, anything that results in a classical branch instruction is considered to be control-flow. Note that OpenQL also supports a special case for `if`-like constructs called *conditional gate execution* that does not rely on control-flow; we'll get to that.

The result of the above is that a kernel behaves just like how you would traditionally draw a quantum circuit, with time on the X-axis and the qubits and classical bits on the Y-axis in the form of horizontal lines.

More complex algorithms that include control-flow can be specified using multiple kernels. Say, for instance, that you have an initialization circuit, then a circuit that you want to repeat until some qubit measures as 1, followed by a circuit that does some final measurements. The first and last circuit would then be added to the program as a normal kernel, while the second would be added as a do-while kernel.

Warning: Most architectures and parts of OpenQL don't fully support nonstandard kernel types yet. Some (mapping) don't support multiple kernels in any form. For now, you have to check the documentation of the passes used by the particular compiler configuration that you intend to use to see what's supported and what isn't.

Control-flow based on measurement results tends to be a costly operation in most architectures, because the time from sending a measurement gate to the instruments to being able to act on the measurement result tends to be quite long compared to the coherence time of NISQ-era qubits. However, sometimes part of this pipeline can be avoided. Say, for instance, that you want to apply an X gate on some qubit only if some other qubit measured as 1. If the instruments themselves (or at least a deeper part of the control architecture) are capable of turning an X gate into an identity/no-op gate based on a measurement, this and subsequent gates can already be queued up before the measurement has actually taken place. This is the conditional gate execution we alluded to earlier. Using this scheme, the condition for whether the gate is executed or not is encoded as part of the gate, instead of being part of the program's control-flow.

1.1.4 Gate representation

Gates in OpenQL fundamentally consist of a name, some set of operands, and a condition. The gate names available for use are defined within the platform configuration file, along with some of their semantics, such as the gate duration.

Warning: As of version 0.9, OpenQL also still assigns semantics and makes assumptions based on the name of a gate however. For example, an `x` gate is assumed to commute with an `x90` gate, and both are assumed to have a single qubit operand and nothing else, or things will probably break. This behavior is unfortunately largely undocumented, so you'll have to search through the code for it. Obviously this is not an ideal situation, and thus this is something that we want to get rid of. All semantics needed by OpenQL should, down the line, be specified in the platform configuration, or, for backward compatibility, be inferred from the gate name in a documented way.

The operand set for each gate consists of the following:

- zero or more qubit operands;
- zero or more creg operands;
- zero or more breg operands;
- zero or one literal integer operand*; and
- zero or one angle operand.

Here, "cregs" refer to classical integer registers, and "bregs" refer to classical bit registers. The former are used for loops and other control flow, while the latter are used for conditional execution.

Finally, the gate's condition consists of a boolean function applied to zero, one, or two bregs. Unconditional gates are simply modelled using a unit-one boolean function acting on zero bregs.

1.1.5 Configuring the compilation process

We've now described the way in which you specify the input and the target for the compiler, but there's one more thing OpenQL must know: *how* to compile for the given target. This is also known as the compilation *strategy*. When the strategy is incorrect or insufficient, the resulting circuits may not actually be completely valid for the target, unless the incoming algorithm is carefully written such that constraints not dealt with by OpenQL have already been met.

Generally, the compilation process consists of the following steps:

- decomposition;

- optimization;
- mapping;
- scheduling; and
- code generation.

Decomposition is the act of converting gates that cannot be executed using a single instruction in the target gateset into a list of gates that have the same behavior. For example, a SWAP gate may be decomposed into three CNOT gates.

Optimization tries to reduce the algorithm to a more compact form. This is particularly relevant after decomposition, as the decomposition rules may lead to sequences of gates that trivially cancel each other out.

Mapping is the act of changing the qubit indices in the circuit such that the connectivity constraints of the target device are met. For complex circuits, no single mapping will suffice (or it may be too time-consuming to compute, as this is an NP problem); in this case, SWAP gates will be inserted to route non-nearest-neighbor qubits toward each other.

Scheduling is the act of assigning cycle numbers to each gate in a kernel. This can of course be done trivially by assigning monotonously increasing cycle numbers to each gate in the order in which they were written by the user, but this is highly inefficient; instead, heuristics and commutation rules are used to try to find a more optimal solution that makes efficient use of the parallelism provided by the control architecture.

Finally, code generation takes the completed program and converts it to the assembly or machine-code format that the architecture-specific tools expect at their input.

A strategy consists of a list of *passes*, along with pass-specific configuration options for each pass. OpenQL provides default pass lists for the available architectures, as listed in the *architecture reference*. You can modify this default strategy using API calls prior to compilation if need be, or you can override the defaults entirely by writing a *compiler configuration file*.

1.2 Installation

OpenQL is available from PyPI as a pre-built package for Windows, MacOS, and Linux for all active Python 3.x versions. Once you have Python and have access to a command line you can get it as follows:

```
pip install qutechopenql
```

after which you should be able to run

```
python -c 'import openql; print(openql.get_version())'
```

to see if it works.

Note: Depending on your OS and Python configuration, you may need to use `python3` instead of just `python` to disambiguate with a Python 2.7 installation, and/or use `pip3` or `python -m pip` instead of just `pip`. You may also need to add `--user` at the end of the `pip` command to avoid permission problems. If you're unsure of what all of the above means, first read up on how Python and pip work on your operating system in the relevant Python (or Linux distribution) documentation; installation of Python packages is rather fundamental to Python and out of scope for this manual.

Warning: The documentation you're reading now is generated for version **0.10.0**. If there is a mismatch, be aware that there may be an API mismatch as well! The reference information can, however, be queried from within Python using the `help()` builtin and (from 0.8.1.dev6 onwards) using the various `dump_*()` functions.

If you're on MacOS and want to use the visualizer, you'll need XQuartz in addition. You can install this using Brew.

Some of OpenQL's components are optional when OpenQL itself is compiled. In general, the pre-built package includes everything, *except* for initial placement due to a license conflict. If you need initial placement, you'll need to *compile manually*.

OpenQL used to support Conda in addition to PyPI/pip for Python package management, but ultimately this was disabled due to excessive time spent on dependency resolution. Nevertheless, the Conda recipe is still available, so it may or may not work, but for this you'll also have to *compile manually*, as the prebuilt Conda packages are likely out of date.

1.3 Creating your first program

In the OpenQL framework, the quantum `Program`, its `Kernel`s, and its `gate`s are created using API calls contained in a C++ or Python 3 program. You then run this program in order to compile the quantum program. In the manual, we'll use Python exclusively, but *the API is largely identical in C++*.

You can set up Python however you like; via a Jupyter/IPython notebook, a Python file created in a text editor that you then run, various Python IDEs like IDLE, and so on. Just make sure that OpenQL is actually installed for the interpreter that your preferred environment uses.

The very first step is to import the OpenQL module:

```
import openql as ql
```

Note: In versions before 0.9, you had to use the more verbose `from openql import openql as ql` syntax. This is still supported for backward compatibility, but is deprecated.

Next, you should call the `initialize()` function:

```
ql.initialize()
```

This function ensures that OpenQL is (re)initialized to its default configuration. This is especially important in the context of a test suite or IPython notebook, where one might want to do multiple compilation runs in a single Python instance. For backward compatibility, OpenQL will automatically call this function when you first use a dependent API function, warning you as it does.

After initialization, you may want to change some of OpenQL's global *options*. One of the important ones is `output_dir`, which is used to specify which directory the compiler's output will be placed in. If you don't set it, OpenQL will default to outputting to a `test_output` directory within the current working directory. Another important one is `log_level`, which sets the verbosity of OpenQL's logging; if you don't set that one, you won't see any log messages, at least until something has already gone horribly wrong.

```
ql.set_option('output_dir', 'output')
ql.set_option('log_level', 'LOG_INFO')
```

Note: OpenQL will automatically recursively create directories whenever it tries to write a file. Thus, you don't have to manually create the output directory.

Note: As of version 0.9, you can also opt to use the more powerful, but slightly more complicated `Compiler` API to set options and manipulate the compilation strategy. Since version 0.9, almost all of the global options have no effect other than manipulating the default compilation strategy and pass options. You can obtain a reference to the *Compiler* object used by a *Platform* or *Program* using the *get_compiler()* method, or you can construct a *Compiler* manually and use its *compile()* function to compile the program (rather than `program.compile()`).

Before you can start building a quantum program, you must create a *Platform* object. One of its constructor parameters is either the name of the *architecture* you want to compile for, a reference to a *platform configuration file*, or (via the *from_json()* constructor) a JSON object specified by way of Python dictionaries, lists, strings, integers, and booleans with the same structure as the platform configuration file. The platform configuration is consulted by the APIs creating the program, kernels, and gates, to generate the matching internal representation of each gate.

For now, let's use the "none" architecture:

```
platform = ql.Platform('my_platform', 'none')
```

This is a basic architecture that is most useful for simulating with QX. The first argument is only used to identify the platform in error messages; you can set it to whatever you like.

After creating the platform, the *program* and its *Kernel*s may be created. The program and kernel constructors take the program/kernel name, the associated platform, and the number of qubits used in it as parameters. We'll use 3 in this example:

```
nqubits = 3
program = ql.Program('my_program', platform, nqubits)
kernel = ql.Kernel('my_kernel', platform, nqubits)
```

When needed, the number of used CRegs (classical integer registers) and BRegs (bit registers) used by the program/kernel must also be specified, but we don't use these for now.

Again, the first argument is just a name. However, unlike for the platform, the name is actually used. Specifically, the program name is used as a prefix for the output files, and the kernel names are used in various places where a unique name is needed (thus, they must actually be unique).

Once you have a kernel, you can add gates to it:

```
for i in range(nqubits):
    kernel.prepz(i)

kernel.x(0)
kernel.h(1)
kernel.cz(2, 0)
kernel.measure(0)
kernel.measure(1)
```

Most gates have a shorthand function, as used above. However, some architecture-specific gates might not, or might need additional arguments. For these cases, the *gate()* method can be used.

Note: You can only add gates that are registered via the instruction set definition in the platform configuration structure, or for which a decomposition rule exists. If a gate doesn't exist there, you will immediately get an exception. In future versions, this exception may be delayed to when you call *compile()*.

When you're done adding gates to a kernel, you can add the kernel to the program using *add_kernel()*:

```
program.add_kernel(kernel)
```

Note: The number of qubits, CRegs, and BRegs used by a kernel must be less than or equal to the number used by the program, and the number for the program must be less than or equal to what the number available in the platform. Also, a kernel can only be added to a program when the kernel and program were constructed using the same platform.

Finally, when you have completed the program, you can compile it using the `compile() <openql.Program.compile()` function:

```
program.compile()
```

Here's the completed program, taken from `examples/simple.py`:

```
import openql as ql

ql.initialize()

ql.set_option('output_dir', 'output')
ql.set_option('log_level', 'LOG_INFO')

platform = ql.Platform('my_platform', 'none')

nqubits = 3
program = ql.Program('my_program', platform, nqubits)
kernel = ql.Kernel('my_kernel', platform, nqubits)

for i in range(nqubits):
    kernel.prepz(i)

kernel.x(0)
kernel.hadamard(1)
kernel.cz(2, 0)
kernel.measure(0)
kernel.measure(1)

program.add_kernel(kernel)

program.compile()
```

When you run this file with Python, two files will be generated in the `output` directory: `my_program.qasm` and `my_program_scheduled.qasm`. The first look like this:

```
# Generated by OpenQL 0.10.0 for program my_program
version 1.2

pragma @ql.name("my_program")


.my_kernel
    prep_z q[0]
    prep_z q[1]
    prep_z q[2]
    x q[0]
    h q[1]
    cz q[2], q[0]
    measure q[0]
    measure q[1]
```

This file is generated by a cQASM writer pass before anything else is done. As you can see, it contains exactly what was generated by the Python program, but in cQASM 1.0 format.

The second is a little more interesting:

```
# Generated by OpenQL 0.10.0 for program my_program
version 1.2

pragma @ql.name("my_program")


.my_kernel
    prep_z q[0]
    skip 1
    { # start at cycle 2
        prep_z q[2]
        x q[0]
    }
    skip 1
    { # start at cycle 4
        prep_z q[1]
        cz q[2], q[0]
    }
    skip 1
    h q[1]
    skip 1
    { # start at cycle 8
        measure q[0]
        measure q[1]
    }
    skip 14
```

It is generated after basic ALAP (as late as possible) scheduling, using the (rather arbitrary) instruction durations specified in the default platform configuration file for the "none" architecture.

Depending on the architecture and compiler configuration, different output files may be generated. The above only applies because of the default pass list doe the "none" architecture: a cQASM writer, followed by a scheduler, followed by another cQASM writer. This is fully configurable.

1.4 Simulation using QX

This tutorial explains how to compile an OpenQL program and execute it on QX. We will use the example of rolling an 8-faced dice. Rolling this dice results in 1 out of 8 outcomes. The complete code for this example is available in `examples/dice.py`. You can also copy the snippits over to your own script as we walk through it.

We start by importing openql, qxelerator and some python packages. We also set some options for openql. For this example we will be using 3 qubits. All this is done by the following code snippet:

```
from openql import openql as ql
import qxelarator
from functools import reduce
import os
import matplotlib.pyplot as plt

ql.set_option('output_dir', 'output')
ql.set_option('log_level', 'LOG_INFO')

nqubits = 3
```

Next, we create a platform, a program and a kernel. We populate the kernel with 3 hadamard gates being applied on each qubits. This will put each qubit in superposition. Measuring each qubit will collapse the state resulting in getting either 0 or 1. This is done by dice_compile() as shown below:

```
def dice_compile():
    platform = ql.Platform('myPlatform', 'none')
    p = ql.Program('dice', platform, nqubits)
    k = ql.Kernel('aKernel', platform, nqubits)

    for q in range(nqubits):
        k.gate('h', [q])

    for q in range(nqubits):
        k.gate('measure', [q])

    p.add_kernel(k)
    p.compile()
```

Compiling the above code snippet will produce the following quantum assembly code in cQASM 1.0 format:

- `output/dice.qasm`, the generated un-scheduled cQASM code; and
- `output/dice_scheduled.qasm`, the generated cQASM code after scheduling.

For instance, `dice.qasm` contents are shown below:

```
version 1.0
# this file has been automatically generated by the OpenQL compiler please do not␣
↪modify it manually.
qubits 3

.aKernel
    h q[0]
    h q[1]
    h q[2]
    measure q[0]
    measure q[1]
    measure q[2]
```

These cQASM codes can be simulated on QX simulator. For this we are using the simplified python interface to QX known as QXelarator. This is done by the following code snippet:

```
def dice_execute_singleshot():
    print('executing 8-face dice program on qxelarator')
    qx = qxelarator.QX()

    # set the qasm to be executed
    qx.set('output/dice.qasm')

    # execute the qasm
    qx.execute()

    # get the measurement results
    res = [int(qx.get_measurement_outcome(q)) for q in range(nqubits)]

    # convert the measurement results from 3 qubits to dice face value
    dice_face = reduce(lambda x, y: 2*x+y, res, 0) + 1
    print('Dice face : {}'.format(dice_face))
```

Running `dice.py` will produce output as shown below:

```
Dice face : 2
```

where the Dice face can be any number between 1 and 8.

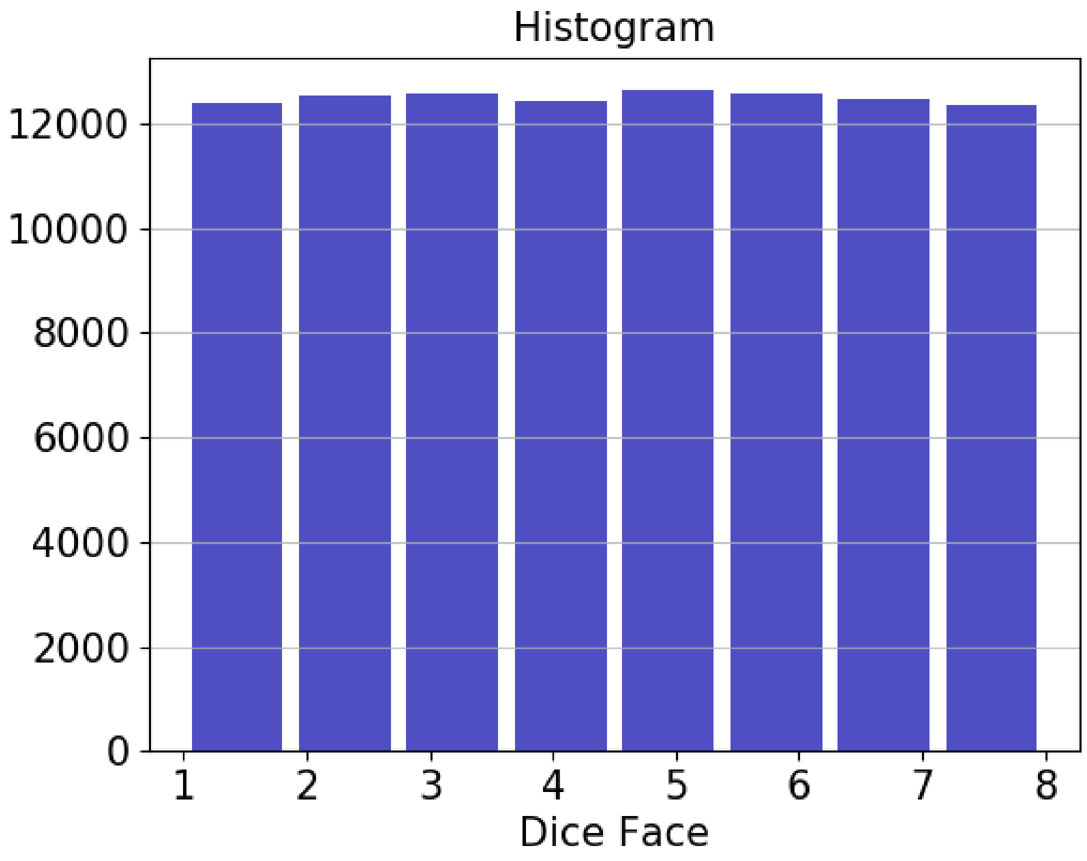
Next we can also roll the dice 100000 times and plot the frequency of occurance of each face by the following code snippet:

```
def plot_histogram(dice_faces):
    plt.hist(dice_faces, bins=8, color='#0504aa',alpha=0.7, rwidth=0.85)
    plt.grid(axis='y', alpha=0.75)
    plt.xlabel('Dice Face',fontsize=15)
    plt.ylabel('Frequency',fontsize=15)
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.ylabel('Frequency',fontsize=15)
    plt.title('Histogram',fontsize=15)
    plt.show()
    plt.savefig('hist.png')

def dice_execute_multishot():
    print('executing 8-face dice program on qxelarator')
    qx = qxelarator.QX()
    qx.set('output/dice.qasm')
    dice_faces = []
    ntests = 100
    for i in range(ntests):
        qx.execute()
        res = [int(qx.get_measurement_outcome(q)) for q in range(nqubits)]
        dice_face = reduce(lambda x, y: 2*x+y, res, 0) +1
        dice_faces.append(dice_face)

    plot_histogram(dice_faces)
```

This will produce the histogram similar to the one shown below:


Histogram
12000
10000
8000
6000
4000
2000
0
1
2
3
4
5
6
7
8
Dice Face

1.5 DQCsim Simulation

This tutorial modifies the QX simulation tutorial to use DQCsim. In short, DQCsim is a framework that allows simulations to be constructed by chaining plugins operating on a stream of gates and measurement results, thus making it easier to play around with error models, gather runtime statistics, and connect different quantum simulators to different algorithm file formats. In this tutorial, we will use it to simulate the toy example modelling an 8-faced die with QX and QuantumSim's error models.

Note that DQCsim currently does not work on Windows. If you're using a Windows workstation, you'll need to work in a virtual machine or on a Linux server.

1.5.1 Dependencies

DQCsim and the plugins we'll be using can be installed using pip as follows:

```
python -m pip install dqcsim dqcsim-qx dqcsim-quantumsim dqcsim-cqasm
```

You'll probably need to prefix `sudo` to make that work, and depending on your Linux distribution you may need to substitute `python3`. If you don't have superuser access, you can add the `--user` flag, but you'll need to make sure that DQCsim's executables are in your system path. The easiest way to do that is figure out the path using `python -m pip uninstall dqcsim`, observe the directory that the `bin/dqcsim` file lives in, and add that to your path using `export PATH=$PATH:...`, replacing the `...` with the listed path from `/` to `bin`.

We'll also need to add some modules to the Python file from the QX die example:

```
from dqcsim.host import *
import shutil
```

1.5.2 Replicating the QXelarator results

The results we got when using QX directly are pretty easy to replicate. Here's how:

```
def dice_execute_singleshot():
    print('executing 8-face dice program on DQCsim using QX')

    # DQCsim disambiguates between input file formats based on file extension.
    # .qasm is already in use for OpenQASM files, so DQCsim uses .cq for cQASM
    shutil.copyfile('output/dice.qasm', 'output/dice.cq')

    # open the simulation context and run the simulation. the cQASM frontend
    # returns the results as a JSON object for us to parse througn run()
    with Simulator('output/dice.cq', 'qx') as sim:
        results = sim.run()

    # parse the measurement results
    res = [results['qubits'][q]['value'] for q in range(nqubits)]

    # convert the measurement results from 3 qubits to dice face value
    dice_face = reduce(lambda x, y: 2*x+y, res, 0) +1
    print('Dice face : {}'.format(dice_face))
```

The key is the `Simulator('test_output/dice.cq', 'qx')` expression wrapped in the `with` block, which constructs a DQCsim simulation using the `cq` frontend (based on the file extension, that's why we have to make a copy and rename OpenQL's output first) and the `qx` backend, wrapping the libqasm cQASM parser and QX's internals respectively.

1.5.3 Enabling QX's depolarizing channel error model

While not exactly useful for this particular algorithm, we can use DQCsim to enable QX's error model without having to edit the cQASM file. The easiest way to do that is to add a line before `sim.run()` to form

```
with Simulator('test_output/dice.cq', 'qx') as sim:
    sim.arb('back', 'qx', 'error', model='depolarizing_channel', error_probability=0.
↪2)
    results = sim.run()
```

This requires some explanation. The `sim.arb()` function (docs here) instructs DQCsim to send a so-called ArbCmd (short for "arbitrary command") to one of its plugins. In short, ArbCmds are DQCsim's way to let its users communicate intent between plugins, without DQCsim itself needing to know what's going on. DQCsim has no concept of error models and the likes built-in, so we need to use ArbCmds to configure them.

Its first argument specifies the plugin that the ArbCmd is intended for, where `'back'` is simply the default name for the backend plugin. You could also use the integer 1 to select the second plugin from the front, or -1 to select the first plugin from the back, as if it's indexing a Python list.

The second and third argument specify the interface and operation identifiers respectively. The interface identifier is usually just the name of the plugin, acting like a namespace or the name of a class, while the operation identifier specifies what to do, acting as a function or method name. You'll have to read the plugin documentation to see which interface/operation pairs are supported. Usually these are listed in the form `<interface>.<operation>`, as if we're using a parameter named `<operation>` from a class named `<interface>`.

Note that the semantics of ArbCmds are defined such that plugins will happily ignore any ArbCmd specifying an interface they don't support, but will complain when they support the interface but don't understand the operation. More information and the rationale for this can be found here.

Any remaining arguments are interpreted as arguments. Specifically, keyword arguments are transformed into the keys and values of a JSON object, in this case `{"model": "depolarizing_channel", "error_probability": 0.2}`. Positional arguments are interpreted as binary strings, but those are out of the scope of this tutorial (they're not that relevant in the Python world). Again, you'll have to read the plugin documentation to see what arguments are expected.

You won't be able to see much in the result of the algorithm, because it was already purely random. But you may notice that the log output of DQCsim now includes a *Depolarizing channel model inserted* ... *errors* from the backend.

1.5.4 Using QuantumSim instead

More interesting in terms of DQCsim's functionality is just how easy it is to change the simulator. All you have to do to simulate using QuantumSim instead of QX is change the `'qx'` in the `Simulation` constructor with `'quantumsim'`.

While QuantumSim is capable of much more, its DQCsim plugin currently only supports a qubit error model based on t1/t2 times. The arb for that, along with the modified *Simulator* constructor, looks like this:

```
with Simulator('test_output/dice.cq', 'quantumsim') as sim:
    sim.arb('back', 'quantumsim', 'error', t1=10.0, t2=20.0)
    results = sim.run()
```

For that to have any merit whatsoever, you'll have to modify the code such that we're at least simulating OpenQL's scheduled output, because it's based entirely on the timing of the circuit:

```
shutil.copyfile('output/dice_scheduled.qasm', 'output/dice.cq')
```

One thing the QuantumSim plugin does that the QX plugin doesn't is report the actual probability of a qubit measurement result. The *results* variable looks like this:

```
{
  "qubits": [
    {
      "value": 0,
      "raw": 0,
      "average": 0.0,
      "json": {"probability": 0.5},
      "binary": [[0, 0, 0, 0, 0, 0, 224, 63]]
    },
    {
      "value": 0,
      "raw": 0,
      "average": 0.0,
      "json": {"probability": 0.5},
      "binary": [[0, 0, 0, 0, 0, 0, 224, 63]]
    },
    {
      "value": 0,
      "raw": 0,
      "average": 0.0,
      "json": {"probability": 0.5},
      "binary": [[0, 0, 0, 0, 0, 0, 224, 63]]
    }
  ]
}
```

In particular, the `"json"` parameter lists data that the cQASM frontend received from the backend but doesn't know about, in this case showing that the probability for this outcome was exactly 0.5 for each of the three individual measurements.

1.5.5 Further reading

A more extensive Python tutorial for DQCsim can be found here. It (intentionally) does not depend on any of the plugins and doesn't use OpenQL, but hopefully the above illustrates that swapping out plugins is about the easiest thing you can do with DQCsim.

1.6 Where to go from here

This manual is not really complete yet, but hopefully the most important things have been treated, and you can figure out the rest from the much more complete *reference*. If not, you may try to read through some of the *older pages*. These have not yet been updated for version 0.9, but may still provide useful background information.

1.7 Python API

To use OpenQL from Python, you need to install the `qutechopenql` module using `pip`, and then

```
import openql as ql
```

Note: It used to be necessary to use `import openql.openql as ql`. This is still supported for backward compatibility.

The typical usage pattern for OpenQL is as follows:

- call `initialize()` to initialize the OpenQL library and clean up any leftovers from compiling a previous program;
- set some global options with `set_option()`;
- build a `Platform`;
- build a `Program` using the platform;
- build one or more `Kernel` s using the platform, and add them to the program with `add_kernel()`;
- compile the program with `compile()`.

Note: The `initialize()` didn't use to exist. Therefore, for backward compatibility, it is called automatically by the constructor of `Platform`, the constructor of `Compiler`, or by `set_option()` if it has not been called yet within this Python interpreter.

Warning: Calling `Program.compile()` or `Compiler.compile()` multiple times on the same program is currently *not* a supported use case: the `compile()` function mutates the contents of the program as compilation progresses. There are currently no API methods on `Program` or `Kernel` to read back the compilation result, but these may be added in the future. Therefore, if you want to compile a program multiple times, you'll have to rebuild the program from scratch each time.

For more advanced usage of the OpenQL compiler, the default compilation strategy might not be good enough, or the global options may be too restrictive for what you want. For this reason, the `Compiler` interface was recently added. The easiest way to make use of it is through `Platform.get_compiler()` or `Program.get_compiler()`; this returns a reference that allows you to change the default compilation strategy or set options for particular passes. Once you do this, however, any changes made to global options will cease to have an effect on that particular Platform/Program/Compiler triplet; you *must* use `Compiler.set_option()` and friends from that point onwards. Note that the names of the options in this interface have been revised compared to the global options, so you can't just replace a global `set_option()` with a `Compiler.set_option()` without a bit of work.

1.7.1 Index

Regular functions

initialize()	Initializes the OpenQL library, for as far as this must be done.
ensure_initialized()	Calls initialize() if it hasn't been called yet.
compile(-> None)	Entry point for compiling from a cQASM file directly, rather than using the Python API to build the program and platform.
get_version()	Returns the compiler's version string.
set_option(option, value)	Sets a global option for the compiler.
get_option(option)	Returns the current value for a global option.

Classes

Platform(*args)	Quantum platform description.
Program(name, platform[, qubit_count, . . .])	Represents a complete quantum program.
Kernel(name, platform[, qubit_count, . . .])	Represents a kernel of a quantum program, a.k.a.
CReg(id)	Wrapper for a classical integer register with the given index.
Operation(*args)	Wrapper for a classical operation.
Unitary(name, matrix)	Unitary matrix interface.
Compiler(*args)	Wrapper for the compiler/pass manager.
Pass()	Wrapper for a pass that belongs to some pass manager.
cQasmReader(*args)	Legacy cQASM reader interface.

Documentation retrieval functions

print_options()	Prints the documentation for all available global options.
dump_options()	Returns the result of print_options() as a string.
print_architectures()	Prints the documentation for all available target architectures.
dump_architectures()	Returns the result of print_architectures() as a string.
print_passes()	Prints the documentation for all available passes.
dump_passes()	Returns the result of print_passes() as a string.
print_resources()	Prints the documentation for all available scheduler resources.
dump_resources()	Returns the result of print_resources() as a string.
print_platform_docs()	Prints the documentation for platform configuration files.
dump_platform_docs()	Returns the result of print_platform_docs() as a string.

1.7.2 Platform class

class openql.**Platform**(**args*)

Quantum platform description. This describes everything that the compiler needs to know about the target quantum chip, instruments, etc. Platforms are created from either the default configuration for a particular architecture variant or from JSON (+ comments) configuration files: there is no way to modify a platform using the API, and introspection is limited. Instead, if you want to use a custom configuration, you will need to write a JSON configuration file for it, or use get_platform_json() and from_json() to modify an existing one from within Python.

The syntax of the platform configuration file is too extensive to describe here. It has its own section in the manual.

In addition to the platform itself, the Platform object provides an interface for obtaining a Compiler object. This object describes the *strategy* for transforming the quantum algorithm to something that can be executed on the device described by the platform. You can think of the difference between them as the difference between a verb and a noun: the platform describes something that just exists, while the compilation strategy describes how to get there.

The (initial) strategy can be set using a separate configuration file (compiler_config), directly from within the platform configuration file, or one can be inferred based on the previously hardcoded defaults. Unlike the platform itself however, an extensive API is available for adjusting the strategy as you see fit; just use get_compiler() to get a reference to a Compiler object that may be used for this purpose. If you don't do anything with the compiler methods and object, don't specify the compiler_config_file parameter, and the "eqasm_compiler" key of the platform configuration file refers to one of the previously-hardcoded compiler, a strategy will be generated to mimic the old logic for backward compatibility.

Eight constructors are provided:

- Platform(): shorthand for Platform('none', 'none').
- Platform(name): shorthand for Platform(name, name).
- Platform(name, platform_config): builds a platform with the given name (only used for log messages) and platform configuration, the latter of which can be either a recognized platform name with or without variant suffix (for example "cc" or "cc_light.s7"), or a path to a JSON configuration filename.
- Platform(name, platform_config, compiler_config): as above, but specifies a custom compiler configuration file in addition.
- Platform.from_json(name, platform_config_json): instead of loading the platform JSON data from a file, it is taken from its Python object representation (as per json.loads()/dumps()).
- Platform.from_json(name, platform_config_json, compiler_config): as above, with compiler JSON file override.
- Platform.from_json_string(name, platform_config_json): as from_json, but loads the data from a string rather than a Python object.
- Platform.from_json_string(name, platform_config_json, compiler_config): as from_json, but loads the data from a string rather than a Python object.

property name

The user-given name of the platform.

property config_file

The configuration file that the platform was loaded from.

__init__(*self*, *name: str*, *platform_config: str*, *compiler_config: str*) → *Platform*
__init__(*self*, *name: str*, *platform_config: str*) → *Platform*
__init__(*self*, *name: str*) → *Platform*

`__init__`(*self*) → *Platform*

`static from_json_string`(*name: str*, *platform_config_json: str*, *compiler_config: str*) → *Platform*

`static from_json_string`(*name: str*, *platform_config_json: str*) → *Platform*

Alternative constructor. Instead of the platform JSON data being loaded from a file, they are loaded from the given string. See also from_json().

Parameters

- **`name`** (`str`) – The name for the platform.
- **`platform_config_json`** (`str`) – The platform JSON configuration data as a string. This will accept anything that the normal constructor accepts when it reads the configuration from a file.
- **`compiler_config`** (`str`) – Optional compiler configuration JSON filename. This is *NOT* JSON data.

Returns The constructed platform.

Return type *Platform*

`static get_platform_json_string`(*platform_config: str*) → str

`static get_platform_json_string`() → str

Returns the default platform configuration data as a JSON + comments string. The comments use double-slash syntax. Note that JSON itself does not support such comments (or comments of any kind), so these comments need to be removed from the data before the JSON data can be parsed.

Parameters **`platform_config`** (`str`) – The platform configuration. Same syntax as the platform constructor, so this supports architecture names, architecture variant names, or JSON filenames. In the latter case, this function just loads the file contents into a string and returns it.

Returns The JSON + comments data for the given platform configuration string.

Return type str

`get_qubit_number`(*self*) → int

Returns the number of qubits in the platform.

Parameters **`None`** –

Returns The number of qubits in the platform.

Return type int

`print_info`(*self*) → None

Prints some basic information about the platform.

Parameters **`None`** –

Returns

Return type None

`dump_info`(*self*) → str

Returns the result of print_info() as a string.

Parameters **`None`** –

Returns The result of print_info() as a string.

Return type str

`get_info`(*self*) → str

Old alias for dump_info(). Deprecated.

Parameters **`None`** –

Returns The result of print_info() as a string.

Return type str

`has_compiler`(*self*) → bool

Returns whether a custom compiler configuration has been attached to this platform. When this is the case, programs constructed from this platform will use it to implement Program.compile(), rather than generating the compiler in-place from defaults and global options during the call.

Parameters **`None`** –

Returns Whether a custom compiler configuration has been attached to this platform.

Return type bool

`get_compiler`(*self*) → *Compiler*

Returns the custom compiler configuration associated with this platform. If no such configuration exists yet, the default one is created, attached, and returned.

Parameters **`None`** –

Returns A Compiler object that may be used to introspect or modify the compilation strategy associated with this platform.

Return type *Compiler*

`set_compiler`(*self*, *compiler:* Compiler) → None

Sets the compiler associated with this platform. Any programs constructed from this platform after this call will use the given compiler.

Parameters **`compiler`** (`Compiler`) – The new compiler configuration.

Returns

Return type None

`static from_json`(*name: str*, *platform_config_json: Dict[. . .]*, *compiler_config: str*) → *Platform*
`static from_json`(*name: str*, *platform_config_json: Dict[. . .]*) → *Platform*

Alternative constructor. Instead of the platform JSON data being loaded from a file, they are loaded from the given Python object representation of the JSON platform configuration data.

This is useful when you only need to change a builtin platform for some architecture variant a little bit. In this case, you can get the default JSON data using get_platform_json(), introspect and modify it programmatically, and then use this to build the platform from the modified configuration.

Parameters

- **`name`** (*`str`*) – The name for the platform.
- **`platform_config_json`** (*`JSON-like object`*) – The platform JSON configuration data in Python object representation (anything accepted by json.dumps()).
- **`compiler_config`** (*`str`*) – Optional compiler configuration JSON filename. This is *NOT* JSON data.

Returns The constructed platform.

Return type *Platform*

static get_platform_json(*platform_config: str) -> Dict[...] get_platform_json()* → Dict[...]
Returns the default platform configuration data as the Python object representation of the JSON data (as returned by json.loads()).

Parameters **platform_config** (*str*) – The platform configuration. Same syntax as the platform constructor, so this supports architecture names, architecture variant names, or JSON filenames. In the latter case, this function just parses the file contents and returns it.

Returns The Python object representation of the JSON data corresponding to the given platform configuration string.

Return type str

1.7.3 Program class

class openql.**Program**(*name*, *platform*, *qubit_count=0*, *creg_count=0*, *breg_count=0*)
Represents a complete quantum program.

The constructor creates a new program with the given name, using the given platform. The third, fourth, and fifth arguments optionally specify the desired number of qubits, classical integer registers, and classical bit registers. If not specified, the number of qubits is taken from the platform, and no classical or bit registers will be allocated.

property name
The name given to the program by the user.

property platform
The platform associated with the program.

property qubit_count
The number of (virtual) qubits allocated for the program.

property creg_count
The number of classical integer registers allocated for the program.

property breg_count
The number of classical bit registers allocated for the program.

__init__(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*, *creg_count: int = 0*, *breg_count: int = 0*) → *Program*
__init__(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*, *creg_count: int = 0*) → *Program*
__init__(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*) → *Program*
__init__(*self*, *name: str*, *platform:* Platform) → *Program*

add_kernel(*self*, *k:* Kernel) → None
Adds an unconditionally-executed kernel to the end of the program.

Parameters **k** (Kernel) – The kernel to add.

Returns

Return type None

add_program(*self*, *p:* Program) → None
Adds an unconditionally-executed subprogram to the end of the program.

Parameters **p** (Program) – The subprogram to add.

Returns

Return type None

add_if(*self*, *k*: Kernel, *operation*: Operation) → None
add_if(*self*, *p*: Program, *operation*: Operation) → None

Adds a conditionally-executed kernel or subprogram to the end of the program. The kernel/subprogram will be executed if the given classical condition evaluates to true.

Parameters

- **k** (`Kernel`) – The kernel to add.
- **p** (`Program`) – The subprogram to add.
- **operation** (`Operation`) – The operation that must evaluate to true for the kernel/subprogram to be executed.

Returns

Return type None

add_if_else(*self*, *k_if*: Kernel, *k_else*: Kernel, *operation*: Operation) → None
add_if_else(*self*, *p_if*: Program, *p_else*: Program, *operation*: Operation) → None

Adds two conditionally-executed kernels/subprograms with inverted conditions to the end of the program. The first kernel/subprogram will be executed if the given classical condition evaluates to true; the second kernel/subprogram will be executed if it evaluates to false.

Parameters

- **k_if** (`Kernel`) – The kernel to execute when the condition evaluates to true.
- **p_if** (`Program`) – The subprogram to execute when the condition evaluates to true.
- **k_else** (`Kernel`) – The kernel to execute when the condition evaluates to false.
- **p_else** (`Program`) – The subprogram to execute when the condition evaluates to false.
- **operation** (`Operation`) – The operation that determines which kernel/subprogram will be executed.

Returns

Return type None

add_do_while(*self*, *k*: Kernel, *operation*: Operation) → None
add_do_while(*self*, *p*: Program, *operation*: Operation) → None

Adds a kernel/subprogram that will be repeated until the given classical condition evaluates to true. The kernel/subprogram is executed at least once, since the condition is evaluated at the end of the loop body.

Parameters

- **k** (`Kernel`) – The kernel that represents the loop body.
- **p** (`Program`) – The subprogram that represents the loop body.
- **operation** (`Operation`) – The operation that must evaluate to true at the end of the loop body for the loop body to be executed again.

Returns

Return type None

add_for(*self*, *k*: Kernel, *iterations*: *int*) → None
add_for(*self*, *p*: Program, *iterations*: *int*) → None

Adds an unconditionally-executed kernel/subprogram that will loop for the given number of iterations.

Parameters

- **k** (`Kernel`) – The kernel that represents the loop body.

- **p** (`Program`) – The subprogram that represents the loop body.
- **iterations** (*int*) – The number of loop iterations.

Returns

Return type None

set_sweep_points (*self*, *sweep_points: List[float]*) → None

Sets sweep point information for the program.

NOTE: sweep points functionality is deprecated and may be removed at any time. Do not use it in new programs.

Parameters **sweep_points** (*List[float]*) – The list of sweep points.

Returns

Return type None

get_sweep_points (*self*) → List[float]

Returns the configured sweep point information for the program.

NOTE: sweep points functionality is deprecated and may be removed at any time. Do not use it in new programs.

Parameters **None** –

Returns The previously configured sweep point information for the program, or an empty list if none were configured.

Return type List[float]

set_config_file (*self*, *config_file_name: str*) → None

Sets the name of the file that the sweep points will be written to.

NOTE: sweep points functionality is deprecated and may be removed at any time. Do not use it in new programs.

Parameters **config_file_name** (*str*) – The name of the file that the sweep points are to be written to.

Returns

Return type None

has_compiler (*self*) → bool

Whether a custom compiler configuration has been attached to this program. When this is the case, it will be used to implement compile(), rather than generating the compiler in-place from defaults and global options during the call.

Parameters **None** –

Returns Whether a custom compiler configuration has been attached to this program.

Return type bool

get_compiler (*self*) → *Compiler*

Returns the custom compiler configuration associated with this program. If no such configuration exists yet, the default one is created, attached, and returned.

Parameters **None** –

Returns A Compiler object that may be used to introspect or modify the compilation strategy associated with this program.

Return type *Compiler*

set_compiler(*self*, *compiler: Compiler*) → None
Sets the compiler associated with this program. It will then be used for compile().

Parameters **compiler** (`Compiler`) – The new compiler configuration.

Returns

Return type None

compile(*self*) → None
Compiles the program.

Parameters **None** –

Returns

Return type None

print_interaction_matrix(*self*) → None
Prints the interaction matrix for each kernel in the program.

Parameters **None** –

Returns

Return type None

write_interaction_matrix(*self*) → None
Writes the interaction matrix for each kernel in the program to a file. This is one of the few functions that still uses the global output_dir option.

Parameters **None** –

Returns

Return type None

1.7.4 Kernel class

class openql.**Kernel**(*name*, *platform*, *qubit_count=0*, *creg_count=0*, *breg_count=0*)
Represents a kernel of a quantum program, a.k.a. a basic block. Kernels are just sequences of gates with no classical control-flow in between: they may end in a (conditional) branch to the start of another kernel, but otherwise, they may only consist of quantum gates and mixed quantum-classical data flow operations.

The constructor creates a new kernel with the given name, using the given platform. The third, fourth, and fifth arguments optionally specify the desired number of qubits, classical integer registers, and classical bit registers. If not specified, the number of qubits is taken from the platform, and no classical or bit registers will be allocated.

Currently, the contents of a kernel can only be constructed by adding gates and classical data flow instructions in the order in which they are to be executed, and there is no way to get information about which gates are in the kernel after the fact. If you need this kind of bookkeeping, you will have to wrap OpenQL's kernels for now.

Classical flow-control is configured when a completed kernel is added to a program, via basic structured control-flow paradigms (if-else, do-while, and loops with a fixed iteration count).

NOTE: the way gates are represented in OpenQL is on the list to be completely revised. Currently OpenQL works using a mixture of "default gates" and the "custom gates" that you can specify in the platform configuration file, but these two things are not orthogonal and largely incompatible with each other, yet are currently used interchangeably. Furthermore, there is no proper way to specify lists of generic arguments to a gate, leading to lots of code duplication inside OpenQL and long gate() argument lists. Finally, the semantics of gates are largely derived by undocumented and somewhat heuristic string comparisons with the names of gates, which is terrible design in combination with user-specified instruction sets via the platform configuration file. The interface for adding simple *quantum* gates to a kernel is something we want to keep 100% backward compatible, but the more advanced gate() signatures may change in the (near) future.

NOTE: classical logic is on the list to be completely revised. This interface may change in the (near) future.

NOTE: the higher-order functions for constructing controlled kernels and conjugating kernels have not been maintained for a while and thus probably won't work right. They may be removed entirely in a later version of OpenQL.

`property name`
The name of the kernel as given by the user.

`property platform`
The platform that the kernel was built for.

`property qubit_count`
The number of (virtual) qubits allocated for the kernel.

`property creg_count`
The number of classical integer registers allocated for the kernel.

`property breg_count`
The number of classical bit registers allocated for the kernel.

`__init__`(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*, *creg_count: int = 0*, *breg_count: int = 0*) → *Kernel*
`__init__`(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*, *creg_count: int = 0*) → *Kernel*
`__init__`(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*) → *Kernel*
`__init__`(*self*, *name: str*, *platform:* Platform) → *Kernel*

`get_custom_instructions`(*self*) → str
Old alias for dump_custom_instructions(). Deprecated.

Parameters **`None`** –

Returns A newline-separated list of all custom gates supported by the platform.

Return type str

`print_custom_instructions`(*self*) → None
Prints a list of all custom gates supported by the platform.

Parameters **`None`** –

Returns

Return type None

`dump_custom_instructions`(*self*) → str
Returns the result of print_custom_instructions() as a string.

Parameters **`None`** –

Returns A newline-separated list of all custom gates supported by the platform.

Return type str

`gate_preset_condition`(*self*, *condstring: str*, *condregs: List[int]*) → None
Sets the condition for all gates subsequently added to this kernel. Thus, essentially shorthand notation. Reset with gate_clear_condition().

Parameters

- **`condstring`** (`str`) – Must be one of:
  - "COND_ALWAYS" or "1": no condition; gate is always executed.

  - ”COND_NEVER” or “0”: no condition; gate is never executed.
  - ”COND_UNARY” or “” (empty): gate is executed if the single bit specified via condregs is 1.
  - ”COND_NOT” or “!”: gate is executed if the single bit specified via condregs is 0.
  - ”COND_AND” or “&”: gate is executed if the two bits specified via condregs are both 1.
  - ”COND_NAND” or “!&”: gate is executed if either of the two bits specified via condregs is zero.
  - ”COND_OR” or “|”: gate is executed if either of the two bits specified via condregs is one.
  - ”COND_NOR” or “!|”: no condition; gate is always executed.
- **condregs** (*List[int]*) – Depending on condstring, must be a list of 0, 1, or 2 breg indices.

Returns

Return type None

gate_clear_condition (*self*) → None

Clears a condition previously set via gate_preset_condition().

Parameters **None** –

Returns

Return type None

gate (*self*, *name: str*, *q0: int*) → None
gate (*self*, *name: str*, *q0: int*, *q1: int*) → None
gate (*self, name: str, qubits: List[int], duration: int = 0, angle: float = 0.0, bregs: List[int], condstring: str, condregs: List[int]*) → None
gate (*self, name: str, qubits: List[int], duration: int = 0, angle: float = 0.0, bregs: List[int], condstring: str*) → None
gate (*self, name: str, qubits: List[int], duration: int = 0, angle: float = 0.0, bregs: List[int]*) → None
gate (*self*, *name: str*, *qubits: List[int]*, *duration: int = 0*, *angle: float = 0.0*) → None
gate (*self*, *name: str*, *qubits: List[int]*, *duration: int = 0*) → None
gate (*self*, *name: str*, *qubits: List[int]*) → None
gate (*self*, *name: str*, *qubits: List[int]*, *destination:* CReg) → None
gate (*self*, *u:* Unitary, *qubits: List[int]*) → None

Main function for appending arbitrary quantum gates.

Parameters

- **name** (*str*) – The name of the gate. Note that OpenQL currently uses string comparisons with these names all over the place to derive functionality, and to derive what the actual arguments do. This is inherently a bad idea and something we want to move away from, so documenting it all would not be worthwhile. For now, just use common sense, and you’ll probably be okay.
- **q0** (*int*) – Index of the first qubit to apply the gate to. For controlled gates, this is the control qubit.
- **q1** (*int*) – Index of the second qubit to apply the gate to. For controlled gates, this is the target qubit.
- **qubits** (*List[int]*) – The full list of qubit indices to apply the gate to.

- **`duration`** (*`int`*) – Gate duration in nanoseconds, or 0 to use the default value from the platform configuration file. This is primarily intended to be used for wait gates.
- **`angle`** (*`float`*) – Rotation angle in radians for gates that use it (rx, ry, rz, etc). Ignored for all other gates.
- **`bregs`** (*`List[int]`*) – The full list of bit register argument indices for the gate, excluding any bit registers used for conditional execution. Currently only used for the measure gate, which may be given an explicit bit register index to return its result in. If no such register is specified, the result is assumed to implicitly go to the bit register with the same index as the qubit being measured. Ignored for gates that don't use bit registers.
- **`condstring`** (*`str`*) – If specified, must be one of:
  - "COND_ALWAYS" or "1": no condition; gate is always executed.
  - "COND_NEVER" or "0": no condition; gate is never executed.
  - "COND_UNARY" or "" (empty): gate is executed if the single bit specified via condregs is 1.
  - "COND_NOT" or "!": gate is executed if the single bit specified via condregs is 0.
  - "COND_AND" or "&": gate is executed if the two bits specified via condregs are both 1.
  - "COND_NAND" or "!&": gate is executed if either of the two bits specified via condregs is zero.
  - "COND_OR" or "|": gate is executed if either of the two bits specified via condregs is one.
  - "COND_NOR" or "!|": no condition; gate is always executed.
- **`condregs`** (*`List[int]`*) – Depending on condstring, must be a list of 0, 1, or 2 breg indices.
- **`destination`** (`CReg`) – An integer control register that receives the result of the mixed quantum-classical gate identified by name.
- **`u`** (`Unitary`) – The unitary gate to insert.

Returns

Return type None

`condgate` (*self*, *name: str*, *qubits: List[int]*, *condstring: str*, *condregs: List[int]*) → None

Alternative function for appending normal conditional quantum gates. Avoids having to specify duration, angle, and bregs for gates that don't need it.

Parameters

- **`name`** (*`str`*) – The name of the gate. Note that OpenQL currently uses string comparisons with these names all over the place to derive functionality, and to derive what the actual arguments do. This is inherently a bad idea and something we want to move away from, so documenting it all would not be worthwhile. For now, just use common sense, and you'll probably be okay.
- **`qubits`** (*`List[int]`*) – The full list of qubit indices to apply the gate to.
- **`condstring`** (*`str`*) – If specified, must be one of:
  - "COND_ALWAYS" or "1": no condition; gate is always executed.
  - "COND_NEVER" or "0": no condition; gate is never executed.

  - ”COND_UNARY” or “” (empty): gate is executed if the single bit specified via condregs is 1.
  - ”COND_NOT” or “!”: gate is executed if the single bit specified via condregs is 0.
  - ”COND_AND” or “&”: gate is executed if the two bits specified via condregs are both 1.
  - ”COND_NAND” or “!&”: gate is executed if either of the two bits specified via condregs is zero.
  - ”COND_OR” or “|”: gate is executed if either of the two bits specified via condregs is one.
  - ”COND_NOR” or “!|”: no condition; gate is always executed.
- **condregs** (*List[int]*) – Depending on condstring, must be a list of 0, 1, or 2 breg indices.

Returns

Return type None

`classical`(*self*, *destination:* CReg, *operation:* Operation) → None
`classical`(*self*, *operation: str*) → None

Appends a classical assignment gate to the circuit. The classical integer register is assigned to the result of the given operation.

Parameters

- **destination** (CReg) – An integer control register at the left-hand side of the classical assignment gate.
- **operation** (Operation) – The expression to evaluate on the right-hand side of the classical assignment gate.

Returns

Return type None

`identity`(*self*, *q0: int*) → None

Shorthand for an “identity” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

`hadamard`(*self*, *q0: int*) → None

Shorthand for appending a “hadamard” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

`s`(*self*, *q0: int*) → None

Shorthand for appending an “s” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

sdag (*self*, *q0: int*) → None
Shorthand for appending an "sdag" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

t (*self*, *q0: int*) → None
Shorthand for appending a "t" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

tdag (*self*, *q0: int*) → None
Shorthand for appending a "tdag" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

x (*self*, *q0: int*) → None
Shorthand for appending an "x" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

y (*self*, *q0: int*) → None
Shorthand for appending a "y" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

z (*self*, *q0: int*) → None
Shorthand for appending a "z" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

rx90 (*self*, *q0: int*) → None
Shorthand for appending an "rx90" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

mrx90 (*self*, *q0: int*) → None
Shorthand for appending an "mrx90" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

rx180 (*self*, *q0: int*) → None
Shorthand for appending an “rx180” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

ry90 (*self*, *q0: int*) → None
Shorthand for appending an “ry90” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

mry90 (*self*, *q0: int*) → None
Shorthand for appending an “mry90” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

ry180 (*self*, *q0: int*) → None
Shorthand for appending an “ry180” gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

rx (*self*, *q0: int*, *angle: float*) → None
Shorthand for appending an “rx” gate with a single qubit and the given rotation in radians.

Parameters

- **q0** (*int*) – The qubit to apply the gate to.
- **angle** (*float*) – The rotation angle in radians.

Returns

Return type None

ry (*self*, *q0: int*, *angle: float*) → None
Shorthand for appending an “ry” gate with a single qubit and the given rotation in radians.

Parameters

- **q0** (*int*) – The qubit to apply the gate to.
- **angle** (*float*) – The rotation angle in radians.

Returns

Return type None

rz (*self*, *q0: int*, *angle: float*) → None
Shorthand for appending an “rz” gate with a single qubit and the given rotation in radians.

Parameters

- **`q0`** (*`int`*) – The qubit to apply the gate to.
- **`angle`** (*`float`*) – The rotation angle in radians.

Returns

Return type None

`measure` (*self*, *q0: int*) → None
`measure` (*self*, *q0: int*, *b0: int*) → None
Shorthand for appending a "measure" gate with a single qubit and implicit or explicit result bit register.

Parameters

- **`q0`** (*`int`*) – The qubit to measure.
- **`b0`** (*`int`*) – The bit register to store the result in. If not specified, the result will be placed in the bit register corresponding to the index of the measured qubit.

Returns

Return type None

`prepz` (*self*, *q0: int*) → None
Shorthand for appending a "prepz" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to prepare in the Z basis.

Returns

Return type None

`cnot` (*self*, *q0: int*, *q1: int*) → None
Shorthand for appending a "cnot" gate with two qubits.

Parameters

- **`q0`** (*`int`*) – The control qubit index.
- **`q1`** (*`int`*) – The target qubit index

Returns

Return type None

`cphase` (*self*, *q0: int*, *q1: int*) → None
Shorthand for appending a "cphase" gate with two qubits.

Parameters

- **`q0`** (*`int`*) – The first qubit index.
- **`q1`** (*`int`*) – The second qubit index.

Returns

Return type None

`cz` (*self*, *q0: int*, *q1: int*) → None
Shorthand for appending a "cz" gate with two qubits.

Parameters

- **`q0`** (*`int`*) – The first qubit index.
- **`q1`** (*`int`*) – The second qubit index.

Returns

Return type None

toffoli (*self*, *q0: int*, *q1: int*, *q2: int*) → None

Shorthand for appending a "toffoli" gate with three qubits.

Parameters

- **q0** (*int*) – The first control qubit index.
- **q1** (*int*) – The second control qubit index.
- **q2** (*int*) – The target qubit index.

Returns

Return type None

clifford (*self*, *id: int*, *q0: int*) → None

Shorthand for appending the Clifford gate with the specific number using the minimal number of rx90, rx180, mrx90, ry90, ry180, and mry90 gates.

Parameters

- **id** (*int*) – The Clifford gate expansion index:
  - 0: no gates inserted.
  - 1: ry90; rx90
  - 2: mrx90, mry90
  - 3: rx180
  - 4: mry90, mrx90
  - 5: rx90, mry90
  - 6: ry180
  - 7: mry90, rx90
  - 8: rx90, ry90
  - 9: rx180, ry180
  - 10: ry90, mrx90
  - 11: mrx90, ry90
  - 12: ry90, rx180
  - 13: mrx90
  - 14: rx90, mry90, mrx90
  - 15: mry90
  - 16: rx90
  - 17: rx90, ry90, rx90
  - 18: mry90, rx180
  - 19: rx90, ry180
  - 20: rx90, mry90, rx90
  - 21: ry90
  - 22: mrx90, ry180

  - 23: rx90, ry90, mrx90
- **q0** (*int*) – The target qubit.

Returns

Return type None

wait (*self*, *qubits: List[int]*, *duration: int*) → None

Shorthand for appending a "wait" gate with the specified qubits and duration in nanoseconds. If no qubits are specified, the wait applies to all qubits instead (a wait with no qubits is meaningless). Note that the duration will usually end up being rounded up to multiples of the platform's cycle time.

Parameters

- **qubits** (*List[int]*) – The list of qubits to apply the wait gate to. If empty, the list will be replaced with the set of all qubits.
- **duration** (*int*) – The duration of the wait gate in nanoseconds.

Returns

Return type None

barrier (*self*, *qubits: List[int]*) → None
barrier (*self*) → None

Shorthand for appending a "wait" gate with the specified qubits and duration 0. If no qubits are specified, the wait applies to all qubits instead (a wait with no qubits is meaningless).

Parameters **qubits** (*List[int]*) – The list of qubits to apply the wait gate to. If empty or unspecified, the list will be replaced with the set of all qubits.

Returns

Return type None

display (*self*) → None

Shorthand for appending a "display" gate with no qubits.

Parameters **None** –

Returns

Return type None

diamond_excite_mw (*self*, *envelope: int*, *duration: int*, *frequency: int*, *phase: int*, *amplitude: int*, *qubit: int*) → None

Appends the diamond "excite_mw" instruction.

Parameters

- **envelope** (*int*) – The envelope of the microwave.
- **duration** (*int*) – The duration of the microwave in nanoseconds.
- **frequency** (*int*) – The frequency of the microwave in kilohertz.
- **phase** (*int*) – The phase of the microwave.
- **amplitude** (*int*) – The amplitude of the microwave.
- **qubit** (*int*) – The target qubit index.

Returns

Return type None

diamond_memswap (*self*, *qubit: int*, *nuclear_qubit: int*) → None
Appends the diamond "memswap" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **nuclear_qubit** (*int*) – The index of the nuclear spin qubit of the color center.

Returns

Return type None

diamond_qentangle (*self*, *qubit: int*, *nuclear_qubit: int*) → None
Appends the diamond "qentangle" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **nuclear_qubit** (*int*) – The index of the nuclear spin qubit of the color center.

Returns

Return type None

diamond_sweep_bias (*self*, *qubit: int*, *value: int*, *dacreg: int*, *start: int*, *step: int*, *max: int*, *memaddress: int*) → None
Appends the diamond "sweep_bias" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **value** (*int*) – The value that has to be send to the dac for biasing.
- **dacreg** (*int*) – The index or address of the register of the dac.
- **start** (*int*) – The start frequency value of the sweep.
- **step** (*int*) – The step frequency value of the sweep.
- **max** (*int*) – The maximum frequency value of the sweep.
- **memaddress** (*int*) – The memory address to write the results to.

Returns

Return type None

diamond_crc (*self*, *qubit: int*, *threshold: int*, *value: int*) → None
Appends the diamond "crc" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **treshold** (*int*) – The threshold value that has to be matched.
- **value** (*int*) – The value of the voltage sent to the dac.

Returns

Return type None

diamond_rabi_check (*self*, *qubit: int*, *measurements: int*, *duration: int*, *t_max: int*) → None
Appends the diamond "rabi_check" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **measurements** (*int*) – How manu measurements have to be recorded.
- **duration** (*int*) – The starting value of the duration of the microwave.
- **t_max** (*int*) – The value of the voltage sent to the dac.

Returns

Return type None

controlled(*self*, *k: Kernel*, *control_qubits: List[int]*, *ancilla_qubits: List[int]*) → None

Appends a controlled kernel. The number of control and ancilla qubits must be equal.

Parameters

- **k** (Kernel) – The kernel to make controlled.
- **control_qubits** (*List[int]*) – The qubits that control the kernel.
- **ancilla_qubits** (*List[int]*) – The ancilla qubits to use to make the kernel controlled. The number of ancilla qubits must equal the number of control qubits.

Returns

Return type None

conjugate(*self*, *k: Kernel*) → None

Appends the conjugate of the given kernel to this kernel.

NOTE: this high-level functionality is poorly/not maintained, and relies on default gates, which are on the list for removal.

Parameters **k** (Kernel) – The kernel to conjugate.

Returns

Return type None

1.7.5 CReg class

class openql.**CReg**(*id*)

Wrapper for a classical integer register with the given index.

NOTE: classical logic is on the list to be completely revised. This interface may change in the (near) future.

__init__(*self*, *id: int*) → *CReg*

1.7.6 Operation class

class openql.**Operation**(**args*)

Wrapper for a classical operation.

A classical operation acts as a simple expression that returns an integer or a boolean. The expression can be a literal number (val), a single CReg (lop, primarily used for assignments), a unary function applied to one CReg (op/rop), or a binary function applied to two CRegs (lop/op/rop).

Function selection is done via strings. The following unary functions are recognized:

- '~': bitwise NOT.

The following binary functions are recognized:

- '+': addition.
- '-': subtraction.
- '&': bitwise AND.
- '|': bitwise OR.
- '^': bitwise XOR.
- '==': equality.
- '!=': inequality.
- '>': greater-than.
- '>=': greater-or-equal.
- '<': less-than.
- '<=': less-or-equal.

NOTE: classical logic is on the list to be completely revised. This interface may change in the (near) future.

`__init__`(*self*, *lop: CReg*, *op: str*, *rop: CReg*) → *Operation*
`__init__`(*self*, *op: str*, *rop: CReg*) → *Operation*
`__init__`(*self*, *lop: CReg*) → *Operation*
`__init__`(*self*, *val: int*) → *Operation*

1.7.7 Unitary class

class `openql.`**`Unitary`**(*name*, *matrix*)
Unitary matrix interface.

The constructor creates a unitary gate from the given row-major, square, unitary matrix.

property name
The name given to the unitary gate.

`__init__`(*self*, *name: str*, *matrix: vectorc*) → *Unitary*

`decompose`(*self*) → None
Explicitly decomposes the gate. Does not need to be called; it will be called automatically when the gate is added to the kernel.

Parameters **`None`** –

Returns

Return type None

static is_decompose_support_enabled() → bool
Returns whether OpenQL was built with unitary decomposition support enabled.

Parameters **`None`** –

Returns Whether OpenQL was built with unitary decomposition support enabled.

Return type bool

1.7.8 Compiler class

class openql.**Compiler**(**args*)

Wrapper for the compiler/pass manager.

This allows you to change the compilation strategy, if the defaults are insufficient for your application. You can get access to a Compiler via several methods:

- using Platform.get_compiler();
- using Program.get_compiler();
- using one of the constructors.

Using the constructors, you can get an empty compiler (by specifying no arguments or only specifying name), a default compiler for a given platform (by specifying a name and a platform), or a compiler based on a compiler configuration JSON file (by specifying a name and a filename). For the structure of this JSON file, refer to the configuration section of the ReadTheDocs documentation or, equivalently, the result of *openql.print_compiler_docs()*.

property name

User-given name for this compiler.

NOTE: not actually used for anything. It's only here for consistency with the rest of the API objects.

__init__(*self*, *name: str*) → *Compiler*
__init__(*self*) → *Compiler*
__init__(*self*, *name: str*, *filename: str*) → *Compiler*
__init__(*self*, *name: str*, *platform:* Platform) → *Compiler*

print_pass_types(*self*) → None

Prints documentation for all available pass types, as well as the option documentation for the passes.

Parameters **None** –

Returns

Return type None

dump_pass_types(*self*) → str

Returns documentation for all available pass types, as well as the option documentation for the passes.

Parameters **None** –

Returns The list of pass types and their documentation as a multiline string.

Return type str

print_strategy(*self*) → None

Prints the currently configured compilation strategy.

Parameters **None** –

Returns

Return type None

dump_strategy(*self*) → str

Returns the currently configured compilation strategy as a string.

Parameters **None** –

Returns The current compilation strategy as a multiline string.

Return type str

set_option(*self*, *path: str*, *value: str*, *must_exist: bool = True*) → int
set_option(*self*, *path: str*, *value: str*) → int

Sets a pass option. The path must consist of the target pass instance name and the option name, separated by a period. The former may include * or ? wildcards. If must_exist is set an exception will be thrown if none of the passes were affected, otherwise 0 will be returned.

Parameters

- **path** (*str*) – The path to the option, consisting of the pass name and the option name separated by a period.
- **value** (*str*) – The value to set the option to.
- **must_exist** (*bool*) – When set, an exception will be thrown when no options matched the path.

Returns The number of pass options affected.

Return type int

get_option(*self*, *path: str*) → str

Returns the current value of an option.

Parameters **path** (*str*) – The path to the option, consisting of pass name and the actual option name separated by a period.

Returns The value of the option. If the option has not been set, the default value is returned.

Return type str

append_pass(*self*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
append_pass(*self*, *type_name: str*, *instance_name: str*) → *Pass*
append_pass(*self*, *type_name: str*) → *Pass*

Appends a pass to the end of the pass list. Returns a reference to the constructed pass.

Parameters

- **type_name** (*str*) – The type of the pass to add.
- **instance_name** (*str*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **options** (*dict[str, str]*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

prefix_pass(*self*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
prefix_pass(*self*, *type_name: str*, *instance_name: str*) → *Pass*
prefix_pass(*self*, *type_name: str*) → *Pass*

Appends a pass to the beginning of the pass list. Returns a reference to the constructed pass.

Parameters

- **type_name** (*str*) – The type of the pass to add.
- **instance_name** (*str*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **options** (*dict[str, str]*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

insert_pass_after(*self*, *target: str*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
insert_pass_after(*self*, *target: str*, *type_name: str*, *instance_name: str*) → *Pass*
insert_pass_after(*self*, *target: str*, *type_name: str*) → *Pass*

Inserts a pass immediately after the target pass (named by instance). If target does not exist, an exception is thrown. Returns a reference to the constructed pass.

Parameters

- **target** (*str*) – The name of the pass to insert the new pass after.
- **type_name** (*str*) – The type of the pass to add.
- **instance_name** (*str*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **options** (*dict[str, str]*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

insert_pass_before(*self*, *target: str*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
insert_pass_before(*self*, *target: str*, *type_name: str*, *instance_name: str*) → *Pass*
insert_pass_before(*self*, *target: str*, *type_name: str*) → *Pass*

Inserts a pass immediately before the target pass (named by instance). If target does not exist, an exception is thrown. Returns a reference to the constructed pass.

Parameters

- **target** (*str*) – The name of the pass to insert the new pass before.
- **type_name** (*str*) – The type of the pass to add.
- **instance_name** (*str*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **options** (*dict[str, str]*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

get_pass(*self*, *target: str*) → *Pass*

Returns a reference to the pass with the given instance name. If no such pass exists, an exception is thrown.

Parameters **target** (*str*) – The name of the pass to retrieve a reference to.

Returns A reference to the targeted pass.

Return type *Pass*

does_pass_exist(*self*, *target: str*) → bool

Returns whether a pass with the target instance name exists.

Parameters **target** (*str*) – The name of the pass to query existence of.

Returns Whether a pass with the target name exists.

Return type bool

get_num_passes(*self*) → int

Returns the total number of passes in the root hierarchy.

Parameters **None** –

Returns The number of passes (or groups) within the root pass list.

Return type int

get_passes(*self*) → List[*Pass*]

Returns a list with references to all passes in the root hierarchy.

Parameters **None** –

Returns The list of all passes in the root hierarchy.

Return type list[*Pass*]

get_passes_by_type(*self*, *target: str*) → List[*Pass*]

Returns a list with references to all passes in the root hierarchy with the given type.

Parameters **target** (`str`) – The target pass type name.

Returns The list of all passes in the root hierarchy with the given type.

Return type list[*Pass*]

remove_pass(*self*, *target: str*) → None

Removes the pass with the given target instance name, or throws an exception if no such pass exists.

Parameters **target** (`str`) – The name of the pass to remove.

Returns

Return type None

clear_passes(*self*) → None

Clears the entire pass list.

Parameters **None** –

Returns

Return type None

compile(*self*, *program:* Program) → None

Ensures that all passes have been constructed, and then runs the passes on the given program. This is the same as Program.compile() when the program is referencing the same compiler.

Parameters **program** (`Program`) – The program to compile.

Returns

Return type None

compile_with_frontend(*self*, *platform:* Platform) → None

Ensures that all passes have been constructed, and then runs the passes without specification of an input program. The first pass should then act as a language frontend. The cQASM reader satisfies this requirement, for instance.

If no platform is specified, it will default to the *none* architecture, but the intended use case is to have the first pass load the platform. Again, the cQASM reader can do this.

Parameters **platform** (`Platform`) – The platform to compile for.

Returns

Return type None

1.7.9 Pass class

class openql.**Pass**

Wrapper for a pass that belongs to some pass manager.

NOTE: while it's possible to construct a pass manually, the resulting object cannot be used in any way. The only way to obtain a valid pass object is through a Compiler object.

`__init__`(*self*) → *Pass*

`get_type`(*self*) → str

Returns the full, desugared type name that this pass was constructed with.

Parameters **`None`** –

Returns The type name.

Return type str

`get_name`(*self*) → str

Returns the instance name of the pass within the surrounding group.

Parameters **`None`** –

Returns The instance name.

Return type str

`print_pass_documentation`(*self*) → None

Prints the documentation for this pass.

Parameters **`None`** –

Returns

Return type None

`dump_pass_documentation`(*self*) → str

Returns the documentation for this pass as a string.

Parameters **`None`** –

Returns The documentation for this pass as a multiline string.

Return type str

`print_options`(*self*, *only_set: bool = False*) → None
`print_options`(*self*) → None

Prints the current state of the options.

Parameters **`only_set`** (`bool`) – When set, only the options that were explicitly configured are dumped.

Returns

Return type None

`dump_options`(*self*, *only_set: bool = False*) → str
`dump_options`(*self*) → str

Returns the string printed by print_options().

Parameters **`only_set`** (`bool`) – When set, only the options that were explicitly configured are dumped.

Returns The option documentation as a multiline string.

Return type str

set_option (*self*, *option: str*, *value: str*) → None
Sets an option.

Parameters

- **option** (`str`) – The option name.
- **value** (`str`) – The value to set the option to.

Returns

Return type None

get_option (*self*, *option: str*) → str
Returns the current value of an option.

Parameters **path** (`str`) – The path to the option.

Returns The value of the option. If the option has not been set, the default value is returned.

Return type str

1.7.10 cQasmReader class

class openql.**cQasmReader** (**args*)
Legacy cQASM reader interface.

The preferred way to read cQASM files is to use the cQASM reader pass (`io.cqasm.read`). The pass supports up to cQASM 1.2, and handles all features that OpenQL supports within its intermediate representation properly, whereas this interface only supports version 1.0 and requires an additional JSON file with gate conversion rules to work.

To read a cQASM file using this interface, build a cQASM reader for the an already-existing program, and then call file2circuit or string2circuit to add the kernels from the cQASM file/string to the program. Optionally a platform can be specified as well, but this is redundant (it must be the same platform as the one that the program was constructed with); these overloads only exist for backward compatibility.

Because OpenQL supports custom gates and cQASM (historically) does not, and also because OpenQL's internal representation of gates is still a bit different from what cQASM uses (at least when constructing the IR using the Python API), you may need custom conversion rules for the gates. This can be done by specifying a gateset configuration JSON file using gateset_fname. This file must consist of a JSON array containing objects with the following structure:

```
{
    "name": "<name>",                   # mandatory
    "params": "<typespec>",             # mandatory
    "allow_conditional": <bool>,        # whether conditional gates of this type are␣
↪accepted, defaults to true
    "allow_parallel": <bool>,           # whether parallel gates of this type are␣
↪accepted, defaults to true
    "allow_reused_qubits": <bool>,      # whether reused qubit args for this type are␣
↪accepted, defaults to false
    "ql_name": "<name>",                # defaults to "name"
    "ql_qubits": [                      # list or "all", defaults to the "Q" args
        0,                              # hardcoded qubit index
        "%0"                            # reference to argument 0, which can be a␣
↪qubitref, bitref, or int
    ],
    "ql_cregs": [                       # list or "all", defaults to the "I" args
        0,                              # hardcoded creg index
```

(continues on next page)

(continued from previous page)

```
        "%0"                        # reference to argument 0, which can be an␣
↪int variable reference, or int for creg index
    ],
    "ql_bregs": [                   # list or "all", defaults to the "B" args
        0,                          # hardcoded breg index
        "%0"                        # reference to argument 0, which can be an␣
↪int variable reference, or int for creg index
    ],
    "ql_duration": 0,               # duration; int to hardcode or "%i" to take␣
↪from param i (must be of type int), defaults to 0
    "ql_angle": 0.0,                # angle; float to hardcode or "%i" to take␣
↪from param i (must be of type int or real), defaults to first arg of type real␣
↪or 0.0
    "ql_angle_type": "<type>",      # interpretation of angle arg; one of "rad"␣
↪(radians), "deg" (degrees), or "pow2" (2pi/2^k radians), defaults to "rad"
    "implicit_sgmq": <bool>,        # if multiple qubit args are present, a␣
↪single-qubit gate of this type should be replicated for these qubits (instead␣
↪of a single gate with many qubits)
    "implicit_breg": <bool>         # the breg operand(s) that implicitly belongs␣
↪to the qubit operand(s) in the gate should be added to the OpenQL operand list
}
```

The typespec string defines the expected argument types for the gate. Each character in the string represents an argument. The following characters are supported by libqasm:

- Q = qubit
- B = assignable bit/boolean (measurement register)
- b = bit/boolean
- a = axis (x, y, or z)
- I = assignable integer
- i = integer
- r = real
- c = complex
- u = complex matrix of size 4^n, where n is the number of qubits in the parameter list (automatically deduced)
- s = (quoted) string
- j = json

Note that OpenQL only uses an argument if it is referred to in one of the "ql_*" keys, either implicitly or explicitly.

If no custom configuration is specified, the reader defaults to the logic that was hardcoded before it was made configurable. This corresponds to the following JSON:

```
[
    {"name": "measure",     "params": "Q",      "ql_name": "measz"},
    {"name": "measure",     "params": "QB",     "ql_name": "measz"},
    {"name": "measure_x",   "params": "Q",      "ql_name": "measx"},
    {"name": "measure_x",   "params": "QB",     "ql_name": "measx"},
    {"name": "measure_y",   "params": "Q",      "ql_name": "measy"},
    {"name": "measure_y",   "params": "QB",     "ql_name": "measy"},
```

(continues on next page)

(continued from previous page)

```
    {"name": "measure_z",   "params": "Q",      "ql_name": "measz"},
    {"name": "measure_z",   "params": "QB",     "ql_name": "measz"},
    {"name": "prep",        "params": "Q",      "ql_name": "prepz"},
    {"name": "prep_x",      "params": "Q",      "ql_name": "prepx"},
    {"name": "prep_y",      "params": "Q",      "ql_name": "prepy"},
    {"name": "prep_z",      "params": "Q",      "ql_name": "prepz"},
    {"name": "i",           "params": "Q"},
    {"name": "h",           "params": "Q"},
    {"name": "x",           "params": "Q"},
    {"name": "y",           "params": "Q"},
    {"name": "z",           "params": "Q"},
    {"name": "s",           "params": "Q"},
    {"name": "sdag",        "params": "Q"},
    {"name": "t",           "params": "Q"},
    {"name": "tdag",        "params": "Q"},
    {"name": "x90",         "params": "Q",      "ql_name": "rx90"},
    {"name": "mx90",        "params": "Q",      "ql_name": "xm90"},
    {"name": "y90",         "params": "Q",      "ql_name": "ry90"},
    {"name": "my90",        "params": "Q",      "ql_name": "ym90"},
    {"name": "rx",          "params": "Qr"},
    {"name": "ry",          "params": "Qr"},
    {"name": "rz",          "params": "Qr"},
    {"name": "cnot",        "params": "QQ"},
    {"name": "cz",          "params": "QQ"},
    {"name": "swap",        "params": "QQ"},
    {"name": "cr",          "params": "QQr"},
    {"name": "crk",         "params": "QQi",    "ql_angle": "%2", "ql_angle_type
↪": "pow2"},
    {"name": "toffoli",     "params": "QQQ"},
    {"name": "measure_all", "params": "",       "ql_qubits": "all", "implicit_sgmq
↪": true},
    {"name": "display",     "params": ""},
    {"name": "wait",        "params": ""},
    {"name": "wait",        "params": "i"}
]
```

property platform
The platform associated with the reader.

property program
The program that the cQASM circuits will be added to.

__init__(*self*, *platform:* Platform, *program:* Program, *gateset_fname: str*) → *cQasmReader*
__init__(*self*, *platform:* Platform, *program:* Program) → *cQasmReader*
__init__(*self*, *program:* Program, *gateset_fname: str*) → *cQasmReader*
__init__(*self*, *program:* Program) → *cQasmReader*

string2circuit(*self*, *cqasm_str: str*) → None
Interprets a string as cQASM file and adds its contents to the program associated with this reader.

Parameters **cqasm_str** (`str`) – The string representing the contents of the cQASM file.

Returns

Return type None

file2circuit(*self*, *cqasm_file_path: str*) → None
Interprets a string as cQASM file and adds its contents to the program associated with this reader.

Parameters **cqasm_file_path** (`str`) – The path to the cQASM file to load.

Returns

Return type None

1.7.11 Functions and miscellaneous

`openql.`**`initialize`**`()` → None

Initializes the OpenQL library, for as far as this must be done. This should ideally be called by the user (in Python) before anything else, but set_option() and the constructors of Compiler and Platform will automatically call this when it hasn't been done yet as well.

Currently this just resets the options to their default values to give the user a clean slate to work with in terms of global variables (in case someone else has used the library in the same interpreter before them, for instance, as might happen with ipython/Jupyter in a shared notebook server, or during test suites), but it may initialize more things in the future.

Parameters **`None`** –

Returns

Return type None

`openql.`**`ensure_initialized`**`()` → None

Calls initialize() if it hasn't been called yet.

Parameters **`None`** –

Returns

Return type None

`openql.`**`compile`**`(`*fname: str*, *read_options: Dict[str, str]*`)` → None
`openql.`**`compile`**`(`*fname: str*`)` → None

Entry point for compiling from a cQASM file directly, rather than using the Python API to build the program and platform.

The platform must be encoded using a *pragma @ql.platform(. . .)* annotation at the front of the file; refer to the documentation of the cQASM reader pass for more information. If specified, the read_options parameter is passed to the cQASM reader pass that is automatically prefixed to the pass list.

Parameters

- **`fname`** (*`str`*) – Path to the cQASM file to read.
- **`read_options`** (*`dict[str, str]`*) – A list of options to set for the cQASM reader pass.

Returns

Return type None

`openql.`**`get_version`**`()` → str

Returns the compiler's version string.

Parameters **`None`** –

Returns version number as a string

Return type str

`openql.`**`set_option`**`(`*option: str*, *value: str*`)` → None

Sets a global option for the compiler. Use print_options() to get a list of all available options.

Parameters

- **`option`** (*`str`*) – Name of the option to set.
- **`value`** (*`str`*) – The value to set the option to.

Returns

Return type None

`openql.`**`get_option`**`(`*option: str*`)` → str

Returns the current value for a global option. Use print_options() to get a list of all available options.

Parameters **`option`** (*`str`*) – Name of the option to retrieve the value of.

Returns The value that the option has been set to, or its default value if the option has not been set yet.

Return type str

`openql.`**`print_options`**`()` → None

Prints the documentation for all available global options.

Parameters **`None`** –

Returns

Return type None

`openql.`**`dump_options`**`()` → str

Returns the result of print_options() as a string.

Parameters **`None`** –

Returns The documentation for the options.

Return type str

`openql.`**`print_architectures`**`()` → None

Prints the documentation for all available target architectures.

Parameters **`None`** –

Returns

Return type None

`openql.`**`dump_architectures`**`()` → str

Returns the result of print_architectures() as a string.

Parameters **`None`** –

Returns The documentation for the supported architectures.

Return type str

`openql.`**`print_passes`**`()` → None

Prints the documentation for all available passes.

Parameters **`None`** –

Returns

Return type None

`openql.`**`dump_passes`**`()` → str

Returns the result of print_passes() as a string.

Parameters **`None`** –

Returns The documentation for the supported passes.

Return type str

`openql.`**`print_resources`**`()` → None

Prints the documentation for all available scheduler resources.

Parameters **`None`** –

Returns

Return type None

`openql.`**`dump_resources`**`()` → str

Returns the result of print_resources() as a string.

Parameters **`None`** –

Returns The documentation for the supported scheduler resources.

Return type str

`openql.`**`print_platform_docs`**`()` → None

Prints the documentation for platform configuration files.

Parameters **`None`** –

Returns

Return type None

`openql.`**`dump_platform_docs`**`()` → str

Returns the result of print_platform_docs() as a string.

Parameters **`None`** –

Returns The documentation for the platform configuration file.

Return type str

`openql.`**`print_compiler_docs`**`()` → None

Prints the documentation for compiler configuration files.

Parameters **`None`** –

Returns

Return type None

`openql.`**`dump_compiler_docs`**`()` → str

Returns the result of print_compiler_docs() as a string.

Parameters **`None`** –

Returns The documentation for the compiler configuration file.

Return type str

`class` `openql.`**`Platform`**`(`**args*`)`

Quantum platform description. This describes everything that the compiler needs to know about the target quantum chip, instruments, etc. Platforms are created from either the default configuration for a particular architecture variant or from JSON (+ comments) configuration files: there is no way to modify a platform using the API, and introspection is limited. Instead, if you want to use a custom configuration, you will need to write a JSON configuration file for it, or use get_platform_json() and from_json() to modify an existing one from within Python.

The syntax of the platform configuration file is too extensive to describe here. It has its own section in the manual.

In addition to the platform itself, the Platform object provides an interface for obtaining a Compiler object. This object describes the *strategy* for transforming the quantum algorithm to something that can be executed on the device described by the platform. You can think of the difference between them as the difference between a verb and a noun: the platform describes something that just exists, while the compilation strategy describes how to get there.

The (initial) strategy can be set using a separate configuration file (compiler_config), directly from within the platform configuration file, or one can be inferred based on the previously hardcoded defaults. Unlike the platform itself however, an extensive API is available for adjusting the strategy as you see fit; just use get_compiler() to get a reference to a Compiler object that may be used for this purpose. If you don't do anything with the compiler methods and object, don't specify the compiler_config_file parameter, and the "eqasm_compiler" key of the platform configuration file refers to one of the previously-hardcoded compiler, a strategy will be generated to mimic the old logic for backward compatibility.

Eight constructors are provided:

- Platform(): shorthand for Platform('none', 'none').
- Platform(name): shorthand for Platform(name, name).
- Platform(name, platform_config): builds a platform with the given name (only used for log messages) and platform configuration, the latter of which can be either a recognized platform name with or without variant suffix (for example "cc" or "cc_light.s7"), or a path to a JSON configuration filename.
- Platform(name, platform_config, compiler_config): as above, but specifies a custom compiler configuration file in addition.
- Platform.from_json(name, platform_config_json): instead of loading the platform JSON data from a file, it is taken from its Python object representation (as per json.loads()/dumps()).
- Platform.from_json(name, platform_config_json, compiler_config): as above, with compiler JSON file override.
- Platform.from_json_string(name, platform_config_json): as from_json, but loads the data from a string rather than a Python object.
- Platform.from_json_string(name, platform_config_json, compiler_config): as from_json, but loads the data from a string rather than a Python object.

property name
The user-given name of the platform.

property config_file
The configuration file that the platform was loaded from.

__init__(*self*, *name: str*, *platform_config: str*, *compiler_config: str*) → *Platform*
__init__(*self*, *name: str*, *platform_config: str*) → *Platform*
__init__(*self*, *name: str*) → *Platform*
__init__(*self*) → *Platform*

static from_json_string(*name: str*, *platform_config_json: str*, *compiler_config: str*) → *Platform*
static from_json_string(*name: str*, *platform_config_json: str*) → *Platform*
Alternative constructor. Instead of the platform JSON data being loaded from a file, they are loaded from the given string. See also from_json().

Parameters

- **name** (`str`) – The name for the platform.
- **platform_config_json** (`str`) – The platform JSON configuration data as a string. This will accept anything that the normal constructor accepts when it reads the configuration from a file.

- **`compiler_config`** (*`str`*) – Optional compiler configuration JSON filename. This is *NOT* JSON data.

Returns The constructed platform.

Return type *Platform*

`static get_platform_json_string`(*platform_config: str*) → str
`static get_platform_json_string`() → str

Returns the default platform configuration data as a JSON + comments string. The comments use double-slash syntax. Note that JSON itself does not support such comments (or comments of any kind), so these comments need to be removed from the data before the JSON data can be parsed.

Parameters **`platform_config`** (*`str`*) – The platform configuration. Same syntax as the platform constructor, so this supports architecture names, architecture variant names, or JSON filenames. In the latter case, this function just loads the file contents into a string and returns it.

Returns The JSON + comments data for the given platform configuration string.

Return type str

`get_qubit_number`(*self*) → int

Returns the number of qubits in the platform.

Parameters **`None`** –

Returns The number of qubits in the platform.

Return type int

`print_info`(*self*) → None

Prints some basic information about the platform.

Parameters **`None`** –

Returns

Return type None

`dump_info`(*self*) → str

Returns the result of print_info() as a string.

Parameters **`None`** –

Returns The result of print_info() as a string.

Return type str

`get_info`(*self*) → str

Old alias for dump_info(). Deprecated.

Parameters **`None`** –

Returns The result of print_info() as a string.

Return type str

`has_compiler`(*self*) → bool

Returns whether a custom compiler configuration has been attached to this platform. When this is the case, programs constructed from this platform will use it to implement Program.compile(), rather than generating the compiler in-place from defaults and global options during the call.

Parameters **`None`** –

Returns Whether a custom compiler configuration has been attached to this platform.

Return type bool

get_compiler(*self*) → *Compiler*

Returns the custom compiler configuration associated with this platform. If no such configuration exists yet, the default one is created, attached, and returned.

Parameters **None** –

Returns A Compiler object that may be used to introspect or modify the compilation strategy associated with this platform.

Return type *Compiler*

set_compiler(*self*, *compiler:* Compiler) → None

Sets the compiler associated with this platform. Any programs constructed from this platform after this call will use the given compiler.

Parameters **compiler** (`Compiler`) – The new compiler configuration.

Returns

Return type None

static from_json(*name: str*, *platform_config_json: Dict[. . .]*, *compiler_config: str*) → *Platform*

static from_json(*name: str*, *platform_config_json: Dict[. . .]*) → *Platform*

Alternative constructor. Instead of the platform JSON data being loaded from a file, they are loaded from the given Python object representation of the JSON platform configuration data.

This is useful when you only need to change a builtin platform for some architecture variant a little bit. In this case, you can get the default JSON data using get_platform_json(), introspect and modify it programmatically, and then use this to build the platform from the modified configuration.

Parameters

- **name** (*str*) – The name for the platform.
- **platform_config_json** (*JSON-like object*) – The platform JSON configuration data in Python object representation (anything accepted by json.dumps()).
- **compiler_config** (*str*) – Optional compiler configuration JSON filename. This is *NOT* JSON data.

Returns The constructed platform.

Return type *Platform*

static get_platform_json(*platform_config: str) -> Dict[...] get_platform_json()* → Dict[. . .]

Returns the default platform configuration data as the Python object representation of the JSON data (as returned by json.loads()).

Parameters **platform_config** (*str*) – The platform configuration. Same syntax as the platform constructor, so this supports architecture names, architecture variant names, or JSON filenames. In the latter case, this function just parses the file contents and returns it.

Returns The Python object representation of the JSON data corresponding to the given platform configuration string.

Return type str

class openql.**Program**(*name*, *platform*, *qubit_count=0*, *creg_count=0*, *breg_count=0*)

Represents a complete quantum program.

The constructor creates a new program with the given name, using the given platform. The third, fourth, and fifth arguments optionally specify the desired number of qubits, classical integer registers, and classical bit registers. If not specified, the number of qubits is taken from the platform, and no classical or bit registers will be allocated.

`property name`
The name given to the program by the user.

`property platform`
The platform associated with the program.

`property qubit_count`
The number of (virtual) qubits allocated for the program.

`property creg_count`
The number of classical integer registers allocated for the program.

`property breg_count`
The number of classical bit registers allocated for the program.

`__init__`(*self*, *name: str*, *platform: Platform*, *qubit_count: int = 0*, *creg_count: int = 0*, *breg_count: int = 0*) → *Program*
`__init__`(*self*, *name: str*, *platform: Platform*, *qubit_count: int = 0*, *creg_count: int = 0*) → *Program*
`__init__`(*self*, *name: str*, *platform: Platform*, *qubit_count: int = 0*) → *Program*
`__init__`(*self*, *name: str*, *platform: Platform*) → *Program*

`add_kernel`(*self*, *k: Kernel*) → None
Adds an unconditionally-executed kernel to the end of the program.

Parameters **`k`** (`Kernel`) – The kernel to add.

Returns

Return type None

`add_program`(*self*, *p: Program*) → None
Adds an unconditionally-executed subprogram to the end of the program.

Parameters **`p`** (`Program`) – The subprogram to add.

Returns

Return type None

`add_if`(*self*, *k: Kernel*, *operation: Operation*) → None
`add_if`(*self*, *p: Program*, *operation: Operation*) → None
Adds a conditionally-executed kernel or subprogram to the end of the program. The kernel/subprogram will be executed if the given classical condition evaluates to true.

Parameters

- **`k`** (`Kernel`) – The kernel to add.
- **`p`** (`Program`) – The subprogram to add.
- **`operation`** (`Operation`) – The operation that must evaluate to true for the kernel/subprogram to be executed.

Returns

Return type None

`add_if_else`(*self*, *k_if: Kernel*, *k_else: Kernel*, *operation: Operation*) → None
`add_if_else`(*self*, *p_if: Program*, *p_else: Program*, *operation: Operation*) → None
Adds two conditionally-executed kernels/subprograms with inverted conditions to the end of the program. The first kernel/subprogram will be executed if the given classical condition evaluates to true; the second kernel/subprogram will be executed if it evaluates to false.

Parameters

- **`k_if`** (`Kernel`) – The kernel to execute when the condition evaluates to true.
- **`p_if`** (`Program`) – The subprogram to execute when the condition evaluates to true.
- **`k_else`** (`Kernel`) – The kernel to execute when the condition evaluates to false.
- **`p_else`** (`Program`) – The subprogram to execute when the condition evaluates to false.
- **`operation`** (`Operation`) – The operation that determines which kernel/subprogram will be executed.

Returns

Return type None

`add_do_while`(*self*, *k: Kernel*, *operation: Operation*) → None
`add_do_while`(*self*, *p: Program*, *operation: Operation*) → None

Adds a kernel/subprogram that will be repeated until the given classical condition evaluates to true. The kernel/subprogram is executed at least once, since the condition is evaluated at the end of the loop body.

Parameters

- **`k`** (`Kernel`) – The kernel that represents the loop body.
- **`p`** (`Program`) – The subprogram that represents the loop body.
- **`operation`** (`Operation`) – The operation that must evaluate to true at the end of the loop body for the loop body to be executed again.

Returns

Return type None

`add_for`(*self*, *k: Kernel*, *iterations: int*) → None
`add_for`(*self*, *p: Program*, *iterations: int*) → None

Adds an unconditionally-executed kernel/subprogram that will loop for the given number of iterations.

Parameters

- **`k`** (`Kernel`) – The kernel that represents the loop body.
- **`p`** (`Program`) – The subprogram that represents the loop body.
- **`iterations`** (*`int`*) – The number of loop iterations.

Returns

Return type None

`set_sweep_points`(*self*, *sweep_points: List[float]*) → None

Sets sweep point information for the program.

NOTE: sweep points functionality is deprecated and may be removed at any time. Do not use it in new programs.

Parameters **`sweep_points`** (*`List[float]`*) – The list of sweep points.

Returns

Return type None

`get_sweep_points`(*self*) → List[float]

Returns the configured sweep point information for the program.

NOTE: sweep points functionality is deprecated and may be removed at any time. Do not use it in new programs.

Parameters **`None`** –

Returns The previously configured sweep point information for the program, or an empty list if none were configured.

Return type List[float]

`set_config_file`(*self*, *config_file_name: str*) → None

Sets the name of the file that the sweep points will be written to.

NOTE: sweep points functionality is deprecated and may be removed at any time. Do not use it in new programs.

Parameters **`config_file_name`** (`str`) – The name of the file that the sweep points are to be written to.

Returns

Return type None

`has_compiler`(*self*) → bool

Whether a custom compiler configuration has been attached to this program. When this is the case, it will be used to implement compile(), rather than generating the compiler in-place from defaults and global options during the call.

Parameters **`None`** –

Returns Whether a custom compiler configuration has been attached to this program.

Return type bool

`get_compiler`(*self*) → *Compiler*

Returns the custom compiler configuration associated with this program. If no such configuration exists yet, the default one is created, attached, and returned.

Parameters **`None`** –

Returns A Compiler object that may be used to introspect or modify the compilation strategy associated with this program.

Return type *Compiler*

`set_compiler`(*self*, *compiler:* Compiler) → None

Sets the compiler associated with this program. It will then be used for compile().

Parameters **`compiler`** (`Compiler`) – The new compiler configuration.

Returns

Return type None

`compile`(*self*) → None

Compiles the program.

Parameters **`None`** –

Returns

Return type None

`print_interaction_matrix`(*self*) → None

Prints the interaction matrix for each kernel in the program.

Parameters **`None`** –

Returns

Return type None

write_interaction_matrix(*self*) → None

Writes the interaction matrix for each kernel in the program to a file. This is one of the few functions that still uses the global output_dir option.

Parameters **`None`** –

Returns

Return type None

class `openql.`**`Kernel`**(*name*, *platform*, *qubit_count=0*, *creg_count=0*, *breg_count=0*)

Represents a kernel of a quantum program, a.k.a. a basic block. Kernels are just sequences of gates with no classical control-flow in between: they may end in a (conditional) branch to the start of another kernel, but otherwise, they may only consist of quantum gates and mixed quantum-classical data flow operations.

The constructor creates a new kernel with the given name, using the given platform. The third, fourth, and fifth arguments optionally specify the desired number of qubits, classical integer registers, and classical bit registers. If not specified, the number of qubits is taken from the platform, and no classical or bit registers will be allocated.

Currently, the contents of a kernel can only be constructed by adding gates and classical data flow instructions in the order in which they are to be executed, and there is no way to get information about which gates are in the kernel after the fact. If you need this kind of bookkeeping, you will have to wrap OpenQL's kernels for now.

Classical flow-control is configured when a completed kernel is added to a program, via basic structured control-flow paradigms (if-else, do-while, and loops with a fixed iteration count).

NOTE: the way gates are represented in OpenQL is on the list to be completely revised. Currently OpenQL works using a mixture of "default gates" and the "custom gates" that you can specify in the platform configuration file, but these two things are not orthogonal and largely incompatible with each other, yet are currently used interchangeably. Furthermore, there is no proper way to specify lists of generic arguments to a gate, leading to lots of code duplication inside OpenQL and long gate() argument lists. Finally, the semantics of gates are largely derived by undocumented and somewhat heuristic string comparisons with the names of gates, which is terrible design in combination with user-specified instruction sets via the platform configuration file. The interface for adding simple *quantum* gates to a kernel is something we want to keep 100% backward compatible, but the more advanced gate() signatures may change in the (near) future.

NOTE: classical logic is on the list to be completely revised. This interface may change in the (near) future.

NOTE: the higher-order functions for constructing controlled kernels and conjugating kernels have not been maintained for a while and thus probably won't work right. They may be removed entirely in a later version of OpenQL.

property name

The name of the kernel as given by the user.

property platform

The platform that the kernel was built for.

property qubit_count

The number of (virtual) qubits allocated for the kernel.

property creg_count

The number of classical integer registers allocated for the kernel.

property breg_count

The number of classical bit registers allocated for the kernel.

__init__(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*, *creg_count: int = 0*, *breg_count: int = 0*) → *Kernel*

__init__(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*, *creg_count: int = 0*) → *Kernel*

__init__(*self*, *name: str*, *platform:* Platform, *qubit_count: int = 0*) → *Kernel*

`__init__`(*self*, *name: str*, *platform:* Platform) → *Kernel*

`get_custom_instructions`(*self*) → str

Old alias for dump_custom_instructions(). Deprecated.

Parameters **`None`** –

Returns A newline-separated list of all custom gates supported by the platform.

Return type str

`print_custom_instructions`(*self*) → None

Prints a list of all custom gates supported by the platform.

Parameters **`None`** –

Returns

Return type None

`dump_custom_instructions`(*self*) → str

Returns the result of print_custom_instructions() as a string.

Parameters **`None`** –

Returns A newline-separated list of all custom gates supported by the platform.

Return type str

`gate_preset_condition`(*self*, *condstring: str*, *condregs: List[int]*) → None

Sets the condition for all gates subsequently added to this kernel. Thus, essentially shorthand notation. Reset with gate_clear_condition().

Parameters

- **`condstring`** (`str`) – Must be one of:
  - ”COND_ALWAYS” or “1”: no condition; gate is always executed.
  - ”COND_NEVER” or “0”: no condition; gate is never executed.
  - ”COND_UNARY” or “” (empty): gate is executed if the single bit specified via condregs is 1.
  - ”COND_NOT” or “!”: gate is executed if the single bit specified via condregs is 0.
  - ”COND_AND” or “&”: gate is executed if the two bits specified via condregs are both 1.
  - ”COND_NAND” or “!&”: gate is executed if either of the two bits specified via condregs is zero.
  - ”COND_OR” or “|”: gate is executed if either of the two bits specified via condregs is one.
  - ”COND_NOR” or “!|”: no condition; gate is always executed.
- **`condregs`** (`List[int]`) – Depending on condstring, must be a list of 0, 1, or 2 breg indices.

Returns

Return type None

`gate_clear_condition`(*self*) → None

Clears a condition previously set via gate_preset_condition().

Parameters **`None`** –

Returns

Return type None

gate(*self*, *name: str*, *q0: int*) → None
gate(*self*, *name: str*, *q0: int*, *q1: int*) → None
gate(*self, name: str, qubits: List[int], duration: int = 0, angle: float = 0.0, bregs: List[int], condstring: str, condregs: List[int]*) → None
gate(*self, name: str, qubits: List[int], duration: int = 0, angle: float = 0.0, bregs: List[int], condstring: str*) → None
gate(*self, name: str, qubits: List[int], duration: int = 0, angle: float = 0.0, bregs: List[int]*) → None
gate(*self*, *name: str*, *qubits: List[int]*, *duration: int = 0*, *angle: float = 0.0*) → None
gate(*self*, *name: str*, *qubits: List[int]*, *duration: int = 0*) → None
gate(*self*, *name: str*, *qubits: List[int]*) → None
gate(*self*, *name: str*, *qubits: List[int]*, *destination:* CReg) → None
gate(*self*, *u:* Unitary, *qubits: List[int]*) → None

Main function for appending arbitrary quantum gates.

Parameters

- **name** (*str*) – The name of the gate. Note that OpenQL currently uses string comparisons with these names all over the place to derive functionality, and to derive what the actual arguments do. This is inherently a bad idea and something we want to move away from, so documenting it all would not be worthwhile. For now, just use common sense, and you'll probably be okay.
- **q0** (*int*) – Index of the first qubit to apply the gate to. For controlled gates, this is the control qubit.
- **q1** (*int*) – Index of the second qubit to apply the gate to. For controlled gates, this is the target qubit.
- **qubits** (*List[int]*) – The full list of qubit indices to apply the gate to.
- **duration** (*int*) – Gate duration in nanoseconds, or 0 to use the default value from the platform configuration file. This is primarily intended to be used for wait gates.
- **angle** (*float*) – Rotation angle in radians for gates that use it (rx, ry, rz, etc). Ignored for all other gates.
- **bregs** (*List[int]*) – The full list of bit register argument indices for the gate, excluding any bit registers used for conditional execution. Currently only used for the measure gate, which may be given an explicit bit register index to return its result in. If no such register is specified, the result is assumed to implicitly go to the bit register with the same index as the qubit being measured. Ignored for gates that don't use bit registers.
- **condstring** (*str*) – If specified, must be one of:
  - "COND_ALWAYS" or "1": no condition; gate is always executed.
  - "COND_NEVER" or "0": no condition; gate is never executed.
  - "COND_UNARY" or "" (empty): gate is executed if the single bit specified via condregs is 1.
  - "COND_NOT" or "!": gate is executed if the single bit specified via condregs is 0.
  - "COND_AND" or "&": gate is executed if the two bits specified via condregs are both 1.
  - "COND_NAND" or "!&": gate is executed if either of the two bits specified via condregs is zero.

- "COND_OR" or "|": gate is executed if either of the two bits specified via condregs is one.
- "COND_NOR" or "!|": no condition; gate is always executed.

- **condregs** (*List[int]*) – Depending on condstring, must be a list of 0, 1, or 2 breg indices.
- **destination** (CReg) – An integer control register that receives the result of the mixed quantum-classical gate identified by name.
- **u** (Unitary) – The unitary gate to insert.

Returns

Return type None

condgate (*self*, *name: str*, *qubits: List[int]*, *condstring: str*, *condregs: List[int]*) → None

Alternative function for appending normal conditional quantum gates. Avoids having to specify duration, angle, and bregs for gates that don't need it.

Parameters

- **name** (*str*) – The name of the gate. Note that OpenQL currently uses string comparisons with these names all over the place to derive functionality, and to derive what the actual arguments do. This is inherently a bad idea and something we want to move away from, so documenting it all would not be worthwhile. For now, just use common sense, and you'll probably be okay.
- **qubits** (*List[int]*) – The full list of qubit indices to apply the gate to.
- **condstring** (*str*) – If specified, must be one of:
  - "COND_ALWAYS" or "1": no condition; gate is always executed.
  - "COND_NEVER" or "0": no condition; gate is never executed.
  - "COND_UNARY" or "" (empty): gate is executed if the single bit specified via condregs is 1.
  - "COND_NOT" or "!": gate is executed if the single bit specified via condregs is 0.
  - "COND_AND" or "&": gate is executed if the two bits specified via condregs are both 1.
  - "COND_NAND" or "!&": gate is executed if either of the two bits specified via condregs is zero.
  - "COND_OR" or "|": gate is executed if either of the two bits specified via condregs is one.
  - "COND_NOR" or "!|": no condition; gate is always executed.
- **condregs** (*List[int]*) – Depending on condstring, must be a list of 0, 1, or 2 breg indices.

Returns

Return type None

classical (*self*, *destination:* CReg, *operation:* Operation) → None
classical (*self*, *operation: str*) → None

Appends a classical assignment gate to the circuit. The classical integer register is assigned to the result of the given operation.

Parameters

- **`destination`** (`CReg`) – An integer control register at the left-hand side of the classical assignment gate.
- **`operation`** (`Operation`) – The expression to evaluate on the right-hand side of the classical assignment gate.

Returns

Return type None

`identity`(*self*, *q0: int*) → None

Shorthand for an "identity" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`hadamard`(*self*, *q0: int*) → None

Shorthand for appending a "hadamard" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`s`(*self*, *q0: int*) → None

Shorthand for appending an "s" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`sdag`(*self*, *q0: int*) → None

Shorthand for appending an "sdag" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`t`(*self*, *q0: int*) → None

Shorthand for appending a "t" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`tdag`(*self*, *q0: int*) → None

Shorthand for appending a "tdag" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`x`(*self*, *q0: int*) → None

Shorthand for appending an "x" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

y (*self*, *q0: int*) → None
Shorthand for appending a "y" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

z (*self*, *q0: int*) → None
Shorthand for appending a "z" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

rx90 (*self*, *q0: int*) → None
Shorthand for appending an "rx90" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

mrx90 (*self*, *q0: int*) → None
Shorthand for appending an "mrx90" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

rx180 (*self*, *q0: int*) → None
Shorthand for appending an "rx180" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

ry90 (*self*, *q0: int*) → None
Shorthand for appending an "ry90" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

mry90 (*self*, *q0: int*) → None
Shorthand for appending an "mry90" gate with a single qubit.

Parameters **q0** (*int*) – The qubit to apply the gate to.

Returns

Return type None

ry180 (*self*, *q0: int*) → None
Shorthand for appending an "ry180" gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to apply the gate to.

Returns

Return type None

`rx` (*self*, *q0: int*, *angle: float*) → None
Shorthand for appending an “rx” gate with a single qubit and the given rotation in radians.

Parameters

- **`q0`** (*`int`*) – The qubit to apply the gate to.
- **`angle`** (*`float`*) – The rotation angle in radians.

Returns

Return type None

`ry` (*self*, *q0: int*, *angle: float*) → None
Shorthand for appending an “ry” gate with a single qubit and the given rotation in radians.

Parameters

- **`q0`** (*`int`*) – The qubit to apply the gate to.
- **`angle`** (*`float`*) – The rotation angle in radians.

Returns

Return type None

`rz` (*self*, *q0: int*, *angle: float*) → None
Shorthand for appending an “rz” gate with a single qubit and the given rotation in radians.

Parameters

- **`q0`** (*`int`*) – The qubit to apply the gate to.
- **`angle`** (*`float`*) – The rotation angle in radians.

Returns

Return type None

`measure` (*self*, *q0: int*) → None
`measure` (*self*, *q0: int*, *b0: int*) → None
Shorthand for appending a “measure” gate with a single qubit and implicit or explicit result bit register.

Parameters

- **`q0`** (*`int`*) – The qubit to measure.
- **`b0`** (*`int`*) – The bit register to store the result in. If not specified, the result will be placed in the bit register corresponding to the index of the measured qubit.

Returns

Return type None

`prepz` (*self*, *q0: int*) → None
Shorthand for appending a “prepz” gate with a single qubit.

Parameters **`q0`** (*`int`*) – The qubit to prepare in the Z basis.

Returns

Return type None

`cnot` (*self*, *q0: int*, *q1: int*) → None
Shorthand for appending a "cnot" gate with two qubits.

Parameters

- **`q0`** (*`int`*) – The control qubit index.
- **`q1`** (*`int`*) – The target qubit index

Returns

Return type None

`cphase` (*self*, *q0: int*, *q1: int*) → None
Shorthand for appending a "cphase" gate with two qubits.

Parameters

- **`q0`** (*`int`*) – The first qubit index.
- **`q1`** (*`int`*) – The second qubit index.

Returns

Return type None

`cz` (*self*, *q0: int*, *q1: int*) → None
Shorthand for appending a "cz" gate with two qubits.

Parameters

- **`q0`** (*`int`*) – The first qubit index.
- **`q1`** (*`int`*) – The second qubit index.

Returns

Return type None

`toffoli` (*self*, *q0: int*, *q1: int*, *q2: int*) → None
Shorthand for appending a "toffoli" gate with three qubits.

Parameters

- **`q0`** (*`int`*) – The first control qubit index.
- **`q1`** (*`int`*) – The second control qubit index.
- **`q2`** (*`int`*) – The target qubit index.

Returns

Return type None

`clifford` (*self*, *id: int*, *q0: int*) → None
Shorthand for appending the Clifford gate with the specific number using the minimal number of rx90, rx180, mrx90, ry90, ry180, and mry90 gates.

Parameters

- **`id`** (*`int`*) – The Clifford gate expansion index:
  - 0: no gates inserted.
  - 1: ry90; rx90
  - 2: mrx90, mry90
  - 3: rx180

  - 4: mry90, mrx90
  - 5: rx90, mry90
  - 6: ry180
  - 7: mry90, rx90
  - 8: rx90, ry90
  - 9: rx180, ry180
  - 10: ry90, mrx90
  - 11: mrx90, ry90
  - 12: ry90, rx180
  - 13: mrx90
  - 14: rx90, mry90, mrx90
  - 15: mry90
  - 16: rx90
  - 17: rx90, ry90, rx90
  - 18: mry90, rx180
  - 19: rx90, ry180
  - 20: rx90, mry90, rx90
  - 21: ry90
  - 22: mrx90, ry180
  - 23: rx90, ry90, mrx90
- **`q0`** (*`int`*) – The target qubit.

Returns

Return type None

`wait` (*self*, *qubits: List[int]*, *duration: int*) → None

Shorthand for appending a "wait" gate with the specified qubits and duration in nanoseconds. If no qubits are specified, the wait applies to all qubits instead (a wait with no qubits is meaningless). Note that the duration will usually end up being rounded up to multiples of the platform's cycle time.

Parameters

- **`qubits`** (*`List[int]`*) – The list of qubits to apply the wait gate to. If empty, the list will be replaced with the set of all qubits.
- **`duration`** (*`int`*) – The duration of the wait gate in nanoseconds.

Returns

Return type None

`barrier` (*self*, *qubits: List[int]*) → None
`barrier` (*self*) → None

Shorthand for appending a "wait" gate with the specified qubits and duration 0. If no qubits are specified, the wait applies to all qubits instead (a wait with no qubits is meaningless).

Parameters **`qubits`** (*`List[int]`*) – The list of qubits to apply the wait gate to. If empty or unspecified, the list will be replaced with the set of all qubits.

Returns

Return type None

`display` (*self*) → None
Shorthand for appending a “display” gate with no qubits.

Parameters **`None`** –

Returns

Return type None

`diamond_excite_mw` (*self*, *envelope: int*, *duration: int*, *frequency: int*, *phase: int*, *amplitude: int*, *qubit: int*) → None
Appends the diamond “excite_mw” instruction.

Parameters

- **`envelope`** (*`int`*) – The envelope of the microwave.
- **`duration`** (*`int`*) – The duration of the microwave in nanoseconds.
- **`frequency`** (*`int`*) – The frequency of the microwave in kilohertz.
- **`phase`** (*`int`*) – The phase of the microwave.
- **`amplitude`** (*`int`*) – The amplitude of the microwave.
- **`qubit`** (*`int`*) – The target qubit index.

Returns

Return type None

`diamond_memswap` (*self*, *qubit: int*, *nuclear_qubit: int*) → None
Appends the diamond “memswap” instruction.

Parameters

- **`qubit`** (*`int`*) – The index of the qubit.
- **`nuclear_qubit`** (*`int`*) – The index of the nuclear spin qubit of the color center.

Returns

Return type None

`diamond_qentangle` (*self*, *qubit: int*, *nuclear_qubit: int*) → None
Appends the diamond “qentangle” instruction.

Parameters

- **`qubit`** (*`int`*) – The index of the qubit.
- **`nuclear_qubit`** (*`int`*) – The index of the nuclear spin qubit of the color center.

Returns

Return type None

`diamond_sweep_bias` (*self*, *qubit: int*, *value: int*, *dacreg: int*, *start: int*, *step: int*, *max: int*, *memaddress: int*) → None
Appends the diamond “sweep_bias” instruction.

Parameters

- **`qubit`** (*`int`*) – The index of the qubit.
- **`value`** (*`int`*) – The value that has to be send to the dac for biasing.

- **dacreg** (*int*) – The index or address of the register of the dac.
- **start** (*int*) – The start frequency value of the sweep.
- **step** (*int*) – The step frequency value of the sweep.
- **max** (*int*) – The maximum frequency value of the sweep.
- **memaddress** (*int*) – The memory address to write the results to.

Returns

Return type None

diamond_crc(*self*, *qubit: int*, *threshold: int*, *value: int*) → None
Appends the diamond "crc" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **treshold** (*int*) – The threshold value that has to be matched.
- **value** (*int*) – The value of the voltage sent to the dac.

Returns

Return type None

diamond_rabi_check(*self*, *qubit: int*, *measurements: int*, *duration: int*, *t_max: int*) → None
Appends the diamond "rabi_check" instruction.

Parameters

- **qubit** (*int*) – The index of the qubit.
- **measurements** (*int*) – How manu measurements have to be recorded.
- **duration** (*int*) – The starting value of the duration of the microwave.
- **t_max** (*int*) – The value of the voltage sent to the dac.

Returns

Return type None

controlled(*self*, *k:* Kernel, *control_qubits: List[int]*, *ancilla_qubits: List[int]*) → None
Appends a controlled kernel. The number of control and ancilla qubits must be equal.

Parameters

- **k** (*Kernel*) – The kernel to make controlled.
- **control_qubits** (*List[int]*) – The qubits that control the kernel.
- **ancilla_qubits** (*List[int]*) – The ancilla qubits to use to make the kernel controlled. The number of ancilla qubits must equal the number of control qubits.

Returns

Return type None

conjugate(*self*, *k:* Kernel) → None
Appends the conjugate of the given kernel to this kernel.

NOTE: this high-level functionality is poorly/not maintained, and relies on default gates, which are on the list for removal.

Parameters **k** (*Kernel*) – The kernel to conjugate.

Returns

Return type None

class `openql.`**`CReg`**(*id*)

Wrapper for a classical integer register with the given index.

NOTE: classical logic is on the list to be completely revised. This interface may change in the (near) future.

`__init__`(*self*, *id: int*) → *CReg*

class `openql.`**`Operation`**(**args*)

Wrapper for a classical operation.

A classical operation acts as a simple expression that returns an integer or a boolean. The expression can be a literal number (val), a single CReg (lop, primarily used for assignments), a unary function applied to one CReg (op/rop), or a binary function applied to two CRegs (lop/op/rop).

Function selection is done via strings. The following unary functions are recognized:

- '~': bitwise NOT.

The following binary functions are recognized:

- '+': addition.
- '-': subtraction.
- '&': bitwise AND.
- '|': bitwise OR.
- '^': bitwise XOR.
- '==': equality.
- '!=': inequality.
- '>': greater-than.
- '>=': greater-or-equal.
- '<': less-than.
- '<=': less-or-equal.

NOTE: classical logic is on the list to be completely revised. This interface may change in the (near) future.

`__init__`(*self*, *lop:* CReg, *op: str*, *rop:* CReg) → *Operation*
`__init__`(*self*, *op: str*, *rop:* CReg) → *Operation*
`__init__`(*self*, *lop:* CReg) → *Operation*
`__init__`(*self*, *val: int*) → *Operation*

class `openql.`**`Unitary`**(*name*, *matrix*)

Unitary matrix interface.

The constructor creates a unitary gate from the given row-major, square, unitary matrix.

property name

The name given to the unitary gate.

`__init__`(*self*, *name: str*, *matrix: vectorc*) → *Unitary*

`decompose`(*self*) → None

Explicitly decomposes the gate. Does not need to be called; it will be called automatically when the gate is added to the kernel.

Parameters **`None`** –

Returns

Return type None

static is_decompose_support_enabled() → bool

Returns whether OpenQL was built with unitary decomposition support enabled.

Parameters **None** –

Returns Whether OpenQL was built with unitary decomposition support enabled.

Return type bool

class openql.**Compiler**(**args*)

Wrapper for the compiler/pass manager.

This allows you to change the compilation strategy, if the defaults are insufficient for your application. You can get access to a Compiler via several methods:

- using Platform.get_compiler();
- using Program.get_compiler();
- using one of the constructors.

Using the constructors, you can get an empty compiler (by specifying no arguments or only specifying name), a default compiler for a given platform (by specifying a name and a platform), or a compiler based on a compiler configuration JSON file (by specifying a name and a filename). For the structure of this JSON file, refer to the configuration section of the ReadTheDocs documentation or, equivalently, the result of *openql.print_compiler_docs()*.

property name

User-given name for this compiler.

NOTE: not actually used for anything. It's only here for consistency with the rest of the API objects.

__init__(*self*, *name: str*) → *Compiler*
__init__(*self*) → *Compiler*
__init__(*self*, *name: str*, *filename: str*) → *Compiler*
__init__(*self*, *name: str*, *platform:* Platform) → *Compiler*

print_pass_types(*self*) → None

Prints documentation for all available pass types, as well as the option documentation for the passes.

Parameters **None** –

Returns

Return type None

dump_pass_types(*self*) → str

Returns documentation for all available pass types, as well as the option documentation for the passes.

Parameters **None** –

Returns The list of pass types and their documentation as a multiline string.

Return type str

print_strategy(*self*) → None

Prints the currently configured compilation strategy.

Parameters **None** –

Returns

Return type None

dump_strategy(*self*) → str

Returns the currently configured compilation strategy as a string.

Parameters **None** –

Returns The current compilation strategy as a multiline string.

Return type str

set_option(*self*, *path: str*, *value: str*, *must_exist: bool = True*) → int
set_option(*self*, *path: str*, *value: str*) → int

Sets a pass option. The path must consist of the target pass instance name and the option name, separated by a period. The former may include * or ? wildcards. If must_exist is set an exception will be thrown if none of the passes were affected, otherwise 0 will be returned.

Parameters

- **path** (*str*) – The path to the option, consisting of the pass name and the option name separated by a period.
- **value** (*str*) – The value to set the option to.
- **must_exist** (*bool*) – When set, an exception will be thrown when no options matched the path.

Returns The number of pass options affected.

Return type int

get_option(*self*, *path: str*) → str

Returns the current value of an option.

Parameters **path** (*str*) – The path to the option, consisting of pass name and the actual option name separated by a period.

Returns The value of the option. If the option has not been set, the default value is returned.

Return type str

append_pass(*self*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
append_pass(*self*, *type_name: str*, *instance_name: str*) → *Pass*
append_pass(*self*, *type_name: str*) → *Pass*

Appends a pass to the end of the pass list. Returns a reference to the constructed pass.

Parameters

- **type_name** (*str*) – The type of the pass to add.
- **instance_name** (*str*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **options** (*dict[str, str]*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

prefix_pass(*self*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
prefix_pass(*self*, *type_name: str*, *instance_name: str*) → *Pass*
prefix_pass(*self*, *type_name: str*) → *Pass*

Appends a pass to the beginning of the pass list. Returns a reference to the constructed pass.

Parameters

- **type_name** (*str*) – The type of the pass to add.

- **`instance_name`** (*`str`*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **`options`** (*`dict[str, str]`*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

`insert_pass_after`(*self*, *target: str*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
`insert_pass_after`(*self*, *target: str*, *type_name: str*, *instance_name: str*) → *Pass*
`insert_pass_after`(*self*, *target: str*, *type_name: str*) → *Pass*

Inserts a pass immediately after the target pass (named by instance). If target does not exist, an exception is thrown. Returns a reference to the constructed pass.

Parameters

- **`target`** (*`str`*) – The name of the pass to insert the new pass after.
- **`type_name`** (*`str`*) – The type of the pass to add.
- **`instance_name`** (*`str`*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **`options`** (*`dict[str, str]`*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

`insert_pass_before`(*self*, *target: str*, *type_name: str*, *instance_name: str*, *options: Dict[str, str]*) → *Pass*
`insert_pass_before`(*self*, *target: str*, *type_name: str*, *instance_name: str*) → *Pass*
`insert_pass_before`(*self*, *target: str*, *type_name: str*) → *Pass*

Inserts a pass immediately before the target pass (named by instance). If target does not exist, an exception is thrown. Returns a reference to the constructed pass.

Parameters

- **`target`** (*`str`*) – The name of the pass to insert the new pass before.
- **`type_name`** (*`str`*) – The type of the pass to add.
- **`instance_name`** (*`str`*) – A unique name for the pass instance. If empty or unspecified, a name will be generated.
- **`options`** (*`dict[str, str]`*) – A list of initial options to set for the pass. This is just shorthand notation for calling set_option() on the returned Pass object.

Returns A reference to the added pass.

Return type *Pass*

`get_pass`(*self*, *target: str*) → *Pass*

Returns a reference to the pass with the given instance name. If no such pass exists, an exception is thrown.

Parameters **`target`** (*`str`*) – The name of the pass to retrieve a reference to.

Returns A reference to the targeted pass.

Return type *Pass*

`does_pass_exist`(*self*, *target: str*) → bool

Returns whether a pass with the target instance name exists.

Parameters **`target`** (*`str`*) – The name of the pass to query existence of.

Returns Whether a pass with the target name exists.

Return type bool

`get_num_passes` (*self*) → int

Returns the total number of passes in the root hierarchy.

Parameters **`None`** –

Returns The number of passes (or groups) within the root pass list.

Return type int

`get_passes` (*self*) → List[*Pass*]

Returns a list with references to all passes in the root hierarchy.

Parameters **`None`** –

Returns The list of all passes in the root hierarchy.

Return type list[*Pass*]

`get_passes_by_type` (*self*, *target: str*) → List[*Pass*]

Returns a list with references to all passes in the root hierarchy with the given type.

Parameters **`target`** (*`str`*) – The target pass type name.

Returns The list of all passes in the root hierarchy with the given type.

Return type list[*Pass*]

`remove_pass` (*self*, *target: str*) → None

Removes the pass with the given target instance name, or throws an exception if no such pass exists.

Parameters **`target`** (*`str`*) – The name of the pass to remove.

Returns

Return type None

`clear_passes` (*self*) → None

Clears the entire pass list.

Parameters **`None`** –

Returns

Return type None

`compile` (*self*, *program:* Program) → None

Ensures that all passes have been constructed, and then runs the passes on the given program. This is the same as Program.compile() when the program is referencing the same compiler.

Parameters **`program`** (`Program`) – The program to compile.

Returns

Return type None

`compile_with_frontend` (*self*, *platform:* Platform) → None

Ensures that all passes have been constructed, and then runs the passes without specification of an input program. The first pass should then act as a language frontend. The cQASM reader satisfies this requirement, for instance.

If no platform is specified, it will default to the *none* architecture, but the intended use case is to have the first pass load the platform. Again, the cQASM reader can do this.

Parameters **`platform`** (`Platform`) – The platform to compile for.

Returns

Return type None

`class openql.Pass`

Wrapper for a pass that belongs to some pass manager.

NOTE: while it's possible to construct a pass manually, the resulting object cannot be used in any way. The only way to obtain a valid pass object is through a Compiler object.

`__init__`(*self*) → *Pass*

`get_type`(*self*) → str

Returns the full, desugared type name that this pass was constructed with.

Parameters **`None`** –

Returns The type name.

Return type str

`get_name`(*self*) → str

Returns the instance name of the pass within the surrounding group.

Parameters **`None`** –

Returns The instance name.

Return type str

`print_pass_documentation`(*self*) → None

Prints the documentation for this pass.

Parameters **`None`** –

Returns

Return type None

`dump_pass_documentation`(*self*) → str

Returns the documentation for this pass as a string.

Parameters **`None`** –

Returns The documentation for this pass as a multiline string.

Return type str

`print_options`(*self*, *only_set: bool = False*) → None

`print_options`(*self*) → None

Prints the current state of the options.

Parameters **`only_set`** (*`bool`*) – When set, only the options that were explicitly configured are dumped.

Returns

Return type None

`dump_options`(*self*, *only_set: bool = False*) → str

`dump_options`(*self*) → str

Returns the string printed by print_options().

Parameters **`only_set`** (*`bool`*) – When set, only the options that were explicitly configured are dumped.

Returns The option documentation as a multiline string.

Return type str

set_option (*self*, *option: str*, *value: str*) → None

Sets an option.

Parameters

- **option** (*str*) – The option name.
- **value** (*str*) – The value to set the option to.

Returns

Return type None

get_option (*self*, *option: str*) → str

Returns the current value of an option.

Parameters **path** (*str*) – The path to the option.

Returns The value of the option. If the option has not been set, the default value is returned.

Return type str

class openql.**cQasmReader** (**args*)

Legacy cQASM reader interface.

The preferred way to read cQASM files is to use the cQASM reader pass (`io.cqasm.read`). The pass supports up to cQASM 1.2, and handles all features that OpenQL supports within its intermediate representation properly, whereas this interface only supports version 1.0 and requires an additional JSON file with gate conversion rules to work.

To read a cQASM file using this interface, build a cQASM reader for the an already-existing program, and then call file2circuit or string2circuit to add the kernels from the cQASM file/string to the program. Optionally a platform can be specified as well, but this is redundant (it must be the same platform as the one that the program was constructed with); these overloads only exist for backward compatibility.

Because OpenQL supports custom gates and cQASM (historically) does not, and also because OpenQL's internal representation of gates is still a bit different from what cQASM uses (at least when constructing the IR using the Python API), you may need custom conversion rules for the gates. This can be done by specifying a gateset configuration JSON file using gateset_fname. This file must consist of a JSON array containing objects with the following structure:

```
{
    "name": "<name>",                # mandatory
    "params": "<typespec>",          # mandatory
    "allow_conditional": <bool>,     # whether conditional gates of this type are␣
↪accepted, defaults to true
    "allow_parallel": <bool>,        # whether parallel gates of this type are␣
↪accepted, defaults to true
    "allow_reused_qubits": <bool>,   # whether reused qubit args for this type are␣
↪accepted, defaults to false
    "ql_name": "<name>",             # defaults to "name"
    "ql_qubits": [                   # list or "all", defaults to the "Q" args
        0,                           # hardcoded qubit index
        "%0"                         # reference to argument 0, which can be a␣
↪qubitref, bitref, or int
    ],
    "ql_cregs": [                    # list or "all", defaults to the "I" args
        0,                           # hardcoded creg index
```

(continues on next page)

(continued from previous page)

```
        "%0"                         # reference to argument 0, which can be an␣
↪int variable reference, or int for creg index
    ],
    "ql_bregs": [                    # list or "all", defaults to the "B" args
        0,                           # hardcoded breg index
        "%0"                         # reference to argument 0, which can be an␣
↪int variable reference, or int for creg index
    ],
    "ql_duration": 0,                # duration; int to hardcode or "%i" to take␣
↪from param i (must be of type int), defaults to 0
    "ql_angle": 0.0,                 # angle; float to hardcode or "%i" to take␣
↪from param i (must be of type int or real), defaults to first arg of type real␣
↪or 0.0
    "ql_angle_type": "<type>",       # interpretation of angle arg; one of "rad"␣
↪(radians), "deg" (degrees), or "pow2" (2pi/2^k radians), defaults to "rad"
    "implicit_sgmq": <bool>,         # if multiple qubit args are present, a␣
↪single-qubit gate of this type should be replicated for these qubits (instead␣
↪of a single gate with many qubits)
    "implicit_breg": <bool>          # the breg operand(s) that implicitly belongs␣
↪to the qubit operand(s) in the gate should be added to the OpenQL operand list
}
```

The typespec string defines the expected argument types for the gate. Each character in the string represents an argument. The following characters are supported by libqasm:

- Q = qubit
- B = assignable bit/boolean (measurement register)
- b = bit/boolean
- a = axis (x, y, or z)
- I = assignable integer
- i = integer
- r = real
- c = complex
- u = complex matrix of size 4^n, where n is the number of qubits in the parameter list (automatically deduced)
- s = (quoted) string
- j = json

Note that OpenQL only uses an argument if it is referred to in one of the "ql_*" keys, either implicitly or explicitly.

If no custom configuration is specified, the reader defaults to the logic that was hardcoded before it was made configurable. This corresponds to the following JSON:

```
[
    {"name": "measure",     "params": "Q",      "ql_name": "measz"},
    {"name": "measure",     "params": "QB",     "ql_name": "measz"},
    {"name": "measure_x",   "params": "Q",      "ql_name": "measx"},
    {"name": "measure_x",   "params": "QB",     "ql_name": "measx"},
    {"name": "measure_y",   "params": "Q",      "ql_name": "measy"},
    {"name": "measure_y",   "params": "QB",     "ql_name": "measy"},
```

(continues on next page)

(continued from previous page)

```
    {"name": "measure_z",   "params": "Q",      "ql_name": "measz"},
    {"name": "measure_z",   "params": "QB",     "ql_name": "measz"},
    {"name": "prep",        "params": "Q",      "ql_name": "prepz"},
    {"name": "prep_x",      "params": "Q",      "ql_name": "prepx"},
    {"name": "prep_y",      "params": "Q",      "ql_name": "prepy"},
    {"name": "prep_z",      "params": "Q",      "ql_name": "prepz"},
    {"name": "i",           "params": "Q"},
    {"name": "h",           "params": "Q"},
    {"name": "x",           "params": "Q"},
    {"name": "y",           "params": "Q"},
    {"name": "z",           "params": "Q"},
    {"name": "s",           "params": "Q"},
    {"name": "sdag",        "params": "Q"},
    {"name": "t",           "params": "Q"},
    {"name": "tdag",        "params": "Q"},
    {"name": "x90",         "params": "Q",      "ql_name": "rx90"},
    {"name": "mx90",        "params": "Q",      "ql_name": "xm90"},
    {"name": "y90",         "params": "Q",      "ql_name": "ry90"},
    {"name": "my90",        "params": "Q",      "ql_name": "ym90"},
    {"name": "rx",          "params": "Qr"},
    {"name": "ry",          "params": "Qr"},
    {"name": "rz",          "params": "Qr"},
    {"name": "cnot",        "params": "QQ"},
    {"name": "cz",          "params": "QQ"},
    {"name": "swap",        "params": "QQ"},
    {"name": "cr",          "params": "QQr"},
    {"name": "crk",         "params": "QQi",    "ql_angle": "%2", "ql_angle_type
↪": "pow2"},
    {"name": "toffoli",     "params": "QQQ"},
    {"name": "measure_all", "params": "",       "ql_qubits": "all", "implicit_sgmq
↪": true},
    {"name": "display",     "params": ""},
    {"name": "wait",        "params": ""},
    {"name": "wait",        "params": "i"}
]
```

property platform
The platform associated with the reader.

property program
The program that the cQASM circuits will be added to.

__init__(*self*, *platform:* Platform, *program:* Program, *gateset_fname: str*) → *cQasmReader*
__init__(*self*, *platform:* Platform, *program:* Program) → *cQasmReader*
__init__(*self*, *program:* Program, *gateset_fname: str*) → *cQasmReader*
__init__(*self*, *program:* Program) → *cQasmReader*

string2circuit(*self*, *cqasm_str: str*) → None
Interprets a string as cQASM file and adds its contents to the program associated with this reader.

Parameters **cqasm_str** (*str*) – The string representing the contents of the cQASM file.

Returns

Return type None

file2circuit(*self*, *cqasm_file_path: str*) → None
Interprets a string as cQASM file and adds its contents to the program associated with this reader.

Parameters **cqasm_file_path** (*str*) – The path to the cQASM file to load.

Returns

Return type None

1.8 C++ API

If you're more of a C++ than a Python person, the same API exposed to Python can also be used from within C++.

There is currently no supported way to install OpenQL as a system library. Instead, you can use CMake to include OpenQL as a dependency of your program. This is pretty straight-forward:

```
cmake_minimum_required(VERSION 3.1 FATAL_ERROR)

# This would be just add_subdirectory(OpenQL) for your program, or perhaps
# add_subdirectory(deps/OpenQL) if you prefer; wherever your OpenQL git
# submodule is.
add_subdirectory(../.. OpenQL)

# Use whatever CMake magic you need to build your program, but linking
# something against OpenQL should be as easy as the second line. This should
# take care of both the libraries and header file include directories.
add_executable(example example.cc)
target_link_libraries(example ql)
```

With that configuration, `#include <openql>` becomes available, which places the OpenQL API in the `ql` namespace. Here's a basic example of what a program might look like:

```
#include <iostream>
#include <openql>

int main(int argc, char **argv) {
    // create platform
    auto platf = ql::Platform("seven_qubits_chip", "cc_light");

    // create program
    auto prog = ql::Program("aProgram", platf, 2);

    // create kernel
    auto k = ql::Kernel("aKernel", platf, 2);

    k.gate("prepz", 0);
    k.gate("prepz", 1);
    k.gate("x", 0);
    k.gate("y", 1);
    k.measure(0);
    k.measure(1);

    // add kernel to program
    prog.add_kernel(k);

    // compile the program
    prog.compile();

    std::cout << "Seems good to me!" << std::endl;
    return 0;
}
```

The API is documented here.

Note: The API classes are merely wrappers of the classes used internally by OpenQL. You can of course also use the internal classes, but their interfaces should not be assumed to be stable from version to version.

1.9 Configuration

Configurability is a primary design goal of OpenQL: instead of hardcoding the way in which an algorithm is compiled for a particular platform, both the platform and the strategy for compiling to it are completely configurable. As such, OpenQL has quite a complex configuration system.

Most of the configuration is provided to OpenQL via JSON files. OpenQL uses a superset of the JSON file format for all input files, namely one that allows `//`-based single-line comments; therefore, a configuration file written for OpenQL is not strictly valid JSON, but OpenQL can parse any valid JSON file (as long as it complies with the expected structure).

The two most important configuration file types are the *platform* and *compiler* configuration files.

- The platform configuration file includes everything OpenQL needs to know about the target platform (i.e., what the quantum chip and microarchitecture looks like), and optionally includes information about how to compile for it. This file is passed to OpenQL when you construct a `ql.Platform`. OpenQL also has a number of default platform configuration files built into it; one for each architecture variant. Furthermore, architecture variants may include preprocessing logic for the platform configuration file, such that repetitive things for a particular platform can automatically be expanded; in this case, the description below documents the *resulting* structure, not necessarily what you would write (although the preprocessing logic should be minimal, such as only providing additional default values). See also the section on supported architectures.
- The compiler configuration file describes the steps that OpenQL should take to transform the incoming program to something that can run on the platform (or at least is no further away from being able to run on it). This is also referred to as the pass list or compilation strategy. Besides configuration via JSON, the strategy can also be configured using the Python/C++ API directly; this is particularly useful when you for example only want to insert a visualizer pass into the existing, default pass list, or when you're doing design-space exploration to determine the optimal compilation strategy for a particular algorithm.

The structure of these files is documented below.

1.9.1 Platform configuration

The platform configuration JSON file (or JSON data, as it's not necessarily always in file form) represents a complete description of what the target platform looks like, and optionally how to compile for it. At the top level, the structure is a JSON object, with the following keys recognized by OpenQL's platform-agnostic logic, customarily written in the following order.

- `"eqasm_compiler"`: an optional description of how to compile for this platform.
- `"hardware_settings"`: contains basic descriptors for the hardware, such as qubit count and cycle time.
- `"topology"`: optionally provides a more in-depth description of how the qubits are organized.
- `"resources"`: optionally provides information about scheduling constraints, for example due to a number of qubits sharing a single waveform generator.
- `"instructions"`: lists the instruction set supported by the platform.

- `"gate_decomposition"`: optionally lists a set of decomposition rules that are immediately applied when a gate with a particular name is added.

Note: The plan is to move away from on-the-fly gate decomposition, and instead make a gate decomposition pass. The exact design for this has not been made yet, but it's possible that the gate decomposition section will change in minor ways in the future, or will be deprecated in favor of an entirely new configuration file section.

Depending on the architecture being compiled for, as specified through the `"eqasm_compiler"` key or via the compiler configuration file override during platform construction, additional sections or keys may be optional or required, or entire sections may even be generated. Refer to the architecture documentation for details to this end. The `"none"` architecture by definition does not do any of this, and can thus always be used as an override of sorts for this behavior; the only thing that the architecture variant specification does is provide better defaults for the platform and compiler configuration, so everything can always be specified using the `"none"` architecture if need be. The remainder of this page thus describes the "universal" structure as used by `"none"`, while the architecture documentation may make achitecture-specific addenda.

In addition, passes are allowed to make use of additional structures in the configuration file. This implies that the common OpenQL code will *not* check for or warn you about unrecognized keys: it assumes that these will be read by something it is not aware of.

`"eqasm_compiler"` section

The `"eqasm_compiler"` key can take any of the following types of values.

- No value/unspecified: the defaults for the "none" architecture will implicitly be used.
- A string matching one of the available architectures or architecture variants, for example `"cc"` or `"cc_light.s7"`: the defaults for that architecture (variant) will be used. Legacy values, such as `"eqasm_compiler_cc"` or `"qx"` may also be used. Refer to the architecture documentation for a full and up-to-date list of recognized values.
- A filename: the specified file will be interpreted as a compiler configuration file, fully specifying what the compiler looks like.
- A JSON object: the contents of the object will be interpreted as a compiler configuration file, again fully specifying what the compiler looks like, but without the extra file indirection.

This key can also be completely overridden by explicitly specifying a compiler configuration file during platform construction, thus allowing the platform and compiler configuration files to be completely disjoint, if this is preferred for the intended application.

`"hardware_settings"` section

This must map to an object containing the basic parameters that describe the platform. OpenQL's common code recognizes the following.

- `"qubit_number"`: must map to an integer specifying the total number of qubits in the platform. Qubit indices start at zero, so all indices must be in the range 0..N-1, where N is this value.
- `"creg_number"`: optionally specifies the number of 32-bit integer classical registers available in the platform. If not specified, the value will be inferred from the constructor of `ql.Program`.
- `"breg_number"`: optionally specifies the number of single-bit classical registers available in the platform, used for receiving measurement results and predicates. If not specified, the value will be inferred from the constructor of `ql.Program`.

- "cycle_time": optionally specifies the cycle time used by the platform in nanoseconds. Currently this must be an integer value. If not specified, 1 will be used as a default, thus equating the nanosecond values to cycle values.

"topology" section

The topology JSON object must have the following structure.

```
{
    "form": <optional string, either "xy" or "irregular">,
    "x_size": <optional integer for form="xy">,
    "y_size": <optional integer for form="xy">,
    "qubits": <mandatory array of objects for form="xy", unused for "irregular">,
    "number_of_cores": <optional positive integer, default 1>,
    "comm_qubits_per_core": <optional positive integer, num_qubits / number_of_cores>,
    "connectivity": <optional string, either "specified" or "full">,
    "edges": <mandatory array of objects for connectivity="specified", unused for
↪"full">
    ...
}
```

The "form" key specifies whether the qubits can be arranged in a 2D grid of integer coordinates ("xy") or not ("irregular"). If irregular, mapper heuristics that rely on sorting possible paths by angle are unavailable. If "xy", "x_size" and "y_size" specify the coordinate ranges (from zero to the limit minus one), and "qubits" specifies the coordinates. "qubits" must then be an array of objects of the following form:

```
{
    "id": <qubit index, mandatory>,
    "x": <X coordinate, mandatory>,
    "y": <Y coordinate, mandatory>,
    ...
}
```

Each qubit must be specified exactly once. Any additional keys in the object are silently ignored, as other parts of OpenQL may use the structure as well.

If the "form" key is missing, its value is derived from whether a "qubits" list is given. If "x_size" or "y_size" are missing, the values are inferred from the largest coordinate found in "qubits".

The "number_of_cores" key is used to specify multi-core architectures. It must be a positive integer. Each core is assumed to have the same number of qubits, so the total number of qubits must be divisible by this number. The first N qubits belong to core 0, the next N belong to core 1, etc, where N equals the total number of qubits divided by the number of cores.

Cores can communicate only via communication qubits. The amount of these qubits per cores may be set using the "comm_qubits_per_core" key. Its value must range between 1 and the number of qubits per core, and defaults to the latter. The first N qubits for each core are considered to be communication qubits, whereas the remainder are local qubits.

The "connectivity" key specifies whether there are qubit connectivity constraints ("specified") or all qubits (within a core) are connected ("full"). In the former case, the "edges" key must map to an array of objects of the following form:

```
{
    "id": <optional unique identifying integer>,
    "src": <source qubit index, mandatory>,
    "dst": <target qubit index, mandatory>,
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Edges are directional; to allow qubits to interact "in both ways," both directions must be specified. If any identifiers are specified, all edges should get one, and they should all be unique; otherwise, indices are generated using src*nq+dst. Any additional keys in the object are silently ignored, as other parts of OpenQL may use the structure as well (although they should preferably just extend this class).

When `"connectivity"` is set to `"full"` in a multi-core environment, inter-core edges are only generated when both the source and destination qubit is a communication qubit.

If the `"connectivity"` key is missing, its value is derived from whether an "edges" list is given.

Any additional keys in the topology root object are silently ignored, as other parts of OpenQL may use the structure as well.

`"resources"` section

Two JSON structures are supported: one for compatibility with older platform configuration files, and one extended structure. The extended structure has the following syntax:

```
{
    "architecture": <optional string, default "">,
    "dnu": <optional list of strings, default []>,
    "resources": {
        "<name>": {
            "type": "<type>",
            "config": {
                <optional configuration>
            }
        }
        ...
    }
}
```

The optional `"architecture"` key may be used to make shorthands for architecture- specific resources, normally prefixed with `"arch.<architecture>."`. If it's not specified or an empty string, the architecture is derived from the compiler configuration (either `"eqasm_compiler"` or as overridden by the compiler configuration file passed to the platform constructor).

The optional `"dnu"` key may be used to specify a list of do-not-use resource types (experimental, deprecated, or any other resource that's considered unfit for "production" use) that you explicitly want to use, including the `"dnu"` namespace they are defined in. Once specified, you'll be able to use the resource type without the `"dnu"` namespace element. For example, if you would include `"dnu.whatever"` in the list, the resource type `"whatever"` may be used to add the resource.

The `"resources"` key specifies the actual resource list. This consists of a map from unique resource names matching `[a-zA-Z0-9_\-]+` to a resource configuration. The configuration object must have a "type" key, which must identify a resource type that OpenQL knows about. The "config" key is optional, and is used to pass type-specific configuration data to the resource. If not specified, an empty JSON object will be passed to the resource instead.

If the `"resources"` key is not present, the old structure is used instead. This has the following simplified form:

```
{
    "<type>": {
        <configuration>
```

(continues on next page)

(continued from previous page)

```
    },
    ...
}
```

`"instructions"` section

This section specifies the instruction set of the architecture. It must be an object, where each key represents the name of a gate, and the value is again an object, containing any semantical information needed to describe the instruction.

Note: OpenQL currently derives much of the semantics from the gate name. For example, the cQASM writer determines whether it should emit a gate angle parameter based on whether the name of the gate equals a set of gate names that would logically have an angle. This is behavior is very much legacy, and is to be replaced with checking for keys in the instruction definition (though of course using appropriate defaults for backward compatibility).

OpenQL supports two classes of instructions: *generalized gates* and *specialized gates*. For generalized gates, the gate name (i.e. the JSON object key) must be a single identifier matching `[a-zA-Z_][a-zA-Z0-9_]*`. This means that the gate can be applied to any set of operands. Specialized gates, on the other hand, have a fixed set of qubit operands. Their JSON key must be of the form `<name> <qubits>`, where `<name>` is as above, and `<qubits>` is a comma-separated list (without spaces) of qubit indices, each of the form `q<index>`. For example, a correct name for a specialized two-qubit gate would be `cz q0,q1`. Specialized gates allow different semantical parameters to be specified for each possible set of qubit operands: for example, the duration for a particular gate on a particular architecture may depend on the operands.

Note: If you're using the mapper, and thus the input to your program is unmapped, OpenQL will also use the gate specializations when the gates still operate on virtual qubits. Therefore, you *must* specify a specialization for all possible qubit operand combinations of a particular gate, even if the gate does not exist for a particular combination due to for example connectivity constraints. Alternatively, you may specify a fallback using a generalized gate, as OpenQL will favor specialized gates when they exist.

Within the instruction definition object, OpenQL's architecture/pass-agnostic logic currently only recognizes the following keys.

Note: Older versions of OpenQL recognized and required the existence of many more keys, such as `"matrix"` and `"latency"`. All passes relying on this information have since been cleaned out as they were no longer in use, and all requirements on the existence of these keys have likewise been lifted.

`"cqasm_name"` key

Specifies an alternative name for the instruction when it is printed as cQASM or when read from cQASM. This must be a valid identifier. If not specified, it defaults to the normal instruction name.

`"prototype"` key

Specifies the amount and type of operands that the instruction expects. If specified, it must be an array of strings. Each of these strings represents an operand, and must be set to the name of its expected type, optionally prefixed with the access mode, separated by a colon. By default, the available types are:

- `qubit` for qubits;
- `bit` for classical bits;
- `int` for 32-bit signed integers; and
- `real` for floating-point numbers.

The available access modes are:

- `B` for barriers (DDG write, liveness ignore);
- `W` for write access or qubit state preparation (DDG write, liveness kill);
- `U` for read+write/update access or regular non-commuting qubit usage (DDG write, liveness use);
- `R` for read-only access (DDG read, liveness use);
- `L` for operands that must be literals;
- `X` for qubit access that behaves like an X rotation (DDG X, liveness use);
- `Y` for qubit access that behaves like an Y rotation (DDG Y, liveness use);
- `Z` for qubit access that behaves like an Z rotation (DDG Z, liveness use);
- `M` for a measurement of the qubit for which the result is written to its implicitly associated bit register (DDG W for the qubit and its bit, liveness use for the qubit, and liveness kill for the bit); and
- `I` for operands that should be ignored (DDG ignore, liveness ignore).

If the access mode is not specified for an operand, `U` is assumed, as it is the most pessimistic mode available. If no prototype is specified at all, it will be inferred based on the instruction name for backward compatibility, using the following rules (first regex-matching rule applies):

- `move_init|prep(_?[xyz])?` -> `["W:qubit"]`
- `h|i` -> `["U:qubit"]`
- `rx` -> `["X:qubit", "L:real"]`
- `(m|mr|r)?xm?[0-9]*` -> `["X:qubit"]`
- `ry` -> `["Y:qubit", "L:real"]`
- `(m|mr|r)?ym?[0-9]*` -> `["Y:qubit"]`
- `rz` -> `["Z:qubit", "L:real"]`
- `crz?` -> `["Z:qubit", "Z:qubit", "L:real"]`
- `crk` -> `["Z:qubit", "Z:qubit", "L:int"]`
  - `[st](dag)?|(m|mr|r)?zm?[0-9]*` -> `["Z:qubit"]`
- `meas(ure)?(_?[xyz])?(_keep)?` -> `["M:qubit"]` and `["U:qubit", "W:bit"]`
- `(teleport)?(move|swap)` -> `["U:qubit", "U:qubit"]`
- `cnot|cx` -> `["Z:qubit", "X:qubit"]`
- `cphase|cz` -> `["Z:qubit", "Z:qubit"]`

- `cz_park` -> `["Z:qubit", "Z:qubit", "I:qubit"]`
- `toffoli` -> `["Z:qubit", "Z:qubit", "X:qubit"]`
- no operands otherwise.

Furthermore, when gates are added via the API or old IR that don't match an existing instruction due to prototype mismatch, and the prototype was inferred per the above rules, a clone is made of the instruction type with the prototype inferred by means of the actual operands, using the `U` access mode for reference operands and `R` for anything else. When a default gate is encountered in the old IR and needs to be converted to the new IR, the entire instruction type is inferred from the default gate. From now on, however, it is strongly recommended to explicitly specify prototypes and not rely on this inference logic.

Note: It is possible to define multiple overloads for an instruction with the same name. Passes using the new IR will be able to distinguish between these overloads based on the types and writability of the operands, but be aware that any legacy pass will use one of the gate definitions at random (so differing duration or other attributes won't work right). The JSON syntax for this is rather awkward, since object keys must be unique; the best thing to do is to just append spaces for the key, since these spaces are cleaned up when the instruction type is parsed.

`"barrier"` key

An optional boolean that specifies that an instruction is to behave as a complete barrier, preventing it from being commuted with any other instruction during scheduling, and preventing optimizations on it. If not specified, the flag defaults to false.

`"duration"` or `"duration_cycles"` key

These keys specify the duration of the instruction in nanoseconds or cycles. It is illegal to specify both of them for a single instruction. If neither is specified, the duration defaults to a single cycle.

Note: OpenQL currently only supports durations that are an integer number of nanoseconds, so any fractions will be rounded up to the nearest nanosecond. Furthermore, in almost all contexts, the duration of an instruction will be rounded up to the nearest integer cycle count.

`"decomposition"` key

May be used to specify one or more decomposition rules for the instruction type. Unlike the rules in the `"gate_decomposition"` section, these rules are normally only applied by an explicit decomposition pass, of which the predicate matches the name and/or additional JSON data for the rule. The value for the `"decomposition"` key can take a number of shapes:

- a single string: treated as a single, anonymous decomposition rule of which the decomposition is defined by the string parsed as a single-line cQASM 1.2 block;
- an array of strings: as above, but each string represents a new line in the cQASM block;
- a single object: treated as a single decomposition specification; or
- an array of objects: treated as multiple decomposition specifications.

A decomposition specification object must have an `"into"` key that specifies the decomposition, which must be a single string or an array of strings as above. In addition, it may be given a name via the `"name"` key, or any number of other keys for passes to use to determine whether to apply a decomposition rule, or which decomposition rule to apply if multiple options are defined.

The cQASM 1.2 block must satisfy the following rules:

- the version header must not be specified (it is added automatically);
- subcircuits and goto statements are not supported;
- the operands of the to-be-decomposed gate can be accessed using the `op(int) -> ...` function, where the integer specifies the operand index (which must constant-propagate to an integer literal); and
- the duration of the decomposed block may not be longer than the duration of the to-be-decomposed instruction (either make the instruction duration long enough, or define the decomposition as a single bundle and (re)schedule after applying the decomposition).

Other than that, the cQASM code is interpreted using the default cQASM 1.2 rules. Note that this also means that the cQASM name of an instruction must be used if said instructions has differing OpenQL and cQASM names. Refer to the documentation of the cQASM 1.2 reader pass (`io.cqasm.Read`) for more information.

As an example, a CNOT gate with its usual decomposition might be specified as follows.

```
{
    "prototype": ["Z:qubit", "X:qubit"],
    "duration_cycles": 4,
    "decomposition": {
        "name": "to_cz",
        "into": [
            "ym90 op(1)",
            "cz op(0), op(1)",
            "skip 1",
            "y90 op(1)"
        ]
    }
}
```

Note that application of this decomposition rule would retain program validity with respect to schedule and data dependencies (if it was valid before application) for platforms where single-qubit rotations are single-cycle and the CZ gate is two-cycle, because the CNOT gate is defined to take four cycles, and the schedule of the decomposition is valid.

Note: The `"decomposition"` key is only supported when the instruction prototype is explicitly specified using the `"prototype"` key. Without a fixed prototype, type checking the `op()` function would be impossible.

`"qubits"` key

This *must* map to a single qubit index or a list of qubit indices that corresponds to the qubits in the specialization. For generalized instructions, the list must either be empty or unspecified. The qubit indices can be specified as either a string of the form `"q<index>"` or an integer with just the index.

Note: This field is obviously redundant. As such, it may be removed in the future, from which point onward it will be ignored.

"gate_decomposition" section

This section specifies legacy decomposition rules for gates/instructions. They are applied in the following cases:

- when a gate matching a decomposition rule is added to a kernel using the API;
- immediately before a legacy pass (i.e. one that still operates on the old IR) is run; and
- when a decomposition pass matching rules named "legacy" is run.

Rules in this section support only a subset of what the new decomposition system supports. For example, scheduling information cannot be represented, the to-be-decomposed instruction can only have qubit operands, and the to-be-decomposed instruction can't exist in the old IR without being decomposed (preventing operations on it before decomposition). For these reasons, this decomposition system is deprecated, and only still exists for backward compatibility.

If the section is specified, it must be an object, where each key represents the name of the to-be-decomposed gate, along with capture groups for the qubit operands. The keys must map to arrays of strings, wherein each string represents a gate in the decomposition.

Examples of two decompositions are shown below. `%0` and `%1` refer to the first argument and the second argument. This means according to the decomposition on line 2, `rx180 %0` will allow us to decompose `rx180 q0` to `x q0`. Similarly, the decomposition on line 3 will allow us to decompose `cnot q2, q0` to three instructions, namely: `ry90 q0`, `cz q2, q0`, and `ry90 q0`.

```
"gate_decomposition": {
    "rx180 %0" : ["x %0"],
    "cnot %0,%1" : ["ry90 %1","cz %0,%1","ry90 %1"]
}
```

These decompositions are simple macros (in-place substitutions) which allow programmer to manually specify a decomposition. These take place at the time of creation of a gate in a kernel. This means the scheduler will schedule decomposed instructions.

Note: Decomposition rules may only refer to custom gates that have already been defined in the instruction set.

Note: Recursive decomposition rules, i.e. decompositions that make use of other decomposed gate definitions, are not supported. Behavior for this is undefined; the nested rules may or may not end up being expanded, and if they're not, internal compiler errors may result.

Note: These decomposition rules are intended to be replaced by a more powerful system in the future.

1.9.2 Compiler configuration

The compiler configuration JSON file (or JSON substructure) is expected to have the following structure:

```
{
    "architecture": <optional string, default "">,
    "dnu": <optional list of strings, default []>,
    "pass-options": <optional object, default {}>,
    "compatibility-mode": <optional boolean, default false>,
    "passes": [
```

(continues on next page)

(continued from previous page)

```
        <pass description>
    ]
}
```

The optional "`architecture`" key may be used to make shorthands for architecture- specific passes, normally prefixed with "`arch.<architecture>.`". If it's not specified or an empty string, no shorthand aliases are made.

The optional "`dnu`" key may be used to specify a list of do-not-use pass types (experimental passes, deprecated passes, or any other pass that's considered unfit for "production" use) that you explicitly want to use, including the "dnu" namespace they are defined in. Once specified, you'll be able to use the pass type without the "`dnu`" namespace element. For example, if you would include "`dnu.whatever`" in the list, the pass type "`whatever`" may be used to add the pass.

The optional "`pass-options`" key may be used to specify options common to all passes. The values may be booleans, integers, strings, or null, but nothing else. Null is used to reset an option to its hardcoded default value. An option need not exist for each pass affected by it; if it doesn't, the default value is silently ignored for that pass. However, if it *does* exist, it must be a valid value for the option with that name. These option values propagate through the pass tree recursively, so setting a default option in the root using this record will affect all passes.

If "`compatibility-mode`" is enabled, some of OpenQL's global options add implicit entries to the "`pass-options`" structure when set, for backward compatibility. However, entries in "`pass-options`" always take precedence. The logic for which options map to which is mostly documented in the global option docs now, since those options don't do anything else anymore. Note that the global options by their original design have no way to specify what pass they refer to, so each option is attempted for each pass type! Which means we have to be a bit careful with picking option names for the passes that are included in compatibility mode.

Pass descriptions can either be strings (in which case the string is interpreted as a pass type alias and everything else is inferred/default), or an object with the following structure.

```
{
    "type": <optional string, default "">,
    "name": <optional string, default "">,
    "options": <optional object, default {}>
}
```

The "`type`" key, if specified, must identify a pass type that OpenQL knows about. You can call `print_pass_types()` on a `ql.Compiler` object to get the list of available pass types (and their documentation) for your particular configuration (just make an empty compiler object initially), or you can read the documentation section on supported passes. If the "`type`" key is not specified or empty, a group is made instead, and "`group`" must be specified for the group to do anything.

The "`name`" key, if specified, is a user-defined name for the pass, that must match `[a-zA-Z0-9_\-]+` and be unique within the surrounding pass list. If not specified, a name that complies with these requirements is generated automatically, but the actual generated name should not be relied upon to be consistent between OpenQL versions. The name may be used to programmatically refer to passes after construction, and passes may use it for logging or unique output filenames. However, passes should not use the name for anything that affects the behavior of the pass.

The "`options`" key, if specified, may be an object that maps option names to option values. The values may be booleans, integers, strings, or null, but nothing else. Null is used to enforce usage of the OpenQL-default value for the option. The option names and values must be supported by the particular pass type.

1.10 Supported architectures

This section lists the backend architectures currently supported by OpenQL.

Architectures are organized into *families*, *variants*, and *platforms*. The architecture family typically refers to a particular control architecture, and may include custom passes and scheduler resources needed to compile for that control architecture. The variant usually refers to the kind of qubit chip being controlled by said control architecture, such as surface-5, surface-7, or surface-17 for CC-light. Finally, architecture variants may be configured with a JSON configuration file to get a complete target description, referred to as the platform.

Note: In older versions of OpenQL, there was no hierarchical definition like this. Instead, there was only a platform, defined by a hard-to-write JSON configuration file, with a backend selection using the `"eqasm_compiler"` key in the configuration file. The new system is fully backward-compatible: you can still just pass any custom JSON configuration file to the platform constructor, in which case the architecture family will be chosen based on the value in `"eqasm_compiler"`, and the default variant for that family will be used. The purpose of the new system is to make the learning curve for using OpenQL less steep; instead of having to pluck one of the many JSON files from OpenQL's `tests` directory and usually needing to modify it manually, you can start off by just using one of the architecture variants as default. Furthermore, each architecture variant may have its own set of defaults and preprocessing rules for the platform configuration file, such that even if you do need to make changes, it should be way easier to do so.

The active architecture is selected in one of the following ways:

- by specifying the namespace name/variant pair for the desired architecture directly to the `ql.Platform` constructor (the architecture will then use the default platform configuration for that pair);
- by specifying a recognized string for the `"eqasm_compiler"` key in the platform configuration file;
- if `"eqasm_compiler"` is instead used for an inline compiler configuration, by setting `"eqasm_compiler"."architecture"` to the namespace name and variant of the desired architecture;
- if none of the above apply, the dummy *"none"* architecture will be selected.

The variant is separated from the namespace name or `eqasm_compiler` name using a `.`.

Ultimately, the architecture variant system only serves to inject sane defaults into the platform and compiler configuration structures, and to separate unavoidable architecture-specific logic from architecture-agnostic logic in OpenQL's codebase. That is, everything boils down to these (internal) platform and compiler configuration structures during platform construction, after which everything is agnostic to the selected architecture. This means that if you want to compile for a new architecture that's sufficiently similar to an existing one to not need any new passes or resources, you may not even have to change or add to OpenQL's codebase; you can just use the `none.default` architecture and build the platform and compiler configuration structures from scratch. This is intentional: the control architectures are still as much in flux as the quantum chips themselves, so being able to quickly piece together a compiler for an architecture we haven't even thought of yet is important. It's also very useful for design-space exploration, and doing research into compilation strategies and control architectures that will become relevant only when the quantum chips mature further.

1.10.1 QuTech Central Controller

- Pass/resource/C++ namespace: `arch.cc`
- Acceptable `"eqasm_compiler"` values: `"cc"` or `"eqasm_backend_cc"`

This architecture allows compilation for the QuTech Central Controller, as currently in use in DiCarloLab for the Starmon chip.

Platform configuration file additions

For the CC backend, contrary to the original one for CC-light, the final hardware output is *entirely determined* by the contents of the configuration file. That is, there is no built-in knowledge of instrument connectivity or codeword organization. This requires a few additional target-specific sections in the platform configuration.

Instrument definitions

The CC-specific `"instrument_definitions"` section of the configuration file defines immutable properties of instruments, i.e. independent of the actual control setup. The required structure is as follows:

```
"instrument_definitions": {
    "qutech-qwg": {
        "channels": 4,
        "control_group_sizes": [1, 4],
    },
    "zi-hdawg": {
        "channels": 8,
        // NB: size=1 needs special treatment of waveforms because one
        // AWG unit drives 2 channels
        "control_group_sizes": [1, 2, 4, 8],
    },
    "qutech-vsm": {
        "channels": 32,
        "control_group_sizes": [1],
    },
    "zi-uhfqa": {
        "channels": 9,
        "control_group_sizes": [1],
    }
}
```

In this:

- `"channels"` defines the number of logical channels of the instrument. For most instruments there is one logical channel per physical channel, but the 'zi-uhfqa' provides 9 logical channels on one physical channel pair.
- `"control_group_sizes"` states possible arrangements of channels operating as a vector.

Control modes

The "`control_modes`" section defines modes to control instruments. These define which bits are used to control groups of channels and/or get back measurement results. The expected structure is as follows:

```
"control_modes": {
    "awg8-mw-vsm-hack": {                     // ZI_HDAWG8.py::cfg_codeword_
↪protocol() == 'microwave'. Old hack to skip DIO[8]
        "control_bits": [
            [7,6,5,4,3,2,1,0],                // group 0
            [16,15,14,13,12,11,10,9]          // group 1
        ],
        "trigger_bits": [31]
    },
    "awg8-mw-vsm": {                          // the way the mode above should have␣
↪been
        "control_bits": [
            [7,6,5,4,3,2,1,0],                // group 0
            [15,14,13,12,11,10,9,8]           // group 1
        ],
        "trigger_bits": [31]
    },
    "awg8-mw-direct-iq": {                    // just I&Q to generate microwave␣
↪without VSM. HDAWG8: "new_novsm_microwave"
        "control_bits": [
            [6,5,4,3,2,1,0],                  // group 0
            [13,12,11,10,9,8,7],              // group 1
            [22,21,20,19,18,17,16],           // group 2. NB: starts at bit 16 so␣
↪twin-QWG can also support it
            [29,28,27,26,25,24,23]            // group 4
        ],
        "trigger_bits": [15,31]
    },
    "awg8-flux": {                             // ZI_HDAWG8.py::cfg_codeword_
↪protocol() == 'flux'
        // NB: please note that internally one AWG unit handles 2 channels, which␣
↪requires special handling of the waveforms
        "control_bits": [
            [2,1,0],                          // group 0
            [5,4,3],
            [8,7,6],
            [11,10,9],
            [18,17,16],                       // group 4. NB: starts at bit 16 so␣
↪twin-QWG can also support it
            [21,20,19],
            [24,23,22],
            [27,26,25]                        // group 7
        ],
        "trigger_bits": [31]
    },
    "awg8-flux-vector-8": {                    // single code word for 8 flux␣
↪channels.
        "control_bits": [
            [7,6,5,4,3,2,1,0]
        ],
        "trigger_bits": [31]
    },
```

(continues on next page)

(continued from previous page)

```
    "uhfqa-9ch": {
        "control_bits": [[17],[18],[19],[20],[21],[22],[23],[24],[25]],     //␣
↪group[0:8]
        "trigger_bits": [16],
        "result_bits": [[1],[2],[3],[4],[5],[6],[7],[8],[9]],              //␣
↪group[0:8]
        "data_valid_bits": [0]
    },
    "vsm-32ch":{
        "control_bits": [
            [0],[1],[2],[3],[4],[5],[6],[7],                          // group[0:7]
            [8],[9],[10],[11],[12],[13],[14],[15],                    // group[8:15]
            [16],[17],[18],[19],[20],[21],[22],[23],                  // group[16:23]
            [24],[25],[26],[27],[28],[28],[30],[31]                   // group[24:31]
        ],
        "trigger_bits": []                                           // no trigger
    }
}
```

In this:

- `<key>` is a name which can be referred to from key `instruments/[]/ref_control_mode`.
- `"control_bits"` defines G groups of B bits, where:
  - G determines which `instrument_definitions/<key>/control_group_sizes` is used; and
  - B is an ordered list of bits (MSB to LSB) used for the code word.
- `"trigger_bits"` must be a vector of bits, used to trigger the instrument. Must either be size 1 (common trigger) or size G (separate trigger per group), or 2 (common trigger duplicated on 2 bits, to support dual-QWG).
- `"result_bits"` is reserved for future use.
- `"data_valid_bits"` is reserved for future use.

Signals

The `"signals"` section provides a signal library that gate definitions can refer to. The expected structure is as follows:

```
"signals": {
    "single-qubit-mw": [
        {   "type": "mw",
            "operand_idx": 0,
            "value": [
                "{gateName}-{instrumentName}:{instrumentGroup}-gi",
                "{gateName}-{instrumentName}:{instrumentGroup}-gq",
                "{gateName}-{instrumentName}:{instrumentGroup}-di",
                "{gateName}-{instrumentName}:{instrumentGroup}-dq"
            ]
        },
        {   "type": "switch",
            "operand_idx": 0,
            "value": ["dummy"]                                      // NB: no actual␣
↪signal is generated
        }
```

(continues on next page)

(continued from previous page)

```
    ],
    "two-qubit-flux": [
        {   "type": "flux",
            "operand_idx": 0,                                // control
            "value": ["flux-0-{qubit}"]
        },
        {   "type": "flux",
            "operand_idx": 1,                                // target
            "value": ["flux-1-{qubit}"]
        }
    ]
}
```

In this, the toplevel object key is a name which can be referred to from key `instructions/<>/cc/ref_signal`. It defines an array of records with the fields below:

- `"type"` defines a signal type. This is used to select an instrument that provides that signal type through key `instruments/*/signal_type`. The types are entirely user defined, there is no builtin notion of their meaning.
- `"operand_idx"` states the operand index of the instruction/gate this signal refers to. Signals must be defined for all `operand_idx` the gate refers to, so a 3-qubit gate needs to define 0 through 2. Several signals with the same operand_idx can be defined to select several signal types, as shown in "single-qubit-mw" which has both "mw" (provided by an AWG) and "switch" (provided by a VSM).
- `"value"` defines a vector of signal names. Supports the following macro expansions:
  - `{gateName}`
  - `{instrumentName}`
  - `{instrumentGroup}`
  - `{qubit}`

Instruments

The `"instruments"` section defines instruments used in this setup, their configuration and connectivity. The expected structure is as follows:

```
"instruments": [
    // readout.
    {
        "name": "ro_0",
        "qubits": [[6], [11], [], [], [], [], [], [], []],
        "signal_type": "measure",
        "ref_instrument_definition": "zi-uhfqa",
        "ref_control_mode": "uhfqa-9ch",
        "controller": {
            "name": "cc",
            "slot": 0,
            "io_module": "CC-CONN-DIO"
        }
    },
    // ...

    // microwave.
```

(continues on next page)

(continued from previous page)

```
    {
        "name": "mw_0",
        "qubits": [                                                 // data qubits:
            [2, 8, 14],                                             // [freq L]
            [1, 4, 6, 10, 12, 15]                                   // [freq H]
        ],
        "signal_type": "mw",
        "ref_instrument_definition": "zi-hdawg",
        "ref_control_mode": "awg8-mw-vsm-hack",
        "controller": {
            "name": "cc",
            "slot": 3,
            "io_module": "CC-CONN-DIO-DIFF"
        }
    },
    // ...

    // VSM
    {
        "name": "vsm_0",
        "qubits": [
            [2], [8], [14], [],  [], [], [], [],                    // [freq L]
            [1], [4], [6], [10], [12], [15], [], [],                // [freq H]
            [0], [5], [9], [13], [], [], [], [],                    // [freq Mg]
            [3], [7], [11], [16], [], [], [], []                    // [freq My]
        ],
        "signal_type": "switch",
        "ref_instrument_definition": "qutech-vsm",
        "ref_control_mode": "vsm-32ch",
        "controller": {
            "name": "cc",
            "slot": 5,
            "io_module": "cc-conn-vsm"
        }
    },

    // flux
    {
        "name": "flux_0",
        "qubits": [[0], [1], [2], [3], [4], [5], [6], [7]],
        "signal_type": "flux",
        "ref_instrument_definition": "zi-hdawg",
        "ref_control_mode": "awg8-flux",
        "controller": {
            "name": "cc",
            "slot": 6,
            "io_module": "CC-CONN-DIO-DIFF"
        }
    },
    // ...
]
```

In this:

- `"name"` is a friendly name for the instrument.
- `"ref_instrument_definition"` selects a record under `"instrument_definitions"`, which must exist or an error is raised.

- "ref_control_mode" selects a record under "control_modes", which must exist or an error is raised.
- "signal_type" defines which signal type this instrument instance provides.
- "qubits" G groups of 1 or more qubits. G must match one of the available group sizes of instrument_definitions/<ref_instrument_definition>/control_group_sizes. If more than 1 qubits are stated per group - e.g. for an AWG used in conjunction with a VSM - they may not produce conflicting signals at any time slot, or an error is raised.
- "force_cond_gates_on" is optional, reserved for future use.
- "controller/slot" is the slot number of the CC this instrument is connected to.
- "controller/name" is reserved for future use.
- "controller/io_module" is reserved for future use.

Additional instruction data

The CC backend extends the instruction definitions with a "cc" subsection, as shown in the example below:

```
"ry180": {
    "duration": 20,
    "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
    "cc": {
        "ref_signal": "single-qubit-mw",
        "static_codeword_override": [2]
    }
},
"cz_park": {
    "duration": 40,
    "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
    "cc": {
        "signal": [
            {   "type": "flux",
                "operand_idx": 0,                                   // control
                "value": ["flux-0-{qubit}"]
            },
            {   "type": "flux",
                "operand_idx": 1,                                   // target
                "value": ["flux-1-{qubit}"]
            },
            {   "type": "flux",
                "operand_idx": 2,                                   // park
                "value": ["park_cz-{qubit}"]
            }
        ],
        "static_codeword_override": [1,2,3]
    }
}
"_wait_uhfqa": {
    "duration": 220,
    "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
    "cc": {
        "signal": []
    }
},
"_dist_dsm": {
```

(continues on next page)

(continued from previous page)

```
    "duration": 20,
    "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
    "cc": {
        "readout_mode": "feedback",
        "signal": [
            {   "type": "measure",
                "operand_idx": 0,
                "value": []
            }
        ]
    }
},
"_wait_dsm": {
    "duration": 80,
    "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
    "cc": {
        "signal": []
    }
},
"if_1_break": {
    "duration": 60,
    "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
    "cc": {
        "signal": [],
        "pragma": {
            "break": 1
        }
    }
}
```

In this:

- `"ref_signal"` points to a signal definition in `hardware_settings/eqasm_backend_cc/signals`, which must exist or

    an error is raised.

- `"signal"` defines a signal in place, in an identical fashion as `hardware_settings/eqasm_backend_cc/signals`. May be empty (`[]`) to disable signal generation.
- `"static_codeword_override"` provides a user defined array of codeword (one entry per operand) for this instruction. Currently, this key is compulsory (if signal is non-empty), but in the future, codewords will be assigned automatically to make better use of the limited codeword space.
- `"readout_mode"` defines an instruction to perform readout if non-empty. If the value "feedback" is used, code is generated to read and distribute the instrument result.
- `"pragma/break"` enables special functionality which makes the gate break out of a for loop if the associated qubit was measured as 1 (`"pragma" { "break": 1 }`) or 0 (`"pragma" { "break": 0 }`).

Program flow feedback

To support Repeat Until Success type experiments, two special fields were added to the gate definition for the CC, as shown in the previous section:

- the `"readout_mode": "feedback"` clause in the `"_dist_dsm"` gate causes the backend to generate code to retrieve the measurement result from the DIO interface and distribute it across the CC;
- the `"pragma": { "break": 1 }` clause in the `"if_1_break"` gate causes the backend to generate code to break out of a OpenQL loop if the associated qubit is read as 1 (or similarly if 0).

For convenience, the gate decomposition section can be extended with `"measure_fb %0": ["measure %0", "_wait_uhfqa %0", "_dist_dsm %0", "_wait_dsm %0"]`

This creates a `measure_fb` instruction consisting of four parts:

- triggering a measurement (on the UHFQA);
- waiting for the internal processing time of the UHFQA;
- retrieve the measurement result, and distribute it across the CC; and
- wait fot the data distribution to finish.

The following example code contains a real RUS experiment using PycQED:

```python
from pycqed.measurement.openql_experiments import openql_helpers as oqh
for i, angle in enumerate(angles):
    oqh.ql.set_option('output_dir', 'd:\\githubrepos\\pycqed_py3\\pycqed\\measurement\
↪\openql_experiments\\output')
    p = oqh.create_program('feedback_{}'.format(angle), config_fn)
    k = oqh.create_kernel("initialize_block_{}".format(angle), p)

    # Initialize
    k.prepz(qidx)

    # Block do once (prepare |1>)
    k.gate("rx180", [qidx])
    p.add_kernel(k)

    # Begin conditional block
    q = oqh.create_kernel("conditional_block_{}".format(angle), p)
    # Repeat until success 0
    q.gate("measure_fb", [qidx])
    q.gate("if_0_break", [qidx])

    # Correction for result 1
    q.gate("rx180", [qidx])
    p.add_for(q, 1000000)

    # Block finalize
    r = oqh.create_kernel("finalize_block_{}".format(angle), p)
    cw_idx = angle // 20 + 9
    r.gate('cw_{:02}'.format(cw_idx), [qidx])

    # Final measurement
    r.gate("measure_fb", [qidx])
    p.add_kernel(r)

    oqh.compile(p, extra_openql_options=[('backend_cc_run_once', 'yes')])
```

Caveats:

- It is not possible to mix `measure_fb` and `measure` in a single program. This is a consequence of the way measurements are read from the input DIO interface of the CC: every measurement (both from `measure_fb` and `measure`) is pushed onto an input FIFO. This FIFO is only popped by a `measure_fb` instruction. If the two types are mixed, misalignment occurs between what is written and read. No check is currently performed by the backend.
- `break` statements may only occur inside a `for` loop. No check is currently performed by the backend.
- `break` statements implicitly refer to the last `measure_fb` earlier in code as a result of implicit allocation of variables.

These limitations will vanish when integration with cQASM 2.0 is completed.

Code generation pass

Most of the magic for all of the above happens in the code generation pass, `arch.cc.VQ1Asm`. This generates the following file types:

- `.vq1asm`: 'Vectored Q1 assembly' file for the Central Controller.
- `.vcd`: timing file, can be viewed using GTKWave (http://gtkwave.sourceforge.net).

For configuration options, please refer to the documentation of this pass.

Default pass list

For the current/default global option values and the default variant (`default`), the following backend passes are used by default.

```
- scheduler: sch.ListSchedule
   |- resource_constraints: yes

- codegen: arch.cc.gen.VQ1Asm
   |- output_prefix: test_output/%N
```

Default configuration file

When no platform configuration file is specified, the following default file is used instead.

```
{
    "eqasm_compiler" : "cc",
    "hardware_settings": {
        "qubit_number": 5,
        "cycle_time" : 20,
        "eqasm_backend_cc": {
            "instrument_definitions": {
                "qutech-qwg": {
                    "channels": 4,
                    "control_group_sizes": [1, 4],
                    "latency": 50
                },
                "zi-hdawg": {
                    "channels": 8,
                    "control_group_sizes": [1, 2, 4, 8],
```

(continues on next page)

(continued from previous page)

```
                "latency": 300
            },
            "qutech-vsm": {
                "channels": 32,
                "control_group_sizes": [1],
                "latency": 10
            },
            "zi-uhfqa": {
                "channels": 9,
                "control_group_sizes": [1],
                "latency": 150
            }
        },
        "control_modes": {
            "awg8-mw-vsm-hack": {
                "control_bits": [
                    [7,6,5,4,3,2,1,0],
                    [16,15,14,13,12,11,10,9]
                ],
                "trigger_bits": [31]
            },
            "awg8-mw-vsm": {
                "control_bits": [
                    [7,6,5,4,3,2,1,0],
                    [23,22,21,20,19,18,17,16]
                ],
                "trigger_bits": [31]
            },
            "awg8-mw-direct-iq": {
                "control_bits": [
                    [6,5,4,3,2,1,0],
                    [13,12,11,10,9,8,7],
                    [22,21,20,19,18,17,16],
                    [29,28,27,26,25,24,23]
                ],
                "trigger_bits": [15,31]
            },
            "awg8-flux": {
                "control_bits": [
                    [2,1,0],
                    [5,4,3],
                    [8,7,6],
                    [11,10,9],
                    [18,17,16],
                    [21,20,19],
                    [24,23,22],
                    [27,26,25]
                ],
                "trigger_bits": [31]
            },
            "awg8-flux-vector-8": {
                "control_bits": [
                    [7,6,5,4,3,2,1,0]
                ],
                "trigger_bits": [31]
            },
            "dualqwg-mw-direct-iq": {
```

(continues on next page)

(continued from previous page)

```
                "control_bits": [
                    [6,5,4,3,2,1,0],
                    [13,12,11,10,9,8,7],
                    [22,21,20,19,18,17,16],
                    [29,28,27,26,25,24,23]
                ],
                "trigger_bits": [15,31]
            },
            "uhfqa-9ch": {
                "control_bits": [[17],[18],[19],[20],[21],[22],[23],[24],[25]],
                "trigger_bits": [16],
                "result_bits": [[1],[2],[3],[4],[5],[6],[7],[8],[9]],
                "data_valid_bits": [0]
            },
            "vsm-32ch":{
                "control_bits": [
                    [0],[1],[2],[3],[4],[5],[6],[7],
                    [8],[9],[10],[11],[12],[13],[14],[15],
                    [16],[17],[18],[19],[20],[21],[22],[23],
                    [24],[25],[26],[27],[28],[28],[30],[31]
                ],
                "trigger_bits": []
            }
        },
        "signals": {
            "single-qubit-mw": [
                {
                    "type": "mw",
                    "operand_idx": 0,
                    "value": [
                        "{gateName}-{instrumentName}:{instrumentGroup}-i",
                        "{gateName}-{instrumentName}:{instrumentGroup}-q"
                    ]
                }
            ],
            "two-qubit-flux": [
                {
                    "type": "flux",
                    "operand_idx": 0,
                    "value": ["flux-0-{qubit}"]
                },
                {
                    "type": "flux",
                    "operand_idx": 1,
                    "value": ["flux-1-{qubit}"]
                }
            ],
            "single-qubit-flux": [
                {
                    "type": "flux",
                    "operand_idx": 0,
                    "value": ["flux-0-{qubit}"]
                }
            ]
        },
        "instruments": [
            {
```

(continues on next page)

(continued from previous page)

```
                "name": "ro_1",
                "qubits": [[0], [2], [3], [4], [], [], [], [], []],
                "signal_type": "measure",
                "ref_instrument_definition": "zi-uhfqa",
                "ref_control_mode": "uhfqa-9ch",
                "controller": {
                    "name": "cc",
                    "slot": 0,
                    "io_module": "CC-CONN-DIO"
                }
            },
            {
                "name": "ro_2",
                "qubits": [[1], [], [], [], [], [], [], [], []],
                "signal_type": "measure",
                "ref_instrument_definition": "zi-uhfqa",
                "ref_control_mode": "uhfqa-9ch",
                "controller": {
                    "name": "cc",
                    "slot": 1,
                    "io_module": "CC-CONN-DIO"
                }
            },
            {
                "name": "mw_0",
                "qubits": [
                    [0],
                    [1],
                    [2],
                    [3]
                ],
                "signal_type": "mw",
                "ref_instrument_definition": "zi-hdawg",
                "ref_control_mode": "awg8-mw-direct-iq",
                "controller": {
                    "name": "cc",
                    "slot": 2,
                    "io_module": "CC-CONN-DIO-DIFF"
                }
            },
            {
                "name": "mw_1",
                "qubits": [
                    [4],
                    [],
                    [],
                    []
                ],
                "signal_type": "mw",
                "ref_instrument_definition": "zi-hdawg",
                "ref_control_mode": "awg8-mw-direct-iq",
                "controller": {
                    "name": "cc",
                    "slot": 3,
                    "io_module": "CC-CONN-DIO-DIFF"
                }
            },
```

(continues on next page)

(continued from previous page)

```
                {
                    "name": "flux_0",
                    "qubits": [[0], [1], [2], [3], [4], [], [], []],
                    "signal_type": "flux",
                    "ref_instrument_definition": "zi-hdawg",
                    "ref_control_mode": "awg8-flux",
                    "controller": {
                        "name": "cc",
                        "slot": 4,
                        "io_module": "CC-CONN-DIO-DIFF"
                    }
                }
            ]
        }
    },
    "gate_decomposition": {
        "measz %0": ["measure %0"],
        "x %0": ["rx180 %0"],
        "y %0": ["ry180 %0"],
        "h %0": ["ry90 %0", "ry180 %0"],
        "z %0": ["rx180 %0","ry180 %0"],
        "t %0": ["ry90 %0","rx45 %0","rym90 %0"],
        "tdag %0": ["ry90 %0","rxm45 %0","rym90 %0"],
        "s %0": ["ry90 %0","rx90 %0","rym90 %0"],
        "sdag %0": ["ry90 %0","rxm90 %0","rym90 %0"],
        "cnot %0 %1": ["rym90 %1", "cz %0 %1", "ry90 %1"],
        "cz q0 q2": ["barrier q0,q2", "sf_cz_se q0", "sf_cz_nw q2", "barrier q0,q2"],
        "cz q2 q0": ["barrier q0,q2", "sf_cz_se q0", "sf_cz_nw q2", "barrier q0,q2"],
        "cz q1 q2": ["barrier q1,q2", "sf_cz_sw q1", "sf_cz_ne q2", "barrier q1,q2"],
        "cz q2 q1": ["barrier q1,q2", "sf_cz_sw q1", "sf_cz_ne q2", "barrier q1,q2"],
        "cz q3 q2": ["barrier q2,q3,q4", "sf_cz_sw q2", "sf_cz_ne q3", "sf_park q4",
↪"barrier q2,q3,q4"],
        "cz q2 q3": ["barrier q2,q3,q4", "sf_cz_sw q2", "sf_cz_ne q3", "sf_park q4",
↪"barrier q2,q3,q4"],
        "cz q4 q2": ["barrier q2,q3,q4", "sf_cz_se q2", "sf_cz_nw q4", "sf_park q3",
↪"barrier q2,q3,q4"],
        "cz q2 q4": ["barrier q2,q3,q4", "sf_cz_se q2", "sf_cz_nw q4", "sf_park q3",
↪"barrier q2,q3,q4"],
        "x180 %0": ["rx180 %0"],
        "y180 %0": ["ry180 %0"],
        "y90 %0": ["ry90 %0"],
        "x90 %0": ["rx90 %0"],
        "ym90 %0": ["rym90 %0"],
        "xm90 %0": ["rxm90 %0"],
        "cl_0 %0": ["i %0"],
        "cl_1 %0": ["ry90 %0", "rx90 %0"],
        "cl_2 %0": ["rxm90 %0", "rym90 %0"],
        "cl_3 %0": ["rx180 %0"],
        "cl_4 %0": ["rym90 %0", "rxm90 %0"],
        "cl_5 %0": ["rx90 %0", "rym90 %0"],
        "cl_6 %0": ["ry180 %0"],
        "cl_7 %0": ["rym90 %0", "rx90 %0"],
        "cl_8 %0": ["rx90 %0", "ry90 %0"],
        "cl_9 %0": ["rx180 %0", "ry180 %0"],
        "cl_10 %0": ["ry90 %0", "rxm90 %0"],
        "cl_11 %0": ["rxm90 %0", "ry90 %0"],
        "cl_12 %0": ["ry90 %0", "rx180 %0"],
```

(continues on next page)

(continued from previous page)

```
        "cl_13 %0": ["rxm90 %0"],
        "cl_14 %0": ["rx90 %0", "rym90 %0", "rxm90 %0"],
        "cl_15 %0": ["rym90 %0"],
        "cl_16 %0": ["rx90 %0"],
        "cl_17 %0": ["rx90 %0", "ry90 %0", "rx90 %0"],
        "cl_18 %0": ["rym90 %0", "rx180 %0"],
        "cl_19 %0": ["rx90 %0", "ry180 %0"],
        "cl_20 %0": ["rx90 %0", "rym90 %0", "rx90 %0"],
        "cl_21 %0": ["ry90 %0"],
        "cl_22 %0": ["rxm90 %0", "ry180 %0"],
        "cl_23 %0": ["rx90 %0", "ry90 %0", "rxm90 %0"],
        "measure_fb %0": ["measure %0", "_wait_uhfqa %0", "_dist_dsm %0", "_wait_dsm
↪%0"]
    },
    "instructions": {
        "i": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "signal": [],
                "static_codeword_override": [0]
            }
        },
        "rx180": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [1]
            }
        },
        "ry180": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [2]
            }
        },
        "rx90": {
            "prototype": ["X:qubit", "L:real"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [3]
            }
        },
        "ry90": {
            "prototype": ["Y:qubit", "L:real"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [4]
            }
        },
        "rxm90": {
            "prototype": ["X:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [5]
            }
        },
        "rym90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [6]
            }
        },
        "ry45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [1]
            }
        },
        "rym45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [2]
            }
        },
        "rx45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [3]
            }
        },
        "rxm45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [4]
            }
        },
        "cz": {
            "prototype": ["Z:qubit", "Z:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [1,1]
            }
        },
        "cz_park": {
            "prototype": ["Z:qubit", "Z:qubit", "I:qubit"],
            "duration": 80,
```

(continues on next page)

(continued from previous page)

```
            "cc": {
                "signal": [
                    {   "type": "flux",
                        "operand_idx": 0,
                        "value": ["flux-0-{qubit}"]
                    },
                    {   "type": "flux",
                        "operand_idx": 1,
                        "value": ["flux-1-{qubit}"]
                    },
                    {   "type": "flux",
                        "operand_idx": 2,
                        "value": ["park_cz-{qubit}"]
                    }
                ],
                "static_codeword_override": [0,0,0]
            }
        },
        "park_cz" : {
            "prototype": ["U:qubit"],
            "duration" : 80,
            "cc": {
                "signal": [
                    {    "type": "flux",
                        "operand_idx": 0,
                        "value": ["park_cz-{qubit}"]
                    }
                ],
                "static_codeword_override": [0]
            }
        },
        "park_measure" : {
            "prototype": ["B:qubit"],
            "duration" : 2000,
            "cc": {
                "signal": [
                    {    "type": "flux",
                        "operand_idx": 0,
                        "value": ["park_measure-{qubit}"]
                    }
                ],
                "static_codeword_override": [0]
            }
        },
        "prepz": {
            "prototype": ["W:qubit"],
            "duration": 200000,
            "cc": {
                "signal": [],
                "static_codeword_override": [0]
            }
        },
        "measure": {
            "prototype": ["M:qubit"],
            "duration": 2000,
            "cc": {
                "readout_mode": "",
```

(continues on next page)

(continued from previous page)

```
                "signal": [
                    {    "type": "measure",
                        "operand_idx": 0,
                        "value": ["dummy"],
                        "weight": ["dummy"]
                    }
                ],
                "static_codeword_override": [0]
            },
            "decomposition": {
                "name": "desugar",
                "into": "measure op(0), bit(op(0))"
            }
        },
        "measure ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 2000,
            "cc": {
                "readout_mode": "",
                "signal": [
                    {    "type": "measure",
                        "operand_idx": 0,
                        "value": ["dummy"],
                        "weight": ["dummy"]
                    }
                ],
                "static_codeword_override": [0]
            }
        },
        "_wait_uhfqa": {
            // not sure what the prototype should be
            "duration": 220,
            "cc": {
                "signal": []
            }
        },
        "_dist_dsm": {
            // not sure what the prototype should be
            "duration": 20,
            "cc": {
                "readout_mode": "feedback",
                "signal": [
                    {    "type": "measure",
                        "operand_idx": 0,
                        "value": []
                    }
                ]
            }
        },
        "_wait_dsm": {
            // not sure what the prototype should be
            "duration": 80,
            "cc": {
                "signal": []
            }
        },
        "if_0_break": {
```

(continues on next page)

(continued from previous page)

```
            "prototype": ["R:bit"],
            "barrier": true,
            "duration": 60,
            "cc": {
                "signal": [],
                "pragma": {
                    "break": 0
                }
            }
        },
        "if_1_break": {
            "prototype": ["R:bit"],
            "barrier": true,
            "duration": 60,
            "cc": {
                "signal": [],
                "pragma": {
                    "break": 1
                }
            }
        },
        "square": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "spec": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "rx12": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "cw_00": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "cw_01": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
```

(continues on next page)

(continued from previous page)

```
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [1]
            }
        },
        "cw_02": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [2]
            }
        },
        "cw_03": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [3]
            }
        },
        "cw_04": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [4]
            }
        },
        "cw_05": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [5]
            }
        },
        "cw_06": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [6]
            }
        },
        "cw_07": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [7]
            }
        },
        "cw_08": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
```

(continues on next page)

(continued from previous page)

```
                "static_codeword_override": [8]
            }
        },
        "cw_09": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [9]
            }
        },
        "cw_10": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "cw_11": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [1]
            }
        },
        "cw_12": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [2]
            }
        },
        "cw_13": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [3]
            }
        },
        "cw_14": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [4]
            }
        },
        "cw_15": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [5]
```

(continues on next page)

(continued from previous page)

```
            }
        },
        "cw_16": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [6]
            }
        },
        "cw_17": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [7]
            }
        },
        "cw_18": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [8]
            }
        },
        "cw_19": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [9]
            }
        },
        "cw_20": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "cw_21": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [1]
            }
        },
        "cw_22": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [2]
            }
```

(continues on next page)

(continued from previous page)

```
        },
        "cw_23": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [3]
            }
        },
        "cw_24": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [4]
            }
        },
        "cw_25": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [5]
            }
        },
        "cw_26": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [6]
            }
        },
        "cw_27": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [7]
            }
        },
        "cw_28": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [8]
            }
        },
        "cw_29": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [9]
            }
        },
```

(continues on next page)

(continued from previous page)

```
        "cw_30": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [0]
            }
        },
        "cw_31": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "cc": {
                "ref_signal": "single-qubit-mw",
                "static_codeword_override": [1]
            }
        },
        "fl_cw_00": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [0]
            }
        },
        "fl_cw_01": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [1]
            }
        },
        "fl_cw_02": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [2]
            }
        },
        "fl_cw_03": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [3]
            }
        },
        "fl_cw_04": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [4]
            }
        },
        "fl_cw_05": {
```

(continues on next page)

(continued from previous page)

```
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [5]
            }
        },
        "fl_cw_06": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [6]
            }
        },
        "fl_cw_07": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "two-qubit-flux",
                "static_codeword_override": [7]
            }
        },
        "sf_cz_ne q2": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [1]
            }
        },
        "sf_cz_ne q3": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [1]
            }
        },
        "sf_cz_se q0": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [2]
            }
        },
        "sf_cz_se q2": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [2]
            }
        },
        "sf_cz_sw q1": {
            "prototype": ["U:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [3]
            }
        },
        "sf_cz_sw q2": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [3]
            }
        },
        "sf_cz_nw q2": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [4]
            }
        },
        "sf_cz_nw q4": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [4]
            }
        },
        "sf_park q3": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [5]
            }
        },
        "sf_park q4": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [5]
            }
        },
        "sf_sp_park": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [5]
            }
        },
        "sf_square": {
            "prototype": ["U:qubit"],
            "duration": 80,
```

(continues on next page)

(continued from previous page)

```
            "cc": {
                "ref_signal": "single-qubit-flux",
                "static_codeword_override": [6]
            }
        }
    }
}
```

1.10.2 CC-light

- Pass/resource/C++ namespace: `arch.cc_light`
- Acceptable `"eqasm_compiler"` values: `"cc_light"` or `"cc_light_compiler"`

This architecture represents what remains of the CC-light backend from past versions of OpenQL. The CC-light is being/has been phased out in our labs, thus code generation was no longer necessary, and has thus been removed entirely. However, most test cases and most compiler-development-related activities still rely on parts of the CC-light architecture, hence the architecture itself remains. It is also useful as an example for what a basic architecture should look like within OpenQL's codebase.

For extensive documentation on what the architecture was and how it worked, please refer to the documentation pages of older versions of OpenQL. What still remains in OpenQL now is almost entirely based on configuring reusable generalizations of CC-light specific code; therefore, its function can largely be derived from the default configuration file and the documentation that documents the relevant sections of it.

Default pass list

For the current/default global option values and the default variant (`default`), the following backend passes are used by default.

```
- rcscheduler: sch.ListSchedule
   |- resource_constraints: yes

- lastqasmwriter: io.cqasm.Report
   |- output_prefix: test_output/%N
   |- output_suffix: _last.qasm
```

`default` variant

This is the default CC-light configuration, based on what used to be `tests/hardware_config_cc_light.json`, which in turn is a simplified version of the surface-7 configuration (the instruction durations are comparatively short and uniform).

When no platform configuration file is specified, the following default file is used instead.

```
{
    "eqasm_compiler": "cc_light",

    "hardware_settings": {
        "qubit_number": 7,
        "cycle_time": 20
    },

```

(continues on next page)

(continued from previous page)

```
    "topology": {
        "form": "xy",
        "qubits": [
            { "id": 0, "x": 1, "y": 2 },
            { "id": 1, "x": 3, "y": 2 },
            { "id": 2, "x": 0, "y": 1 },
            { "id": 3, "x": 2, "y": 1 },
            { "id": 4, "x": 4, "y": 1 },
            { "id": 5, "x": 1, "y": 0 },
            { "id": 6, "x": 3, "y": 0 }
        ],
        "edges": [
            { "id": 0, "src": 2, "dst": 0 },
            { "id": 1, "src": 0, "dst": 3 },
            { "id": 2, "src": 3, "dst": 1 },
            { "id": 3, "src": 1, "dst": 4 },
            { "id": 4, "src": 2, "dst": 5 },
            { "id": 5, "src": 5, "dst": 3 },
            { "id": 6, "src": 3, "dst": 6 },
            { "id": 7, "src": 6, "dst": 4 },
            { "id": 8, "src": 0, "dst": 2 },
            { "id": 9, "src": 3, "dst": 0 },
            { "id": 10, "src": 1, "dst": 3 },
            { "id": 11, "src": 4, "dst": 1 },
            { "id": 12, "src": 5, "dst": 2 },
            { "id": 13, "src": 3, "dst": 5 },
            { "id": 14, "src": 6, "dst": 3 },
            { "id": 15, "src": 4, "dst": 6 }
        ]
    },

    "resources": {
        "qubits": {},
        "qwgs": {
            "connection_map": {
                "0": [0, 1],
                "1": [2, 3, 4],
                "2": [5, 6]
            }
        },
        "meas_units": {
            "connection_map": {
                "0": [0, 2, 3, 5, 6],
                "1": [1, 4]
            }
        },
        "edges": {
            "connection_map": {
                "0": [2, 10],
                "1": [3, 11],
                "2": [0, 8],
                "3": [1, 9],
                "4": [6, 14],
                "5": [7, 15],
                "6": [4, 12],
                "7": [5, 13],
                "8": [2, 10],
```

(continues on next page)

(continued from previous page)

```
                "9": [3, 11],
                "10": [0, 8],
                "11": [1, 9],
                "12": [6, 14],
                "13": [7, 15],
                "14": [4, 12],
                "15": [5, 13]
            }
        },
        "detuned_qubits": {
            "connection_map": {
                "0": [3],
                "1": [2],
                "2": [4],
                "3": [3],
                "4": [],
                "5": [6],
                "6": [5],
                "7": [],
                "8": [3],
                "9": [2],
                "10": [4],
                "11": [3],
                "12": [],
                "13": [6],
                "14": [5],
                "15": []
            }
        }
    },

    "instructions": {
        "prepx": {
            "prototype": ["W:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "prepx"
        },
        "prepz": {
            "prototype": ["W:qubit"],
            "duration": 40,
            "type": "none",
            "cc_light_instr": "prepz"
        },
        "cprepz": {
            "prototype": ["W:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "cprepz"
        },
        "measz": {
            "prototype": ["M:qubit"],
            "duration": 40,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "measz ": {
```

(continues on next page)

(continued from previous page)

```
            "prototype": ["U:qubit", "W:bit"],
            "duration": 40,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "measure": {
            "prototype": ["M:qubit"],
            "duration": 40,
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measure op(0), bit(op(0))"
            }
        },
        "measure ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 40,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "i": {
            "prototype": ["U:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "i"
        },
        "x": {
            "prototype": ["X:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "x"
        },
        "y": {
            "prototype": ["Y:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "y"
        },
        "z": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "z"
        },
        "h": {
            "prototype": ["U:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "h"
        },
        "s": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "s"
        },
```

(continues on next page)

(continued from previous page)

```
        "sdag": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "sdag"
        },
        "rx90": {
            "prototype": ["X:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "x90"
        },
        "xm90": {
            "prototype": ["X:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "xm90"
        },
        "ry90": {
            "prototype": ["Y:qubit"],
            "duration": 40,
            "qubits": ["q0"],
            "type": "mw",
            "cc_light_instr": "y90"
        },
        "ym90": {
            "prototype": ["Y:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "ym90"
        },
        "t": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "t"
        },
        "tdag": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "tdag"
        },
        "x45": {
            "prototype": ["X:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "x45"
        },
        "xm45": {
            "prototype": ["X:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "xm45"
        },
        "ry180": {
            "prototype": ["Y:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "ry180"
        },
        "cnot": {
            "prototype": ["Z:qubit", "X:qubit"],
            "duration": 80,
            "type": "flux",
            "cc_light_instr": "cnot"
        },
        "sqf": {
            "prototype": ["U:qubit"],
            "duration": 80,
            "type": "flux",
            "cc_light_instr": "sqf"
        },
        "cz": {
            "prototype": ["Z:qubit", "Z:qubit"],
            "duration": 80,
            "type": "flux",
            "cc_light_instr": "cz"
        }
    },

    "gate_decomposition": {
        "rx180 %0" : ["x %0"],
        "cnot %0,%1" : ["ym90 %1","cz %0,%1","ry90 %1"]
    }
}
```

`s5` variant

This variant models the surface-5 chip. It is primarily intended as a baseline configuration for testing mapping and scheduling, as the eQASM backend is no longer part of OpenQL.

When no platform configuration file is specified, the following default file is used instead.

```
{
    "eqasm_compiler": "cc_light",

    "hardware_settings": {
        "qubit_number": 5,
        "cycle_time": 20
    },

    "topology": {
        "form": "xy",
        "qubits": [
            { "id": 0,  "x": 1, "y": 2 },
            { "id": 1,  "x": 3, "y": 2 },
            { "id": 2,  "x": 2, "y": 1 },
            { "id": 3,  "x": 1, "y": 0 },
            { "id": 4,  "x": 3, "y": 0 }
        ],
        "edges": [
```

(continues on next page)

(continued from previous page)

```
            { "id": 0,  "src": 0, "dst": 2 },
            { "id": 1,  "src": 2, "dst": 1 },
            { "id": 2,  "src": 3, "dst": 2 },
            { "id": 3,  "src": 2, "dst": 4 },
            { "id": 4,  "src": 2, "dst": 0 },
            { "id": 5,  "src": 1, "dst": 2 },
            { "id": 6,  "src": 2, "dst": 3 },
            { "id": 7,  "src": 4, "dst": 2 }
        ]
    },

    "resources": {
        "qubits": {},
        "meas_units": {
            "connection_map": {
                "0": [0, 2, 3, 4],
                "1": [1]
            }
        },
        "detuned_qubits": {
            "connection_map": {
                "0": [],
                "1": [],
                "2": [4],
                "3": [3],
                "4": [],
                "5": [],
                "6": [4],
                "7": [3]
            }
        }
    },

    "instructions": {
        "prepx": {
            "prototype": ["W:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "prepx"
        },
        "prepz": {
            "prototype": ["W:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "prepz"
        },
        "measx_keep": {
            "prototype": ["M:qubit"],
            "duration": 340,
            "type": "readout",
            "cc_light_instr": "measx",
            "decomposition": {
                "name": "desugar",
                "into": "measx_keep op(0), bit(op(0))"
            }
        },
        "measx_keep ": {
```

(continues on next page)

(continued from previous page)

```
            "prototype": ["U:qubit", "W:bit"],
            "duration": 340,
            "type": "readout",
            "cc_light_instr": "measx"
        },
        "measz_keep": {
            "prototype": ["M:qubit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measz_keep op(0), bit(op(0))"
            }
        },
        "measz_keep ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "measure": {
            "prototype": ["M:qubit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measure op(0), bit(op(0))"
            }
        },
        "measure ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "i": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "i"
        },
        "x": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x"
        },
        "y": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y"
        },
        "z": {
            "prototype": ["Z:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "z"
        },
        "rx": {
            "prototype": ["X:qubit", "L:real"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "rx"
        },
        "ry": {
            "prototype": ["Y:qubit", "L:real"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ry"
        },
        "rz": {
            "prototype": ["Z:qubit", "L:real"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "rz"
        },
        "h": {
            "prototype": ["U:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "h"
        },
        "s": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "s"
        },
        "sdag": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "sdag"
        },
        "x90": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x90"
        },
        "xm90": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "xm90"
        },
        "y90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y90"
```

(continues on next page)

(continued from previous page)

```
        },
        "ym90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ym90"
        },
        "t": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "t"
        },
        "tdag": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "tdag"
        },
        "x45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x45"
        },
        "xm45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "xm45"
        },
        "y45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y45"
        },
        "ym45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ym45"
        },
        "cz": {
            "prototype": ["Z:qubit", "Z:qubit"],
            "duration": 60,
            "type": "flux",
            "cc_light_instr": "cz"
        },
        "cnot_keep": {
            "prototype": ["Z:qubit", "X:qubit"],
            "duration": 100,
            "type": "flux",
            "cc_light_instr": "cnot"
        },
        "swap_keep": {
            "prototype": ["U:qubit", "U:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 260,
            "type": "flux",
            "cc_light_instr": "swap"
        },
        "move_keep": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 180,
            "type": "flux",
            "cc_light_instr": "move"
        }
    },

    "gate_decomposition": {
        "cnot %0,%1": ["ym90 %1","cz %0,%1","y90 %1"],
        "swap %0,%1": ["ym90 %1","cz %0,%1","y90 %1", "ym90 %0","cz %1,%0","y90 %0",
↪"ym90 %1","cz %0,%1","y90 %1"],
        "move %0,%1": ["ym90 %0","cz %1,%0","y90 %0", "ym90 %1","cz %0,%1","y90 %1"],

        "toffoli %0,%1,%2" : ["y90 %0", "xm45 %0", "y %0", "y90 %1", "xm45 %1", "ym90
↪%1", "x %2", "ym90 %2", "cz %2,%0", "y %0", "x45 %0", "y %0", "ym90 %2","cz %1,%2",
↪"y %2","cz %1,%0","y %0", "x45 %2", "y %2", "xm45 %0", "y %0","cz %1,%2","y90 %2",
↪"cz %2,%0","y %0", "x45 %0", "y %0", "y90 %2", "xm45 %2", "ym90 %2", "cz %1,%0",
↪"y90 %0", "x %2", "ym90 %2"],

        "rx180 %0" : ["x %0"],
        "ry180 %0" : ["y %0"],
        "rx90 %0" : ["x90 %0"],
        "ry90 %0" : ["y90 %0"],
        "mrx90 %0" : ["xm90 %0"],
        "mry90 %0" : ["ym90 %0"],
        "rx45 %0" : ["x45 %0"],
        "ry45 %0" : ["y45 %0"],
        "mrx45 %0" : ["xm45 %0"],
        "mry45 %0" : ["ym45 %0"],
        "measx %0" : ["h %0", "measure %0"],
        "measz %0" : ["measure %0"],

        "swap_real %0,%1": ["cnot %0,%1", "cnot %1,%0", "cnot %0,%1"],
        "move_real %0,%1": ["cnot %1,%0", "cnot %0,%1"],
        "z_real %0" : ["x %0","y %0"],
        "h_real %0" : ["x %0", "ym90 %0"],
        "t_real %0" : ["y90 %0", "x45 %0", "ym90 %0"],
        "tdag_real %0" : ["y90 %0", "xm45 %0", "ym90 %0"],
        "s_real %0" : ["y90 %0", "x90 %0", "ym90 %0"],
        "sdag_real %0" : ["y90 %0", "xm90 %0", "ym90 %0"],

        "cnot_prim %0,%1": ["ym90 %1","cz %0,%1","y90 %1"],
        "swap_prim %0,%1": ["ym90 %1","cz %0,%1","y90 %1", "ym90 %0","cz %1,%0","y90
↪%0", "ym90 %1","cz %0,%1","y90 %1"],
        "move_prim %0,%1": ["ym90 %0","cz %1,%0","y90 %0", "ym90 %1","cz %0,%1","y90
↪%1"],
        "z_prim %0" : ["x %0","y %0"],
        "h_prim %0" : ["x %0", "ym90 %0"],
        "t_prim %0" : ["y90 %0", "x45 %0", "ym90 %0"],
        "tdag_prim %0" : ["y90 %0", "xm45 %0", "ym90 %0"],
        "s_prim %0" : ["y90 %0", "x90 %0", "ym90 %0"],
        "sdag_prim %0" : ["y90 %0", "xm90 %0", "ym90 %0"]
```

(continues on next page)

(continued from previous page)

```
    }
}
```

s7 variant

This variant models the surface-7 chip. It is primarily intended as a baseline configuration for testing mapping and scheduling, as the eQASM backend is no longer part of OpenQL.

When no platform configuration file is specified, the following default file is used instead.

```
{
    "eqasm_compiler": "cc_light",

    "hardware_settings": {
        "qubit_number": 7,
        "cycle_time": 20
    },

    "topology": {
        "form": "xy",
        "qubits": [
            { "id": 0, "x": 1, "y": 2 },
            { "id": 1, "x": 3, "y": 2 },
            { "id": 2, "x": 0, "y": 1 },
            { "id": 3, "x": 2, "y": 1 },
            { "id": 4, "x": 4, "y": 1 },
            { "id": 5, "x": 1, "y": 0 },
            { "id": 6, "x": 3, "y": 0 }
        ],
        "edges": [
            { "id": 0, "src": 2, "dst": 0 },
            { "id": 1, "src": 0, "dst": 3 },
            { "id": 2, "src": 3, "dst": 1 },
            { "id": 3, "src": 1, "dst": 4 },
            { "id": 4, "src": 2, "dst": 5 },
            { "id": 5, "src": 5, "dst": 3 },
            { "id": 6, "src": 3, "dst": 6 },
            { "id": 7, "src": 6, "dst": 4 },
            { "id": 8, "src": 0, "dst": 2 },
            { "id": 9, "src": 3, "dst": 0 },
            { "id": 10, "src": 1, "dst": 3 },
            { "id": 11, "src": 4, "dst": 1 },
            { "id": 12, "src": 5, "dst": 2 },
            { "id": 13, "src": 3, "dst": 5 },
            { "id": 14, "src": 6, "dst": 3 },
            { "id": 15, "src": 4, "dst": 6 }
        ]
    },

    "resources": {
        "qubits": {},
        "qwgs": {
            "connection_map": {
                "0": [0, 1],
                "1": [2, 3, 4],
```

(continues on next page)

(continued from previous page)

```
                "2": [5, 6]
            }
        },
        "meas_units": {
            "connection_map": {
                "0": [0, 2, 3, 5, 6],
                "1": [1, 4]
            }
        },
        "edges": {
            "connection_map": {
                "0": [2, 10],
                "1": [3, 11],
                "2": [0, 8],
                "3": [1, 9],
                "4": [6, 14],
                "5": [7, 15],
                "6": [4, 12],
                "7": [5, 13],
                "8": [2, 10],
                "9": [3, 11],
                "10": [0, 8],
                "11": [1, 9],
                "12": [6, 14],
                "13": [7, 15],
                "14": [4, 12],
                "15": [5, 13]
            }
        },
        "detuned_qubits": {
            "connection_map": {
                "0": [3],
                "1": [2],
                "2": [4],
                "3": [3],
                "4": [],
                "5": [6],
                "6": [5],
                "7": [],
                "8": [3],
                "9": [2],
                "10": [4],
                "11": [3],
                "12": [],
                "13": [6],
                "14": [5],
                "15": []
            }
        }
    },

    "instructions": {
        "prepx": {
            "prototype": ["W:qubit"],
            "duration": 640,
            "type": "mw",
            "cc_light_instr": "prepx"
```

(continues on next page)

(continued from previous page)

```
        },
        "prepz": {
            "prototype": ["W:qubit"],
            "duration": 620,
            "type": "mw",
            "cc_light_instr": "prepz"
        },
        "measx_keep": {
            "prototype": ["M:qubit"],
            "duration": 340,
            "type": "readout",
            "cc_light_instr": "measx",
            "decomposition": {
                "name": "desugar",
                "into": "measx_keep op(0), bit(op(0))"
            }
        },
        "measx_keep ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 340,
            "type": "readout",
            "cc_light_instr": "measx"
        },
        "measz_keep": {
            "prototype": ["M:qubit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measz_keep op(0), bit(op(0))"
            }
        },
        "measz_keep ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "measure": {
            "prototype": ["M:qubit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measure op(0), bit(op(0))"
            }
        },
        "measure ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "i": {
            "prototype": ["W:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "i"
        },
        "x": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x"
        },
        "y": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y"
        },
        "z": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "z"
        },
        "rx": {
            "prototype": ["X:qubit", "L:real"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "rx"
        },
        "ry": {
            "prototype": ["Y:qubit", "L:real"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "ry"
        },
        "rz": {
            "prototype": ["Z:qubit", "L:real"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "rz"
        },
        "h": {
            "prototype": ["W:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "h"
        },
        "s": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "s"
        },
        "sdag": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "sdag"
```

(continues on next page)

(continued from previous page)

```
        },
        "x90": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x90"
        },
        "xm90": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "xm90"
        },
        "y90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y90"
        },
        "ym90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ym90"
        },
        "t": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "t"
        },
        "tdag": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "tdag"
        },
        "x45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x45"
        },
        "xm45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "xm45"
        },
        "y45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y45"
        },
        "ym45": {
            "prototype": ["Y:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ym45"
        },
        "cz": {
            "prototype": ["Z:qubit", "Z:qubit"],
            "duration": 40,
            "type": "flux",
            "cc_light_instr": "cz"
        },
        "cnot": {
            "prototype": ["Z:qubit", "X:qubit"],
            "duration": 80,
            "type": "flux",
            "cc_light_instr": "cnot"
        },
        "toffoli" : {
            "prototype": ["Z:qubit", "Z:qubit", "X:qubit"],
            "duration": 80,
            "type": "multi",
            "cc_light_instr": "toffoli"
        },
        "swap": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 200,
            "type": "flux",
            "cc_light_instr": "swap"
        },
        "move": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 140,
            "type": "flux",
            "cc_light_instr": "move"
        }
    },

    "gate_decomposition": {
        "toffoli_decomp %0,%1,%2" : ["y90 %0", "xm45 %0", "y %0", "y90 %1", "xm45 %1",
↪ "ym90 %1", "x %2", "ym90 %2", "cz %2,%0", "y %0", "x45 %0", "y %0", "ym90 %2","cz
↪%1,%2","y %2","cz %1,%0","y %0", "x45 %2", "y %2", "xm45 %0", "y %0","cz %1,%2",
↪"y90 %2","cz %2,%0","y %0", "x45 %0", "y %0", "y90 %2", "xm45 %2", "ym90 %2", "cz
↪%1,%0","y90 %0", "x %2", "ym90 %2"],

        "rx180 %0" : ["x %0"],
        "ry180 %0" : ["y %0"],
        "rx90 %0" : ["x90 %0"],
        "ry90 %0" : ["y90 %0"],
        "mrx90 %0" : ["xm90 %0"],
        "mry90 %0" : ["ym90 %0"],
        "rx45 %0" : ["x45 %0"],
        "ry45 %0" : ["y45 %0"],
        "mrx45 %0" : ["xm45 %0"],
        "mry45 %0" : ["ym45 %0"],
        "measx %0" : ["h %0", "measure %0"],
        "measz %0" : ["measure %0"],

        "swap_real %0,%1": ["cnot %0,%1", "cnot %1,%0", "cnot %0,%1"],
```

(continues on next page)

(continued from previous page)

```
        "move_real %0,%1": ["cnot %1,%0", "cnot %0,%1"],
        "z_real %0" : ["x %0","y %0"],
        "h_real %0" : ["x %0", "ym90 %0"],
        "t_real %0" : ["y90 %0", "x45 %0", "ym90 %0"],
        "tdag_real %0" : ["y90 %0", "xm45 %0", "ym90 %0"],
        "s_real %0" : ["y90 %0", "x90 %0", "ym90 %0"],
        "sdag_real %0" : ["y90 %0", "xm90 %0", "ym90 %0"],

        "cnot_prim %0,%1": ["ym90 %1","cz %0,%1","y90 %1"],
        "swap_prim %0,%1": ["ym90 %1","cz %0,%1","y90 %1", "ym90 %0","cz %1,%0","y90
↪%0", "ym90 %1","cz %0,%1","y90 %1"],
        "move_prim %0,%1": ["ym90 %0","cz %1,%0","y90 %0", "ym90 %1","cz %0,%1","y90
↪%1"],
        "z_prim %0" : ["x %0","y %0"],
        "h_prim %0" : ["x %0", "ym90 %0"],
        "t_prim %0" : ["y90 %0", "x45 %0", "ym90 %0"],
        "tdag_prim %0" : ["y90 %0", "xm45 %0", "ym90 %0"],
        "s_prim %0" : ["y90 %0", "x90 %0", "ym90 %0"],
        "sdag_prim %0" : ["y90 %0", "xm90 %0", "ym90 %0"]
    }
}
```

s17 variant

This variant models the surface-17 chip. It is primarily intended as a baseline configuration for testing mapping and scheduling, as the eQASM backend is no longer part of OpenQL.

When no platform configuration file is specified, the following default file is used instead.

```
{
    "eqasm_compiler": "cc_light",

    "hardware_settings": {
        "qubit_number": 17,
        "cycle_time": 20
    },

    "topology": {
        "form": "xy",
        "qubits": [
            { "id": 0, "x": 4, "y": 6 },
            { "id": 1, "x": 1, "y": 5 },
            { "id": 2, "x": 3, "y": 5 },
            { "id": 3, "x": 5, "y": 5 },
            { "id": 4, "x": 0, "y": 4 },
            { "id": 5, "x": 2, "y": 4 },
            { "id": 6, "x": 4, "y": 4 },
            { "id": 7, "x": 1, "y": 3 },
            { "id": 8, "x": 3, "y": 3 },
            { "id": 9, "x": 5, "y": 3 },
            { "id": 10, "x": 2, "y": 2 },
            { "id": 11, "x": 4, "y": 2 },
            { "id": 12, "x": 6, "y": 2 },
            { "id": 13, "x": 1, "y": 1 },
            { "id": 14, "x": 3, "y": 1 },
```

(continues on next page)

(continued from previous page)

```
            { "id": 15, "x": 5, "y": 1 },
            { "id": 16, "x": 2, "y": 0 }
        ],
        "edges": [
            { "id": 0, "src": 2, "dst": 0 },
            { "id": 1, "src": 0, "dst": 3 },
            { "id": 2, "src": 4, "dst": 1 },
            { "id": 3, "src": 1, "dst": 5 },
            { "id": 4, "src": 5, "dst": 2 },
            { "id": 5, "src": 2, "dst": 6 },
            { "id": 6, "src": 6, "dst": 3 },
            { "id": 7, "src": 4, "dst": 7 },
            { "id": 8, "src": 7, "dst": 5 },
            { "id": 9, "src": 5, "dst": 8 },
            { "id": 10, "src": 8, "dst": 6 },
            { "id": 11, "src": 6, "dst": 9 },
            { "id": 12, "src": 7, "dst": 10 },
            { "id": 13, "src": 10, "dst": 8 },
            { "id": 14, "src": 8, "dst": 11 },
            { "id": 15, "src": 11, "dst": 9 },
            { "id": 16, "src": 9, "dst": 12 },
            { "id": 17, "src": 13, "dst": 10 },
            { "id": 18, "src": 10, "dst": 14 },
            { "id": 19, "src": 14, "dst": 11 },
            { "id": 20, "src": 11, "dst": 15 },
            { "id": 21, "src": 15, "dst": 12 },
            { "id": 22, "src": 13, "dst": 16 },
            { "id": 23, "src": 16, "dst": 14 },

            { "id": 24, "src": 0, "dst": 2 },
            { "id": 25, "src": 3, "dst": 0 },
            { "id": 26, "src": 1, "dst": 4 },
            { "id": 27, "src": 5, "dst": 1 },
            { "id": 28, "src": 2, "dst": 5 },
            { "id": 29, "src": 6, "dst": 2 },
            { "id": 30, "src": 3, "dst": 6 },
            { "id": 31, "src": 7, "dst": 4 },
            { "id": 32, "src": 5, "dst": 7 },
            { "id": 33, "src": 8, "dst": 5 },
            { "id": 34, "src": 6, "dst": 8 },
            { "id": 35, "src": 9, "dst": 6 },
            { "id": 36, "src": 10, "dst": 7 },
            { "id": 37, "src": 8, "dst": 10 },
            { "id": 38, "src": 11, "dst": 8 },
            { "id": 39, "src": 9, "dst": 11 },
            { "id": 40, "src": 12, "dst": 9 },
            { "id": 41, "src": 10, "dst": 13 },
            { "id": 42, "src": 14, "dst": 10 },
            { "id": 43, "src": 11, "dst": 14 },
            { "id": 44, "src": 15, "dst": 11 },
            { "id": 45, "src": 12, "dst": 15 },
            { "id": 46, "src": 16, "dst": 13 },
            { "id": 47, "src": 14, "dst": 16 }
        ]
    },

    "resources": {
```

(continues on next page)

(continued from previous page)

```
        "qubits": {},
        "qwgs": {
            "connection_map": {
                "0" : [1, 2, 3, 13, 14, 15],
                "1" : [7, 8, 9],
                "2" : [0, 5, 11, 16, 4, 6, 10, 12]
            }
        },
        "meas_units": {
            "connection_map": {
                "0" : [13,16],
                "1" : [1, 4, 5, 7, 8, 10, 11, 14, 15],
                "2" : [0, 2, 3, 6, 9, 12]
            }
        },
        "edges": {
            "connection_map": {
                "0": [3, 27, 6, 30],
                "1": [5, 29, 4, 28],
                "2": [4, 28],
                "3": [0, 24, 5, 29],
                "4": [2, 26, 1, 25, 6, 30],
                "5": [3, 27, 1, 25],
                "6": [4, 28, 0, 24],
                "7": [9, 23, 13, 37],
                "8": [10, 34, 13, 37, 14, 28],
                "9": [7, 31, 12, 36, 11, 35, 15, 39],
                "10": [8, 32, 12, 36, 15, 39],
                "11": [9, 33, 13, 37, 14, 38],
                "12": [9, 33, 10, 34, 14, 38],
                "13": [8, 32, 11, 35, 15, 39],
                "14": [8, 32, 12, 36, 11, 35, 16, 40],
                "15": [9, 33, 13, 37, 10, 34],
                "16": [10, 34, 14, 38],
                "17": [19, 43],
                "18": [22, 46, 20, 44],
                "19": [17, 41, 22, 46, 21, 45],
                "20": [18, 42, 23, 47],
                "21": [19, 43],
                "22": [18, 42, 19, 43],
                "23": [17, 41, 20, 44],

                "24": [3, 27, 6, 30],
                "25": [5, 29, 4, 28],
                "26": [4, 28],
                "27": [0, 24, 5, 29],
                "28": [2, 26, 1, 25, 6, 30],
                "29": [3, 27, 1, 25],
                "30": [4, 28, 0, 24],
                "31": [9, 23, 13, 37],
                "32": [10, 34, 13, 37, 14, 28],
                "33": [7, 31, 12, 36, 11, 35, 15, 39],
                "34": [8, 32, 12, 36, 15, 39],
                "35": [9, 33, 13, 37, 14, 38],
                "36": [9, 33, 10, 34, 14, 38],
                "37": [8, 32, 11, 35, 15, 39],
                "38": [8, 32, 12, 36, 11, 35, 16, 40],
```

(continues on next page)

(continued from previous page)

```
                "39": [9, 33, 13, 37, 10, 34],
                "40": [10, 34, 14, 38],
                "41": [19, 43],
                "42": [22, 46, 20, 44],
                "43": [17, 41, 22, 46, 21, 45],
                "44": [18, 42, 23, 47],
                "45": [19, 43],
                "46": [18, 42, 19, 43],
                "47": [17, 41, 20, 44]
            }
        },
        "detuned_qubits": {
            "connection_map": {
                "0": [5, 6],
                "1": [6],
                "2": [5],
                "3": [4],
                "4": [0, 6],
                "5": [5, 0],
                "6": [0],
                "7": [],
                "8": [8],
                "9": [7],
                "10": [9],
                "11": [8],
                "12": [8],
                "13": [7],
                "14": [9],
                "15": [8],
                "16": [],
                "17": [16],
                "18": [16, 11],
                "19": [10, 16],
                "20": [12],
                "21": [11],
                "22": [10],
                "23": [10, 11],

                "24": [5, 6],
                "25": [6],
                "26": [5],
                "27": [4],
                "28": [0, 6],
                "29": [5, 0],
                "30": [0],
                "31": [],
                "32": [8],
                "33": [7],
                "34": [9],
                "35": [8],
                "36": [8],
                "37": [7],
                "38": [9],
                "39": [8],
                "40": [],
                "41": [16],
                "42": [16, 11],
```

(continues on next page)

(continued from previous page)

```
                "43": [10, 16],
                "44": [12],
                "45": [11],
                "46": [10],
                "47": [10, 11]
            }
        }
    },

    "instructions": {
        "prepx": {
            "prototype": ["W:qubit"],
            "duration": 640,
            "type": "mw",
            "cc_light_instr": "prepx"
        },
        "prepz": {
            "prototype": ["W:qubit"],
            "duration": 620,
            "type": "mw",
            "cc_light_instr": "prepz"
        },
        "measx_keep": {
            "prototype": ["M:qubit"],
            "duration": 340,
            "type": "readout",
            "cc_light_instr": "measx",
            "decomposition": {
                "name": "desugar",
                "into": "measx_keep op(0), bit(op(0))"
            }
        },
        "measx_keep ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 340,
            "type": "readout",
            "cc_light_instr": "measx"
        },
        "measz_keep": {
            "prototype": ["M:qubit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measz_keep op(0), bit(op(0))"
            }
        },
        "measz_keep ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "measure": {
            "prototype": ["M:qubit"],
            "duration": 300,
```

(continues on next page)

(continued from previous page)

```
            "type": "readout",
            "cc_light_instr": "measz",
            "decomposition": {
                "name": "desugar",
                "into": "measure op(0), bit(op(0))"
            }
        },
        "measure ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300,
            "type": "readout",
            "cc_light_instr": "measz"
        },
        "i": {
            "prototype": ["U:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "i"
        },
        "x": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x"
        },
        "y": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y"
        },
        "z": {
            "prototype": ["Z:qubit"],
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "z"
        },
        "rx": {
            "prototype": ["X:qubit", "L:real"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "rx"
        },
        "ry": {
            "prototype": ["Y:qubit", "L:real"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "ry"
        },
        "rz": {
            "prototype": ["Z:qubit", "L:real"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "rz"
        },
        "h": {
            "prototype": ["U:qubit"],
```

(continues on next page)

(continued from previous page)

```
            "duration": 40,
            "type": "mw",
            "cc_light_instr": "h"
        },
        "s": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "s"
        },
        "sdag": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "sdag"
        },
        "x90": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x90"
        },
        "xm90": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "xm90"
        },
        "y90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y90"
        },
        "ym90": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ym90"
        },
        "t": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "t"
        },
        "tdag": {
            "prototype": ["Z:qubit"],
            "duration": 60,
            "type": "mw",
            "cc_light_instr": "tdag"
        },
        "x45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "x45"
```

(continues on next page)

(continued from previous page)

```
        },
        "xm45": {
            "prototype": ["X:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "xm45"
        },
        "y45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "y45"
        },
        "ym45": {
            "prototype": ["Y:qubit"],
            "duration": 20,
            "type": "mw",
            "cc_light_instr": "ym45"
        },
        "cz": {
            "prototype": ["Z:qubit", "Z:qubit"],
            "duration": 40,
            "type": "flux",
            "cc_light_instr": "cz"
        },
        "cnot": {
            "prototype": ["Z:qubit", "X:qubit"],
            "duration": 80,
            "type": "flux",
            "cc_light_instr": "cnot"
        },
        "swap": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 200,
            "type": "flux",
            "cc_light_instr": "swap"
        },
        "move": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 140,
            "type": "flux",
            "cc_light_instr": "move"
        }
    },

    "gate_decomposition": {
        "toffoli %0,%1,%2" : ["y90 %0", "xm45 %0", "y %0", "y90 %1", "xm45 %1", "ym90
↪%1", "x %2", "ym90 %2", "cz %2,%0", "y %0", "x45 %0", "y %0", "ym90 %2","cz %1,%2",
↪"y %2","cz %1,%0","y %0", "x45 %2", "y %2", "xm45 %0", "y %0","cz %1,%2","y90 %2",
↪"cz %2,%0","y %0", "x45 %0", "y %0", "y90 %2", "xm45 %2", "ym90 %2", "cz %1,%0",
↪"y90 %0", "x %2", "ym90 %2"],

        "rx180 %0" : ["x %0"],
        "ry180 %0" : ["y %0"],
        "rx90 %0" : ["x90 %0"],
        "ry90 %0" : ["y90 %0"],
        "mrx90 %0" : ["xm90 %0"],
```

(continues on next page)

(continued from previous page)

```
        "mry90 %0" : ["ym90 %0"],
        "rx45 %0" : ["x45 %0"],
        "ry45 %0" : ["y45 %0"],
        "mrx45 %0" : ["xm45 %0"],
        "mry45 %0" : ["ym45 %0"],
        "measx %0" : ["h %0", "measure %0"],
        "measz %0" : ["measure %0"],

        "swap_real %0,%1": ["cnot %0,%1", "cnot %1,%0", "cnot %0,%1"],
        "move_real %0,%1": ["cnot %1,%0", "cnot %0,%1"],
        "z_real %0" : ["x %0","y %0"],
        "h_real %0" : ["x %0", "ym90 %0"],
        "t_real %0" : ["y90 %0", "x45 %0", "ym90 %0"],
        "tdag_real %0" : ["y90 %0", "xm45 %0", "ym90 %0"],
        "s_real %0" : ["y90 %0", "x90 %0", "ym90 %0"],
        "sdag_real %0" : ["y90 %0", "xm90 %0", "ym90 %0"],

        "cnot_prim %0,%1": ["ym90 %1","cz %0,%1","y90 %1"],
        "swap_prim %0,%1": ["ym90 %1","cz %0,%1","y90 %1", "ym90 %0","cz %1,%0","y90
↪%0", "ym90 %1","cz %0,%1","y90 %1"],
        "move_prim %0,%1": ["ym90 %0","cz %1,%0","y90 %0", "ym90 %1","cz %0,%1","y90
↪%1"],
        "z_prim %0" : ["x %0","y %0"],
        "h_prim %0" : ["x %0", "ym90 %0"],
        "t_prim %0" : ["y90 %0", "x45 %0", "ym90 %0"],
        "tdag_prim %0" : ["y90 %0", "xm45 %0", "ym90 %0"],
        "s_prim %0" : ["y90 %0", "x90 %0", "ym90 %0"],
        "sdag_prim %0" : ["y90 %0", "xm90 %0", "ym90 %0"]
    }
}
```

1.10.3 Diamond

- Pass/resource/C++ namespace: `arch.diamond`
- Acceptable `"eqasm_compiler"` values: `"diamond"`

This architecture is aimed towards computing with qubits made in color centers in diamond. It is part of the Fujitsu project and is a work in progress. The backend will, for now as it is in it's early stages, work as a translation tool from a high-level algorithm to our own defined microcode. It is mostly a proof of concept at this time.

Default pass list

For the current/default global option values and the default variant (`default`), the following backend passes are used by default.

```
- diamond_codegen: arch.diamond.gen.Microcode
   |- no options to dump
```

Default configuration file

When no platform configuration file is specified, the following default file is used instead.

```
{
  "eqasm_compiler": "diamond",

  "hardware_settings": {
    "qubit_number": 10,
    "cycle_time": 20
  },

  "gate_decomposition": {
    "toffoli %0, %1, %2" : ["h %2", "cnot %1 %2", "tdag %2", "cnot %0 %2", "t %2",
↪"cnot %1 %2", "tdag %2", "cnot %0 %2", "t %1", "t %2", "cnot %0 %1", "h %2", "t %0",
↪ "tdag %1", "cnot %0 %1"],
    "measure_z %0": ["measure %0"],
    "measure_y %0": ["mprep_y %0", "measure %0"],
    "measure_x %0": ["mprep_x %0", "measure %0"]
  },

  "instructions": {

    "prep_x": {
      "prototype": ["W:qubit"],
      "duration": 40,
      "diamond_type": "prepare"
    },

    "prep_y": {
      "prototype": ["W:qubit"],
      "duration": 40,
      "diamond_type": "prepare"
    },

    "prep_z": {
      "prototype": ["W:qubit"],
      "duration": 40,
      "diamond_type": "prepare"
    },

    "mprep_x": {
      "prototype": ["U:qubit"],
      "duration": 40,
      "diamond_type": "prepare"
    },

    "mprep_y": {
      "prototype": ["U:qubit"],
      "duration": 40,
      "diamond_type": "prepare"
    },

    "qnop": {
      "prototype": ["B:qubit"],
      "duration": 1
    },
```

(continues on next page)

(continued from previous page)

```
    "calculate_current": {
      "prototype": ["U:qubit"],
      "duration": 1,
      "diamond_type": "classical"
    },

    "calculate_voltage": {
      "prototype": ["U:qubit"],
      "barrier": true,
      "duration": 1,
      "diamond_type": "classical"
    },

    "i": {
      "prototype": ["U:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "h": {
      "prototype": ["U:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "x": {
      "prototype": ["X:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "y": {
      "prototype": ["Y:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "z": {
      "prototype": ["Z:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "initialize": {
      "prototype": ["W:qubit"],
      "duration": 40
    },

    "x90": {
      "prototype": ["X:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "y90": {
      "prototype": ["Y:qubit"],
      "duration": 20,
```

(continues on next page)

(continued from previous page)

```
      "diamond_type": "qgate"
    },

    "x180": {
      "prototype": ["X:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "y180": {
      "prototype": ["Y:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "mx90": {
      "prototype": ["X:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "my90": {
      "prototype": ["Y:qubit"],
      "duration": 20,
      "diamond_type": "qgate"
    },

    "rx": {
      "prototype": ["X:qubit", "L:real"],
      "duration": 40,
      "diamond_type": "rotation"
    },

    "ry": {
      "prototype": ["Y:qubit", "L:real"],
      "duration": 40,
      "diamond_type": "rotation"
    },

    "rz": {
      "prototype": ["Z:qubit", "L:real"],
      "duration": 40,
      "diamond_type": "rotation"
    },

    "cr": {
      "prototype": ["Z:qubit", "Z:qubit", "L:real"],
      "duration": 40,
      "diamond_type": "rotation"
    },

    "crk": {
      "prototype": ["Z:qubit", "Z:qubit", "L:int"],
      "duration": 40,
      "diamond_type": "rotation"
    },
```

(continues on next page)

(continued from previous page)

```
    "s": {
      "prototype": ["Z:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "sdag": {
      "prototype": ["Z:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "t": {
      "prototype": ["Z:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "tdag": {
      "prototype": ["Z:qubit"],
      "duration": 40,
      "diamond_type": "qgate"
    },

    "cnot": {
      "prototype": ["Z:qubit", "X:qubit"],
      "duration": 80,
      "diamond_type": "qgate2"
    },

    "pmx90": {
      "prototype": ["U:qubit", "U:qubit"],
      "duration": 80,
      "diamond_type": "qgate2"
    },

    "pmy90": {
      "prototype": ["U:qubit", "U:qubit"],
      "duration": 80,
      "diamond_type": "qgate2"
    },

    "pmx180": {
      "prototype": ["U:qubit", "U:qubit"],
      "duration": 80,
      "diamond_type": "qgate2"
    },

    "pmy180": {
      "prototype": ["U:qubit", "U:qubit"],
      "duration": 80,
      "diamond_type": "qgate2"
    },

    "cz": {
      "prototype": ["Z:qubit", "Z:qubit"],
      "duration": 80,
```

(continues on next page)

(continued from previous page)

```
      "diamond_type": "qgate2"
    },

    "measure": {
      "prototype": ["M:qubit"],
      "duration": 300,
      "decomposition": {
        "name": "desugar",
        "into": "measure op(0), bit(op(0))"
      }
    },

    "measure ": {
      "prototype": ["U:qubit", "W:bit"],
      "duration": 300
    },

    "display": {
      "prototype": [],
      "barrier": true,
      "duration": 20,
      "qubits": []
    },

    "display_binary": {
      "prototype": [],
      "barrier": true,
      "duration": 20,
      "qubits": []
    },

    "swap": {
      "prototype": ["U:qubit", "U:qubit"],
      "duration": 80,
      "diamond_type": "qgate2"
    },

    "memswap": {
      "prototype": ["U:qubit", "L:int"],
      "duration": 80
    },

    "qentangle": {
      "prototype": ["U:qubit", "L:int"],
      "duration": 60
    },

    "nventangle": {
      "prototype": ["U:qubit", "U:qubit"],
      "duration": 100
    },

    "sweep_bias": {
      "prototype": ["U:qubit", "L:int", "L:int", "L:int", "L:int", "L:int", "L:int"],
      "duration": 300
    },
```

(continues on next page)

(continued from previous page)

```
    "excite_mw": {
      "prototype": ["U:qubit", "L:int", "L:int", "L:int", "L:int", "L:int"],
      "duration": 20
    },

    "crc": {
      "prototype": ["U:qubit", "L:int", "L:int"],
      "duration": 20,
      "diamond_type": "initial_checks"
    },

    "mag_bias": {
      "prototype": ["U:qubit"],
      "duration": 300,
      "diamond_type": "initial_checks"
    },

    "rabi_check": {
      "prototype": ["U:qubit", "L:int", "L:int", "L:int"],
      "duration": 300,
      "diamond_type": "initial_checks"
    },

    "decouple": {
      "prototype": ["U:qubit"],
      "duration": 300,
      "diamond_type": "calibration"
    },

    "cal_measure": {
      "prototype": ["U:qubit"],
      "duration": 300,
      "diamond_type": "calibration"
    },

    "cal_pi": {
      "prototype": ["U:qubit"],
      "duration": 300,
      "diamond_type": "calibration"
    },

    "cal_halfpi": {
      "prototype": ["U:qubit"],
      "duration": 300,
      "diamond_type": "calibration"
    }
  }
}
```

1.10.4 None

- Pass/resource/C++ namespace: `arch.none`
- Acceptable `"eqasm_compiler"` values: `"none"`, `"qx"`, or `""`

This is just a dummy architecture that does not include any backend passes by default, does not provide shortcuts for any architecture-specific passes and resources, and does not do any platform-specific preprocessing on the platform configuration file. You can use it when you just want to try OpenQL out, or when your target is an architecture-agnostic simulator.

The default configuration file consists of relatively sane defaults for simulating the resulting cQASM output with the QX simulator.

Default pass list

For the current/default global option values, this architecture does not insert any backend passes.

Default configuration file

When no platform configuration file is specified, the following default file is used instead.

```
{
    "eqasm_compiler": "none",

    "hardware_settings": {
        "qubit_number": 10,
        "cycle_time": 20
    },

    "instructions": {

        "prep_x": {
            "prototype": ["W:qubit"],
            "duration": 40
        },

        "prep_y": {
            "prototype": ["W:qubit"],
            "duration": 40
        },

        "prep_z": {
            "prototype": ["W:qubit"],
            "duration": 40
        },

        "i": {
            "prototype": ["U:qubit"],
            "duration": 40
        },

        "h": {
            "prototype": ["U:qubit"],
            "duration": 40
        },
```

(continues on next page)

(continued from previous page)

```
        "x": {
            "prototype": ["X:qubit"],
            "duration": 40
        },

        "y": {
            "prototype": ["Y:qubit"],
            "duration": 40
        },

        "z": {
            "prototype": ["Z:qubit"],
            "duration": 40
        },

        "x90": {
            "prototype": ["X:qubit"],
            "duration": 40
        },

        "y90": {
            "prototype": ["Y:qubit"],
            "duration": 20
        },

        "x180": {
            "prototype": ["X:qubit"],
            "duration": 40
        },

        "y180": {
            "prototype": ["Y:qubit"],
            "duration": 40
        },

        "mx90": {
            "prototype": ["X:qubit"],
            "duration": 40
        },

        "my90": {
            "prototype": ["Y:qubit"],
            "duration": 20
        },

        "rx": {
            "prototype": ["X:qubit", "L:real"],
            "duration": 40
        },

        "ry": {
            "prototype": ["Y:qubit", "L:real"],
            "duration": 40
        },

        "rz": {
```

(continues on next page)

(continued from previous page)

```
            "prototype": ["Z:qubit", "L:real"],
            "duration": 40
        },

        "s": {
            "prototype": ["Z:qubit"],
            "duration": 40
        },

        "sdag": {
            "prototype": ["Z:qubit"],
            "duration": 40
        },

        "t": {
            "prototype": ["Z:qubit"],
            "duration": 40
        },

        "tdag": {
            "prototype": ["Z:qubit"],
            "duration": 40
        },

        "cnot": {
            "prototype": ["Z:qubit", "X:qubit"],
            "duration": 80
        },

        "toffoli": {
            "prototype": ["Z:qubit", "Z:qubit", "X:qubit"],
            "duration": 80
        },

        "cz": {
            "prototype": ["Z:qubit", "Z:qubit"],
            "duration": 80
        },

        "swap": {
            "prototype": ["U:qubit", "U:qubit"],
            "duration": 80
        },

        "measure": {
            "prototype": ["M:qubit"],
            "duration": 300,
            "decomposition": {
                "name": "desugar",
                "into": "measure op(0), bit(op(0))"
            }
        },

        "measure ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300
        },
```

(continues on next page)

(continued from previous page)

```
        "measure_x": {
            "prototype": ["M:qubit"],
            "duration": 300
        },

        "measure_x ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300
        },

        "measure_y": {
            "prototype": ["M:qubit"],
            "duration": 300
        },

        "measure_y ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300
        },

        "measure_z": {
            "prototype": ["M:qubit"],
            "duration": 300
        },

        "measure_z ": {
            "prototype": ["U:qubit", "W:bit"],
            "duration": 300
        },

        "display": {
            "prototype": [],
            "barrier": true,
            "duration": 20
        },

        "display_binary": {
            "prototype": [],
            "barrier": true,
            "duration": 20
        }
    }
}
```

1.11 Supported global options

This section lists all the global options currently supported by OpenQL.

Note: Most of these options exist only for backward compatibility, having been superseded by pass options. They will be used only when the pass list is automatically generated to mimic legacy behavior, or when compatibility mode is enabled in the compiler configuration file.

1.11.1 `log_level`

Must be one of `LOG_NOTHING`, `LOG_CRITICAL`, `LOG_ERROR`, `LOG_WARNING`, `LOG_INFO`, or `LOG_DEBUG`, default `LOG_NOTHING`. Log levels

1.11.2 `use_default_gates`

Must be `yes` or `no`, default `yes`. Use default gates or not. When set, a number of builtin gates become available as fallback for the gates defined in the platform configuration structure, including the special wait and barrier gates.

1.11.3 `decompose_toffoli`

Must be one of `no`, `NC`, or `AM`, default `no`. Controls the behavior of Kernel.toffoli(); either decompose immediately via the given substitution, or insert the Toffoli gate into the circuit as-is if `no` or unspecified.

1.11.4 `issue_skip_319`

Must be `yes` or `no`, default `no`. Issue skip instead of wait in bundles. TODO: document better, and actually fix skip vs. wait/barrier properly once and for all.

1.11.5 `unique_output`

Must be `yes` or `no`, default `no`. Uniquify the program name as used for constructing output filenames, such that compiling the same program multiple times yields a different name each time. When this option is set during the first construction of a program with a particular name, the program name is used as-is, and a <program>.unique file is generated in the output directory to track how many times a program with this name has been constructed. When a program with the same name is constructed again later, again with this option set, a numeric suffix will be automatically added to the program name, starting from 2. The generated suffix can be reset by simply removing the .unique file. Note that the uniquified name is only used when %N is used in the `output_prefix` common pass option.

1.11.6 `clifford_prescheduler`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether to run the Clifford optimizer before the prescheduler.

1.11.7 `prescheduler`

Must be `yes` or `no`, default `yes`. When no compiler configuration file is specified, this controls whether a basic ASAP/ALAP scheduler without resource constraints should be run before mapping.

1.11.8 `clifford_postscheduler`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether to run the Clifford optimizer after the prescheduler.

1.11.9 `clifford_premapper`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether to run the Clifford optimizer before the mapper.

1.11.10 `clifford_postmapper`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether to run the Clifford optimizer after the mapper.

1.11.11 `output_dir`

Must be any string, default `test_output`. When no compiler configuration file is specified, this controls the `output_prefix` option for all passes; it will be set to `<output_dir>/%N_%P`. Defaults to `test_output` for compatibility reasons. The directory will automatically be created if it does not already exist when the first output file is written.

1.11.12 `write_qasm_files`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this enables writing cQASM files before and after each default pass. When a compiler configuration file is specified, use the `debug` pass option common to all passes instead.

1.11.13 `write_report_files`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this enables writing statistics report files before and after each default pass. When a compiler configuration file is specified, use the `debug` pass option common to all passes instead.

1.11.14 `scheduler`

Must be one of `ASAP` or `ALAP`, default `ALAP`. When no compiler configuration file is specified, this controls whether ALAP or ASAP scheduling is to be used for the default-inserted scheduler passes. Both the pre-mapping and post-mapping schedulers are affected.

1.11.15 `scheduler_uniform`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether uniform scheduling should be done instead of ASAP/ALAP (i.e. the `scheduler` option will be ignored). Both the pre-mapping and post-mapping schedulers are affected. Setting this selects the old scheduler (`sch.Schedule`), because the new scheduler (`sch.ListSchedule`) doesn't support uniform scheduling.

1.11.16 `scheduler_heuristic`

Must be one of `path_length` or `random`, default `path_length`. When no compiler configuration file is specified, this controls what scheduling heuristic should be used for ordering the list of available gates by criticality. These are the heuristics for the old scheduler (`sch.Schedule`), so setting this option will prevent the new scheduler (`sch.ListSchedule`) from being used. To set the heuristic for the new scheduler, you must use its pass options directly; there is no global option for this.

1.11.17 `scheduler_commute`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether the default-inserted scheduler passes are allowed to commute CZ and CNOT gates. This also affects the mapper.

1.11.18 `scheduler_commute_rotations`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether the default-inserted scheduler passes are allowed to commute single-qubit X and Z rotations. This also affects the mapper.

1.11.19 `print_dot_graphs`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, this controls whether data dependency/schedule graphs should be written by default-inserted scheduler passes. The DOT file format is used as output format.

1.11.20 `initialplace`

Must be one of `no`, `yes`, `1s`, `10s`, `1m`, `10m`, `1h`, `1sx`, `10sx`, `1mx`, `10mx`, or `1hx`, default `no`. When no compiler configuration file is specified, this controls whether the MIP-based initial placement algorithm should be run before running the heuristic mapper. A timeout can be specified, as listed in the allowable values. If the timeout value ends in an 'x', compilation fails if the timeout is hit; otherwise, heuristic mapping is performed instead.

1.11.21 `initialplace2qhorizon`

Must be an integer between 0 and 100 inclusive, default `0`. When no compiler configuration file is specified, this controls how many two-qubit gates the MIP-based initial placement pass (if any) considers for each kernel. If 0 or unspecified, all gates are considered.

1.11.22 mapper

Must be one of no, base, baserc, minextend, minextendrc, or maxfidelity, default no. When no compiler configuration file is specified, this controls whether the heuristic mapper will be run, and if so, which heuristic it should use. When no, MIP-based placement is also disabled.

1.11.23 mapmaxalters

Must be an integer less than or equal to 0, default 0. When no compiler configuration file is specified, this controls whether the heuristic mapper will be run, and if so, how many alternative routing solutions it should generate before picking one via the heuristic or tie-breaking method. 0 means unlimited.

1.11.24 mapinitone2one

Must be yes or no, default yes. When no compiler configuration file is specified, and the mapper is enabled, this controls whether the mapper should assume that each kernel starts with a one-to-one mapping between virtual and real qubits. When disabled, the initial mapping is treated as undefined.

1.11.25 mapassumezeroinitstate

Must be yes or no, default no. When no compiler configuration file is specified, and the mapper is enabled, this controls whether the mapper should assume that each qubit starts out as zero at the start of each kernel, rather than with an undefined state.

1.11.26 mapprepinitsstate

Must be yes or no, default no. When no compiler configuration file is specified, and the mapper is enabled, this controls whether the mapper may assume that a user-written prepz gate actually leaves the qubit in the zero state, rather than any other quantum state. This allows it to make some optimizations.

1.11.27 maplookahead

Must be one of no, 1qfirst, noroutingfirst, or all, default noroutingfirst. When no compiler configuration file is specified, and the mapper is enabled, this controls the lookahead_mode option for the mapper, controlling the strategy for selecting the next gate(s) to map. Refer to the mapper pass documentation for lookahead_mode for more information.

1.11.28 mappathselect

Must be one of all, borders, or random, default all. When no compiler configuration file is specified, and the mapper is enabled, this controls whether to consider all paths from a source to destination qubit while routing, or to favor routing along the borders of the search space. The latter is only supported when the qubits are given planar coordinates in the topology section of the platform configuration file. Both all and random consider all paths, but for the latter the order in which the paths are generated is shuffled, which is useful to reduce bias when max_alternative_routes is used.

1.11.29 `mapselectswaps`

Must be one of `one`, `all`, or `earliest`, default `all`. When no compiler configuration file is specified, and the mapper is enabled, this controls how routing interacts with speculation. When`all`, all swaps for a particular routing option are committed immediately, before trying anything else. When `one`, only the first swap in the route from source to target qubit is committed. When `earliest`, the swap that can be done at the earliest point is selected, which might be the one swapping the source or target qubit.

1.11.30 `maprecNN2q`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, and the mapper is enabled, this controls the `recurse_on_nn_two_qubit` option for the mapper; i.e. whether to "recurse" on nearest-neighbor two-qubit gates. NOTE: this is an advanced/unstable option; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

1.11.31 `mapselectmaxlevel`

Must be an integer between 0 and 10 inclusive or inf, default `0`. When no compiler configuration file is specified, and the mapper is enabled, this controls the maximum recursion depth while searching for alternative mapping solutions. NOTE: this is an advanced/unstable option; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

1.11.32 `mapselectmaxwidth`

Must be one of `min`, `minplusone`, `minplushalfmin`, `minplusmin`, or `all`, default `min`. When no compiler configuration file is specified, and the mapper is enabled, this limits how many alternative mapping solutions are considered. `min` means only the best-scoring alternatives are considered, `minplusone` means the best scoring alternatives plus one more are considered, `minplushalfmin` means 1.5x the number of best-scoring alternatives are considered, `minplusmin` means 2x, and `all` means they are all considered. NOTE: this is an advanced/unstable option; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

1.11.33 `maptiebreak`

Must be one of `first`, `last`, `random`, or `critical`, default `random`. When no compiler configuration file is specified, and the mapper is enabled, this controls how to tie-break equally-scoring alternative mapping solutions. `first` and `last` choose respectively the first and last solution in the list (assuming the qubits have planar coordinates specified in the topology section, `first` selects the left-most alternative with the two-qubit gate near target, and `last` selects the right-most alternative with the two-qubit gate near source; when no coordinates are given the choice is undefined, though deterministic), `random` uses random number generation to select an alternative, and `critical` favors the alternative that maps the most critical gate as determined by the scheduler (if any).

1.11.34 `mapusemoves`

Must be an integer between 0 and 20 inclusive or one of `no` or `yes`, default `yes`. When no compiler configuration file is specified, and the mapper is enabled, this controls if/when the mapper inserts move gates rather than swap gates to perform routing. If `no`, swap gates are always used. Otherwise, a move gate is used if the other qubit has been initialized, or if initializing it only extends the circuit by the given number of cycles. `yes` implies this limit is 0 cycles.

1.11.35 `mapreverseswap`

Must be `yes` or `no`, default `yes`. When no compiler configuration file is specified, and the mapper is enabled, this controls whether the mapper will reverse the operands for a swap gate when reversal improves the schedule. NOTE: this currently assumes that the second qubit operand of the swap gate decomposition in the platform configuration file is used before than the first operand; if this is not the case, enabling this will worsen the routing result rather than improve it.

1.11.36 `backend_cc_map_input_file`

Must be any string, no default value. When no compiler configuration file is specified, and the CC backend pass is inserted automatically, this controls its `map_input_file` option, which specifies the input map filename.

1.11.37 `backend_cc_verbose`

Must be `yes` or `no`, default `yes`. When no compiler configuration file is specified, and the CC backend pass is inserted automatically, this controls its `verbose` option, which selects whether verbose comments should be added to the generated .vq1asm file.

1.11.38 `backend_cc_run_once`

Must be `yes` or `no`, default `no`. When no compiler configuration file is specified, and the CC backend pass is inserted automatically, this controls its `run_once` option, which creates a .vq1asm program that runs once instead of repeating indefinitely.

1.11.39 `quantumsim`

Must be one of `no`, default `no`. Quantumsim output is no longer supported by OpenQL. This option only exists to not break existing code that sets the option to `no`.

1.11.40 `cz_mode`

Must be one of `manual` or `auto`, default `manual`. This option is no longer used by OpenQL. It's just there to not break existing code that sets the option.

1.11.41 `scheduler_post179`

Must be `yes` or `no`, default `yes`. This option is no longer used by OpenQL. It's just there to not break existing code that sets the option.

1.11.42 `optimize`

Must be `yes` or `no`, default `no`. This option is no longer used by OpenQL. It's just there to not break existing code that sets the option.

1.11.43 `generate_code`

Must be `yes` or `no`, default `yes`. This option is no longer used by OpenQL. It's just there to not break existing code that sets the option.

1.12 Supported passes

This section lists all the compiler pass types currently available within OpenQL.

1.12.1 Statistics cleaner

Type name(s): `ana.statistics.Clean`.

This pass just discards any statistics that previous passes might have attached to the kernel and program. It is inserted automatically after every normal pass that does not have statistics reporting enabled.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

1.12.2 Statistics reporter

Type name(s): `ana.statistics.Report`.

This pass reports some basic statistics of the program and each kernel to a report file. Some passes may also attach additional pass-specific statistics to the program and kernels, in which case these are printed and subsequently discarded as well.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`output_suffix`

Must be any string, default `.txt`. Suffix to use for the output filename.

`line_prefix`

Must be any string, no default value. Historically, report files contain a “# ” prefix before each line. You can use this option to emulate that behavior.

1.12.3 Circuit visualizer

Type name(s): `ana.visualize.Circuit`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

The circuit visualizer produces an image of the circuit containing the operations on each qubit per cycle. If so configured, it can also render instrument waveforms alongside it.

Configuration file structure

The visualizer is configured by way of the visualizer configuration file. Each attribute has a default setting, so many can be omitted if no change is wanted.

The circuit visualizer supports the following top-level sections:

- `"circuit"`: contains options for the circuit visualization, including pulse visualization.
- `"saveImage"`: a boolean indicating whether the generated image should be saved to disk. When this is true, the file will be saved regardless of/in addition to the interactive window as controlled by the `interactive` option.
- `"backgroundColor"`: the background color of the generated image.

Note: A single visualizer configuration file may be used for all three visualization pass types. The configuration file format is designed to be cross-compatible.

Note: When the to-be-visualized circuit is very large, the interactive window may have trouble rendering the circuit even when zoomed in. Therefore, it is recommended to use non-interactive mode and view the generated bitmap with a more capable external viewer.

The `"circuit"` section has several child sections.

- `"cycles"`: contains parameters that govern cycle labels, edges, cycle compression and cutting.
- `"bitLines"`: defines the labels and lines, including grouping lines for both quantum and classical bitLines.
- `"grid"`: defines several parameters of the image grid.
- `"gateDurationOutlines"`: controls parameters for gate duration outlines.
- `"measurements"`: several parameters controlling measurement visualization.
- `"pulses"`: parameters for pulse visualization.
- `"instructions"`: a map of instruction types (the keys) with that type's gate visualization as value, used for custom instructions.

Example configuration (self-explanatory attributes have no description):

```
"cycles": {
    // parameters for the labels above each cycle
    "labels": {
```

(continues on next page)

(continued from previous page)

```
        "show": true,
        // whether the cycle labels should be shown in nanoseconds or
        // cycle numbers
        "inNanoSeconds": false,
        // the height of the cycle label row
        "rowHeight": 24,
        "fontHeight": 13,
        "fontColor": [0, 0, 0]
    },
    // parameters for the vertical edges between cycles
    "edges": {
        "show": true,
        "color": [0, 0, 0],
        "alpha": 0.2
    },
    // parameters for the cutting of cycles (cycles are cut when no new
    // gates are started)
    "cutting": {
        "cut": true,
        // how many cycles should be without a gate starting before the
        // cycle is cut
        "emptyCycleThreshold": 2,
        "cutCycleWidth": 16,
        // a multiplier on the width of the cut cycles
        "cutCycleWidthModifier": 0.5
    },
    // cycles are compressed by reducing each gate's duration to one cycle
    "compressCycles": false,
    // partitioning a cycle means that each gate in that cycle gets its
    // own column within the cycle; this can be done to remove visual
    // overlap
    "partitionCyclesWithOverlap": true
},
"bitLines": {
    // parameters for the labels on each quantum or classical bit line
    "labels": {
        "show": true,
        // the width of the label column
        "columnWidth": 32,
        "fontHeight": 13,
        // the colors of quantum and classical bit labels
        "qbitColor": [0, 0, 0],
        "cbitColor": [128, 128, 128]
    },
    // parameters specifically for quantum bit lines
    "quantum": {
        "color": [0, 0, 0]
    },
    // parameters specifically for classical bit lines
    "classical": {
        "show": true,
        // grouping classical bit lines collapses them into a double line
        // to reduce visual clutter
        "group": false,
        // controls the gap between the double line indicating the
        // collapsed classical lines
        "groupedLineGap": 2,
```

(continues on next page)

(continued from previous page)

```
        "color": [128, 128, 128]
    },
    // parameters for the horizontal edges between bit lines
    "edges": {
        "show": false,
        "thickness": 5,
        "color": [0, 0, 0],
        "alpha": 0.4
    }
},
"grid": {
    // the size of each cell formed by a the crossing of a single bit line
    // and cycle
    "cellSize": 32,
    // the border at the edges of the generated image
    "borderSize": 32
},
"gateDurationOutlines": {
    "show": true,
    // the gap between the edge of the cell and the gate duration outline
    "gap": 2,
    // the filled background alpha
    "fillAlpha": 0.2,
    // the outline alpha
    "outlineAlpha": 0.3,
    "outlineColor": [0, 0, 0]
},
"measurements": {
    // whether to draw a connection from the measurement gate to the
    // classical line it stores the result in
    "drawConnection": true,
    // the gap between the double line representing the connection
    "lineSpacing": 2,
    "arrowSize": 10
},
"pulses": {
    // set this to true to use the pulse visualization
    "displayGatesAsPulses": false,
    // these heights control the line row heights
    "pulseRowHeightMicrowave": 32,
    "pulseRowHeightFlux": 32,
    "pulseRowHeightReadout": 32,
    // these colors control the line colors
    "pulseColorMicrowave": [0, 0, 255],
    "pulseColorFlux": [255, 0, 0],
    "pulseColorReadout": [0, 255, 0]
},
"instructions" {
    // defined below
}
```

Gate visualization

When using default gates, the visualizations for each gate are built in. However, default gates are mostly deprecated (aside from a few exceptions such as barrier), and will likely be removed in the future.

When using custom gates, the default gate visualizations are not used, so the visualization needs to be defined by the user. In the instructions section of the visualizer configuration file, each instruction "type" has its own corresponding description of gate visualization parameters. These instruction types are mapped to actual custom instructions from the hardware configuration file by adding a `"visual_type"` key to the instructions. For example:

```
{
    ...,
    "instructions" {
        ...,
        "h q1": {
            "duration": 40,
            "qubits": ["q1"],
            "visual_type": "h"
        },
        ...
    },
    ...
}
```

This custom Hadamard gate defined on qubit 1 has one additional attribute `"visual_type"` describing its visualization type. The value of this attribute links to a key in the visualizer configuration file, which has the description of the gate visualization parameters that will be used to visualize this custom instruction. Note that this allows multiple custom instructions to share the same visualization parameters, without having to duplicate the parameters.

The `instructions` section of the visualizer configuration file then defines how each gate type is rendered. Here's an excerpt from an example configuration file:

```
{
    ...,
    "instructions": {
        ...,
        "h": {
            "connectionColor": [0, 0, 0],
            "nodes": [
                {
                    "type": "GATE",
                    "radius": 13,
                    "displayName": "H",
                    "fontHeight": 13,
                    "fontColor": [255, 255, 255],
                    "backgroundColor": [70, 210, 230],
                    "outlineColor": [70, 210, 230]
                }
            ]
        },
        ...
    },
    ...
}
```

Each gate has a `"connectionColor"` which defines the color of the connection line for multi-operand gates, and an array of `"nodes"`. A node is the visualization of the gate acting on a specific qubit or classical bit. If a Hadamard gate is acting on qubit 3, that is represented by one node. If a CNOT gate is acting on qubits 1 and 2, it will have two nodes, one describing the visualization of the CNOT gate at qubit 1 and one describing the visualization on qubit 2. A measurement gate measuring qubit 5 and storing the result in classical bit 0 will again have two nodes.

Each node has several attributes describing its visualization.

- `"type"`: the visualization type of the node, see below for a list of the available types.
- `"radius"`: the radius of the node in pixels.
- `"displayName"`: text that will be displayed on the node (for example `"H"` will be displayed on the Hadamard gate in the example above).
- `"fontHeight"`: the height of the font in pixels used by the `"displayName"`.
- `"fontColor"`: the color of the font used by the `"displayName"`.
- `"backgroundColor"`: the background color of the node.
- `"outlineColor"`: the color of the edge-line of the node.

The colors are defined as RGB arrays: `[R, G, B]`.

The type of the nodes can be one of the following.

- `"NONE"`: the node will not be visible.
- `"GATE"`: a square representing a gate.
- `"CONTROL"`: a small filled circle.
- `"NOT"`: a circle outline with cross inside (a CNOT cross).
- `"CROSS"`: a diagonal cross.

When a gate has multiple operands, each operand should have a node associated with it. Simply create as many nodes in the node array as there are operands and define a type and visual parameters for it. Don't forget the comma to separate each node in the array. Note that nodes are coupled to each operand sequentially, i.e. the first node in the node array will be used for the first qubit in the operand vector.

Pulse visualization

Along with an abstract representation of the gates used in the quantum circuit, the gates can also be represented by the RF pulses used in the real hardware. This will be done when the `"displayGatesAsPulses"` flag in the `"pulses"` section is set to true. In this case, the `waveform_mapping` option must be used to specify a waveform configuration file.

Each qubit consists of three lines, the microwave, flux and readout lines, controlling single-qubit gates, two-qubit gates and readouts respectively. The waveforms used by the hardware should be stored in the waveform mapping configuration file. Then, in the hardware configuration file the `"visual_codeword"` and `"qubits"` attributes of each instruction are used as key into the table contained in the waveform mapping file to find the corresponding waveform for the specific instruction and qubit (waveforms for the same instruction can be different for different qubits). Note that a two-qubit gate has two codeword attributes, one for each qubit: `"visual_right_codeword"` and `"visual_left_codeword"`.

In the waveform mapping configuration file, the waveforms are grouped by codeword first and then by addressed qubit. The waveforms themselves are stored as an array of real numbers. The scale of these numbers does not matter, the visualizer will automatically scale the pulses to fit inside the graph. The time between samples is determined by the sample rate (the sample rate can be different for each of the three lines).

TODO: the structure of the waveform mapping configuration file should still be documented. For now, use the examples in `tests/visualizer` as a baseline.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`config`

Must be any string, default `visualizer_config.json`. Path to the visualizer configuration file.

`waveform_mapping`

Must be any string, default `waveform_mapping.json`. Path to the visualizer waveform mapping file.

`interactive`

Must be `yes` or `no`, default `no`. When yes, the visualizer will open a window when the pass is run. When no, an image will be saved as <output_prefix>.bmp instead.

1.12.4 Qubit interaction graph visualizer

Type name(s): `ana.visualize.Interaction`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

The qubit interaction graph visualizes the interactions between each of the qubits in the circuit. If a gate acts on two or more qubits, those qubits interact with each other and an edge will be drawn in the graph, with a number indicating the amount of times those qubits have interacted with each other. Note that the visualization of this is very simple, and the DOT graph the visualizer can produce should be used with the user's favorite graphing software to create a better looking graph.

Configuration file structure

The visualizer is configured by way of the visualizer configuration file. Each attribute has a default setting, so many can be omitted if no change is wanted.

The circuit visualizer supports the following top-level sections:

- `"interactionGraph"`: contains options for the interaction graph.
- `"saveImage"`: a boolean indicating whether the generated image should be saved to disk. When this is true, the file will be saved regardless of/in addition to the interactive window as controlled by the `interactive` option.
- `"backgroundColor"`: the background color of the generated image.

Note: A single visualizer configuration file may be used for all three visualization pass types. The configuration file format is designed to be cross-compatible.

The `"interactionGraph"` section should have the following structure.

```
"interactionGraph": {
    // whether a DOT file should be generated for use with graphing
    // software
    "outputDotFile": true,
    "borderWidth": 32,
    // the minimum radius of the circle on which the qubits are placed
    "minInteractionCircleRadius": 100,
    "interactionCircleRadiusModifier": 3.0,
    "qubitRadius": 17,
    "labelFontHeight": 13,
    "circleOutlineColor": [0, 0, 0],
    "circleFillColor": [255, 255, 255],
    "labelColor": [0, 0, 0],
    "edgeColor": [0, 0, 0]
}
```

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`config`

Must be any string, default `visualizer_config.json`. Path to the visualizer configuration file.

`interactive`

Must be `yes` or `no`, default `no`. When yes, the visualizer will open a window when the pass is run. When no, an image will be saved as <output_prefix>.bmp instead.

1.12.5 Qubit mapping graph visualizer

Type name(s): `ana.visualize.Mapping`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

The mapping graph tracks the journey of the virtual qubits through the real topology of the quantum hardware as the cycles of the quantum program are executed. The virtual qubits change location whenever a swap/move gate (or their decomposed parts) is finished executing. For convenience, the abstract circuit representation of the quantum program is shown above the qubit mappings for each cycle.

The topology of the quantum hardware is taken from the topology section in the hardware configuration file, together with the edges between the qubits. If no coordinates and/or edges are defined for the qubits, the qubits will simply be spaced sequentially in a grid structure without edges being shown.

Configuration file structure

The visualizer is configured by way of the visualizer configuration file. Each attribute has a default setting, so many can be omitted if no change is wanted.

The circuit visualizer supports the following top-level sections:

- `"mappingGraph"`: contains options for the mapping graph.
- `"saveImage"`: a boolean indicating whether the generated image should be saved to disk. When this is true, the file will be saved regardless of/in addition to the interactive window as controlled by the `interactive` option.
- `"backgroundColor"`: the background color of the generated image.

Note: A single visualizer configuration file may be used for all three visualization pass types. The configuration file format is designed to be cross-compatible.

The "mappingGraph" section should have the following structure.

```
"mappingGraph": {
    // whether qubits should be filled with the corresponding logical
    // qubit index in the first cycle
    "initDefaultVirtuals": false,
    // give each distinct virtual qubit a color
    "showVirtualColors": true,
    // show the real qubit indices above the qubits
    "showRealIndices": true,
    // whether to use the topology from the hardware configuration file
    "useTopology": true,
    // parameters for controlling the layout
    "qubitRadius": 15,
    "qubitSpacing": 7,
    "fontHeightReal": 13,
    "fontHeightVirtual": 13,
    "textColorReal": [0, 0, 255],
    "textColorVirtual": [255, 0, 0],
    // the gap between the qubit and the real index
    "realIndexSpacing": 1,
    "qubitFillColor": [255, 255, 255],
    "qubitOutlineColor": [0, 0, 0]
}
```

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`config`

Must be any string, default `visualizer_config.json`. Path to the visualizer configuration file.

`interactive`

Must be `yes` or `no`, default `no`. When yes, the visualizer will open a window when the pass is run. When no, an image will be saved as <output_prefix>.bmp instead.

1.12.6 Central Controller code generator

Type name(s): `arch.cc.gen.VQ1Asm`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

Assembly code generator for the Q1 processor in the QuTech Central Controller, version 0.3.1

This pass actually generates three files:

- `<prefix>.vq1asm`: the assembly code output file;
- `<prefix>.map`: the instrument configuration file; and
- `<prefix>.vcd`: a VCD (value change dump) file for viewing the waveforms that the program outputs.

The pass is compile-time configured with the following options:

- `OPT_CC_SCHEDULE_RC` = 1
- `OPT_SUPPORT_STATIC_CODEWORDS` = 1
- `OPT_STATIC_CODEWORDS_ARRAYS` = 1
- `OPT_VECTOR_MODE` = 0
- `OPT_FEEDBACK` = 1
- `OPT_PRAGMA` = 1

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`map_input_file`

Must be any string, no default value. Specifies the input map filename.

`verbose`

Must be `yes` or `no`, default `yes`. Selects whether verbose comments should be added to the generated .vq1asm file.

`run_once`

Must be `yes` or `no`, default `no`. When set, the emitted .vq1asm program runs once instead of repeating indefinitely.

1.12.7 Diamond microcode generator

Type name(s): `arch.diamond.gen.Microcode`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

Generates the microcode from the algorithm (cQASM/C++/Python) description for quantum computing in diamond.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

1.12.8 Instruction generalizer

Type name(s): `dec.Generalize`.

This pass converts the format of all instructions in the program to their most generalized form. For example, if a specialized CNOT gate exists for qubits 1 and 2 and this specialization is used in the program, the instruction is changed to the generalized version for any set of qubits. This implements the reverse operation of `dec.Specialize`.

Options

output_prefix

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

1.12.9 Instruction decomposer

Type name(s): `dec.Instructions`.

This pass (conditionally) applies instructions decomposition rules as specified in the platform configuration JSON structure. The pass returns the number of rules that were applied.

Rules can be disabled for the purpose of this pass using the `predicate_key` and `predicate_value` options. When set, the key given by `predicate_key` is resolved in the JSON data that may be associated with new-style decomposition rules (the ones associated with instructions, rather than the ones specified in the `"gate_decomposition"` section of the platform JSON file). If this resolves to a string, the `predicate_value` option is matched against it. The rule is then only applied if there is a match. Some special cases:

- if the key does not exist in the JSON data associated with the decomposition rule, or if it exists but maps to something that isn't a string, the predicate will match if `predicate_value` matches an empty string; and
- the effective JSON structure for legacy decomposition rules is `{"name": "legacy"}`.

The `ignore_schedule` option controls how scheduling information is treated. When set to yes (the default), the cycle numbers of the decomposed instructions will be set to the same cycle number as the original instruction. When set to no, the schedule of the decomposed instructions is taken from the decomposition rule, and instructions are reordered accordingly after all decompositions have taken place.

For example, assume that we have the following decomposition rule for a CNOT gate:

```
ym90 op(1)
cz op(0), op(1)
skip 1
y90 op(1)
```

and that we have the following program as input:

```
{
    cnot q[0], q[1]
    cnot q[1], q[2]
}
```

Now, if `ignore_schedule` is enabled, the resulting program would be

```
{
    ym90 q[1]
    cz q[0], q[1]
    y90 q[1]
    ym90 q[2]
    cz q[1], q[2]
    y90 q[2]
}
```

The schedule is obviously invalid, because qubits are being used by multiple gates in the same cycle. But so was the input. Nevertheless, the order of the instructions is what we wanted; after scheduling, the program will be correct.

If we were to turn `ignore_schedule` off, however, this is what we'd get:

```
{
    ym90 q[1]
    ym90 q[2]
}
{
    cz q[0], q[1]
```

(continues on next page)

(continued from previous page)

```
    cz q[1], q[2]
}
skip 1
{
    y90 q[1]
    y90 q[2]
}
```

Which is wrong! The `ym90` and `y90` gates execute out of order with the `cz q[1], q[2]` now. Scheduling won't fix this.

The key takeaway here is that you should leave `ignore_schedule` enabled if A) the program has not been scheduled yet or B) you're not sure that the schedules in the decomposition rules are actually defined correctly.

Of course, there are cases where `ignore_schedule` needs to be disabled, otherwise the option wouldn't need to be there. It's useful specifically when you need to process code expansions *after* scheduling. You will need to make sure that the decomposition rules that the predicate matches are written such that they won't ever break a correctly scheduled program, but if that's the case, you won't have to schedule the program again after the decomposition. For example, if the input program had been

```
cnot q[0], q[1]
skip 3
cnot q[1], q[2]
```

the result with `ignore_schedule` disabled would have been

```
ym90 q[1]
cz q[0], q[1]
skip 1
y90 q[1]
ym90 q[2]
cz q[1], q[2]
skip 1
y90 q[2]
```

which is not an optimal schedule by any means, but a correct one nonetheless. A more reasonable use case for this than CNOT to CZ decomposition would be expanding a CZ gate to single-qubit flux and parking gates; it's vital that these gates will not be shifted around with respect to each other, which scheduling after decomposing them might do.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

predicate_key

Must be any string, default `name`. The key to use for the predicate check.

predicate_value

Must be any string, default `*`. Pattern that must match for the value of the key specified by the `predicate_key` option for a decomposition rule to be applied. `*` and `?` may be used to construct nontrivial patterns. The entire pattern must match; for partial matches, prefix and append an `*`.Nonexistent keys or non-string values are treated as if they are an empty string.

ignore_schedule

Must be `yes` or `no`, default `yes`. When set, the schedule of the decomposition expansions is ignored. This prevents instructions from ever needing to be reordered, and thus prevents the behavior of the program from changing due to incorrect schedules in the decomposition rules, but will almost certainly require the program to be rescheduled. You should only turn this off when you really want to keep scheduling information, and are really sure that the schedules in the decomposition rule expansions are correct.

1.12.10 Instruction specializer

Type name(s): `dec.Specialize`.

This pass converts the format of all instructions in the program to their most specialized form. For example, if a generalized CNOT gate exists for qubits 1 and 2, and a specialization exists for this qubit pair as well, the instruction is changed to the specialized version This implements the reverse operation of `dec.Generalize`.

Options

output_prefix

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

1.12.11 Structure decomposer

Type name(s): `dec.Structure`.

This pass converts the program to basic block form. Specifically, the postcondition for this pass is:

- all blocks consist of only instructions (no control-flow statements like loops or if-conditionals); and
- only the last instruction of each block may be a goto instruction.

All control-flow that exists in the program before this pass is reduced to this basic form. This doesn't change the behavior of the program, but all information about the program structure is lost. Because of this, this should be one of the last passes, if the pass is needed at all; this depends on the code generator used, or on whether there is a need for passes that rely on basic-block form and the corresponding control-flow graph to operate.

Optionally, the control-flow graph of the resulting program can be printed as in graphviz dot format.

Schedule preservation

This pass does its best to preserve the schedule of the original program, so it doesn't need to be rescheduled after the transformation. Note however that this is only valid if classical instructions have (quantum) duration zero and do not use any scheduling resources.

Unfortunately, there are some situations where the resulting schedule ends up being longer than it should be. This has to do with block duration not currently being explicitly encoded in the IR. The schedule *should* at least be correct, though, if the above assumptions about classical instructions are applicable.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of no, yes, stats, qasm, or both, default no. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to no to disable this functionality or to yes to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix _debug_[in|out].ir for the IR dump, and _debug_[in|out].cq for the cQASM file. The option values stats, cqasm, and both are used for backward compatibility with the write_qasm_files and write_report_files global options; for stats and both a statistics report file is written with suffix _[in|out].report, and for qasm and both a cQASM file is written (without stats in the comments) with suffix _[in|out].qasm.

write_dot_graph

Must be yes or no, default no. Writes the control-flow graph of the resulting program in the dot format. The file is written with suffix ".dot".

1.12.12 cQASM reader

Type name(s): io.cqasm.Read.

This pass completely discards the incoming program and replaces it with the program described by the given cQASM file.

The reader supports up to cQASM 1.2. However, rather than supporting the default cQASM instruction and function set, the instructions defined in the platform JSON description are used. In addition, the following special instructions are supported.

- skip <int>: used in conjunction with bundle notation to represent a scheduled program. The instruction behaves like <int> consecutive empty bundles.
- wait <int>: used as an input to the scheduler, forcing all instructions defined after the wait instruction to start at least <int> cycles after all instructions defined before the wait instruction have completed.
- wait q[...], <int>: as above, but only affects instructions that operate on the specified qubit. Effectively, this means that the instruction enforces that the qubit is idled for at least <int> cycles. If single-gate-multiple-qubit notation is used, for example wait q[0,2], 3, the *independent* wait blocks are created as per the regular single-gate-multiple-qubit rules.
- wait <int>, ...: generalization of the above, supporting any kind of object, and any number of them. That is, all preceding instructions operating on any object specified in place of the ellipsis must complete before the wait can be scheduled, and any following instructions operating on any object specified in place of the ellipsis must start after the wait completes. Unlike the above, if single-gate-multiple- qubit notation is used for qubit/bit objects, the result is a *single* wait instruction that waits for all indexed elements, rather than multiple parallel wait instructions (this is semantically different!).
- barrier ...: shorthand for wait 0, ... A barrier without arguments is also valid.
- measure_all: if the measure_all_target option is set, this is automatically expanded to single-qubit measurement instructions of the name defined by the value of the option for each qubit in the main qubit register.
- pragma: supported as a means to place annotations inside statement lists. The reader uses this for some annotations of its own, but otherwise ignores it.

Registers as defined by the platform are implicitly defined by the reader, and must thus not be redefined as variables. The only exception is the main qubit register (qubits <int>), which may optionally be defined at the top of the file, as this statement is mandatory in the cQASM 1.0 language. Non-scalar registers, such as integer control registers (cregs) and bit registers beyond the bits associated with the main qubit register (bregs), must be referred to using a function call of the same name as the register with the index/indices as arguments (for example `creg(2)`), as cQASM doesn't natively support non-scalar objects aside from the main qubit register and associated bits.

Various annotations may be used to fine-tune the behavior of the reader. Most of these are particularly important for accurate reproduction of OpenQL's internal representation of the program after conversion to and from cQASM.

- `pragma @ql.platform(<json>)` may be placed at the top of the program. If the `load_platform` option is not enabled, it is ignored. Otherwise, the following forms are supported (these mirror the constructors of the Platform class in the API for the most part):
  - not specified or `pragma @ql.platform()`: shorthand for `...("none", "none")`.
  - `pragma @ql.platform(name: string)`: shorthand for `...(name, name)`.
  - `pragma @ql.platform(name: string, platform_config: string)`: builds a platform with the given name (only used for log messages) and platform configuration, the latter of which can be either a recognized platform name with or without variant suffix (for example `"cc"` or `"cc_light.s7"`), or a path to a JSON configuration filename.
  - `pragma @ql.platform(name: string, platform_config: string, compiler_config: string)`: as above, but specifies a custom compiler configuration file in addition.
  - `pragma @ql.platform(name: string, platform_config: json)`: instead of loading the platform JSON data from a file, it is read from the given JSON literal directly.
  - `pragma @ql.platform(platform_config: json)`: shorthand for the above, using just `"platform"` for the name.

  Note that the loaded compiler configuration is ignored, because we already have one by the time this pass is run! Use the `compile_cqasm()` API function to load everything from the cQASM file.
- `pragma @ql.name("<name>")` may be placed at the top of the program to set the name of the program, in case no program exists in the IR yet. Otherwise it will simply default to `"program"`.
- Variables may be annotated with `@ql.type("<name>")` to specify the exact OpenQL type that should be used. If not specified, the first type defined in the platform that matches the primitive cQASM type will be used. You should only need this when you're using a platform that, for instance, supports multiple types/sizes of integers.
- Variables may be annotated with `@ql.temp` to specify that they are temporary objects that were automatically inferred. This is normally only used by generated cQASM code.
- The first subcircuit may be annotated with `@ql.entry` if it consists of only a single, unconditional goto instruction. In that case, the subcircuit will be discarded, and the target of the `goto` instruction will be marked as the entry point of the program within OpenQL, rather than the first subcircuit.
- The last subcircuit may be annotated with `@ql.exit` if it contains no instructions. In that case, the subcircuit will be discarded, but any subcircuits that end in an unconditional goto instruction to this subcircuit will be marked as ending the program within OpenQL.

The `schedule` option controls how scheduling information is interpreted.

- If set to `keep`, `skip` instructions and bundles are used to determine the cycle numbers of the instructions within each block, using the following rules:
  - single-gate-multiple-qubit notation (for example `x q[0,1]`) is expanded to a bundle of instructions that start simultaneously;
  - the first instruction or bundle of instructions is assigned to start in cycle 0;
  - each subsequent single instruction or bundle of instructions starts one cycle after the previous instruction/bundle started; and

  - `skip <int>` behaves like `<int>` empty bundles.

- If set to `discard`, `skip` instructions and bundles are ignored, and single-gate-multiple-qubit is ignored in terms of its timing implication; instead, instructions will be assigned consecutive cycle numbers in the order in which they appear in the file.
- If set to `bundles-as-barriers`, timing information is ignored as for `discard`, but barriers are implicitly inserted before and after each bundle, sensitive to exactly those objects used as operands by the instructions that appear in the bundle. Single-gate-multiple-qubit notation expands to a bundle as well.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`cqasm_file`

Must be any string, no default value. cQASM file to read. Mandatory.

`schedule`

Must be one of `keep`, `discard`, or `bundles-as-barriers`, default `keep`. Controls how scheduling/timing information (via bundles and skip instructions) is interpreted. See pass description for more info.

`measure_all_target`

Must be any string, no default value. Standard cQASM has a measure_all instruction that implicitly measures all qubits in a certain way, while OpenQL platforms normally lack this instruction. When this option is set, it is treated as the name of a single-qubit measurement gate, that will be used to implement measure_all; i.e. the measure_all instruction will be expanded to a bundle of

<measure_all_target> instructions, for each qubit in the main qubit register.

`load_platform`

Must be `yes` or `no`, default `no`. When set, the platform is loaded from the cQASM file by means of a `pragma @ql.platform(...)` statement at the top of the code. See pass description for more information.

1.12.13 cQASM writer

Type name(s): `io.cqasm.Report`.

This pass writes the current program out as a cQASM file, targeting the given cQASM version. The writer supports cQASM versions 1.0, 1.1, and 1.2, but note that older cQASM versions do not support everything that OpenQL supports.

Several options are provided to control how the cQASM file is written. These are necessary because, even within a particular cQASM version, various dialects exist, based on instruction set, implicit register definitions, function definitions, and so on.

Regardless of configuration, the written file assumes that the target cQASM reader/interpreter supports the instruction- and function set as defined (or derived from) the platform JSON description. This means that if you want to target a cQASM reader/interpreter that only supports a subset of this instruction/function set, or one that supports a different instruction set entirely, you will have to ensure that all instructions have been decomposed to the instruction set supported by the target prior to printing the cQASM file, or write the program such that the unsupported instructions/functions aren't used in the first place. It is also possible to embed the platform description into the cQASM file in JSON form via a pragma instruction, but of course the target cQASM reader/interpreter would then have to support that instead.

The only instructions that the cQASM writer can print that are not part of the instruction set as defined in the JSON file are pragmas, barriers, wait, and skip instructions, but they can be disabled via options.

- `pragma` instructions are no-op placeholder instructions with no operands that are used to convey metadata via annotations within the context of a statement. If the `with_metadata` and `with_platform` options are disabled, no pragmas will be printed.
- `barrier` and `wait` instructions are used for the builtin wait instruction. If the `with_barriers` option is disabled, they will not be printed. If the option is set to `simple`, the printed syntax and semantics are:
  - `wait <int>`: wait for all previous instructions to complete, then wait `<int>` cycles, where `<int>` may be zero;
  - `barrier q[...]`: wait for all instructions operating on the qubits in the single-gate-multiple-qubit list to complete.

  Note that this syntax only supports barriers acting on qubits, and doesn't support wait instructions depending on a subset of objects. However, it conforms with the default cQASM 1.0 gateset as of libqasm 0.3.1 (in 0.3 and before, `barrier` did not exist in the default gateset). If the option is instead set to `extended`, the syntax is:
  - `wait <int>`: wait for all previous instructions to complete, then wait `<int>` >= 1 cycles;

  - `wait <int>, [...]`: wait for all previous instructions operating on the given objects to complete, when wait `<int>` >= 1 cycles;
  - `barrier`: wait for all previous instructions to complete;
  - `barrier [...]`: wait for all previous instructions operating on the given objects to complete.

  This encompasses all wait instructions possible within OpenQL's IR. OpenQL's cQASM reader supports both notations equally.

- `skip <int>` instructions are printed in addition to the `{}` multiline bundle notation to convey scheduling information: all instructions in a bundle start in the same cycle, the subsequent bundle or instruction starts in the next cycle (regardless of the duration of the instructions in the former bundle), and a `skip <int>` instruction may be used in place of `<int>` empty bundles, thus skipping `<int>` cycles. `skip` instructions and bundles are not printed when the `with_timing` option is disabled.

None of the supported cQASM versions support non-scalar variables or registers, aside from the special-cased main qubit register and corresponding bit register. Therefore, some tricks are needed.

- For non-scalar registers that are expected to be implicitly defined by the target cQASM reader/interpreter, references are printed as a function call, for example `creg(2)` for the integer control register 2.
- For non-scalar variables (including registers when `registers_as_variables` is set), an independent cQASM variable will be printed for every element of the non-scalar OpenQL object, using the name format `<name>_<major>_[...]_<minor>`. For example, the `creg(2)` example above would be printed as `creg_2` if `registers_as_variables` is set. Note that this notation obviously only supports literal indices, and also note that name conflicts may arise in contrived cases (for example, when a scalar variable named `creg_2` was defined in addition to a one-dimensional `creg` variable).

Indices start from 0 in both cases.

Options

output_prefix

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`output_suffix`

Must be any string, default `.cq`. Suffix to use for the output filename.

`cqasm_version`

Must be one of `1.0`, `1.1`, or `1.2`, default `1.2`. The cQASM version to target.

`with_platform`

Must be `yes` or `no`, default `no`. Whether to include an annotation that includes the (preprocessed) JSON description of the platform.

`registers_as_variables`

Must be `yes` or `no`, default `no`. Whether to include variable declarations for registers. This must be enabled if the cQASM file is to be passed to a target that doesn't implicitly define the registers. Note that the size of the main qubit register is always printed for version 1.0, because it can't legally be omitted for that version. Also note that this is a lossy operation if the file is later read by OpenQL again, because register indices are lost (since only scalar variables are supported by cQASM).

`with_statistics`

Must be `yes` or `no`, default `no`. Whether to include the current statistics for each kernel and the complete program in the generated comments.

`with_metadata`

Must be `yes` or `no`, default `yes`. Whether to include metadata supported by the IR but not by cQASM as annotations, to allow the IR to be more accurately reproduced when read again via the cQASM reader pass.

`with_barriers`

Must be one of `no`, `simple`, or `extended`, default `extended`. Whether to include wait and barrier instructions, and if so, using which syntax (see pass description). These are only needed when the program will be fed to another compiler later on.

`with_timing`

Must be `yes` or `no`, default `yes`. Whether to include scheduling/timing information via bundle-and-skip notation.

1.12.14 Sweep points writer

Type name(s): `io.sweep_points.Write`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

Writes a simple JSON file of the following form:

```
{ "measurement_points": [...] }
```

wherein the ellipsis is populated with the contents of the sweep points array specified to the program through the set_sweep_points(). API call. The filename defaults to `<output_prefix>.json`, but this may be overridden using the set_config_file() API call on program.

This pass has no further use and only exists for backward compatibility. It may be removed entirely in a later version of OpenQL.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

1.12.15 Mapper

Type name(s): `map.qubits.Map`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

The purpose of this pass is to ensure that the qubit connectivity constraints are met for all multi-qubit gates in each kernel. This is done by optionally applying a mixed integer linear programming algorithm to look for a perfect solution that does not require routing or figure out a good initial qubit placement, and then by heuristically inserting swap/move gates to change the mapping on the fly as needed. Finally, it decomposes all gates in the circuit to primitive gates.

Note: The substeps of this pass will probably be subdivided into individual passes in the future.

Warning: This pass currently operates purely on a per-kernel basis. Because it may adjust the qubit mapping from input to output, a program consisting of multiple kernels that maintains a quantum state between the kernels may be silently destroyed.

Initial placement

This step attempts to find a single mapping of the virtual qubits of a circuit to the real qubits of the platform's qubit topology that minimizes the sum of the distances between the two mapped operands of all two-qubit gates in the circuit. The distance between two real qubits is the minimum number of swaps that is required to move the state of one of the two qubits to the other. It employs a Mixed Integer Linear Programming (MIP) algorithm to solve this, modelled as a Quadratic Assignment Problem. If enabled, this step may find a mapping that is optimal for the whole circuit, but because its time-complexity is exponential with respect to the size of the circuit, this may take quite some computer time. Also, the result is only really useful when in the mapping found all mapped operands of two-qubit gates are nearest-neighbor (i.e. distance 1). So, there is no guarantee for success: it may take too long and the result may not be optimal.

Note: Availability of this step depends on the build configuration of OpenQL due to license conflicts with the library used for solving the MIP problem. If it is not included, the step is effectively no-op, and a warning message will be printed.

Heuristic routing

This step essentially transforms the program by iterating over its gates from front to back and inserting `swap` or `move` gates when needed. Whenever it does this, it updates its internal virtual to real qubit mapping. While iterating, the virtual qubit indices of the incoming gates are replaced with real qubit indices, i.e. those defined in the topology section of the platform.

Some platforms have gates for which parameters differ based on the qubits they operate on. For example, `cz q0, q1` may have a different duration than `cz q2, q3`, and `cz q0, q2` may not even exist because of topological constraints. However, rules like this make no sense when the cz gate is still using virtual qubit indices: it's perfectly fine for the user to do `cz q0, q2` at the input if the mapper is enabled.

To account for this, the mapper will look for an alternative gate definition when it converts the virtual qubit indices to real qubit indices: specifically, it will look for a gate with `_real` or `_prim` (see also the primitive decomposition step) appended to the original gate name. For example, `cz q0, q2` may, after routing, be transformed to `cz_real q2, q3`. This allows you to define `cz` using a generalized gate definition (i.e. independent on qubit operands), and `cz_real` as a set of specialized gates as required by the platform.

Note: The resolution order is `*_prim`, `*_real`, and finally just the original gate name. Thus, if you don't need this functionality, you don't need to define any `*_real` gates.

Because the mapper inserts swap and/or move gates, it is important that these gates are actually defined in the configuration file (usually by means of a decomposition rule). The semantics for them must be as follows.

- `swap x, y` or `swap_real x, y`: must apply a complete swap gate to the given qubits to exchange their state. If in the final decomposition one of the operands is used before the other, the second operand (`y`) is expected to be used first for the `reverse_swap_if_better` option to work right.
- `move x, y` or `move_real x, y`: if `use_moves` is enabled, the mapper will attempt to use these gates instead of `swap`/`swap_real` if it knows that the `y` qubit is in the `|0>` state (or it can initialize it as such) and the result is better (or not sufficiently worse) than using a normal swap. Such a move gate can be implemented with two CNOTs instead of three.

The order in which non-nearest-neighbor two-qubit gates are routed, the route taken for them, and where along the route the actual two-qubit gate is performed, is determined heuristically. The way in which this is done is controlled by the various options for this pass; it can be made really simple by just iterating over the circuit in the specified order and just choosing a random routing alternative whenever routing is needed, or more intelligent methods can be used at the cost of execution time and memory usage (the latter especially when a lot of alternative solutions are generated before a choice is made). Based on these options, time and space complexity can be anywhere from linear to exponential!

Decomposition into primitives

As a final step, the mapper will try to decompose the "real" gates (i.e. gates with qubit operands referring to real qubits) generated by the previous step into primitive gates, as actually executable by the target architecture. It does this by attempting to suffix the name of each gate with `_prim`. Thus, if you define a decomposition rule named `cz_prim` rather than `cz`, this rule will only be applied after mapping.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`initialize_one_to_one`

Must be `yes` or `no`, default `yes`. Controls whether the mapper should assume that each kernel starts with a one-to-one mapping between virtual and real qubits. When disabled, the initial mapping is treated as undefined.

`assume_initialized`

Must be `yes` or `no`, default `no`. Controls whether the mapper should assume that each qubit starts out as zero at the start of each kernel, rather than with an undefined state.

`assume_prep_only_initializes`

Must be `yes` or `no`, default `no`. Controls whether the mapper may assume that a user-written prepz gate actually leaves the qubit in the zero state, rather than any other quantum state. This allows it to make some optimizations.

`enable_mip_placer`

Must be `yes` or `no`, default `no`. Controls whether the MIP-based initial placement algorithm should be run before resorting to heuristic mapping.

`mip_horizon`

Must be an integer less than or equal to 0, default `0`. This controls how many two-qubit gates the MIP-based initial placement algorithm considers for each kernel (if enabled). If 0 or unspecified, all gates are considered.

`route_heuristic`

Must be one of `base`, `baserc`, `minextend`, `minextendrc`, or `maxfidelity`, default `base`. Controls which heuristic the router should use when selecting between possible routing operations. `base` and `base_rc` are the simplest forms: all routes are considered equally `good`, so the tie-breaking strategy is just applied immediately. `minextend` and `minextendrc` are way more involved (but also take longer to compute): these options will speculate what each option will do in terms of extending the duration of the circuit, optionally recursively, to find the best alternatives in terms of circuit duration within somelookahead window. The existence of the `rc` suffix specifies whether the internal scheduling for fitness determination should be done with or without resource constraints. `maxfidelity` is not supported in this build of OpenQL.

max_alternative_routes

Must be an integer less than or equal to 0, default `0`. Controls the maximum number of alternative routing solutions to generate before applying the heuristic and/or tie-breaking method to choose one. Leave unspecified or set to 0 to disable this limit.

tie_break_method

Must be one of `first`, `last`, `random`, or `critical`, default `random`. Controls how to tie-break equally-scoring alternative mapping solutions. `first` and `last` choose respectively the first and last solution in the list (assuming the qubits have planar coordinates specified in the topology section, `first` selects the left-most alternative with the two-qubit gate near target, and `last` selects the right-most alternative with the two-qubit gate near source; when no coordinates are given the choice is undefined, though deterministic), `random` uses random number generation to select an alternative, and `critical` favors the alternative that maps the most critical gate as determined by the scheduler (if any).

lookahead_mode

Must be one of `no`, `1qfirst`, `noroutingfirst`, or `all`, default `noroutingfirst`. Controls the strategy for selecting the next gate(s) to map. When `no`, just map the gates in the order of the circuit, disregarding commutation as allowed by the circuit's dependency graph. Single-qubit and nearest-neighbor two-qubit gates are mapped trivially; non-nearest-neighbor gates are mapped when encountered by generating alternative routing solutions and picking the best one via `route_heuristic`. For `1qfirst`, the dependency graph is used to greedily map all single-qubit gates, before proceeding with mapping the most critical two-qubit gate. If this gate is not nearest-neighbor, it is routed the same way as for `no`. `noroutingfirst` works the same, but also greedily maps two-qubit gates that don't require any routing regardless of criticality, before routing the most critical non-nearest-neighbor two-qubit gate. Finally, `all` works the same as `noroutingfirst`, but instead of considering only routing alternatives for the most critical non-nearest-neighbor two-qubit gate, alternatives are generated for *all* available non-nearest-neighbor two-qubit gates, thus ignoring criticality and relying only on `route_heuristic` (which may be better depending on the heuristic chosen, but will cost execution time).

path_selection_mode

Must be one of `all`, `borders`, or `random`, default `all`. Controls whether to consider all paths from a source to destination qubit while routing, or to favor routing along the route search space. The latter is only supported and sensible when the qubits are given planar coordinates in the topology section of the platform configuration file. Both `all` and `random` consider all paths, but for the latter the order in which the paths are generated is shuffled, which is useful to reduce bias when `max_alternative_routes` is used.

swap_selection_mode

Must be one of `one`, `all`, or `earliest`, default `all`. This controls how routing interacts with speculation. When `all`, allswaps for a particular routing option are committed immediately, before trying anything else. When `one`, only the first swap in the route from source to target qubit is committed. When `earliest`, the swap that can be done at the earliest point is selected, which might be the one swapping the source or target qubit.

`recurse_on_nn_two_qubit`

Must be `yes` or `no`, default `no`. When a nearest-neighbor two-qubit gate is the next gate to be mapped, this controls whether the mapper will speculate on adding it now or later, or if it will add it immediately without speculation. NOTE: this is an advanced/unstable option that influences `lookahead_mode` in a complex way; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

`recursion_depth_limit`

Must be an integer less than or equal to 0 or inf, default `0`. Controls the maximum recursion depth while searching for alternative mapping solutions. NOTE: this is an advanced/unstable option; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

`recursion_width_factor`

Must be an real number less than or equal to 0, default `1`. Limits how many alternative mapping solutions are considered as a factor of the number of best-scoring alternatives, rounded up. NOTE: this is an advanced/unstable option; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

`recursion_width_exponent`

Must be an real number between 0 and 1 inclusive, default `1`. Adjustment for recursion_width_factor based on the current recursion depth. For each additional level of recursion, the effective width factor is multiplied by this number. NOTE: this is an advanced/unstable option; don't use it unless you know what you're doing. May be removed or changed in a later version of OpenQL.

`use_moves`

Must be an integer less than or equal to 0 or one of `no` or `yes`, default `yes`. Controls if/when the mapper inserts move gates rather than swap gates to perform routing. If `no`, swap gates are always used. Otherwise, a move gate is used if the other qubit has been initialized, or if initializing it only extends the circuit by the given number of cycles. `yes` implies this limit is 0 cycles.

`reverse_swap_if_better`

Must be `yes` or `no`, default `yes`. Controls whether the mapper will reverse the operands for a swap gate when reversal improves the schedule. NOTE: this currently assumes that the second qubit operand of the swap gate decomposition in the platform configuration file is used before than the first operand; if this is not the case, enabling this will worsen the routing result rather than improve it.

commute_multi_qubit

Must be `yes` or `no`, default `no`. Whether to consider commutation rules for the CZ and CNOT quantum gates.

commute_single_qubit

Must be `yes` or `no`, default `no`. Whether to consider commutation rules for single-qubit X and Z rotations.

scheduler_heuristic

Must be one of `path_length` or `random`, default `path_length`. This controls what scheduling heuristic should be used for ordering the list of available gates by criticality.

write_dot_graphs

Must be `yes` or `no`, default `no`. Whether to print dot graphs of the schedules created using the embedded scheduler.

1.12.16 Clifford gate optimizer

Type name(s): `opt.clifford.Optimize`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

This pass tries to minimize sequences of single-qubit gates in the Clifford C1 set to their minimal counterpart in terms of cycles. The pass returns the total number of cycles saved by this optimization per qubit.

Note that the relation between the Clifford state transition corresponding to a particular gate is currently hardcoded based on gate name, and the equivalent cycle counts are also hardcoded.

Options

output_prefix

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of no, yes, stats, qasm, or both, default no. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to no to disable this functionality or to yes to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix _debug_[in|out].ir for the IR dump, and _debug_[in|out].cq for the cQASM file. The option values stats, cqasm, and both are used for backward compatibility with the write_qasm_files and write_report_files global options; for stats and both a statistics report file is written with suffix _[in|out].report, and for qasm and both a cQASM file is written (without stats in the comments) with suffix _[in|out].qasm.

1.12.17 List scheduler

Type name(s): sch.ListSchedule.

This pass analyzes the data dependencies between statements and applies quantum cycle numbers to them using optionally resource-constrained ASAP or ALAP list scheduling. All blocks in the program are scheduled independently.

Options

output_prefix

Must be any string, default %N.%P. Format string for the prefix used for all output products. %n is substituted with the user-specified name of the program. %N is substituted with the optionally uniquified name of the program. %p is substituted with the local name of the pass within its group. %P is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). %U is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,%D is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

debug

Must be one of no, yes, stats, qasm, or both, default no. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to no to disable this functionality or to yes to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix _debug_[in|out].ir for the IR dump, and _debug_[in|out].cq for the cQASM file. The option values stats, cqasm, and both are used for backward compatibility with the write_qasm_files and write_report_files global options; for stats and both a statistics report file is written with suffix _[in|out].report, and for qasm and both a cQASM file is written (without stats in the comments) with suffix _[in|out].qasm.

`resource_constraints`

Must be `yes` or `no`, default `yes`. Whether to respect or ignore resource constraints when scheduling.

`scheduler_target`

Must be one of `asap` or `alap`, default `alap`. Which scheduling target is to be used; ASAP schedules all statements as soon as possible, while ALAP starts from the last statement and schedules all statements as late as possible. ALAP is best for most simple quantum circuits, because the measurements at the end will be done in parallel if possible, and state initialization is postponed as much as possible to reduce state lifetime.

`scheduler_heuristic`

Must be one of `none`, `critical_path`, or `deep_criticality`, default `deep_criticality`. This controls what heuristic is used to sort the list of statements available for scheduling. `none` effectively disables sorting; the available statements will be scheduled in the order in which they were specified in the original program. `critical_path` schedules the statement with the longest critical path first. `deep_criticality` is the same except for statements with equal critical path length; in this case, the deep-criticality of the most critical successor is recursively checked instead.

`commute_multi_qubit`

Must be `yes` or `no`, default `no`. Whether to consider commutation rules for multi-qubit gates.

`commute_single_qubit`

Must be `yes` or `no`, default `no`. Whether to consider commutation rules for single-qubit gates.

`max_resource_block_cycles`

Must be an integer less than or equal to 0, default `10000`. The maximum number of cycles to wait for the resource constraints to unblock a statement when there is nothing else to do. This is used for deadlock detection. It should just be set to a high number, or can be set to 0 to disable deadlock detection (but then the scheduler might end up in an infinite loop).

`write_dot_graphs`

Must be `yes` or `no`, default `no`. Whether to emit a graphviz dot graph representation of the data dependency graph and schedule of each block. The emitted files will use suffix `_<block-name>.dot`, where `<block-name>` is a uniquified name for each block.

1.12.18 Scheduler

Type name(s): `sch.Schedule`.

Note: This is a legacy pass, operating on the old intermediate representation. If the program is using features that the old IR does not support when this pass is run, an internal compiler error will be thrown. Furthermore, kernel/block names may change regardless of whether the pass does anything with them, due to name uniquification logic.

Legacy list scheduler that operates on the old IR. This pass is deprecated in favor of `sch.ListSchedule`, and may be replaced with a wrapper around that in future version.

The pass analyzes the data dependencies between gates and applies cycle numbers to them based on some scheduling heuristic. Depending on options, the scheduler will either be resource-constrained or will ignore resources.

Options

`output_prefix`

Must be any string, default `%N.%P`. Format string for the prefix used for all output products. `%n` is substituted with the user-specified name of the program. `%N` is substituted with the optionally uniquified name of the program. `%p` is substituted with the local name of the pass within its group. `%P` is substituted with the fully-qualified name of the pass, using periods as hierarchy separators (guaranteed unique). `%U` is substituted with the fully-qualified name of the pass, using underscores as hierarchy separators. This may not be completely unique,`%D` is substituted with the fully-qualified name of the pass, using slashes as hierarchy separators. Any directories that don't exist will be created as soon as an output file is written.

`debug`

Must be one of `no`, `yes`, `stats`, `qasm`, or `both`, default `no`. May be used to implicitly surround this pass with cQASM/report file output printers, to aid in debugging. Set to `no` to disable this functionality or to `yes` to write a tree dump and a cQASM file before and after, the latter of which includes statistics as comments. The filename is built using the output_prefix option, using suffix `_debug_[in|out].ir` for the IR dump, and `_debug_[in|out].cq` for the cQASM file. The option values `stats`, `cqasm`, and `both` are used for backward compatibility with the `write_qasm_files` and `write_report_files` global options; for `stats` and `both` a statistics report file is written with suffix `_[in|out].report`, and for `qasm` and `both` a cQASM file is written (without stats in the comments) with suffix `_[in|out].qasm`.

`resource_constraints`

Must be `yes` or `no`, default `yes`. Whether to respect or ignore resource constraints when scheduling.

`scheduler_target`

Must be one of `asap`, `alap`, or `uniform`, default `alap`. Which scheduling target is to be used; ASAP schedules all gates as soon as possible, ALAP starts from the last gate and schedules all gates as late as possible, and uniform tries to smoothen out the amount of parallelism throughout each kernel. Uniform scheduling is only supported without resource constraints. ALAP is best for most simple quantum circuits, because the measurements at the end will be done in parallel if possible, and state initialization is postponed as much as possible to reduce state lifetime.

`scheduler_heuristic`

Must be one of `path_length` or `random`, default `path_length`. This controls what scheduling heuristic should be used for ordering the list of available gates by criticality.

`commute_multi_qubit`

Must be `yes` or `no`, default `no`. Whether to consider commutation rules for the CZ and CNOT quantum gates.

`commute_single_qubit`

Must be `yes` or `no`, default `no`. Whether to consider commutation rules for single-qubit X and Z rotations.

`write_dot_graphs`

Must be `yes` or `no`, default `no`. Whether to emit a graphviz dot graph representation of the schedule of the kernel. The emitted file will use suffix `_<kernel>.dot`.

1.13 Supported resources

This section lists the scheduler resource types currently supported by OpenQL.

Roughly speaking, resources control whether two (or more) quantum gates may execute in parallel, and under what conditions. The most obvious one is that two quantum gates operating on the same qubit physically cannot be executed at the same time, but quantum chips typically have more subtle constraints as well. For example, execution of an X gate on one qubit may require generation of a particular waveform by a waveform generator shared between a number of qubits; in this case, it might be possible to do an X gate on another qubit in parallel, but not a Y gate.

Resources are of course used by the (resource-constrained) scheduler, but other passes may also make use of them. For example, the mapper uses them in its heuristic routing algorithm to try to overlap swaps with the rest of the circuit as much as possible, in such a way that resource constraints are not violated.

1.13.1 Resource specification

Resources are specified using the `"resources"` section of the platform configuration file. Two flavors are supported for its contents: one for compatibility with older platform configuration files, and one extended structure. The extended structure has the following syntax.

```
"resources": {
    "architecture": <optional string, default "">,
    "dnu": <optional list of strings, default []>,
    "resources": {
        "<name>": {
            "type": "<type>",
            "config": {
                <optional configuration>
            }
        }
        ...
    }
}
```

The optional `"architecture"` key may be used to make shorthands for architecture- specific resources, normally prefixed with `"arch.<architecture>."`. If it's not specified or an empty string, the architecture is derived from the `"eqasm_compiler"` key.

The optional `"dnu"` key may be used to specify a list of do-not-use resource types (experimental, deprecated, or any other resource that's considered unfit for "production" use) that you explicitly want to use, including the "dnu" namespace they are defined in. Once specified, you'll be able to use the resource type without the `"dnu"` namespace element. For example, if you would include `"dnu.whatever"` in the list, the resource type `"whatever"` may be used to add the resource.

The `"resources"` subkey specifies the actual resource list. This consists of a map from unique resource names matching `[a-zA-Z0-9_\-]+` to a resource configuration. The configuration object must have a `"type"` key, which must identify a resource type that OpenQL knows about; the type names are listed in the sections below. The `"config"` key is optional, and is used to pass type-specific configuration data to the resource. If not specified, an empty JSON object will be passed to the resource instead.

If the `"resources"` **sub**key is not present, the old structure is used instead. This has the following, simpler form:

```
"resources": {
    "<type>": {
        <configuration>
    },
    ...
}
```

This is limited to one resource per type alias. The names for the resources are inferred, and the architecture namespace is in this case always based upon the contents of the `"eqasm_compiler"` key.

1.13.2 Instrument resource

Type names: `Instrument`.

This resource models an instrument or group of instruments that is needed to apply a certain kind of quantum gate, with the constraint that the instrument is shared between a number of qubits/edges, and can only perform one function at a time. That is, two gates that share an instrument can be parallelized if and only if they use the same instrument function. By default, parallel gates requiring the same instrument also need to start at the same time and have the same duration, but this can be disabled.

The instrument function is configurable for each particular gate based on one or more custom keys in the instruction set definition of the platform configuration file. It's also possible to specify that there is only a single function (i.e., all gates requiring access to the instrument can be parallelized, but only if they start at the same time and have the same duration), or to specify that all functions are mutually exclusive (in which case gates using the same instrument can never be parallelized).

The instrument(s) affected by the gate, if any, are selected based on the qubit operands of the gate and upon whether the gate matches a set of predicates. Like the instrument function selection, the predicates are based on custom keys in the instruction definition in the platform configuration file. A different set of predicates can be provided based on the number of qubit operands of the gate.

Configuration structure

The shared instrument resource is configured using the following JSON structure.

```
{
    "predicate": {
        "<gate-key>": ["<value>", ...],
        ...
    },
    "predicate_1q": ...,
    "predicate_2q": ...,
    "predicate_nq": ...,
    "function": [
        "<gate-key>",
        ...
    ],
    "allow_overlap": [true, false],
    "instruments": [
        {
            "name": "<optional instrument name>",
            "qubit":     [<qubits>],
            "edge":      [<edges>],
            "1q_qubit":  [<qubits>],
            "2q_qubit0": [<qubits>],
            "2q_qubit1": [<qubits>],
            "nq_qubit0": [<qubits>],
            "nq_qubit1": [<qubits>],
            "nq_qubitn": [<qubits>]
        }
    ]
}
```

All sections except `"instruments"` are optional. Unrecognized sections throw an error.

Predicates

The predicate section must be a map of string-string or string-list(string) key-value pairs, representing (custom) keys and values in the instruction set definition section for the incoming gate that must be matched. An incoming gate matches the predicate if and only if:

- its instruction set definition object has values for all keys specified;
- these keys all map to strings; and
- the string values match (one of) the specified value(s) for each key.

For example, if an instruction definition looks like this:

```
"x": {
    "duration": 40,
    "type": "mw",
    "instr": "x"
}
```

the following predicate configuration will match it:

```
"predicate": {
    "type": "mw"
}
```

but will reject a gate defined like this:

```
"cnot": {
    "duration": 80,
    "type": "flux",
    "instr": "cnot"
}
```

because its `"type"` is `"flux"`.

Note: It will also silently reject gates which don't have the `"type"` key, so beware of typos!

Should you want to match both types, but not any other type, you could do

```
"predicate": {
    "type": ["mw", "flux"]
}
```

Different predicates can be specified for single-, two-, and more-than-two-qubit gates. Both the common predicate and the size-specific predicate must match.

Instrument function selection

The function section can be one of three things:

- a list of gate keys as specified in the structure above, in which case the selected function is the combination of the (string) values of these keys in the gate definition;
- an empty list or unspecified, in which case function matching is disabled, always allowing two gates to execute in parallel (but still requiring them to start and end simultaneously); or
- the string `"exclusive"`, in which case exclusive access is modelled, i.e. matching gates can never execute in parallel.

For example, say that an `x` gate requires a different instrument function than an `y` gate, i.e. they cannot be done in parallel, but multiple `x` gates on different qubits can be parallelized (idem for `y`), you might use:

```
"function": [
    "instr"
]
```

for gates defined as follows:

```
"x": {
    "duration": 40,
    "type": "mw",
    "instr": "x"
},
"y": {
    "duration": 40,
    "type": "mw",
    "instr": "y"
}
```

In some cases, it is not necessary for parallel operations requiring the same instrument function to actually start at the same time. For example, an instrument resource modelling qubit detuning would have two states, one indicating that the qubit is detuned, and one indicating that it is not, but as long as it is in one of these states, it doesn't matter when gates requiring it start and end. This behavior can be specified with the `"allow_overlap"` option. If specified, it must be a boolean, defaulting to `false`.

Note: It makes little sense to combine `"allow_overlap"` with an empty `"function"` section, because this would disable all constraints on parallelism.

Instrument definition

A single instrument resource can define multiple independent instruments with the same behavior, distinguished by which qubits or edges they're connected to. This is done using the instruments section. It consists of a list of objects, where each object represents an instrument. The contents of the object define its connectivity, by way of predicates on the qubit operand list of the incoming gates.

- For single-qubit gates, the instrument is used if and only if the single qubit operand is in the `"1q_qubit"` or `"qubit"` list.
- For two-qubit gates, the instrument is used if any of the following is true:
  - either qubit is in the `"qubit"` list;
  - the first qubit operand is in the `"2q_qubit0"` list;

  - the second qubit operand is in the "2q_qubit1" list; or
  - the edge index (from the topology section of the platform) corresponding to the qubit operand pair is in the "edge" list.

- For three-or-more-qubit gates, the instrument is used if any of the following is true:
  - any of the qubit operands is in the "qubit" list;
  - the first qubit operand is in the "nq_qubit0" list;
  - the second qubit operand is in the "nq_qubit1" list; or
  - any of the remaining qubit operands are in the "nq_qubitn" list.

For example, an instrument defined as follows:

```
"instruments": [
    {
        "name": "qubit-2",
        "1q_qubit": [2],
        "edge": [1, 9]
    }
]
```

will be used by single-qubit gates acting on qubit 2, and by two-qubit gates acting on edge 1 or 9 (as defined in the topology section of the platform configuration file). Of course the gate also has to match the gate predicates defined for this resource for it to be considered. When a gate matches all of the above, the instrument function as defined in the function section will (need to) be reserved for this instrument for the duration of that gate, and thus the gate will be postponed if a conflicting reservation already exists.

QWG example

In CC-light, single-qubit rotation gates (instructions with "type": "mw") are controlled by QWGs. Each QWG controls a particular set of qubits. It can control multiple qubits at a time, but only when they perform the same gate (configured using "cc_light_instr") and start at the same time. There are three QWGs, the first of which is connected to qubits 0 and 1, the second one to qubits 2, 3, and 4, and the third to qubits 5 and 6.

This can be modelled with the following configuration:

```
{
    "predicate": { "type": "mw" },
    "function": [ "cc_light_instr" ],
    "instruments": [
        {
            "name": "QWG0",
            "qubit": [0, 1]
        },
        {
            "name": "QWG1",
            "qubit": [2, 3, 4]
        },
        {
            "name": "QWG2",
            "qubit": [5, 6]
        }
    ]
}
```

Before resources were generalized, QWGs were modelled with a specific resource type. Its configuration structure instead looked like this for the same thing:

```
{
    "count": 3,
    "connection_map": {
        "0" : [0, 1],
        "1" : [2, 3, 4],
        "2" : [5, 6]
    }
}
```

For backward compatibility, this structure is desugared to the complete structure when the instrument resource is constructed using `arch.cc_light.qwgs`.

Measurement unit example

In CC-light, single-qubit measurements (instructions with `"type": "readout"`) are controlled by measurement units. Each one controls a private set of qubits. A measurement unit can control multiple qubits at the same time, but only when they start at the same time. There were two measurement units, the first of which being connected to qubits 0, 2, 4, 5, and 6, and the second to qubits 1 and 4.

This can be modelled with the following configuration:

```
{
    "predicate": { "type": "readout" },
    "instruments": [
        {
            "name": "MEAS0",
            "qubit": [0, 2, 4, 5, 6]
        },
        {
            "name": "MEAS1",
            "qubit": [1, 4]
        }
    ]
}
```

Before resources were generalized, this was modelled with a specific resource type. Its configuration structure instead looked like this for the same thing:

```
{
    "count": 2,
    "connection_map": {
        "0" : [0, 2, 3, 5, 6],
        "1" : [1, 4]
    }
}
```

For backward compatibility, this structure is desugared to the complete structure when the instrument resource is constructed using `arch.cc_light.meas_units`.

Qubit detuning example

In CC-light, two-qubit flux gates (`"type": "flux"`) for particular edges require a third qubit to be detuned (also known as parked). This moves the qubit out of the way frequency-wise to prevent it from being affected by the flux gate as well. However, doing so prevents single-qubit microwave quantum gates (`"type": "mw"`) from using the detuned qubit (and vice versa). Specifically:

- a two-qubit gate operating on qubits 0 and 2 detunes qubit 3;
- a two-qubit gate operating on qubits 0 and 3 detunes qubit 2;
- a two-qubit gate operating on qubits 1 and 3 detunes qubit 4;
- a two-qubit gate operating on qubits 1 and 4 detunes qubit 3;
- a two-qubit gate operating on qubits 3 and 5 detunes qubit 6; and
- a two-qubit gate operating on qubits 3 and 6 detunes qubit 5.

This can be modelled with the following configuration:

```
{
    "predicate_1q": { "type": "mw" },
    "predicate_2q": { "type": "flux" },
    "function": [ "type" ],
    "allow_overlap": true,
    "instruments": [
        {
            "name": "qubit-2",
            "1q_qubit": [2],
            "edge": [1, 9]
        },
        {
            "name": "qubit-3",
            "1q_qubit": [3],
            "edge": [0, 3, 8, 11]
        },
        {
            "name": "qubit-4",
            "1q_qubit": [4],
            "edge": [2, 10]
        },
        {
            "name": "qubit-5",
            "1q_qubit": [5],
            "edge": [6, 14]
        },
        {
            "name": "qubit-6",
            "1q_qubit": [6],
            "edge": [5, 13]
        }
    ]
}
```

Before resources were generalized, this was modelled with a specific resource type. Its configuration structure instead looked like this for the same thing:

```
{
    "count": 7,
```

(continues on next page)

(continued from previous page)

```
    "connection_map": {
        "0": [3],
        "1": [2],
        "2": [4],
        "3": [3],
        "4": [],
        "5": [6],
        "6": [5],
        "7": [],
        "8": [3],
        "9": [2],
        "10": [4],
        "11": [3],
        "12": [],
        "13": [6],
        "14": [5],
        "15": []
    }
}
```

To be specific, the connection map mapped each edge index to a list of qubits detuned by doing a flux gate on that edge. For backward compatibility, this structure is desugared to the complete structure when the instrument resource is constructed using `arch.cc_light.detuned_qubits`.

Mutably-exclusive edge example

The above qubit detuning has additional implications for two-qubit gates. Specifically, the following edges are mutably exclusive:

- q0-q2 (edge 0 and 8) is mutually-exclusive with q1-q3 (edge 2 and 10);
- q0-q3 (edge 1 and 9) is mutually-exclusive with q1-q4 (edge 3 and 11);
- q2-q5 (edge 4 and 12) is mutually-exclusive with q3-q6 (edge 6 and 14); and
- q3-q5 (edge 5 and 13) is mutually-exclusive with q4-q6 (edge 7 and 15).

```
{
    "predicate": { "type": "flux" },
    "function": "exclusive",
    "instruments": [
        {
            "name": "edge-0_2-1_3",
            "edge": [0, 2, 8, 10]
        },
        {
            "name": "edge-0_3-1_4",
            "edge": [1, 3, 9, 11]
        },
        {
            "name": "edge-2_5-3_6",
            "edge": [4, 6, 12, 14]
        },
        {
            "name": "edge-3_5-4_6",
            "edge": [5, 7, 13, 15]
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Before resources were generalized, this was modelled with a specific resource type. Its configuration structure instead looked like this for the same thing:

```
{
    "count": 16,
    "connection_map": {
        "0": [2, 10],
        "1": [3, 11],
        "2": [0, 8],
        "3": [1, 9],
        "4": [6, 14],
        "5": [7, 15],
        "6": [4, 12],
        "7": [5, 13],
        "8": [2, 10],
        "9": [3, 11],
        "10": [0, 8],
        "11": [1, 9],
        "12": [6, 14],
        "13": [7, 15],
        "14": [4, 12],
        "15": [5, 13]
    }
}
```

Here, the keys and values in `"connection_map"` specify edge indices, where usage of the edge in key indicates that the edge indices it maps to can no longer be used. Note that the i+8 edge is not in the second list because it maps to the inversed-direction edge, which is already excluded by mutual exclusion of the qubits themselves. For backward compatibility, this structure is desugared to the complete structure when the instrument resource is constructed using `arch.cc_light.edges`.

1.13.3 Multi-core channel resource

Type names: `InterCoreChannel`.

This resource models inter-core communication with limited connectivity between cores. This is modelled as follows.

Each core has a limited number of channels, with which it can connect to other cores. The connectivity between the channels of each core is assumed to be fully connected, but the number of channels per core can be adjusted. Gates matching the predicate (if any) use one of the available core/channel pairs for each core that they use (communication) qubits of. The core is of course determined by the qubit index, but the channel is undefined; the resource will use the first available channel.

The resource is configured using the following structure.

```
{
    "predicate": {
        "<gate-key>": ["<value>", ...],
        ...
    },
    "predicate_1q": ...,
    "predicate_2q": ...,
```

(continues on next page)

(continued from previous page)

```
    "predicate_nq": ...,
    "inter_core_required": <boolean, default true>,
    "communication_qubit_only": <boolean, default false>,
    "num_channels": <number of channels per core, default 1>
}
```

Note that the number of cores and communication qubits per core are configured in the topology section of the platform JSON configuration data. These settings are not repeated here.

The predicate section must be a map of string-string or string-list(string) key-value pairs, representing (custom) keys and values in the instruction set definition section for the incoming gate that must be matched. An incoming gate matches the predicate if and only if:

- its instruction set definition object has values for all keys specified;
- these keys all map to strings; and
- the string values match (one of) the specified value(s) for each key.

Different predicates can be specified for single-, two-, and more-than-two-qubit gates. Both the common predicate and the size-specific predicate must match.

Furthermore, gates can be predicated based on whether they actually use qubits from multiple cores. This is controlled by `"inter_core_required"`. When set or unspecified, a gate must operate on qubits belonging to at least two different cores to match the predicate. Otherwise, this is not required. That is, a gate matching the other predicates that uses only qubits from one core would still use channel resources for that core. This is mostly for compatibility with the original channel resource, which didn't check for this.

The `"communication_qubit_only"` flag controls whether all qubits of a gate use communication channel resources, or whether only qubits marked as communication qubits are considered.

Finally, the `"num_channels"` key specifies how many independent channels each core has. The default and minimum value is 1.

1.13.4 Qubit resource

Type names: `Qubit` or `arch.cc_light.qubits`.

This resource ensures that a qubit is only ever in use by one gate at a time.

Note: It assumes that a gate with a qubit operand actually uses this operand for its entire duration, so it may be overly pessimistic.

This resource does not have any JSON configuration options. Historically it had a "count" key specifying the number of qubits, but this is now taken from the platform's qubit count. Any JSON options that are passed anyway are silently ignored.

1.13.5 CC-light channels resource

Type names: `arch.cc_light.channels`.

Compatibility wrapper for the CC-light channels resource. This does exactly the same thing as the InterCoreChannel resource, but accepts the following configuration structure:

```
{
    "count": <number of channels>
}
```

This structure is converted to the following for use with the InterCoreChannel resource:

```
{
    "predicate": { "type": "extern" },
    "inter_core_required": false,
    "communication_qubit_only": false,
    "num_channels": <taken from "count">
}
```

1.13.6 CC-light detuned_qubits resource

Type names: `arch.cc_light.detuned_qubits`.

Compatibility wrapper for the CC-light detuned qubit resource. This is the same resource type as `Instrument`, but accepting a different JSON configuration structure for backward compatibility. Refer to the documentation of the `Instrument` resource for more information; everything is explained there.

1.13.7 CC-light edges resource

Type names: `arch.cc_light.edges`.

Compatibility wrapper for the CC-light edge resource. This is the same resource type as `Instrument`, but accepting a different JSON configuration structure for backward compatibility. Refer to the documentation of the `Instrument` resource for more information; everything is explained there.

1.13.8 CC-light meas_units resource

Type names: `arch.cc_light.meas_units`.

Compatibility wrapper for the CC-light measurement unit resource. This is the same resource type as `Instrument`, but accepting a different JSON configuration structure for backward compatibility. Refer to the documentation of the `Instrument` resource for more information; everything is explained there.

1.13.9 CC-light qwgs resource

Type names: `arch.cc_light.qwgs`.

Compatibility wrapper for the CC-light QWG resource. This is the same resource type as `Instrument`, but accepting a different JSON configuration structure for backward compatibility. Refer to the documentation of the `Instrument` resource for more information; everything is explained there.

1.14 Where to begin

So you want to contribute to OpenQL? Or perhaps you're employed to help maintain it? Great!

OpenQL has grown to be quite a large project, so you may be feeling a bit overwhelmed. I know I was. I'm assuming you already know what OpenQL is when you get here, otherwise please read through the user documentation first. But after that, where to begin. . . ?

First of all, you should make sure that you're able to build and test OpenQL on your own machine. So follow the *Build instructions*, and if you run into any problems, ask an existing maintainer or open an issue.

If you'll be touching the Python API, you'll also want to follow the instructions for building the documentation locally; there's all kinds of generation magic from the API docstrings and documentation getters that might fall over if you change the wrong thing, and documentation generation is not currently tested by CI.

Once done, you'll want to get some sort of IDE configured, so you can click through the code. I use CLion; they have free educational licenses for anyone with a university e-mail address, and it works okay.

Before changing anything, please read through the *whole* section on *C++ coding conventions* or `CONTRIBUTING.md` (the content is the same). This section describes more than just what should be capitalized and whether braces go on the same or the next line for consistency; it also goes over the general organization of the code, how to include things to make sure everything works everywhere, and what rules need to be followed with regards to the documentation `dump_*()` functions in order for the reStructuredText generators in `docs/` to keep working.

Familiarize yourself with what's available in the `ql::utils` namespace. This was added as a wrapper around the C++ standard library to offer additional runtime safety, improve type naming consistency with the non-STL types defined by OpenQL itself, and improve debugging. Depending on how used you are to C++ programming, you'll probably either love it, hate it, or both. But please, *please* use it anyway, to keep OpenQL's codebase consistent.

In general, please think twice if you feel the need to type `std::` or include a standard library header directly. Most things are wrapped (although it's virtually impossible to be complete).

Avoid adding new native dependencies. If you *really* need to, the build system should be made smart enough for things to work out of the box even if the dependency is not installed: your additions should automatically be disabled if they can't be built, but the rest of OpenQL can. You can do this via preprocessor macros, but be aware that you can only use those in `src`! Files in `include` are public, and can thus be built with any preprocessor macro set when included by user C++ code. You can look at the code for unitary decomposition, MIP-based placement, or the visualizer if you're not sure how to work with these constraints; those pieces of code all do this.

When you've added something, don't forget to add yourself to `CONTRIBUTORS.md`!

These were just some general pointers I came up with on a whim, so this is most likely not complete. If you feel like something is missing, feel free to add to this list!

1.15 Build instructions

This page documents how OpenQL and its documentation pages can be built and installed from scratch.

Note: It is very difficult to maintain these instructions, due to there being so many supported environments, and due to externally-maintained dependencies. Therefore, please let the OpenQL maintainers know if you run into any difficulties with these instructions. If you're a new maintainer, update them accordingly via a PR, but be mindful that something that works on your machine might not work on everyone's machine!

1.15.1 Dependencies

The following packages are required to compile OpenQL from sources:

- a C++ compiler with C++11 support (Linux: gcc, MacOS: LLVM/clang, Windows: MSVC 2015 with update 3 or above)
- git
- flex > 2.6
- bison > 3.0
- cmake >= 3.0
- swig (Linux: >= 3.0.12, Windows: >= 4.0.0)
- **Python 3.x + pip, with the following packages:**
  - `plumbum`
  - `wheel`
  - [Optional] `pytest` (for testing)
  - [Optional] `numpy` (for testing)
  - [Optional] `libqasm` (for testing)
  - [Optional] `sphinx==3.5.4` (for documentation generation)
  - [Optional] `sphinx-rtd-theme` (for documentation generation)
  - [Optional] `m2r2` (for documentation generation)
- [Optional] Doxygen (for documentation generation)
- [Optional] Graphviz Dot utility (to convert graphs from dot to pdf, png etc)
- [Optional] XDot (to visualize generated graphs in dot format)
- [Optional] GLPK (if you want initial placement support)
- [Optional] make (required for documentation generation; other CMake backends can be used for everything else)
- [Optional, MacOS only] XQuartz (only if you want to use the visualizer)

Note: The connection between Sphinx' and SWIG's autodoc functionalities is very iffy, but aside from tracking everything manually or forking SWIG there is not much that can be done about it. Because of this, not all Sphinx versions will build correctly, hence why the Sphinx version is pinned. Sphinx 4.x for example crashes on getting the function signature of property getters/setters.

Windows-specific instructions

Note: The current maintainers of OpenQL all use either Linux or MacOS. While we've checked that these instructions should work on a clean Windows install, things may go out of date. Please let us know if you encounter difficulties with these instructions.

Dependencies can be installed with:

- win_flex_bison 2.5.20
- cmake 3.15.3
- swigwin 4.0.0

Make sure the above mentioned binaries are added to the system path.

For initial placement support, you'll also need winglpk 4.6.5. But just adding this directory to the system path is not enough for CMake to find it. Instead, the toplevel CMake script listens to the `WINGLPK_ROOT_DIR` environment variable. Set that to the root directory of what's in that zip file instead.

Alternatively, you can use Chocolatey to install packages. This is how CI currently does it. They just chain to sourceforge downloads, though.

The actual build and install should be done with PowerShell, for which some modifications (may?) need to be made first.

- Use Power Shell for installation
- Set execution policy by:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

- Install PowerShell Community Extensions:

```
Install-Module -AllowClobber -Name Pscx -RequiredVersion 3.2.2
```

- MSVC 2015 should be added to the path by using the following command:

```
Invoke-BatchFile "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\vcvarsall.bat
↪" amd64
```

- but when you installed Microsoft Visual Studio Community Edition do:

```
Invoke-BatchFile "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\
↪Auxiliary\Build\vcvarsall.bat" amd64
```

- To make your life easier, you can add this command to the profile you are using for power shell, avoiding the need to manually run this command every time you open a power shell. You can see the path of this profile by *echo $PROFILE*. Create/Edit this file to add the above command.
- Python.exe, win_flex.exe, win_bison.exe and swig.exe should be in the path of power shell. To test if swig.exe is the path, run:

```
Get-Command swig
```

- To show the currently defined environment variables do:

```
Gci env:
```

- Make sure the following variables are defined:
  - PYTHON_INCLUDE (should point to the directory containing Python.h)
  - PYTHON_LIB (should point to the python library pythonXX.lib, where XX is for the python version number)
- To set an environment variable in an expression use this syntax:

```
$env:EnvVariableName = "new-value"
```

MacOS-specific instructions

Note: These instructions have not been carefully vetted. If you run into issues, please let the maintainers know.

All dependencies can be installed using Homebrew and pip:

```
brew update
brew install llvm flex bison cmake swig python3 doxygen graphviz glpk xquartz
pip3 install wheel plumbum pytest numpy sphinx==3.5.4 sphinx-rtd-theme m2r2
```

Make sure the above mentioned binaries are added to the system path in front of `/usr/bin`, otherwise CMake finds the default versions.

Linux-specific instructions

Honestly, if you're already used to developing on Linux, and you're using a self-respecting Linux distribution, you should have no problems installing these dependencies. None of them are particularly special, so they should all be available in your package manager.

If you're for some reason using CentOS, you'll need to use a `devtoolset` compiler, because the one shipped with it is too old. Likewise, CentOS ships with cmake 2.9 installed in `/usr/bin` and depends on this; while `cmake3` is in the package manager, you actually need to call `cmake3` instead of `cmake`, which `setup.py` is not smart enough for. On CentOS or other batteries-not-included systems you might also have to compile some dependencies manually (`swig`, `flex`, `bison`, and their dependencies `m4` and possibly `gettext`), but they shouldn't give you too much drama. `cmake` has distro-agnostic binary distributions on github that are a only `wget` and `tar xzv` away. `glpk` might be a bigger issue; I haven't tried.

1.15.2 Obtaining OpenQL

OpenQL sources for each release can be downloaded from github releases as .zip or .tar.gz archive. OpenQL can also be cloned by:

```
git clone https://github.com/QE-Lab/OpenQL.git --recursive
```

Note the `--recursive`: the repository depends on various submodules. If you forgot the `--recursive`, you can get/synchronize them later with `git submodule update --init --recursive`.

1.15.3 Building the `qutechopenql` Python package

Running the following command in a terminal/Power Shell from the root of the OpenQL repository should install the `qutechopenql` package:

```
pip install -v .
```

Or in editable mode by the command:

```
pip install -v -e .
```

Editable mode has the advantage that you'll get incremental compilation if you ever change OpenQL's C++ files, but it's a bit more fragile in that things will break if you move the OpenQL repository around later. Specifically, editable mode just installs an absolute path link to your clone of the OpenQL repository, so if you move it, the link breaks. You'd have to remember to uninstall if you ever end up moving it.

Note: Depending on your system configuration, you may need to use `pip3`, `python -m pip` or `python3 -m pip` instead of `pip`. You may also need to add `--user` to the flags or prefix `sudo`. An exhaustive list of which is needed when is out of scope here; instead, just look for pip usage instructions for your particular operating system online. This works the same for any other Python package.

Warning: NEVER install with `python3 setup.py install` (or similar) directly! This always leads to all kinds of confusion, because `setuptools` does not inform `pip` that the package is installed, allowing `pip` to go out of sync.

Note: The `setup.py` script (as invoked by pip in the above commands, again, do not invoke it directly!) listens to a number of environment variables to configure the installation and the compilation process. The most important ones are:

- `OPENQL_ENABLE_INITIAL_PLACEMENT`: if defined (value doesn't metter), initial placement support will be enabled.
- `OPENQL_DISABLE_UNITARY`: if defined (value doesn't matter), unitary decomposition is disabled. This speeds up compile time if you don't need it.
- `NPROCS`: sets the number of parallel processes to use when compiling (must be a number if defined). Without this, it won't multithread, so it'll be much slower.

In bash-like terminals, you can just put them in front of the pip command like so: `NPROCS=10 pip ...`. In Powershell, you can use `$env:NPROCS = '10'` in a command preceding the `pip` command.

Note: You may find that CMake notes that some packages it's looking for are missing. This is fine: some things are only needed for optional components (which will automatically disable themselves when dependencies are missing) and some things are only quality-of-life things, for example for generating backtraces for the exception messages. As long as the tests pass, the core OpenQL components should all work.

Once installed, and assuming you have the requisite optional dependencies installed, you can run the test suite (still from the root of the OpenQL repository) using

:: pytest -v

Note: If `pytest` is unrecognized, you should be able to use `python -m pytest` or `python3 -m pytest` instead (making sure to use the same Python version that the `pip` you installed the package with corresponds to).

Conda vs pip

A conda recipe also exists in the repository. However, it is in a state of disuse, as conda's ridiculous NP-complete dependency solver implementation is too heavy for CI (it can take literal hours), and none of the maintainers use it. Your mileage may vary.

1.15.4 Building the C++ tests and programs

Existing tests and programs can be compiled by the following instructions. You can use any existing example as a starting point for your own programs, but refer to `examples/cpp-standalone-example` for the build system.

The tests are run with the `tests` directory as the working directory, so they can find their JSON files. The results end up in `tests/test_output`.

Linux/MacOS

Existing tests and examples can be compiled and run using the following commands:

```
mkdir cbuild
cd cbuild
cmake .. -DOPENQL_BUILD_TESTS=ON    # configure the build
make                                # actually build OpenQL and the tests
make test                           # run the tests
```

Windows

Existing tests and examples can be compiled and run using the following commands:

```
mkdir cbuild
cd cbuild
cmake .. -DOPENQL_BUILD_TESTS=ON -DBUILD_SHARED_LIBS=OFF # configure the build
cmake --build .                     # actually build OpenQL and the tests
cmake --build . --target RUN_TESTS  # run the tests
```

Note: `-DBUILD_SHARED_LIBS=OFF` is needed on Windows only because the executables can't find the OpenQL DLL in the build tree that MSVC generates, and static linking works around that. It works just fine when you manually place the DLL in the same directory as the test executables though, so this is just a limitation of the current build system for the tests.

Other CMake flags

CMake accepts a number of flags in addition to the `-DOPENQL_BUILD_TESTS=ON` flag used above:

- `-DWITH_INITIAL_PLACEMENT=ON`: enables initial placement.
- `-DWITH_UNITARY_DECOMPOSITION=OFF`: disables unitary composition (vastly speeds up compile time if you don't need it).
- `-DCMAKE_BUILD_TYPE=Debug`: builds in debug rather than release mode (less optimizations, more debug symbols).
- `-DBUILD_SHARED_LIBS=OFF`: build static libraries rather than dynamic ones. Note that static libraries are not nearly as well tested, but they should work if you need them.

1.15.5 Building the documentation

If you want, you can build the ReadTheDocs and doxygen documentation locally for your particular version of OpenQL. Assuming you have installed the required dependencies to do so, the procedure is as follows.

```
# first build/install the qutechopenql Python package!
cd docs
rm -rf doxygen      # optional: ensures all doxygen pages are rebuilt
make clean          # optional: ensures all Sphinx pages are rebuilt
make html
```

The main page for the documentation will be generated at `docs/_build/html/index.html`.

Note: The Doxygen pages are never automatically rebuilt, as there is no dependency analysis here. You will always need to remove the doxygen output directory manually before calling `make html` to trigger a rebuild.

1.16 Build automation

OpenQL employs continuous integration based on GitHub Actions, to ensure that new features or modifications actually work on all supported systems. The files to this end are in the `.github` directory of the repository. Furthermore, ReadTheDocs can build and publish documentation for OpenQL automatically, for which the files are in the `docs` folder (specifically, it uses the `docs/requirements-docs.txt` for a pip package dependency list and then just runs the makefile in `docs`.

1.16.1 Integration tests

The integration tests are run when you push to a branch for which a pull request is open, or when `develop` changes. The suite runs the following things:

- the Python test suite;
- the ctest suite (tests and examples); and
- the standalone C++ test, built completely out of context.

The former two are run on `ubuntu-latest`, `macos-latest`, and `windows-latest` for all active Python versions (currently 3.6 through 3.9). The latter is only run on `ubuntu-latest`, as it doesn't check much except inclusion of the project via CMake.

Note: To test an incomplete branch that you're still working on, please open a draft pull request.

1.16.2 Release automation

Release artifact generation triggers on a push to a branch that starts with `release_` (used for testing) or when a new release is made via GitHub and/or a tag is pushed.

Warning: Please don't do any of these things until you have read the *Release procedure*!

1.16.3 ReadTheDocs automation

TODO: Razvan, anything noteworthy here?

1.17 Release procedure

Most of the release process is managed by GitHub Actions. If you follow the procedure below, it will automatically test and build all wheels and conda packages for all platform, and publish them to PyPI and conda assuming the secrets are configured correctly.

- Make a branch with a name starting with "release" based upon the commit that is to be released. For example, `release-0.0.1`, but the suffix doesn't matter.
- Change the version in `include/ql/version.h` (this is the only functional place where the version is hard-coded) and change any other files where applicable (changelog, etc) and update `CHANGELOG.md` accordingly, then commit and make a PR for it.
- CI will now run not only the test workflow, but also the assets workflow. The assets workflow builds the wheels and conda packages, but publishes them to the GitHub Actions build artifacts only. You can then test these yourself if you have reason to believe that something might be wrong with them that CI might not catch. To find them, go to Actions -> Assets workflow -> click the run for your branch -> Artifacts.
- Always delete the artifacts of the assets runs when you're done with them or don't need them! OpenQL's binaries are quite big, so if you don't do this, GitHub will soon start rejecting new artifacts due to storage quota.
- If the test or assets workflows fail, fix it before merging the PR (of course).
- Once CI is green, merge the PR into `develop` if there are any changes. If no changes were needed, just delete the branch to clean up.

- If needed, also merge to `master`.
- Draft a new release through the GitHub interface. Set the "tag version" to the same version you put in `include/ql/version.h`, the title to "Release `version: name`", and write release notes in the body. The release notes should include at least:
  - a summary of what has changed, what is new, and what is incompatible with the previous version;
  - if there are incompatibilities, what the user can do for mitigation; and
  - what has been deprecated and may be removed in a later version.

  In principle, these things apply only to the public API. For releases that don't go to master, check the "prerelease" box, so it won't show up as the latest release.
- CI will run the assets workflow again, now with the new version string baked into the wheels and packages. When done, these wheels and packages are automatically added to the GitHub release, and if the secrets/API keys for PyPI and conda are correct, CI will publish them there.
- Remove the temporary `release-*` branch. Users can find particular releases via the tag that GitHub automatically added when you drafted the release.

1.18 C++ coding conventions

In order to maintain the code homogeneous and consistent, all contibutors are invited to follow this coding convention.

NOTE: at the time of writing, not all of OpenQL has been converted to this code style completely yet.

In general, consistency is considered to be more important than any of these rules. If a significant piece of code violates a rule consistently, either change the entire piece of code to conform, or make your changes in the same style as the original code.

1.18.1 File and directory organization

C++ header files should be named `.h`. Header files private to OpenQL go in the `src` directory, preferably in a `detail` subdirectory. Header files that a user needs to access as well (the vast majority) go in `include`.

All definitions must go into source files; header files should only declare things. Therefore, almost all header files need a corresponding source file. This file must have the same name and path relative to the src/include directory as the corresponding header file.

All filenames are lowercase, separated by _ when composed of multiple words.

The filename and directory structure (loosely) follows the namespace structure of OpenQL and vice versa. When a namespace is only comprised of only a single file, the filename will be the name of the namespace, and the directory it's placed in is the name of the parent namespace (and so on). When a namespace consists of multiple files, the entire namespace path is represented as directories, and the contained files should be named after the (main) class (in lower_case) or functionality that they provide.

Subdirectories in `src/ql` may include a `tests` directory. Any `*.cc` file in such a directory is automatically interpreted as a unit test file by the build system. Note however that when you add or remove a test, you must manually regenerate the CMake project. A unit test simply consists of a C++ program with a `main()`, that must return zero on success or nonzero on failure. Unit tests are run using the *toplevel* `tests` directory as the working directory.

1.18.2 Naming conventions

To be consistent with especially Python (since we share an API between it and C++):

- class and type names are written in `TitleCase`.
- variables, fields, and namespaces (compare to modules) are written in `snake_case`.
- constants and macros are written in `UPPER_CASE`.

This is already the standard in most popular languages (aside from some deciding to use `mixedCase` in places). C++ is the only serious language that remains that maintains sort of its own style, but it's also the one largest amount of conflicting styles. Therefore, it makes more sense to just stick to Python. The only annoying conflict is that the standard library types are lowercase.

When naming things, try to be explicit and precise, but only within the context of the current namespace. For example, if you have a class representing a red apple, and you place it in namespace `apple`, call the class `Red` instead of `RedApple`. This saves you typing within the `apple` namespace, doesn't cost someone outside your namespace much extra typing for occasional apple usage (`apple::Red` isn't much longer than `RedApple` after all), and someone using lots of apples within some scope can just do `using namespace apple` locally to save more typing.

When you use polymorphism for a group of objects, the base class is typically called `Base`. Continuing with the apple example, the `apple` namespace may have a class `Base` declared in `base.h`, `Red : public Base` in `red.h`, and `Green : public Base` in `green.h`. Using `Base` instead of `Apple` avoids annoying constructions like `apple::Apple`.

Avoid abbreviations of "words" within a name, except maybe for very local variables like loop iterators. A little typing overhead while writing the code saves a lot of overhead when someone else later has to read and understand your code. However, typedefs (using the `using` keyword, C-style `typedef`s are comparatively hard to read) are encouraged, to remove parts of names or namespace paths that are obvious within context.

1.18.3 Namespaces

Since OpenQL may be used as a C++ library, it's common courtesy not to pollute the global namespace with stuff. Imagine, for instance, if OpenQL would define the type `Bit` to represent a classical bit in the global namespace, and someone using the library from C++ also includes a bit manipulation library that happens to also define `Bit`; this would be a naming conflict that's impossible to resolve for the user. Therefore, everything defined by OpenQL should be within the `ql` namespace, and all preprocessor macros (which can't be namespaced) should start with `QL_`.

Furthermore, nothing except the main C++-style `openql` header in `include` should define anything directly in `ql`. This namespace is reserved for the API layer that the user is expected to access, and must thus remain as consistent from version to version as possible. The main header currently does a `using namespace api` to pull the contents of `ql::api` into `ql`, but if internal changes are made to OpenQL again later, this translation may become more complex.

OpenQL has a well-defined namespace tree used to structure its components and keep things disjoint. Roughly speaking, the namespaces serve as library for dependent namespaces, although some dependency cycles still remain at this abstraction level. The `ql` subnamespaces, roughly ordered by dependencies, are:

- `utils`: extensions to (standard) libraries, wrappers, etc. not specific to OpenQL or compilers in any way.
- `ir`: intermediate representation. Contains most of the data structures needed to represent a quantum program and its target platform as it's being compiled. This is light on functionality; most of the functional behavior should be in `com`. This is because `ir` consistst primarily of generated code.
  - `prim`: "primitive" types and classes used by the generated tree representation. Some of these types actually are primitive, some are more complex classes that would be annoying or inefficient to represent using tree-gen.

  - `annot`: annotation types commonly used within the IR should be defined here.
  - `compat`: contains the pre-tree-gen IR. This is preserved because it is tightly coupled with the API, which must remain backward compatible. So, the API functions still generate this old representation, which is then converted to the tree-gen-based IR just before the pass manager is called for compilation.
- `com`: common operations. This contains all OpenQL/compiler-specific code operating on the platform and IR trees that is reusable for various passes. For example DFG or CFG construction might live here.
- `pmgr`: pass management. This contains all the logic that manages the compilation process.
- `pmgr::pass_types`: defines the abstract base classes for the compiler passes.
- `pass`: pass implementations. This contains a subtree of namespaces that eventually define the architecture-agnostic compiler passes of OpenQL. This tree should correspond exactly to the namespace paths in the path types as the pass factory knows them. The first namespace level is standardized as follows:
  - `pre`: passes that perform pre-processing of the platform tree. (NOTE: at the time of writing these don't exist yet, and pass management isn't quite ready for it yet due to issues with backward compatibility of the API)
  - `io`: I/O passes that load the IR from a file or save (parts of the IR) to a file without significant transformation. Mostly cQASM, but would also include conversion of the IR to different formats (OpenQASM? QuantumSim?).
  - `ana`: passes that leave the IR and platform as is (save for annotations), and only analyze the content of the platform/IR. For example statistics reporting, visualization, error checking, consistency checking for debugging, etc.
  - `dec`: passes that decompose code (instructions, gates, etc) to more primitive components or otherwise lower algorithm abstraction level. Should includes of course gate decomposition passes (once that functionality is pulled out of Kernel), but something like reduction of structured control flow to only labels and goto's would also go here.
  - `map` passes that map qubits or classical storage elements to something closer to hardware. Right now that would be "the mapper," but would also include a hypothetical pass that automatically applies some error correction code to the user-specified algorithm, mapping variables to classical registers and memory, reduction to single-static-assignment form, etc.
  - `opt`: optimization passes, i.e. passes that do not lower IR abstraction level, but instead transmute the IR to a "better" equivalent representation.
  - `sch` passes that shuffle instructions around and add timing information.
  - `gen` passes that internally convert the common IR into their own IR to reduce it further, to eventually generate architecture-specific assembly or machine code. These should only ever be part of `arch`.
  - `misc`: any passes that don't fit in the above categories, for example a Python pass wrapper if we ever make one, which could logically be any kind of pass.
  - *dnu*: "Do Not Use:" code exists only for compatibility purposes, only works in very particular cases, is generally unfinished, or is so old that we're not sure if it even works anymore. This receives special treatment in the pass factory: passes prefixed with `dnu` must be explicitly enabled in the compiler configuration file.
    * `io`..`misc`: the other categories reappear as namespaces within `dnu`.
- `rmgr`: resource management. This contains the logic that functionally describes the scheduling resources of a platform, used to define for example instrument constraints.
- `pmgr::resource_types`: defines the abstract base classes for the scheduling resources.
- `resource`: defines the architecture-agnostic scheduling resources built into OpenQL.

- *resource::dnu*: similar to *pass::dnu*, defunct or work-in-progress resources should be placed in here.
- `arch::<name>`: the place for all architecture-specific stuff. Specifically, this may include `com`, `pass`, and `resource` sub-namespaces that provide architecture-specific additions or overrides for the respective `ql` sub-namespaces.
- `api`: this namespace contains all user-facing API wrappers.

Private functionality for a logical piece of code within OpenQL (usually a pass) should go into a subnamespace named `detail`. This namespace should only have files in `src`, and as such, the non-`detail` parts of the code should only refer to it from `.cc` files (so NOT from the `include` header files). This enforces sectioning off local implementation details from the rest of the OpenQL code, preventing excessive compilation time by keeping the (public) header files as lean as possible.

Within a local namespace, use whatever you want (`using namespace` etc) as you see fit, although more selective inclusions and abbreviations using `namespace x = ...` and `using T = ...` is preferred.

As for code style, please stick to the following.

```
// Namespaces do not receive indentation, because then everything would be
// indented. But the closing brace must be clearly marked as such (using the
// depicted style) to compensate.
namespace ql {

... // namespace contents are not indented...

} // namespace ql

// "using namespace" is allowed only in .cc files, private header files, or
// in local scopes. If you're working deep inside the namespace tree, it is
// sometimes useful to pull another namespace into it, which is fine, but
// it's preferred to include things selectively with `using x = y` or to
// abbreviate namespaces if needed using `namespace x = a::b::c`.
```

1.18.4 Header files and `#include` syntax

Some general rules for `#include` directive consistency.

- Always use `#include "ql/..."` to refer to public header files of OpenQL. That is, use the `""` syntax rather than the `<>` syntax, and use the full path from `ql` onward.
- Usage of relative paths is allowed only to refer to private/`detail` header files.
- The first directive in a `.cc` file must be inclusion of its respective header file. The next line should be blank. Any header files needed for the `.cc` file that are not needed for the corresponding `.h` file follow after this blank line.
- Try to keep `#include` directives ordered as follows:
  - system header files (standard library, etc);
  - OpenQL's dependencies from the `deps` folder (lemon, libqasm, etc); and
  - include OpenQL's own headers in the same order that the namespaces are listed in the previous section.

Sometimes you may end up with an include dependency loop. For example, the platform structure includes a reference to an architecture structure, but the architecture-specific logic certainly makes use of the platform structure. The typical symptom is an error message that some type has not been declared yet, with a long include chain at the top. This can usually be bypassed by making a `declarations.h` file for the namespace for which forward references are needed. This header file should make forward declarations for the types defined in the namespace (just `class X;` etc.) and declare any pointer/reference typedefs needed for them. Note that the files that actually define the classes should always include this file as well, so the compiler will actually warn you when there are inconsistencies!

1.18.5 “Runtime” documentation and dump() functions

In order to aid the synchronization of the user-facing documentation and the internal codebase, and to make it easier for users to access the documentation, a good portion of the documentation is placed in the OpenQL codebase itself as strings. These strings can then be queried via the API by the user directly, or by the ReadTheDocs/Sphinx conf.py script to generate online documentation pages from them. Consistency is key for making this all work smoothly: inconsistensies are not only ugly when reading the documentation (say for instance that one person uses regular English interpunction while the other uses a more comment-like lack of interpunction and capitalization), but may also easily break the generators. After all, the output of these documentation functions is fed through some Python magic to Sphinx’ reStructuredText parser.

In order to make the documentation readable from within Python as well, indentation is used for sectioning, rather than RST section headers. This means that each documentation printing function needs to be aware of the current indentation level; simply returning a string is not enough. To solve this and a few other problems all functions that print documentation-like information to the user must have the following signature:

```
dump_*(std::ostream &os = std::cout, const utils::Str &line_prefix = "")
```

The following contract must be adhered to:

- at least one line must be written to `os`;
- all lines must start with `line_prefix` and end with `"\n"`;
- the `<iomanip>` stream state of `os` must not be mangled;
- the stream should be flushed at the end (either via `std::endl` or an explicit call to `flush()`).

To open a subsection in the output stream (for a recursive call to a dump function, for instance):

- there must be at least one blank line (or the start of the input) before the section header;
- the section header must have the same indentation level as the parent (so whatever is in `line_prefix`);
- the header must be exactly of the form `<line_prefix>* <text> *\n`; and
- the body of the section must be indented by two additional spaces.

Do NOT use RST or markdown headers in the section bodies; use only indented sections as described above. Violating this rule or any of the other rules above will likely break the converter for the RTD pages.

The text inside the documentation strings is interpreted as *markdown*, converted to reStructuredText for Sphinx/ReadTheDocs via `m2r2`. Markdown is used rather than RST because it’s way more pleasing to read raw, for example when dumped from within an interactive Python interpreter. `m2r2` passes most RST tags straight through however, so you still need to be careful not to accidentally put something that looks like RST in a docstring.

In addition `m2r2`‘s logic, the following conversions are made:

- section headers are detected and converted to appropriate RST header levels;
- section bodies are un-indented; and
- `NOTE:` or `WARNING:` at the start of a markdown paragraph (blank line before and after) is converted to an RST `.. note::`/`.. warning::` block. The first letter of the sentence (fragment) following the header is automatically capitalized, so it can be lowercase in the raw output while still being appropriately capitalized on ReadTheDocs.

To aid writing these long documentation strings inside C++, two functions are available in `utils/str.h`:

- `dump_str`: useful for writing long dumpable strings by means of manually-wrapped raw strings. For example:

```
utils::dump_str(os, line_prefix, R"(
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum at
lacus porttitor mi consectetur ultrices. Aenean malesuada tristique nisl,
eu ultrices enim sodales eu. Cras sed nulla enim. Nunc pretium pretium
tortor, ut cursus nulla commodo sit amet.
)");
```

`dump_str` ensures that the C++ indentation level is stripped from each line of the raw string, and that line_prefix is inserted before each line.

- `wrap_str`: similar to the above, but assumes that the input is not wrapped yet. This is more useful for shorter pieces of text where you don't want to be bothered by wrapping manually, or generated text where doing so consistently would otherwise be impossible. However, while the wrapper tries to be smart about maintaining indentation for multiple paragraphs in its input, it is not infallible. Hence, for long pieces of relatively complicated documentation code (that includes code blocks etc.) `dump_str` is more helpful.

1.18.6 Utility types and functions

The `ql::utils` namespace provides a bunch of typedefs and wrappers for C++ standard library stuff, modified to improve safety, reduce undefined behavior, simplify stuff where OpenQL doesn't need the full expressive power of the standard library, improve consistency in terms of naming conventions, or just to reduce typing. In cc files there is sometimes a `using namespace utils` to reduce typing further, but never do this in header files! You should use types and functions from here as much as possible. Here are some important ones.

- From `utils/num.h`:
  - `Bool` for booleans;
  - `Byte` for (unsigned) bytes;
  - `UInt` for unsigned integers (maps to 64-bit unsigned);
  - `Int` for signed integers (maps to 64-bit signed);
  - `Real` for real numbers (maps to `double`);
  - `Complex` for complex numbers;
  - `MAX` as an "undefined" value for Int or UInt;
  - `PI` for pi;
  - `EU` for Euler's constant;
  - `IM` for the imaginary unit;
  - a bunch of common math subroutines from std are copied into utils, so you don't need to type `std::`.
- From `utils/str.h`:
  - `Str` for strings (maps to `std::string`);
  - `StrStrm` for string streams (maps to `std::ostringstream`);
  - `to_string()` for converting things to string (this uses the << stream operator, so it can be overloaded, and IS overloaded for the wrapped container types for instance);
  - `parse_uint()`, `parse_int()`, and `parse_real()` for parsing numbers (these throw better exceptions than the STL counterparts);
  - a couple additional string utility functions are defined here.
- From `utils/json.h`:

  - `Json` for JSON values (from `nlohmann::json`);
  - `load_json()` for loading JSON files with better contextual errors and allowance for `//` comments.
- From `utils/pair.h`:
  - `Pair<A, B>` for `std::pair`; includes stream << overload.
- From `utils/vec.h`:
  - `Vec<T>` for a wrapper of `std::vector<T>` with additional safety.
- From `utils/list.h`:
  - `List<T>` for a wrapper of `std::list<T>` with additional safety.
- From `utils/map.h`:
  - `Map<K, V>` for a wrapper of `std::map<K, V>` with additional safety.
- From `utils/opt.h`:
  - `Opt<V>` for a more-or-less equivalent of `std::optional`, implemented using a smart pointer. Primarily intended for places where you need to own a value but can't construct it immediately (replacing the "virgin constructor" antipattern).
- From `utils/tree.h`:
  - `Maybe<T>` for an optional tree node reference;
  - `One<T>` for a mandatory tree node reference;
  - `Any<T>` for zero or more tree node references;
  - `Many<T>` for one or more tree node references;
  - `OptLink<T>` for an optional link to a tree node elsewhere in a tree;
  - `Link<T>` for a mandatory link to a tree node elsewhere in a tree;
  - `make<T>()` for constructing tree nodes.
- From `utils/exception.h`:
  - `Exception` for exceptions that are unlikely to be caught. This exception automatically adds contextual information, such as a stack trace.
- From `utils/logger.h`:
  - macros for logging to stdout; these are preferred over streaming to `std::cout` directly.
- From `utils/filesystem.h`:
  - `OutFile` for writing files (wraps `std::ofstream` with automatic error-checking, and also ensures directories are created recursively if the path to the file doesn't exist yet);
  - `InFile` for reading files (wraps `std::ifstream` with automatic error-checking);
  - a couple additional FS utilities.

1.18.7 Indentation

Use four spaces. NEVER tabs. Avoid trailing whitespace.

1.18.8 Comments

```
// A good normal comments consists of two slashes at the front of each line,
// followed by a single space, followed by English text in the form of a
// paragraph, preferably manually wrapped at column 80.

// Comment blocks of code like this.
statement;
another statement;

// The next block of code starts after an empty line. The first two statements
// that follow relate to this comment, the third does not.
statement one;
if (condition) {
    statement two;
}

statement three;

// In indented blocks, it works like this.

// Comment A
belongs to A;
belongs to A;
if (condition) {
    belongs to A;

    // Comment B
    belongs to B

}
belongs to A;

// Avoid comments at the end of a line. This almost always just becomes way too
// wide to read easily.

// Avoid /*...*/ for descriptive comments. You can use them to disable code,
// but #if 0 blocks are preferable because you can nest those.

// Functions, variables, classes, etc. should use javadoc-style comments.
// These comments may then be used by Doxygen and IDEs to extract
// documentation. You can use markdown and Doxygen @directives to make the docs
// prettier if you like. Attach the docstring to the *definition* of the object
// rather than its declaration in a header file; if you like, you can add
// regular // comments with a brief description of the function or whatever in
// the header file for people trying to understand the interface of your class
// or module from a birds' eye perspective.
/**
 * Brief documentation.
 *
 * Extended documentation automatically follows after the first sentence.
 */
```

(continues on next page)

(continued from previous page)

```
void some_function();

// When you want to section up your code, you can use breaks like this:

//===============================================================================
// START OF SOME SECTION
//===============================================================================

// Don't use "/*********" etc., as this may be confused with docstrings by some
// tools.
```

1.18.9 Macros and other preprocessor directives

```
// Regardless of code indentation, preprocessor directives are not indented by
// convention.
void test() {
    ...
#if WITH_SOME_FEATURE
    ...
#endif
    ...
}

// Feature flags are controlled by CMake through options passed to the user.
// Refer to the CMakeLists.txt files for more information; it should be fairly
// obvious.

// Feature switches such as the above must only ever be used in .cc files,
// never in headers. The feature flags may get out of sync otherwise (the flags
// may differ between when the OpenQL library was compiled and when the user is
// linking against it).

// Header file guards are done using
#pragma once
// rather than #ifndef etc. This avoids weird problems caused by copypaste
// mistakes, when the same preprocessor definition is used for multiple
// headers. Since OpenQL already requires C++11, you'll be hard-pressed finding
// a compiler that doesn't support that pragma but could compile OpenQL
// otherwise.

// Local header files use "" in their include directives. Only external
// libraries and the standard library uses <>.
#include <vector>
#include <lemon/...>
#include "openql.h"
```

1.18.10 If statements

```
// If statements look like this.
if (condition) {
    ...
} else if (condition) {
    ...
} else {
    ...
}

// Very short statements can go on the same line without braces, but only if
// there is no else block and it's easier to read than writing out the block.
// Typically this is only the case for if-break, if-return, or if-throw.
if (condition) break;

// Do NOT use Yoda conditions. It's harder to read, and unless everything is
// written that way it's inconsistent.

// When the condition becomes too long to be readable, indent as shown:
if (
    (x == "a")
    || (x == "b")
    || (x == "c")
) {
    ...
}
```

1.18.11 Switch statements

```
// Switch statements:
//  - Indent case labels.
//  - Explicitly mark fallthrough with a comment when intended.
//  - Use blocks only if needed (i.e. for variable declarations).
switch (condition) {
    case a:
        ...
        break;
    case b:
    case c: {
        ...
        break;
    }
    case d:
        ...
        // fallthrough
    case e:
        ...
        break;
    default:
        break;
}
```

1.18.12 Loops

```
// Normal for loops:
//  - Declare the loop variable in the loop if it's a local thing.
//  - Use size_t for loops instead of int whenever you're using the loop
//    variable as index to avoid signed-unsigned comparison warnings.
//  - i++ is preferential over ++i unless you have a good reason to believe
//    the additional sequence point will actually cause harm. It's C++ after
//    all, not ++C.
for (size_t i = 0; i < 10; i++) {
    ...
}

// Iterating:
//  - Use iterating loops whenever possible.
//  - Use const auto &x whenever possible. If you intend to mutate the
//    elements, drop the const; if you don't intend to mutate but get
//    errors, drop the const and &.
for (const auto &element : sequence) {
    ...
}

// While loops:
while (condition) {
    ...
}

// Do-while loops (if you need them...):
do {
    ...
} while (condition);

// ALWAYS open a block (this goes for all statements except exceptional
// if statements). In rare cases (when the loop is just annoying boilerplate)
// the block can go in a single line as follows:
for (auto &el : sequence) { el = 0; }
```

1.18.13 Enumerations

```
// Enumerations look like this:
/**
 * Documentation for the complete enumeration.
 */
enum class Enum {

    /**
     * Documentation for option A.
     */
    OPTION_A,

    /**
     * Documentation for option B.
     */
    OPTION_B,

```

(continues on next page)

(continued from previous page)

```
    /**
     * Documentation for option C.
     */
    OPTION_C

};

// Using the C++ "enum class" construct, you can use short names for your enum
// options, as they will be namespaced to the enum;
Enum::OPTION_A
```

1.18.14 Typedefs

```
// Use "using" rather than "typedef". The syntax is just more clear.
using Hello = std::vector<int>*;

// instead of "typedef std::vector<int> *Hello"
```

1.18.15 Classes

```
// Class definitions are indented and whitespaced as follows:
class AClass {
public:
    ...
}

// Classes in header files should not contain any function definitions,
// regardless of triviality (except when templated, then it can't be avoided).
// This keeps compilation times down, the lack of inlining makes stack traces
// less magical, and honestly, I doubt you'll notice the performance penalty of
// the lack of inlining in practice (and there's always LTO nowadays).

// Avoid "virgin constructors": the constructor of a class should also
// initialize that class. C++ utilizes the RAII principle (resource acquisition
// is initialization): if you have a class instance, you should be able to
// assume it has already been initialized and that you can use it. Virgin
// constructors combined with an init method nuke this principle, making it
// relatively easy to accidentally get undefined behavior by using an object
// before initializing it. If you're faced with spaghetti in the member
// initialization part of a constructor (the part between the : and the
// function body) because members don't have a virgin constructor, wrap them in
// utils::Opt or utils::Ptr and call emplace() on them instead of what would
// have been init(); this lets you do everything that you would be able to do
// with the virgin constructor method, but without breaking RAII and with
// runtime detection of use-before-init (since dereferencing an empty Opt or
// Ptr will throw an exception).

// Constructions where you think you need this should be avoided. If you really
// need it anyway, use ql::Opt<T> instead of T directly. You can then use
// emplace(...) to initialize the contained object of type T after the fact
// (or you can just assign it as you otherwise would), and access the fields
// and members of T using -> instead of . (so, as if it were a pointer). The
```

(continues on next page)

(continued from previous page)

```
// practical upshot of this is that Opt will throw an exception your way if
// you try to access it while it's not initialized yet, whereas with the virgin
// constructor method OpenQL would just silently truck along to spit out
// garbage that may or may not look like what you were expecting.

// When using inheritance, the toplevel ancestor class must have a virtual
// destructor. If you don't have anything useful to put there, you can tell the
// compiler to make a default one. If you don't do this, class destruction
// won't work right, and you'll get undefined behavior. It is sufficient to
// only give the toplevel class such a virtual destructor; all descendents get
// virtual destructors automatically because of it.
virtual ~Cls() = default;
```

1.18.16 Variables and fields

```
// References and pointers belong to the name, not to the type:
int *ip;

// so, NOT "int* ip". This is objectively the "correct" way to write this, as
// this is how C++'s syntax tree works. When you write "int* a, b", the result
// will unfortunately be that a is of type int* and b is of type int; to make
// them both int* you need to write "int *a, *b". This is complete nonsense,
// but unfortunately what the C++ overlords decided to go with.

// Public fields are almost always a Bad Idea (TM). Whenever the outside world
// should probably not have write access to a field (variable in a class)
// because this would break things unless a large amount of care is taken, the
// field must be private or protected (depending on whether descendents are
// part of the "outside world" or not). When it makes sense, you can add
// getters (make sure these are const-correct) and/or setters (when
// applicable). Debugging is NOT a good reason to make things public; use
// getters!

// Shared global variables need to be declared as follows in a header file:
OPENQL_DECLSPEC extern int i;

// and as follows in the corresponding source file:
int i;

// OPENQL_DECLSPEC is defined in utils.h. If you don't do this, the OpenQL
// library will break on Windows.
```

1.18.17 Functions and methods

```
// Declaration in header file:

// A simple function.
int name(int a, int b = 2, const std::string &s = "hello");

// Implementation in source file:

/**
```

(continues on next page)

(continued from previous page)

```
 * A simple function.
 *
 * All functions should be documented with javadoc-style comments like these.
 */
int name(int a, int b = 2, const std::string &s) {
    ...
}

// If the parameter list becomes annoyingly long, indent as follows:
int name(
    int a,
    int b,
    int c,
    int d
) {
    ...
}

// Constructor lists either go on a single line along with all parameters:
int name(int a, int b) : a(a), b(b) {
    ...
}

// or look like this:
int name(
    int a,
    int b
) :
    a(a),
    b(b)
{
    ...
}

// Template statements go on the line in front of the function. Templated
// functions belong in header files, unless you really know what you're
// doing (explicitly instantiated specializations, etc.).
template <class A>
int *name(int a, const std::string &s) {
    ...
}

// Methods (functions on a class) should be static if they don't use any
// fields,
class Cls {
    ...
    static void func();
    ...
};

void Cls::func() {
    ...
}

// or const if they only read fields:
class Cls {
    ...
```

(continues on next page)

(continued from previous page)

```
    void func() const;
    ...
};

void Cls::func() const {
    ...
}

// If you're intending to override methods, the method in the ancestor class
// should be marked virtual:
class Cls {
    ...
    virtual void func();
    ...
}

void Cls::func() {
    ...
}

// and the overrides should be marked with override:
class Cls {
    ...
    void func() override;
    ...
}

void ClsImpl::func() {
    ...
}

// Note that static, virtual, and override don't appear in the implementation
// for some reason, and the position of the statements is arbitrary at best.
// This is just how C++ works, unfortunately.

// Pass nontrivial argument types (anything that's larger than a pointer) by
// const reference unless you have a good reason not to. That is, when passing
// some random string, it should be done like
void print(const std::string &str);

// If you really know what you're doing you can deviate (rvalue references,
// perfect forwarding, etc.), but this is the norm.

// Passing mutable references for the purpose of mutating them in the function
// is allowed, but MUST BE CLEARLY DOCUMENTED.
/**
 * Mutate the given string to make it lowercase.
 */
void to_lower(std::string &str);

// Passing by raw pointer is frowned upon, but still happens all over at the
// time of writing. At least be const-correct about it if you can.
```

1.18.18 Expressions and function calls

```
// Apply whitespace and interpunction rules as if you were writing English,
// except for the open-parenthesis that immediately follows a function name,
// where the space is omitted. For example, don't write something like
// "hello ( a+b , ( x+y )*z )", but write
hello(a + b, (x + y) * z);

// When expressions are too long to fit in a single line, preferably indent as
// follows if you have parentheses and a comma-separated list:
hello(
    a + b,
    (x + y) * z
);

// If you don't have anything like that, you might want to introduce
// parentheses and wrap as follows:
long_expression * (
    long_expression + long_expression
) * long_expression

// but in general, do whatever looks right (within context).
```

1.19 Intermediate representation

The IR is the most important data structure of any compiler. It must be able to encapsulate any program that can be compiled, during any compilation stage. The compiler passes can then be defined to just be transformations on this IR structure.

Compilers sometimes have multiple IR structures that they switch between at certain points in the compilation process. This is not the case for OpenQL: there is only one IR structure that all passes must operate on, though it's of course legal for a pass to temporarily build its own private structures, which may be especially important for complex code generation passes. Noteworthy for OpenQL however, is that the platform (a.k.a. target in many other compilers) is part of the IR structure in OpenQL: it shouldn't be modified by any passes, but it lives in the same tree structure for reasons we'll get back to in a few paragraphs.

1.19.1 The old IR

Despite only having a single logical IR, OpenQL is currently schizophrenic about its IR in its own way: at some point a new IR (`ql::ir`) was developed to replace the old one (now in `ql::ir::compat`, previously `ql::ir` and `ql::plat`), but not all passes have been converted to use the new IR yet. Thus, for any pass that hasn't been converted yet, the pass management logic first converts the new IR to the old one, then runs the legacy pass implementation, and then converts back to the new IR. This process is usually transparant, but there are a few gotchas:

- kernel names may change in some cases due to name uniquification logic;
- annotations places on nodes other than the program, a kernel, or a gate will be lost;
- any new-to-old conversion implicitly runs all legacy gate decompositions; and
- obviously, any features supported by the new IR but not by the old IR will result in compatibility errors during the new-to-old conversion.

Unfortunately, even when all passes have been converted, the old IR cannot be removed, because it is highly intertwined with the Python/C++ APIs for program construction (primarily, adding gates to a kernel immediately applies decomposition rules, which have a bunch of oddities that a reimplementation probably wouldn't be able to mimic exactly). Thus, as long as we need to maintain compatibility with old OpenQL programs, we'll be stuck with it. However, the intention is to use the old IR exclusively within the API in the future, run the (only) old-to-new conversion in `program.compile()`/`compiler.compile()` just before calling the pass manager, and remove the new-to-old conversion logic.

The platform construction logic is similarly difficult, partly because the old platform construction logic needs to be maintained because of the above, and partly because the platform construction logic is very intertwined with the old IR and operations on it. Thus, the platform side of the old-to-new conversion will also remain relevant. In fact, this conversion is currently where platform features only supported in the new IR are parsed; there is unfortunately no saner place to do this.

If you're not already familier with the old IR and you're not here to work on or upgrade legacy code, it's probably best to ignore the existence of the old IR as much as possible. If you nevertheless need to learn more about the old IR, you can:

- try to make sense of the classes and functions in `ql::ir::compat`;
- try to make sense of the "old pages" section in these docs, if it still exists when you're reading this; or
- read the documentation of an older version of OpenQL (prior to PR #405, so commit `7f2e2bb` or earlier).

For the rest of this section, we'll ignore the existence of the old IR and its conversions to and from the new IR.

1.19.2 The new IR

Describing a tree structure in C++ requires a lot of boilerplate code and repetition. To avoid some of this, the C++ classes for the IR are generated using tree-gen. This tool was developed specifically for OpenQL and libqasm, but doesn't depend on OpenQL (or libqasm) in any way, so it is written and documented in a generic way. This also means that its quirks are out of scope for this documentation; nevertheless, it is vital that you understand how `tree-gen` tree structures conceptually work before trying to understand OpenQL's IR structure. To prevent you from having to jump back and forth *too* much, here are a few things that might not be immediately apparent.

- The tree definition file format was a bit rushed, and in part because of that its structure might be unintuitive when you're not used to it yet. Most importantly, the `{}`-based structure of the tree definition file does *not* correspond to the tree structure being described, but rather to the class inheritance tree of the nodes that can be used *within* the described tree. Also, while the node names are written in lower_case in the tree description file, they are converted to TitleCase for the C++ class names (this is simply because `tree-gen` needs both forms, and it's easier to convert from lower_case to TitleCase than the other way around).
- There is no well-defined root node in a `tree-gen` tree, and (somewhat equivalently) `tree-gen` nodes do not know who their parent is. For this reason, most functions operating on OpenQL's IR take the root node of the IR (`ql::ir::Root`, or more typically `ql::ir::Ref`, which typedefs to `ql::utils::One<ql::ir::Root>`) as their first argument. This also means that a node can be reused in multiple trees. But be careful: nodes are almost always stored and passed by means of (ultimately) a `std::shared_ptr<>` reference with mutable content, so if a node ends up being shared, changing it in one tree will also effectively change it in the other tree. You can use the generated `copy()` and `clone()` methods to respectively make a shallow or deep copy of a node if need be.
- Besides the usual DAG edges in a tree graph, `tree-gen` trees support so-called "link" edges as well. A tree node can have any number of links pointing to it from anywhere else in the tree, even if this forms a loop. Links are useful to for example implement a variable reference node, or to refer to a data type node within the referenced variable definition node. This avoids having to keep track of unique names everywhere, and avoids a map lookup. Essentially, the "name" of a linked node is the pointer to its data structure. The tree consistency check ensures that a node is only used once in a tree, thus ensuring uniqueness of its pointer (and preventing a lot of other mayhem due to accidental reference reuse). It also ensures that all links in a tree actually link to nodes that are reachable from the root node via non-link edges: this is the primary reason why the platform description has to be part of the IR tree.

- There are six “edge types” for connecting nodes together: Maybe, One, Any, Many, OptLink, and Link. These correspond to zero or one edge, exactly one edge, zero or more edges, one or more edges, zero or one link, and exactly one link. Note however that it’s actually possible for One, Link, and Many to represent zero edges; doing that would merely make the tree consistency check fail. This is useful while building a tree or operating on it.
- Tree nodes, or anything else that implements the `Annotatable` type defined by `tree-gen`’s support library (`ql::utils::tree` in OpenQL), can be “annotated” with zero or one of literally any C++ type. You can think of them like a fancy, type-safe `void*` field in every node. This allows passes to attach information to nodes temporarily, allows storage of metadata that doesn’t really belong in the tree explicitly, and so on. Keep in mind, however, that annotations are stored by reference, and cannot be copied unless you already know which annotation types exist (just like a `void*`). Also, annotations are not copied by the `copy()` and `clone()` methods; if you intend to copy annotation *references* you must use `copy_annotations()` in addition to copying/cloning the node, and if you really need to copy an annotation by value you must use `copy_annotation<AnnotationType>()`.

With `tree-gen`-specific stuff out of the way, the IR definition itself should be rather straight-forward based on its tree definition file. So, instead of duplicating documentation in a way that will inevitably desync with the implementation, here’s the contents of that file (`src/ql/ir/ir.tree`).

```
# Implementation for the IR tree node classes.
source

# Header file for the IR tree node classes.
header "ql/ir/ir.gen.h"

// Include tree base classes.
include "ql/utils/tree.h"
tree_namespace utils::tree::base

// Use the tree support library customized for OpenQL (using utils types where
// applicable).
support_namespace utils::tree

// Include primitive types.
include "ql/ir/prim.h"

// Initialization function to use to construct default values for the tree base
// classes and primitives.
initialize_function prim::initialize
serdes_functions prim::serialize prim::deserialize

// Include SourceLocation annotation object for the debug dump generator.
//src_include "cqasm-parse-helper.hpp"
//location cqasm::parser::SourceLocation

# Namespace for the IR tree node classes.
namespace ql
namespace ir

# Root node for the IR.
root {

    # Root node for the description of the target.
    platform: One<platform>;

    # Root node for the description of the algorithm.
```

(continues on next page)

(continued from previous page)

```
    program: Maybe<program>;

}

# Root node for the description of the target.
platform {

    # User-given name for the platform. No constraints on syntax. May also be
    # empty.
    name: prim::Str;

    # Vector of data types, ordered by name so lookup can be done with log(N)
    # complexity. This represents a list of all data types usable by the
    # algorithm, such as qubits, integers, etc.
    data_types: Many<data_type>;

    # Vector of instruction types, ordered by name so lookup can be done with
    # log(N) complexity. This represents the instruction set as usable by the
    # algorithm at any time during the compilation process (i.e., it also
    # includes non-primitive instructions that may need to be decomposed at
    # some point!).
    instructions: Any<instruction_type>;

    # Vector of (builtin) function types, ordered by name so lookup can be done
    # with log(N) complexity. Functions are the active elements of expression
    # trees. They may at some point be mapped to instructions.
    functions: Any<function_type>;

    # Vector of all physical objects (a.k.a. registers) available in the
    # platform, ordered by name so lookup can be done with log(N) complexity.
    objects: Many<physical_object>;

    # The main qubit register that the generic mapper will map everything to
    # and that topology applies to. The data type must be a vector of qubits.
    # This also indirectly defines the main qubit type.
    qubits: Link<physical_object>;

    # If qubits have an implicit bit associated with them, this must be set to
    # the corresponding bit type. If it doesn't, this should be empty.
    implicit_bit_type: OptLink<data_type>;

    # The bit-like data type used for default-generated instruction and loop
    # conditions.
    default_bit_type: Link<data_type>;

    # The int-like data type used for default-generated indices.
    default_int_type: Link<data_type>;

    # Topology/connectivity information for the main qubit register.
    topology: prim::Topology;

    # Control architecture information structure.
    architecture: prim::Architecture;

    # Resource manager for scheduling.
    resources: prim::ResourceManager;
```

(continues on next page)

(continued from previous page)

```
    # Raw platform configuration JSON data for anything not specified in this
    # record.
    data: prim::Json;

}

# Representation of a data type usable by the algorithm represented by the IR.
# Semantical information may be added using annotations.
data_type {

    # Unique identifier for the data type. Must match `[a-zA-Z_][a-zA-Z0-9_]*`.
    name: prim::Str;

    # A data type that behaves like a qubit.
    qubit_type {}

    # A data type that represents classical information.
    classical_type {

        # A data type that behaves like a boolean/bit.
        bit_type {}

        # A data type that behaves like a two's-complement integer.
        int_type {

            # Whether the data type is signed.
            is_signed: prim::Bool;

            # Number of bits used to represent the type. Must be at most 64 for
            # signed or at most 63 for unsigned, otherwise literals cannot be
            # properly represented in cQASM.
            bits: prim::UInt;

            reorder(name, is_signed, bits);
        }

        # Type of a real number (IEEE double).
        real_type {}

        # Type of a complex number (2x IEEE double).
        complex_type {}

        # Type of a matrix. Matrices are currently special-cased to keep the
        # type system no more complex than cQASM 1.x's and be able to represent
        # gate matrices, but these should ultimately be replaced by dedicated
        # array types if/when these would be added.
        matrix_type {

            # Number of rows. Must be nonzero.
            num_rows: prim::UInt;

            # Number of columns. Must be nonzero.
            num_cols: prim::UInt;

            # A real-valued matrix.
            real_matrix_type {
                reorder(name, num_rows, num_cols);
```

(continues on next page)

(continued from previous page)

```
            }

            # A complex-valued matrix.
            complex_matrix_type {
                reorder(name, num_rows, num_cols);
            }

        }

        # Type of an arbitrary string.
        string_type {}

        # Type of a JSON string.
        json_type {}

    }

}

# Representation of an instruction type usable by the algorithm represented by
# the IR. Semantical information may be added using annotations.
instruction_type {

    # Identifier for the instruction. This only needs to be unique in
    # combination with the operand types. Must match `[a-zA-Z_][a-zA-Z0-9_]*`.
    name: prim::Str;

    # Identifier for the instruction as used in cQASM. Normally this is the same
    # as name; the override exists because historically different conventions
    # have been used for cQASM and OpenQL. Must match `[a-zA-Z_][a-zA-Z0-9_]*`.
    cqasm_name: prim::Str;

    # The types of all the non-template operands that the instance of this
    # instruction must have.
    operand_types: Any<operand_type>;

    # Specializations for this instruction. Specializations allow different
    # semantics (such as different durations) to be attached to instructions,
    # based on one or more of its operands. Each specialization in this list
    # must have:
    #  - the same name and cqasm_name;
    #  - the first element of operand_types removed;
    #  - an additional element at the end of template_operands;
    #  - the type of said element must match the removed operand_type element;
    #  - generalization must link back to this node.
    # The remaining fields may be specialized.
    specializations: Any<instruction_type>;

    # Link to the generalization of this instruction, if any; this must be set
    # iff template_operands is nonempty. The generalization must have a link to
    # this node in its specialization list.
    generalization: OptLink<instruction_type>;

    # The values of any template operands for this specialization of this
    # instruction.
    template_operands: Any<expression>;
```

(continues on next page)

(continued from previous page)

```
    # Decomposition rules for this instruction type. Multiple of these may be
    # defined: it is up to the decomposition pass to choose the decomposition
    # used (if any) based on the name or on some other heuristic.
    decompositions: Any<instruction_decomposition>;

    # The duration of this instruction in quantum cycles. Note that this may be
    # zero, as classical instructions don't usually pass any time in the quantum
    # time domain.
    duration: prim::UInt;

    # When set, the instruction acts as a barrier with respect to the data flow
    # graph, so it cannot commute with anything, regardless of what the operands
    # are. This is useful to represent operations like "measure all qubits" or
    # (for simulation) "print the current state of the program".
    barrier: prim::Bool;

    # Raw platform configuration JSON data for anything not specified in this
    # record.
    data: prim::Json;

}

# A decomposition rule for an instruction.
instruction_decomposition {

    # Name for this decomposition rule. May be used by decomposition logic to
    # determine which decomposition rule to apply. No constraints on syntax.
    name: prim::Str;

    # Objects used to represent the instruction parameters.
    parameters: Any<temporary_object>;

    # Any temporary variables as needed within the decomposition rule.
    objects: Any<virtual_object>;

    # The block of instructions that the decomposition rule expands to.
    expansion: Any<statement>;

    # Raw platform configuration JSON data for anything not specified in this
    # record.
    data: prim::Json;

}

# Representation of a (builtin) function type usable by the algorithm
# represented by the IR within expressions. All functions must be free of side
# effects. Semantical information may be added using annotations.
function_type {

    # Identifier for the function. This only needs to be unique in combination
    # with the operand types. Must match `[a-zA-Z_][a-zA-Z0-9_]*` or be a
    # recognized operator name (such as `operator+`, so just like C++).
    name: prim::Str;

    # The types of the operands that instances of this function must have. All
    # operands must have read or literal access mode.
    operand_types: Any<operand_type>;
```

(continues on next page)

(continued from previous page)

```
    # The type returned by the function.
    return_type: Link<data_type>;

    # The decomposition rule used for converting this function to instructions.
    # If not set, the function either needs to be primitive for the target, or
    # the decomposition must be done by a target-specific pass. During
    # decomposition, a temporary object will always be generated for storing
    # the return value.
    decomposition: Maybe<function_decomposition>;

    # Raw platform configuration JSON data for anything not specified in this
    # record.
    data: prim::Json;

}

# A decomposition rule for a function.
function_decomposition {

    # The type of instruction that this function decomposes to. The prototype of
    # this instruction must exactly match the prototype of the function, after
    # inclusion of the return operand (if dedicated). Also, any operand with
    # mode 'W' in the function must have mode 'W' in the instruction. The return
    # value location must also be mode 'W' if it maps to an operand.
    instruction_type: Link<instruction_type>;

    # The manner in which the return value of the function is mapped to the
    # instruction operand list.
    return_location: One<return_location>;

}

# The manner in which the return value of the function is mapped to the
# instruction operand list.
return_location {

    # Indicates that the return value of the associated function is not stored
    # in an operand, but rather in a special physical register. The
    # decomposition will have the following form:
    #  - barrier <object>
    #  - <insn> [template-operands] <operands>
    #  - barrier <object>
    #  - set <retval> = <object>
    return_in_fixed_object {

        # The physical object/register in which the return value will be stored.
        object: Link<physical_object>;

    }

    # Indicates that the return value of the associated function will be stored
    # in a register indicated by a dedicated output operand. The
    # decomposition will have the following form:
    #  - <retval> [template-operands] <operands-0..idx-1> <temp> <operands-idx..end>
    return_in_dedicated_operand {
```

(continues on next page)

(continued from previous page)

```
        # The index of the return operand in the instruction non-template
        # operand list.
        index: prim::UInt;

    }

    # Indicates that the return value of the associated function will be stored
    # in a register indicated by a shared (read-write) operand. The
    # decomposition will have the following form:
    #  - set <temp> = <operand-idx>
    #  - <retval> [template-operands] <operands-0..idx-1> <temp> <operands-idx+1..end>
    return_in_shared_operand {

        # The index of the return operand in the instruction non-template
        # operand list.
        index: prim::UInt;

    }

}

# A data storage location.
object {

    # Identifier for the object. Must match `[a-zA-Z_][a-zA-Z0-9_]*` or be
    # left unspecified (empty). Names for toplevel physical objects must be
    # specified and unique; only virtual objects and the implicit bit
    # register object may be anonymous. The names of virtual objects need
    # not be unique.
    name: prim::Str;

    # The elemental data type of this object.
    data_type: Link<data_type>;

    # The shape of this object. Empty means scalar, a single element means
    # vector of the given size, two elements means a matrix, and so on.
    shape: prim::UIntVec;

    # A virtual object, i.e. an object that still needs to be mapped to a
    # physical object. These are declared in the program part of the tree.
    virtual_object {

        # A variable declared by the user.
        variable_object {}

        # A temporary object, for example needed as part of a decomposition.
        # These are typically anonymous (i.e. have no specified name).
        temporary_object {}

    }

    # A physical object, i.e. a storage location or qubit that actually exists
    # in the target. These are declared in the platform part of the tree.
    physical_object {}

}
```

(continues on next page)

(continued from previous page)

```
# The type of a function or instruction operand, including access mode for
# commutative data dependency graph construction.
operand_type {

    # Access mode for the operand.
    mode: prim::OperandMode;

    # The data type of the operand.
    data_type: Link<data_type>;

}

# Root node for the algorithm itself.
program {

    # User-given name for the program. No constraints on syntax. May also be
    # empty.
    name: prim::Str;

    # Possibly-uniquified program name in the context of multiple compilations
    # of the same program within some context.
    # TODO: this shouldn't be here.
    unique_name: prim::Str;

    # List of virtual objects (variables and temporary storage locations) in use
    # by the program.
    objects: Any<virtual_object>;

    # The list of blocks that constitute the program.
    blocks: Many<block>;

    # The block that serves as the entry point to the program. Must point into
    # an entry of blocks.
    entry_point: Link<block>;

}

# Base type for sub-blocks and toplevel (named) blocks.
block_base {

    # The list of statements that constitute the body of the block. The cycle
    # numbers of any contained instructions must be non-decreasing.
    statements: Any<statement>;

    # A sub-block of statements, used within structured control-flow statements.
    sub_block {}

    # A block of statements within the program. Depending on the stage of
    # compilation, this may represent a *basic* block. A basic block has the
    # following rules attached:
    #  - all statements must be instructions; and
    #  - no non-goto instructions may follow any goto instruction.
    # Before this stage, there are no requirements.
    block {

        # Optional name of this block. Must match `[a-zA-Z_][a-zA-Z0-9_]*` and be
        # unique if specified (non-empty).
```

(continues on next page)

(continued from previous page)

```
        name: prim::Str;

        # Link to the block that will be processed after this block. If empty, the
        # algorithm terminates at the end.
        next: OptLink<block>;

    }

}

# A statement. This can take the form of an instruction or a special structured
# control-flow statement.
statement {

    # The quantum cycle in which this instruction is scheduled, with respect
    # to the start of the block it is contained in. Note that the order of
    # statements scheduled within the same quantum cycle is still
    # considered to be relevant: all side effects of a particular
    # statement always run before the side effects of the next statement
    # even start, regardless of cycle number and duration. This means that
    # even if `set a = b` and `set b = a` are scheduled in the same cycle,
    # this does NOT swap the values of `a` and `b`. This also means that a
    # program with all statements scheduled in cycle 0 is semantically
    # valid (though not likely to be executable as such, of course), so a
    # program that has not yet been scheduled can be represented with any
    # non-decreasing cycle assignment.
    cycle: prim::Int;

    # A regular instruction instance. May be quantum, classical, or mixed in
    # nature.
    instruction {

        # A conditional instruction instance.
        conditional_instruction {

            # The condition expression. This will usually be literal true, to
            # indicate that the instruction is actually unconditional. The
            # actual data type depends on the target, but should behave like a
            # boolean.
            condition: One<expression>;

            # A custom instruction defined within the target platform by means
            # of an instruction type.
            custom_instruction {

                # The instruction type that this is an instance of.
                instruction_type: Link<instruction_type>;

                # The operands for this instruction instance. The types of the
                # expressions must match the operand types listed in the
                # instruction type.
                operands: Any<expression>;

            }

            # A classical assignment instruction. This simply assigns a value
            # to a target object.
```

(continues on next page)

(continued from previous page)

```
            set_instruction {

                # A reference to the object being assigned.
                lhs: One<reference>;

                # The value that the object is being assigned to. The type of
                # the expression must match the type of the left-hand side
                # exactly (i.e., typecasts of any kind must be made explicit
                # by means of a typecast expression).
                rhs: One<expression>;

            }

            # A goto instruction. Jumps to the start of the target block when
            # executed and the condition evaluates to true.
            goto_instruction {

                # Link to the target block.
                target: Link<block>;

            }

        }

        # A (quantum) wait instruction. Also known as a (quantum) barrier when
        # the duration is zero.
        wait_instruction {

            # The objects that are to be waited on. If empty, the wait
            # instruction must effectively wait for *all* objects in the program
            # (or, equivalently, it must wait for all preceding instructions to
            # complete, and all subsequent instructions must start after it).
            # Also known as a "full barrier" for lack of a better term.
            objects: Any<reference>;

            # The amount of quantum cycles that must be waited in addition after
            # all objects are ready.
            duration: prim::UInt;

        }

    }

    # Structured control-flow statements.
    structured {

        # An if-else chain.
        if_else {

            # The if-else branches.
            branches: Many<if_else_branch>;

            # The final else block, if any.
            otherwise: Maybe<sub_block>;

        }
```

(continues on next page)

(continued from previous page)

```
        # A loop.
        loop {

            # The loop body.
            body: One<sub_block>;

            # A loop with a statically-known range of integers being iterated
            # over. Note that while the iteration count has an upper limit,
            # namely abs(from - to + 1), break and continue statements are
            # allowed.
            static_loop {

                # Reference to the variable used for looping.
                lhs: One<reference>;

                # The first value.
                frm: One<int_literal>;

                # The last value.
                to: One<int_literal>;

            }

            # A dynamic loop, of which the iteration count depends on a
            # condition.
            dynamic_loop {

                # The condition for starting another iteration.
                condition: One<expression>;

                # A C-style for loop. Note that a while loop is a special case
                # of this, specifically one with no initialize/update
                # expression. The condition is evaluated before each iteration,
                # and iteration stops when it yields false.
                for_loop {

                    # The optional initializing assignment, run before the loop
                    # starts.
                    initialize: Maybe<set_instruction>;

                    # The updating assignment, done at the end of the loop body
                    # and upon continue.
                    update: Maybe<set_instruction>;

                }

                # A repeat-until loop. The condition is evaluated at the end of
                # each iteration, and iteration stops when it yields true.
                repeat_until_loop {}

            }

        }

        # A loop control statement, i.e. break or continue.
        loop_control_statement {
```

(continues on next page)

(continued from previous page)

```
            # A break statement.
            break_statement {}

            # A continue statement.
            continue_statement {}

        }

    }

    # A dummy statement that doesn't produce any code. Used within the data
    # dependency graph for the source and sink nodes.
    sentinel_statement {}

}

# A single condition + block for use in an if-else chain.
if_else_branch {

    # The condition.
    condition: One<expression>;

    # The body.
    body: One<sub_block>;

}

# An expression. Expressions are used wherever operands are needed, and can be
# either a literal (i.e. the value is known at compile-time), a reference (i.e.
# the value is not known, but the storage location is), or a call to a builtin
# function. The latter has itself zero or more operands, so arbitrarily-deep
# expression trees can be described.
expression {

    # A literal expression, i.e. one of which the value is known at
    # compile-time.
    literal {

        # Link to the data type represented by this literal.
        data_type: Link<data_type>;

        # A bit/boolean literal. Either 1/true or 0/false.
        bit_literal {

            # The value of the literal.
            value: prim::Bool;

        }

        # An integer literal.
        int_literal {

            # The value of the literal.
            value: prim::Int;

        }
```

(continues on next page)

(continued from previous page)

```
        # A real floating-point literal.
        real_literal {

            # The value of the literal.
            value: prim::Real;

        }

        # A complex floating-point literal.
        complex_literal {

            # The value of the literal.
            value: prim::Complex;

        }

        # A real-valued matrix.
        real_matrix_literal {

            # The value of the literal.
            value: prim::RMatrix;

        }

        # A complex-valued matrix.
        complex_matrix_literal {

            # The value of the literal.
            value: prim::CMatrix;

        }

        # A string literal.
        string_literal {

            # The value of the literal.
            value: prim::Str;

        }

        # A JSON literal.
        json_literal {

            # The value of the literal.
            value: prim::Json;

        }

    }

    # A reference to an object.
    #
    # Besides appearing in the IR, these references are also used to represent
    # data dependencies. In that context, it is also legal to make a reference
    # that does not refer to any object. This is referred to as a null
    # reference. These null references are implicitly "read" by all normal
    # statements, and implicitly "written" by statements of which the order
```

(continues on next page)

(continued from previous page)

```
    # must be preserved, being full barriers (wait/barrier instructions with an
    # empty object list) and control-flow instructions.
    reference {

        # Link to the target object.
        target: Link<object>;

        # The data type that the object is accessed as. In almost all cases,
        # this must be equal to target->data_type. The only exception currently
        # allowed is accessing a qubit type as a bit. This yields the implicit
        # classical bit associated with the qubit in targets which use this
        # paradigm.
        data_type: Link<data_type>;

        # The indices by which the object is indexed. The indices must be
        # integer-like. Depending on context, they may need to be literals.
        # Except within wait instructions, the amount of indices must exactly
        # match the dimensionality of the target object; partial indexation is
        # not supported because the type system doesn't support it. References
        # within wait instructions may have less indices than the dimensionality
        # of the target object.
        indices: Any<expression>;

    }

    # A call to a custom function defined by the target.
    function_call {

        # Link to the function type as defined in the platform.
        function_type: Link<function_type>;

        # The operands for the function. The types of these expressions must
        # match the operand types in the associated function type.
        operands: Any<expression>;

    }

}
```

1.20 Options (of various degrees)

There are a number of ways to make a piece of code execute conditionally in OpenQL, global- and pass options being the most obvious ones. But there are more, with different use cases, so let's go over them.

1.20.1 Pass options

Wherever possible, pass options are what should be used to configure things related to passes.

1.20.2 Global options

Global options are primarily a remnant from before we had passes. They should now only be used for backward compatibility, and for things that cannot be turned into pass options due to needing to be available outside the context of a pass. If you really need them, they are defined in `src/ql/com/options.cc`.

1.20.3 Platform JSON configuration file options

Anything that has access to the IR can read arbitrary JSON from the platform configuration data structure; the raw JSON is always maintained, and unrecognized keys are silently ignored (somewhat by design). So, whenever you have an optional bit of code that's logically related to the platform and that has access to the platform, this is what you should use. Usually, though, it makes more sense to use a pass option.

1.20.4 Preprocessor-based options

The C preprocessor can be used to set options at compile-time. There are various ways to do this (beyond just `#define`) in the context of CMake, which are preferred to the usual C++ way, because they don't require a user to modify code in order to change the option.

Note: All preprocessor macros should use the `QL_` prefix, to avoid name clashes with libraries we're depending on, or (if it ever happens) the codebase of software depending on OpenQL within C++.

CMake `target_compile_definitions`

These are useful for pieces of code that shouldn't even be attempted to be compiled when the option is off. However, the associated preprocessor directive **must not** be used in public header files! After all, if OpenQL would be installed and a user would link against it, they wouldn't use the CMakeLists file at all, and thus you'd end up with arbitrary discrepancies in the header files! When nevertheless used, they should be added at the bottom of the "OpenQL library target" section of the main `CMakeLists.txt` file.

Whether the macro is defined or now would normally be controlled via a CMake option/cache variable, created with the `option()` function. These directives should be placed at the top of the main `CMakeLists.txt` file, in the "Configuration options" section.

CMake `configure_file` options

When the above is not possible because the definition is also needed in public header files, you can instead use `ql/config.h`, again in conjunction with an `option()` directive. To do so:

- add a `#cmakedefine` directive to `src/ql/config.h.template`;
- make sure that the name of the preprocessor directive exists as a yes/no CMake variable when `config.h` is created (also in the "OpenQL library target" section of the main `CMakeLists.txt` file); and
- ensure that `ql/config.h` is included in all files that use the definition.

The latter is of course very important, otherwise you'll get the same kind of discrepancies that you would for `target_compile_definitions`. Basically, this just shifts that problem. Choose the one that's more convenient.

`#define` in some header file somewhere

You can do this if you want, but it should only be done for things that rarely change. After all, you can't change the option without changing sources.

1.21 Passes

Fundamentally, compiler passes are bits of code that transform or analyze the current IR in some way. For example, a scheduler pass will change the cycle numbers of instructions and possibly reorder them accordingly.

Conceptually, OpenQL's passes can get a bit more complicated than that, at least internally. Instead of a linear pass list, OpenQL's pass manager supports a tree of passes (useful for instance to set options for a whole group of passes at once, for establishing logging or performance monitoring regions if that's ever implemented, or otherwise manipulate a group of passes as if it's a single pass), and even has basic support for conditional passes, or repeating a group of passes until a condition is true. To support all of this without making simple cases needlessly complex for users, the lifecycle of a pass is nontrivial.

- First, the pass is created via an `append_pass()` API call (or equivalent).
- Options may then be set on the pass via `set_option()` API calls (or equivalent).
- At some point, the pass is "constructed." This doesn't imply C++ construction of the pass class (this happens at the start of this list); rather, this is about the `construct()` method. The user can either call this directly, or the pass manager will do it automatically when needed. During construction, a pass can choose to become a *group* of passes, rather than staying a single pass. This can be a normal group, a conditional group, a "while" group, or a "repeat until" group. It can make this decision based on its options; therefore, to avoid potential confusion, it's illegal to change the options of a pass after it has been constructed. Instead, if the pass turned itself into a group, the user is subsequently allowed to modify its list of sub-passes, including setting options directly on the sub-passes, or adding new sub-passes. These sub-passes will eventually be constructed again, repeating this process recursively.
- If the pass did not turn into a group, it will eventually be "run" for the given IR tree. A single pass may be run multiple times if it's a sub-pass of a looping block, or it may never be run. If the pass did decide to turn into a group upon construction, the overridable `run()` method is never called; instead, the base class for the pass will ensure that its sub-passes are run appropriately.

As an example of when this might become useful, imagine that eventually the mapper pass is split up into its logical sub-passes, namely optional initial placement, routing, and primitive decomposition. At this point, legacy user code may still assume that the old combined mapper pass exists, and behaves as a single pass within the context of pass management. To support this, the old mapper pass can be defined to construct into a group of its sub-passes, with the initial placement pass added only if initial placement is actually enabled. Now, until the user explicitly constructs the mapper pass (which the old code would have no reason to ever do), the mapper pass behaves exactly as it would have before, i.e. a single pass that can have options applied on it or be deleted, yet the compiler necessarily behaves exactly as if the user had created the initial placement, routing, and primitive decomposition passes manually.

Note: Most of the APIs related to pass groups and management thereof are currently disabled via the `QL_HIERARCHICAL_PASS_MANAGEMENT` preprocessor directive, explicitly `#undef`'d in `src/ql/config.h.template`. Internally, however, everything should already be there.

1.21.1 Implementation

Pass classes

Pass instances and/or groups of passes are represented by a class that (ultimately) derives from `ql::pmgr::pass_types::Base`. There is no associated object for a pass type; rather, the C++ type itself is used for the pass type, so in principle you only have to implement one class to define a pass.

Passes don't normally implement `ql::pmgr::pass_types::Base` directly. Instead, they may use:

- `ql::pmgr::pass_types::Transformation` for regular transformation passes operating on the new IR;
- `ql::pmgr::pass_types::Analysis` for new-IR passes that don't change the IR, aside from possibly adding metadata to it via annotations;
- `ql::pmgr::pass_types::ProgramTransformation` for old-IR passes that transform the complete program in one go;
- `ql::pmgr::pass_types::KernelTransformation` for old-IR passes that transform one kernel at a time;
- `ql::pmgr::pass_types::ProgramAnalysis` for old-IR passes that analyze the complete program in one go; or
- `ql::pmgr::pass_types::KernelAnalysis` for old-IR passes that analyze one kernel at a time.

Note that there is currently no difference between the `*Transformation` and `*Analysis` passes, because there is currently no good way to guarantee constness with the IR tree. They're really just hints right now.

Most passes override the following methods to define their functionality:

- `dump_docs(...)`: a dump function that writes the documentation for the pass type. This must not depend on any pass options; it is only called on "virgin" objects (unfortunately, there is no such thing as overriding static methods in C++).
- `get_friendly_name()`: used by the documentation generation logic to get a user-friendly name for the pass, to use as section header.
- The constructor: used to define pass-specific options.
- `run(...)`: actually runs the pass.

The pass class itself is not the correct place to store variables/fields for the actual algorithm that `run()` implements. Instead, if the implementation of a pass is complex, it's better to make a `detail` namespace for the pass-specific types and functions, and leave the pass class as a thin wrapper around it. Ideally, these wrappers (and pass registration) can then ultimately be generated, preventing a lot of boilerplate code. Even without the generator, it's good to have the boilerplate and documentation generation stuff separate from the actual implementation; the implementation will probably be complex enough as it is.

The pass factory

For a pass type to be usable within a compilation strategy, its class must be registered with the pass factory (`ql::pmgr::Factory`) used to build the strategy with. While the code is written such that it's possible for a user program to eventually make its own pass factory (which would probably be necessary to let them define their own passes), currently everything just uses a default-constructed `Factory` object initially, and its default constructor is where the pass types are registered. For example, this constructor currently contains the following line, among others like it:

```
register_pass<::ql::pass::io::cqasm::read::Pass>("io.cqasm.Read");
```

The template argument (typedefs to) the pass class, while the string argument defines its externally-usable type name.

Note: The C++ namespace path and externally-usable type name path should be kept in sync! Please avoid using differing naming conventions for the two. If needed for backward compatibility, different aliases can be made for the same pass type, but the complement of the C++ name should also be usable as a pass type externally.

Note: The capitalization of the pass types is chosen such to be as familiar as possible to Python users: the last entry represents a class, while the remaining period-separated entries represent module names. In C++ it works the same, except that passes have their own namespace in addition, so you end up with `...::name::Pass` rather than `...::Name`.

After default-construction, the `Factory` object will be "configured" by the pass manager. During configuration, aliases are added for the architecture-specific passes of the selected architecture, preventing the user from having to explicitly prefix these passes using `arch.<arch-name>.`. This mechanism also allows an architecture to override the implementation of a generic pass if it needs to, without breaking backward compatibility, as architecture-specific passes take precedence over generic passes when these aliases are created. Aliases may also be generated for "dnu" (do-not-use) passes that are explicitly requested by the user.

The pass manager

Pass instances are glued together into a pass strategy by the pass manager (`ql::pmgr::Manager`), also known as just the `Compiler` in API terminology. For the most part, this class is just boilerplate around a factory and a single group pass that represents the first level of the pass group hierarchy. However, it also contains a bunch of backward compatibility logic from the olden days when there was no pass management at all by way of the `from_defaults()` and `convert_global_to_pass_options()` methods, and the compiler configuration JSON file loading logic by way of the `from_json()` method.

`convert_global_to_pass_options()` especially requires a bit of attention, because its implementation is currently very stupid: whenever a global option is defined, it effectively calls `set_option()` on any default pass that has an option going by the (converted) global option name. This may not be good enough when more passes are added eventually; for example, if multiple passes have a `heuristic` option, the global option conversion logic has no way of only setting the option for a particular pass type (incidentally, this is why the scheduler heuristic pass option is redundantly named `scheduler_heuristic` instead).

1.21.2 Adding a new pass

Having read the above, adding a new pass should be a fairly straightforward process. Nevertheless, here's a checklist that should handle the common cases.

- Figure out what you want to call the pass, keeping in mind the naming conventions and organizing groups (i.e. `ana`, `io`, `map`, `opt`, and `sch`, see namespaces).
- Create a source file for the pass corresponding to the pass type you settled on in `src/ql/pass`, and an accompanying header file in `include/ql/pass`. The contents can mostly be copypasted from existing passes; much of it is boilerplate.
- Derive from the right base class for your pass (probably `Transformation` or `Analysis`). If needed, change the prototype of the `run()` function accordingly.

- Implement the documentation generation functions. If you can't be bothered to put anything useful there until you're done with the implementation yet then that's on you, but at least put a one-liner placeholder there. Don't just copypaste the documentation of another pass!
- Update the constructor to define the pass options you want for your pass.
- Put an appropriate placeholder in `run()`, such as `QL_ICE("not yet implemented")`.
- Register your pass with the pass factory by adding it to its default constructor.
- At this point, you should have everything needed for the user to be able to create the pass, and for the documentation generation system to detect and add it.
- If you want the pass to become part of the default pass list, add it to `ql::pmgr::Manager::from_defaults()`. Note that it should probably be guarded by a global option that defaults to not inserting the pass for backward compatibility; these are defined in `ql::com::options::make_ql_options()`.
- If you want the pass to become part of an architecture-specific default pass list, add it to the `populate_backend_passes()` method of its `Info` class.
- Actually implement and document the pass. If the implementation is complex, it should be put in a `detail` namespace within the pass namespace, with all (private!) header files and source files in the `src` directory. Any header file that must be public or is used elsewhere within OpenQL, for example one containing annotation types that other passes may want to do something with as well, should *not* be in `detail`; `detail` is your private implementation, anything outside of it is public.

1.22 Resources

In OpenQL, the term "resources" is reserved for *scheduling* resources. Conceptually, a resource models a physical thing that prevents two instructions from executing simulaneously due to resource contention, but ultimately it can be anything that prevents an instruction from starting in a particular cycle.

1.22.1 Implementation

Resources can be used by various passes that somehow relate to scheduling, so they are not an OpenQL-wide thing. Because of how fundamental they are, they received their own namespaces and management logic, not unlike the pass manager and pass namespaces. In fact, much of what applies for passes also applies for resources.

Resource classes

Resource instances are represented by a class that (ultimately) derives from `ql::rmgr::resource_types::Base`. Unlike their pass equivalent, resource classes normally derive from this directly. Like passes, there is no associated object for a resource type; rather, the C++ type itself is used for the resource type, so in principle you only have to implement one class to define a new resource type.

Resource classes override the following methods to define their functionality:

- `on_dump_docs(...)`: a dump function that writes the documentation for the resource type. This must not depend on its JSON configuration; it is only called on "virgin" objects (unfortunately, there is no such thing as overriding static methods in C++).
- `on_dump_config(...)`: a dump function that prints the configuration of the resource in a user-friendly way.
- `on_dump_state(...)`: a dump function that prints the current state of the resource, mid-scheduling.

- `get_friendly_name()`: used by the documentation generation logic to get a user-friendly name for the resource, to use as section header.
- `on_initialize()`: used to initialize the state of the resource for a particular scheduling direction.
- `on_gate()`: used to test or update the state during scheduling. Its arguments include a gate reference, the desired start cycle for the gate, and whether to update the state to reflect that the given gate has actually been scheduled in that cycle.

The semantics of `on_gate()` should be as follows:

- If the given gate can *not* be scheduled in the given cycle and `commit` is false, return false.
- If the given gate can *not* be scheduled in the given cycle and `commit` is true, throw an exception.
- If the given gate *can* be scheduled in the given cycle and `commit` is false, return true.
- If the given gate *can* be scheduled in the given cycle and `commit` is true, update the resource state accordingly and return true.

Resource class instances must be copy-constructable, in such a way that the actual state of the resource is appropriately copied. This is to allow recursive-descent scheduling algorithms that try different solutions to be created with them. Ideally, configuration data should *not* be copied to save space and increase performance of the copy operation; it's advisable to store configuration data in a separate structure and embed it into the resource class via a `utils::Ptr<>`.

Depending on the direction argument passed to `on_initialize()`, resources may assume that cycle numbers are non-decreasing or non-increasing. This often allows old state information to be deleted, thus further reducing the overhead of recursive-descent algorithms.

The resource factory

For a resource type to be usable within the platform configuration file, its class must be registered with the resource factory (`ql::rmgr::Factory`). Registration happens in the default constructor of the class. For example, this constructor currently contains the following line, among others like it:

```
register_resource<resource::qubit::Resource>("Qubit");
```

The template argument (typedefs to) the resource class, while the string argument defines its externally-usable type name.

Note: The C++ namespace path and externally-usable type name path should be kept in sync! Please avoid using differing naming conventions for the two. If needed for backward compatibility, different aliases can be made for the same resource type, but the complement of the C++ name should also be usable as a resource type within the platform.

Note: The capitalization of the resource types is chosen such to be as familiar as possible to Python users: the last entry represents a class, while the remaining period-separated entries represent module names. In C++ it works the same, except that resources have their own namespace in addition, so you end up with `...::name::Resource` rather than `...::Name`. Note however that the CC-light resource types use lowercase names for backward compatibility.

After default-construction, the `Factory` object will be "configured" by the resource manager. During configuration, aliases are added for the architecture-specific resources of the selected architecture, preventing the user from having to explicitly prefix these resources using `arch.<arch-name>.`. This mechanism also allows an architecture to override the implementation of a generic resource if it needs to, without breaking backward compatibility, as architecture-specific resources take precedence over generic resources when these aliases are created. Aliases may also be generated for "dnu" (do-not-use) resources that are explicitly requested by the user.

The resource manager

Resource instances are bundled together by the resource manager (`ql::rmgr::Manager`), stored as part of the platform structures. The resource manager is not much more than a wrapper around a list of resources, the JSON loading logic to create this list, and boilerplate code for documentation generation stuff. Once the list of resources is complete, schedulers can use its `build()` method to construct a clean `ql::rmgr::State` object.

The state object

The state object, like the resource manager, is just a wrapper around a list of resources. It differs in where it's used: the state object tracks the state of all the resources during scheduling, while the resource manager is conceptually just a builder for that state. The state object presents the following interface to scheduling algorithms:

- `available()`: used to query whether a gate can be scheduled in a particular cycle (it calls `on_gate()` for all resources with commit set to false, returning true only if all resources returned true);
- `reserve()`: used to update the state to reflect that the given gate is scheduled in the given cycle; and
- the copy constructor/copy assignment operator: to allow a state to be copied for recursive-descent scheduling algorithms.

1.22.2 Adding a new resource

Having read the above, adding a new resource should be a fairly straightforward process. Nevertheless, here's a checklist that should handle the common cases.

- Figure out what you want to call the resource, keeping in mind the naming conventions.
- Create a source file for the resource corresponding to the resource type you settled on in `src/ql/resource`, and an accompanying header file in `include/ql/resource`. It's probably easiest to copypaste the contents from an existing resource.
- Implement the documentation generation functions. If you can't be bothered to put anything useful there until you're done with the implementation yet then that's on you, but at least put a one-liner placeholder there. Don't just copypaste the documentation of another pass!
- Put an appropriate placeholders in `on_initialize()` and `on_gate()`, such as `QL_ICE("not yet implemented")`.
- Register your resource with the resource factory by adding it to its default constructor.
- At this point, you should have everything needed for the user to be able to add the resource to the platform configuration file, and for the documentation generation system to detect and add it.
- Actually implement and document the resource. Unlike for passes, any state tracked by the resource must be part of the resource object, and the implementation of the resource is usually (at least for the existing resources) not so complicated that it warrants its own detail namespace or even types and functions outside of the resource class, but of course you're free to make one anyway as per the naming conventions if you prefer.

1.23 Doxygen documentation

Documentation for the C++ source code is generated by Doxygen. Properly documenting everything with docstrings is unfortunately still an ongoing process, but at the very least all the functions and classes should be there, as automatically documented by Doxygen.

1.24 Changelog

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

1.24.1 [next] - [TBD]

Added

- . . .

Changed

- . . .

Removed

- . . .

Fixed

- . . .

1.24.2 [0.10.0] - [2021-07-15]

Added

- scalability options for coping with large multi-core systems, including a progress bar for the mapping process
- initial implementation of the Diamond architecture developed for Fujitsu (lead by Stephan Wong)
- full cQASM 1.2 read and write support, with options for different version levels and various language quirks
- new internal representation that encompasses the entire cQASM 1.2 language, and has many new generalized platform features
- lossless conversion functions between the two IR representations until all passes have been converted to the new representation
- new pass-based decomposition logic that supports arbitrary cQASM 1.2 code for the expansion and doesn't clobber scheduling information
- new pass for converting structured cQASM 1.2 control flow to basic block form
- new list scheduler based on the new IR

Changed

- all written cQASM files are now in 1.2 format by default
- the cQASM reader no longer has a JSON configuration file for mapping cQASM gates to OpenQL gates; this translation is now part of OpenQL's platform data
- the old scheduler is replaced with a new implementation for most option variations, that outputs slightly different schedules
- statistics report output is also formatted slightly different, though information content is the same
- CC backend:
  - scheduling is now done using resource constraints by default

Fixed

- excessive memory usage and slow platform construction for large multi-core systems

1.24.3 [0.9.0] - [2021-05-27]

Added

- architecture system: platform and compilation strategy defaults are now built into OpenQL, preventing the need for users to copypaste configuration files from the tests directory
- interface (C++ and Python) to compile cQASM 1.x
- allow 'wait' and 'barrier' in JSON section 'gate_decomposition'
- CC backend:
  - improved reporting on JSON semantic errors
  - added check for dimension of "instruments/qubits" against "instruments/ref_control_mode/control_bits"
  - added check for dimension of "instructions//cc/[signals,ref_signal]/value" against "instruments/ref_control_mode/control_bits"
  - added cross check of "instruments/ref_control_mode" against "instrument_definitions"
  - added support for "pragma/break" in JSON definition to define 'gate' that breaks out of loop
  - added support to distribute measurement results via DSM
  - added support for conditional gates
  - added compile option "–backend_cc_run_once"
  - added compile option "–backend_cc_verbose"

Changed

- pass management: instead of a hardcoded compilation strategy, the strategy can be adjusted and fine-tuned manually
- pass options: instead of doing everything with global options, global options were replaced with pass options as much as possible
- most documentation is now generated from code and can be queried using API calls
- scheduler resources are completely reworked to be made more generic
- major internal refactoring and restructuring to facilitate the above two things
- CC backend:
  - renamed JSON field “signal_ref” to “ref_signal”
  - renamed JSON field “ref_signals_type” to “signal_type”
  - changed JSON field “static_codeword_override” to be a vector with one element per qubit parameter. To edit a JSON file using Sublime, use Replace with Regular Expressions: find=`"static_codeword_override": ([0-9])+,` replace=`"static_codeword_override": [\1]`
  - adopted new module synchronization scheme (“seq_bar semantics”, requires CC software >= v0.2.0, PycQED after commit 470df5b)
  - JSON field “instruction/type” no longer used by backend, use “instruction/cc/readout_mode” to flag measurement instructions
  - allow specification of 2 triggers in JSON field “control_modes/*/trigger_bits” to support dual-QWG
  - changed label in generated code from “mainLoop” to “**mainLoop”. Do not start kernel names with “**” (this should be specified by the API)
  - removed initial 1 cycle (20 ns) delay at start of kernels (resulting from bundle start_cycle starting at 1)
  - correctly handle kernel names containing “_” in conjunction with looping (formerly duplicate labels could arise)
  - added “seq_out 0,1” to program start to allow tracing of actual program start

Removed

- CC-light code generation, as the CC-light is being phased out in the lab, and its many passes were obstacles for pass management and refactoring
- rotation optimization based on matrices; matrices in general were removed entirely because no one was using it
- the commute variation pass, as it has been superseded by in-place commutations within the scheduler
- the toffoli decomposition pass, as it wasn’t really used; to decompose a toffoli gate, use generic platform-driven decomposition instead
- the defunct fidelity estimation logic from metrics.cc; this may be added again later, but requires lots of cleanup and isn’t currently in use
- quantumsim and qsoverlay output; apparently this was no longer being used, and it was quite intertwined with the CC-light backend

Fixed

- changed register used for FOR loop, so it doesn't clash with delay setting
- fixed documentation for python setup and running tests
- various miscellaneous bugs, dangling pointers, and memory leaks

1.24.4 [0.8.0] - [2019-10-31]

Added

- support for CC backend

Changed

Removed

Fixed

- fixed issue with duplicate kernel names
- updated json library to fix osx builds

1.24.5 [0.7.1] - [2019-09-02]

Added

Changed

- re-factored folders

Removed

Fixed

- fixed issue with correct python library picking on tud win systems

1.24.6 [0.7.0] - [2019-06-03]

Added

- support for single qubit flux options (auto/manual modes)
- option to control generation of qasm files and dot graphs
- NPROCS=n variable can now be set for faster compilation to use n threads
- conda build recipe
- conda binary releases for Linux, Windows platform (not yet available for OSX due to a conda distribution issue)

Changed

- openql is now public
- improved resource-constrained scheduling
- sweep point array is now optional
- support for barrier/wait on all qubits

Removed

- set_sweep_points(sweep_points list, num of sweep points)

Fixed

- resource-constrained qasm is generated by same scheduler for cc-light as is used to generate qisa
- Illegal parameter in gate_decomposition

1.24.7 [0.6] - [2018-10-29]

Added

- generated qasm code conforms to cQASM v1.0 specification
- added libqasm to pytest to test conformance of generated qasm

Changed

- ALAP scheduler is the default option (Issue #193)
- compiling an empty program raises error (Issue #164)

Removed

Fixed

- tests are added to test option setting/getting (Issue #190)

1.24.8 [0.5.5] - [2018-10-25]

Added

Changed

- simplified interface of Program.set_sweep_points (Issue #184)

Removed

Fixed

- instruction ordering to generate consistent qisa (Issue #190)
- stateful behaviour in OpenQL (Issue #171)

1.24.9 [0.5.4] - [2018-10-17]

Added

Changed

Removed

Fixed

- qubit ordering in SMIS and SMIT instructions

1.24.10 [0.5.3] - [2018-10-11]

Added

- added detuning constraints for cclight

Changed

Removed

Fixed

- alap scheduling for cclight

1.24.11 [0.5.2] - [2018-10-10]

Added

Changed

Removed

Fixed

- wrong target qubits in the configuration files
- Jenkins test build profile to test against assembler

1.24.12 [0.5.1] - [2018-09-12]

Added

- API to obtain version number

Changed

Removed

Fixed

- qisa format (removed comma)

1.24.13 [0.5] - [2018-06-26]

Added

- support for classical instructions
- support for flow control (selection and repetition)
- classical register manager implementation

Changed

- measure instruction updated to support classical target register
- kernels are not any more fused to generate a single qisa program

Removed

- kernel does not recieve iteration count, deprecated in favor of for-loop

Fixed

- qisa format (pre-interval syntax updated)

1.24.14 [0.4.1] - [2018-05-31]

Added

-

Changed

-

Removed

-

Fixed

- getting started example

1.24.15 [0.4] - [2018-05-19]

Added

- kernel conjugation/un-compute feature
- multi-qubit control decomposition
- toffoli decomposition
- QASM loader for QASM v1.0 syntax check
- initial support for Quantumsim backend
- vebosity levels

Changed

- program options can be set/get with simple api calls
- when adding gates, qubits should always be specified as list
- updated qisa-as support for tests

Removed

- qisa-as is not a part of openql
- prog.compile() does not get optimiz/schedule/verbose options

Fixed

- static iteration count for scheduled qasm
- roation angle printing

1.24.16 [0.3] - [2017-10-24]

Added

- CCLight eQASM compiler
- unittests using qisa-as
- simplified gate decompositions
- wait/barrier instructions

Changed

-

Removed

-

Fixed

- varying prepz duration
- M_PI issue in windows install

1.24.17 [0.2] - [2017-08-18]

Added

- CBox eQASM compiler
- Python and C++ interface
- Configuration file specifiction
- trace support for qumis code
- cmake based builds

Changed

-

Removed

-

Fixed

1.25 Contributors

OpenQL framework has been created initially by Nader Khammassi.

Note: please fill your contributions in this file

- Nader Khammassi
  - CBox Backend
  - Configuration file support
  - QASM loader for QASM syntax check
  - C++ exceptions
  - QISA map file generation
  - QISA Control store generation
- Imran Ashraf
  - support for hybrid classical/quantum compilation
  - support for control flow (selection and repetition)
  - kernel un-compute/conjugation feature
  - multi-qubit control decomposition
  - toffoli decompositions
  - openql intermediate representation
  - quantumsim simulator Backend
  - compilation for CC-Light architecture

    ```
    - resource-constrained scheduling
    - parallel (SIMD and VLIW) QISA code generation
    ```

  - flexible platform constraints specification and its implementation
  - support for multi-qubit gates
  - scheduling (ASAP/ALAP) algorithms
  - parametrized gate decomposition
  - unit-tests
  - python Package for OpenQL
  - cmake-based Compilation for cross-platform build setup
  - conda recipies and packages
  - single qubit flux operations
  - cQASM v1.0 support

  - OpenQL documentation
- Adriaan Rol
  - Contributed to the Hardware Configuration Specification
  - Utilizing qisa-as in unit-tests
  - Testing OpenQL on the Hardware
- Xiang Fu
  - Contributed to the Hardware Configuration Specification
  - Testing OpenQL on the Hardware
- Wouter Vlothuizen
  - backend for Central Controller (CC)
  - new simplified qubit numbering scheme (rotated surface code fabric by 45 deg)
  - support for comments in JSON file
  - show line number and position on JSON syntax errors
  - cleanup
- Hans van Someren
  - uniform scheduling algorithm
  - resource constraint framework design
  - resource constraint description for CC-Light architecture
  - resource constrained list scheduling algorithms
  - backward resource constraint checking
  - forward and backward list scheduling algorithms
  - gate commutation while scheduling
  - clifford gate sequence optimization
  - out of order gate creation
  - staged decomposition description
  - generalized passes, dumping and reporting
  - platform topology specification and its implementation
  - single qubit flux operations design
  - initial placement mapping implementation
  - basic routing implementation
  - latency sensitive routing
  - resource constrained routing
  - scheduler integration into routing
  - use moves next to swaps while routing
  - crossbar spin-qubit scheduling and resource management
  - recursive look-back and look-ahead routing

  - arbitrary topology routing
  - OpenQL documentation
- Fer Grooteman
  - added interface (C++ and Python) to compile cQASM 1.0
- Anneriet Krol
  - unitary decomposition support
- Razvan Nane
  - compiler API and modularity support
  - added C printer pass
- Jeroen van Straten
  - tutorial on DQCsim + OpenQL interoperation
  - doxygen documentation
  - improved pass management
  - extensive cleanup; basically a rewrite of everything at this point
- Quinten van Wingerden
  - added support for diamond architecture

1.26 Program

Warning: This page has not been revised yet since modularization and refactoring, and may thus be out of date.

In the OpenQL programming model, one first creates the platform object and then with it the program object. After that, one creates kernels with gates and adds these kernels to the program. Finally, one compiles the program and executes it. At any time, options can be set and got.

We saw an example of this in *Creating your first program*. Here it is again but then with everything glued together:

```
from openql import openql as ql

platform = ql.Platform("myPlatform", "hardware_config_cc_light.json")

nqubits = 3

p = ql.Program("aProgram", platform, nqubits)

k = ql.Kernel("aKernel", platform, nqubits)

for i in range(nqubits):
    k.gate('prepz', [i])

k.gate('x', [0])
k.gate('h', [1])
k.gate('cz', [2, 0])
k.gate('measure', [0])
```

(continues on next page)

(continued from previous page)

```
k.gate('measure', [1])

p.add_kernel(k)

p.compile()
```

Platform creation takes a name (to use in information messages) and the name of the platform configuration file. The latter is used to initialize the platform attributes, e.g. to create custom gates.

A program is created by specifying a name, the platform, and the numbers of quantum and classical registers. The name can be used as seed to create output file names and is used in information messages.

The main structural attribute of a program is its vector of kernels. This vector is in the simplest form initialized by adding kernels one by one to it. The order of execution is then the order of the kernels in the vector. But there are also program APIs to create control flow between kernels such as if/then, if/then/else, do/while and for. These take one or more kernels representing the then-part, the else-part, or the loop-body, and add special kernels around them to represent the control flow. These latter APIs also take the particular branch condition or the number of iterations as parameter. See *Kernel* for an overview of these APIs and see *Classical gate attributes in the internal representation* for a definition of the control flow internal representation.

The gates of a kernel's circuit are always executed sequentially. At the end of a circuit, control passes on to the next kernel in the program's vector of kernels.

After having completed adding kernels, the program has been completely specified. It is represented by a vector of kernels, each with a circuit. And in this form, the program is compiled by invoking its `p.compile()` method.

In the `p.compile()` method, the platform independent compiler passes and then the platform dependent compiler passes are called one by one in the order specified by the OpenQL compiler's internals. After compilation, the `p.compile()` method returns, with the internal representation still available. Compilation will have resulted in the creation of several external representations, to be used by e.g. simulation, assembly/execution or human inspection.

[API TBD]

1.27 Kernel

Warning: This page has not been revised yet since modularization and refactoring, and may thus be out of date.

A kernel conventionally models a circuit with quantum gates ending in one or more measurements. In OpenQL, this has been extended with:

- control flow that can jump at the end of a kernel to the start of another kernel; see this section
- classical gates mixed with quantum gates (including measurements) in a single circuit with the design objective to support control flow changes; see *Classical Instructions*; measurements are just gates with classical results and can be anywhere, bridging quantum code to classical code; a kernel doesn't necessarily have to contain a measurement

In OpenQL a `kernel` is an object; it has a name, a type, and a circuit as its main structural attributes. This circuit is a vector of gates.

The type of a kernel with a non-empty circuit with gates is `STATIC`. During execution, all gates of such a circuit are executed from the start to the end. After executing the last gate, control will be transferred to the next kernel in the vector of kernels. This vector of kernels is an attribute of the governing program object.

You saw a first kernel which was a `STATIC` one being created in the example program in *Program*.

Kernels of other types are used to represent control flow. This is the topic of the remainder of this section. If you are not interested in this now, you can read this later.

Let us first look at some example Python OpenQL code (adapted from *tests/test_hybrid.py*):

```
num_qubits = 5
num_cregs = 10

p = ql.Program('test_classical', platform, num_qubits, num_cregs)
kfirst = ql.Kernel('First', platform, num_qubits, num_cregs)

# create classical registers
rs1 = ql.CReg()
rs2 = ql.CReg()

# if (rs1 == rs2) then Thenpart else Elsepart endif
kfirst.classical(rs1, ql.Operation(...))
kfirst.classical(rs2, ql.Operation(...))
kthen = ql.Kernel('Thenpart', platform, num_qubits, num_cregs)
kthen.gate('x', [0])
kelse = ql.Kernel('Elsepart', platform, num_qubits, num_cregs)
kelse.gate('y', [0])
p.add_if_else(kthen, kelse, ql.Operation(rs1, '==', rs2))

# loop 10 times over Loopbody endloop;
kloopbody = ql.Kernel('Loopbody', platform, num_qubits, num_cregs)
kloopbody.gate('x', [0])
p.add_for(kloopbody, 10)
# Afterloop
kafterloop = ql.Kernel('Afterloop', platform, num_qubits, num_cregs)
kafterloop.gate('y', [0])
p.add_kernel(kafterloop)

# do Dowhileloopbody while (rs1 < rs2)
kdowhileloopbody = ql.Kernel('Dowhileloopbody', platform, num_qubits, num_cregs)
kdowhileloopbody.gate('x', [0])
kdowhileloopbody.classical(rs1, ql.Operation(...))
kdowhileloopbody.classical(rs2, ql.Operation(...))
p.add_do_while(kdowhileloopbody, ql.Operation(rs1, '<', rs2))
# Afterdowhile
kafterdowhile = ql.Kernel('Afterdowhile', platform, num_qubits, num_cregs)
kafterdowhile.gate('y', [0])
p.add_kernel(kafterdowhile)

p.compile()
```

These are three examples in one:

- the first creates an if-then-else construct under the condition that `rs1` equals `rs2`
- the second creates a for loop with 10 iterations
- the third one creates a do-while construct executing the `Dowhileloopbody` as long as `rs1` is less than `rs2`

We see that ordinary kernels are created and filled with a single gate; these are the `STATIC` kernels. These kernels serve as the thenpart, elsepart, loopbody, etc. And then we see three examples of the creation of a control-flow construct, with the ordinary kernels as parameters.

After this, we'll have the following 15 kernels in the `kernels` vector of program `test_classical` (some are named after their `type`, see below): `First`, `IF_START`, `Thenpart`, `IF_END`, `ELSE_START`, `Elsepart`, `ELSE_END`, `FOR_START`, `Loopbody`, `FOR_END`, `Afterloop`, `DO_WHILE_START`, `Dowhileloopbody`, `DO_WHILE_END`, `Afterdowhile`.

1.27.1 Control flow in the internal representation

The classical gates in *Classical Instructions* deal with classical computation. Control flow is represented in the internal representation as kernels of a special type, with their special attributes.

The relevant kernel attributes are `type`, `name`, `iterations`, and `br_condition`. How these relate, is summarized in the next table:

type	name	circuit	br_condition	iterations	example OpenQL creating this kernel
STATIC	label	gates			p.add(ql.kernel(label, . . .))
FOR_START	body.name+'for_start'			loopcount	p.add_for(body, loopcount)
FOR_END	body.name+'for_end'				
DO_WHILE_START	body.name+'do_while_start'		loopcond		p.add_do_while(body, loopcond)
DO_WHILE_END	body.name+'do_while'				
IF_START	then.name+'if'		thencond		p.add_if(then, thencond)
IF_END	then.name+'if_end'				
ELSE_START	else.name+'else'				p.add_if_else(then, else, thencond)
ELSE_END	else.name+'else_end'				

The example OpenQL in the last column shows how a kernel of the type is created. The table also shows how the parameters of the OpenQL call creating the kernel are used to initialize the kernel's attributes.

Further information on these attributes:

- `name` is unique among the other names of kernels and is often used to construct a label before the first gate of the circuit; for non-`STATIC` kernels it is generated in a systematic way from the name of the first kernel of the body (or then or else part) and from the kernel type to make it easy to generate the conditional branches to the respective label; the `name` column suggests a way but in practice this can more complicated in the presence of nested constructs (then additional counts are needed) or in the presence of multiple kernels (a `program` object) constituting the body (or then or else part)
- `circuit` (the real kernel attribute name is `c` but this is very non-descriptive) contains the gates and is empty for non-`STATIC` kernels
- `br_condition` is an expression that is created by a call to an `Operation()` method (see *Classical Instructions*); it represents a condition so it must be of `RELATIONAL` type; this attribute stores the condition under which the (first) body of the conditional construct is executed; the latter is the kernel referenced by `then` in case of an if or an if-else; and it is the kernel representing the loop's body in case of a do-while. `body`, `then`, and `else` all stand for references to the other kernels in the respective constructs. Similarly, `loopcond`, and `thencond` stand for the expressions representing the condition.

`loopcount` and `iterations` are of type `size_t` and so are non-negative and are assumed to have a value of at least 1.

The semantics of a kernel with respect to control flow is described next, separately for each kernel type:

- `type` is `STATIC`: the kernel's circuit is meant to be executed sequentially from start to end; after executing the last gate, control is transferred to the next kernel in the vector of kernels
- `type` is `FOR_START`: the kernel sets up a loop with `iterations` specifying the iteration count, of which the loop body starts with the next kernel, and of which the loop body ends with the first kernel with type `FOR_END`
- `type` is `FOR_END`: the kernel takes care of control transfer to the start of the loop by decrementing the iteration counter and conditionally branching to the start of the loop body as long as the counter is not `0`

- `type` is `DO_WHILE_START`: the kernel sets up a conditional loop of do-while type, of which the loop body starts with the next kernel, and of which the loop body ends with a matching kernel with type `DO_WHILE_END`
- `type` is `DO_WHILE_END`: the kernel takes care of conditional control transfer to the start of the loop by checking the specified branch condition `br_condition` and conditionally branching to the start of the loop body as long as it evaluates to `true`
- `type` is `IF_START`: the kernel takes care of checking the specified branch condition `br_condition` and conditionally branching to a matching kernel with type `IF_END` when it evaluates to `false`
- `type` is `IF_END`: the kernel signals a merge of control flow from an `IF_START` type kernel
- `type` is `ELSE_START`: the kernel takes care of checking the specified branch condition `br_condition` and conditionally branching to a matching kernel with type `ELSE_END` when it evaluates to `true`
- `type` is `ELSE_END`: the kernel signals a merge of control flow from an `ELSE_START` type kernel

The kernel's `name` functions as a label to be used in control transfers.

Note There aren't gates for control flow (*control gates*), only kernel attributes.

Note Control flow gates cannot be configured in the platform configuration file.

Note Control flow instructions/gates cannot be scheduled.

Note Code generation of control flow, i.e. the mapping from the internal representation to the target platform's instruction set and to QASM requires code inside the OpenQL compiler that is at a different place than the mapping of gates in the internal representation to the target platform's instruction set or QASM; that there have to be these parallel pieces of code inside the OpenQL compiler complicates the compiler unnecessarily.

Note Scheduling around control flow, i.e. defining durations, dependences, relation to resources, is irregularly organized as well; a property of scheduling is that once scheduling of the main code has been done, all later additional scheduling must not disturb the first schedule, and thus that usually to accomplish this, more strict constraints are applied with less optimal code as result; and any attempt is error-prone as well. It also means that the number of cycles to transfer control flow from one kernel to the next kernel is not modeled and that loop scheduling and other forms of inter-kernel scheduling are unnecessarily hard to support.

1.27.2 Control flow in the output external representation

As explained above in *Kernel*, the kernels in the `kernels` vector of a program by default execute in the order of appearance in this vector, i.e. at the end of each kernel, control is transferred to the next kernel in the vector. This holds for kernels of `type STATIC`, the type of kernels that store the gates.

When generating control flow, before the start and/or after the end of a kernel additional code is generated, depending on the kernel's `type`. The code before the start of a kernel is called `prologue`. The code of the kernel itself is called `body`. The code after the end of a kernel is called `epilogue`.

In this, frequently a QASM conditional branch or the conditional branch with the condition inversed is generated. The following table shows by example which conditional branch and inversed conditional branch is generated for a particular `br_condition`, `operands`, and `target label`:

br_condition	operands	target label	QASM cond. branch	QASM inv. cond branch
"eq"	rs1, rs2	label	beq rs1, rs2, label	bne rs1, rs2, label
"ne"			bne rs1, rs2, label	beq rs1, rs2, label
"lt"			blt rs1, rs2, label	bge rs1, rs2, label
"gt"			bgt rs1, rs2, label	ble rs1, rs2, label
"le"			ble rs1, rs2, label	bgt rs1, rs2, label
"ge"			bge rs1, rs2, label	blt rs1, rs2, label

The following is generated for a QASM prologue:

- the `name` of the kernel as label
- in case of `IF_START`: an inverse conditional branch for the given `br_condition` over the `then` part to the corresponding IF_END kernel
- in case of `ELSE_START`: a conditional branch for the given `br_condition` over the `else` part to the corresponding ELSE_END kernel
- in case of `FOR_START`: the initialization using `ldi``s of r29, r30 and r31 with ``iterations`, 1 and 0, respectively, in which r30 is the increment, and r31 the loop counter

The following is generated for a QASM epilogue:

- the `name` of the kernel as label
- in case of `DO_WHILE_END`: a conditional branch for the given `br_condition` back over the `body` part to the corresponding `DO_WHILE_START` kernel
- in case of `FOR_END`: an "add" to r31 of r30 (which increments the loop counter by 1), and a conditional branch as long as r31 is less than r29, the number of iterations, to the loop body

1.27.3 API

In OpenQL this kernel object also supports adding gates to its circuit using the kernel API. To that end, a kernel object has attributes such as `qubit_count`, and `creg_count` to check validity of the operands of the gates that are to be created. And it needs to know the platform configuration file that is to be used to create custom gates; for this, the API that creates a kernel object has the platform object as one of its parameters. Next to this, the kernel object has a method to create each particular default gate.

[TBD]

1.28 Quantum Gates

Warning: This page has not been revised yet since modularization and refactoring, and may thus be out of date.

Gates in OpenQL are the constructs that refer to operations to be executed somehow on the computing platform.

A gate refers to an operation and to zero or more operands.

Gates are organized in circuits as vectors of gates, i.e. linear sequences of gates. A circuit defines the operation of a kernel. And a program consists of multiple kernels.

Gates can be subdivided in several kinds. This is useful in the description of the passes below.

First, gates can be subdivided according to where their execution has effect:

- quantum gates; these gates execute in the quantum computing hardware; these gates have at least one qubit as (implicit or explicit) operand; these gates can have classical registers as operand as well and may rely on some execution capability in classical hardware
- classical gates; these gates execute in classical hardware only; these don't have any qubit as operand, only zero or more classical registers
- directives; these gates execute neither in quantum nor in classical hardware; these look like gates but don't influence execution, e.g. the display gate

Quantum gates can also be subdivided seen from the state of a qubit:

- preparation gates; (usually one-qubit) gates taking qubits in an undefined state and bringing them in a particular defined state
- rotation gates; gates that perform unitary rotations on the state of the operand qubits; examples are identity, x, rx(pi), cnot, swap, and toffoli.
- measurement gates; gates that measure out the operand qubits, leaving them in a base state; the measurement result can be left in a classical register
- scheduling gates; gates that only influence execution timing regarding the operand qubits; they provide a cycle window for the qubit state to be operated upon before further use; examples are the wait and barrier gates

Quantum gates can further be subdivided from the number of operands they take; this becomes relevant when gates are mapped on a quantum computing platform that only supports two-qubit rotation gates when the operand qubits are physically adjacent, as is the case in CC-Light:

- one-qubit gates; quantum gates operating on one qubit
- two-qubit gates; quantum gates operating on two qubits; e.g. two-qubit rotation gates are the main objective in the current mapping pass since these gates require their qubit operands to be connected in CC-Light
- multi-qubit gates; quantum gates operating (implicitly or explicitly) on more than two qubits; e.g. multi-qubit rotation gates must be decomposed to one-qubit and two-qubit gates because more-qubit primitive rotation gates are not supported by CC-Light

Particular classes of quantum gates can be further recognized; these definitions are given mainly to refer to from other chapters of this documentation, especially from the compiler passes chapter and the quantum gate chapter:

- primitive gates; quantum gates natively supported by instructions of the quantum computing platform
- pauli gates; the Identity, X, Y and Z rotation gates
- clifford gates; the one-qubit clifford gates form a group of 24 elements / equivalence classes each composed from a sequence of one or more rotations by a multiple of 90 degrees in one dimension (X, Y or Z)
- default gates; quantum gates predefined by OpenQL
- custom gates; quantum gates defined by the platform configuration file
- composite gates; custom gates that are decomposed to their component gates when created
- specialized gates; custom gates with a definition in the configuration file that is specific for the particular qubit operands that are specified in it; the semantic attributes of several specialized gates with the same quantum operation but different qubit operands may differ (in-line with the purpose of a gate being specialized)
- parameterized gates; custom gates that are not specialized, i.e. with a definition that is not specific for particular qubit operands; all gates created (usually for different qubits) from the same parameterized gate in the platform configuration file have the same semantic attributes

1.28.1 Quantum gate attributes in the internal representation

Quantum gate attributes can be subdivided in the following kinds:

- structural attribute; these attributes define the gate, and are mandatory; key examples are operation name and operands. These attributes are taken from the OpenQL program or the QASM external representation of a gate. These never change after creation and usually are identical over multiple compilations.
- semantic attribute; these attributes define more of the semantics of the gate, usually for a specific purpose; their value fully depends on and is derived from the gate's structural attributes. In OpenQL they are defined in the configuration file. Furthermore, these attributes usually don't change during compilation, although that would be possible when done in a consistent way over all gates. The latter is consistent with changing the configuration file with respect to the values of the semantic attributes.
- result attribute; the value of these attributes is computed during compilation. Usually there is a choice from various strategies and platform parameters how to compute these and so result attributes are seen as an independent kind of attributes. A key example is the cycle attribute as computed by the scheduler. At the start of compilation, their value is undefined.

Below the OpenQL gate attributes are summarized in a table together with their key characteristics.

Attribute	kind	example	used by	updated by	C++ type
name	structural	"CZ q0,q1"	all passes	never	string
operands		[q0,q1]			vector<size_t>
creg_operands		[r23]			vector<size_t>
angle		numpy.pi			double
type		__t_gate__			gate_type_t
duration	semantic	80	schedulers, etc.		size_t
mat			optimizer pass		cmat_t
cycle	result	4	code generation	scheduler	size_t

Custom gates have an additional set of attributes, primarily supporting the initialization of the gate attributes from configuration file parameters.

Some further notes on the gate attributes:

- the name of a gate includes the string representations of its qubit operands in case of a specialized gate; so in general, when given a name, one has to take care to isolate the operation from it; one may assume that the operation is a single identifier optionally followed by white space and the operands
- gates are most directly distinguished by their name

  Note Distinguishing gates internally in the compiler by their name is problematic; distinguishing by their type (see the table below) would be better; the latter conveys the semantic definition and is independent of the representation (e.g. `mrx90`, `mx90`, and `Rmx90` all could be names of a -90 degrees X rotation); furthermore, a name is something of the external representation and is mapped to the internal representation using the platform configuration file; however, the enumeration type of type can never include all possible gates (e.g. those with arbitrary angles, any number of operands, etc.) so the type inevitably can be imprecise; but it can be precise when the type refers to the operation only, i.e. excluding the operands

- qubit and classical operands are represented by unsigned valued indices starting from 0 in their respective register spaces
- `angle` is in radians; it specifies the value of the arbitrary angle of those operations that need one; it is initialized only from an explicit specification as parameter value of the `gate` creation API; expressions initializing this parameter are usually based on some definition of `pi` such as from `numpy`

- `duration` is in nanoseconds, just as the timing specifications in the platform configuration file; scheduling-like passes divide it (rounding up) by the cycle_time to compute the number of cycles that an operation takes; it is initialized implicitly when the gate is a default gate or a custom gate, or explicitly from a parameter value of a gate creation API
- `mat` is of a two-dimensional complex double valued matrix type with dimensions equal to twice the number of operands; it is only used by the optimizer pass; it is initialized implicitly when the gate is a default gate or a custom gate
- `cycle` is in units of cycle_time as defined in the platform; the undefined value is `std::numeric_limits<int>::max()` also known as `INT_MAX`. A gate's cycle attribute gets defined by applying a scheduler or a mapper pass, and remains defined until any pass is done that invalidates the cycle attribute. As long as the gate's cycle attribute is defined (and until it is invalidated), the gates must be ordered in the circuit in non-decreasing cycle order. Also, there is then a derived internal circuit representation, the bundled representation. See *Circuits and bundles in the internal representation*. The cycle attribute invalidation generally is the result of adding a gate to a circuit, or any optimization or decomposition pass.
- type is an enumeration type; the following table enumerates the possible types and their characteristics:

type	operands	example in QASM	kind
__identity_gate__	1 qubit	i q[0]	rotation
__hadamard_gate__		h q[0]	
__pauli_x_gate__		x q[0]	
__pauli_y_gate__		y q[0]	
__pauli_z_gate__		z q[0]	
__phase_gate__		s q[0]	
__phasedag_gate__		sdag q[0]	
__t_gate__		t q[0]	
__tdag_gate__		tdag q[0]	
__rx90_gate__		rx90 q[0]	
__mrx90_gate__		xm90 q[0]	
__rx180_gate__		x q[0]	
__ry90_gate__		ry90 q[0]	
__mry90_gate__		ym90 q[0]	
__ry180_gate__		y q[0]	
__rx_gate__	1 qubit, 1 angle	rx q[0],3.14	
__ry_gate__		ry q[0],3.14	
__rz_gate__		rz q[0],3.14	
__cnot_gate__	2 qubits	cnot q[0],q[1]	
__cphase_gate__		cz q[0],q[1]	
__swap_gate__		swap q[0],q[1]	
__toffoli_gate__	3 qubits	toffoli q[0],q[1],q[2]	
__prepz_gate__	1 qubit	prepz q[0]	preparation
__measure_gate__		measure q[0]	measurement
__nop_gate__	none	nop	scheduling
__dummy_gate__		sink	
__wait_gate__	0 or more qubits, duration	wait 1	
__display__	0 or more qubits	display	directive
__display_binary__		display_binary	
__classical_gate__	0 or more classical regs.	add r[0],r[1]	classical
__custom_gate__	defined by config file		
__composite_gate__			

The example column shows in the form of an example the QASM representation of the gate. For custom gates, the QASM representation is the gate name followed by the representation of the operands, as with the default gates.

There is an API for each of the above gate types using default gates.

Some notes on the semantics of these gates:

- the wait gate waits for all its (qubit) operands to be ready; then it takes a duration of the given number of cycles for each of its qubit operands to execute; in external representations it is usually possible to not specify operands, it then applies to all qubits of the program; the `barrier` gate is sometimes found in external representations but is identical to a wait with 0 duration on its operand qubits (or all when none were specified)
- the nop gate is identical to `wait 1`, i.e. a one cycle execution duration applied to all program qubits
- dummy gates are SOURCE and SINK; these gates don't have an external representation; these are internal to the scheduler
- custom and composite gates are fully specified in the configuration file; these shouldn't have this type because it doesn't serve a purpose but have a type that reflects its semantics

1.28.2 Circuits and bundles in the internal representation

A circuit of one kernel is represented by a vector of gates in the internal representation, and is a structural attribute of the kernel object. The gates in this vector are assumed to be executed from the first to the last in the vector.

During a scheduling pass, the `cycle` attribute of each gate gets defined. See its definition in *Quantum gate attributes in the internal representation*. The gates in the vector then are ordered in non-decreasing cycle order.

The schedulers also produce a `bundled` version of each circuit. That is done by the `bundler` function available as `ql::ir::bundler(circuit, cycle_time)`. It constructs and returns the bundled representation of the given circuit. The cycle attribute of each gate of the circuit must be valid, and the gates in the circuit must have been sorted by their cycle value.

In the internal bundles representation a circuit is represented by a list of bundles in which each bundle represents the gates that are to be started in a particular cycle. A bundle can contain a mixture of quantum and classical gates. Each bundle is structured as a list of sections and each section as a list of gates (actually gate pointers). The gates in each section share the same operation but have different operands, obviously. The latter prepares for code generation for a SIMD instruction set in which a single instruction with one operation can have multiple operands. Each bundle has two additional attributes:

- `start_cycle` representing the cycle in which all gates of the bundle start
- `duration_in_cycles` representing the maximum duration in cycles of the gates in the bundle

This internal bundles representation is used during QISA generation instead of the original circuit.

1.28.3 Input external representation

OpenQL supports as input external representation currently only the OpenQL program, written in C++ and/or Python. This is an API-level interface based on platform, program, kernel and gate objects and their methods. Calls to these methods transfer the external representation into the internal representation (also called intermediate representation or IR) as sketched above: a program (object) consisting of a vector of kernels, each containing a single circuit, each circuit being a vector of gates.

Quantum gates are created using an API of the general form:

```
k.gate(name, qubit operand vector, classical operand vector, duration, angle)
```

in which particular operands can be empty or 0 depending on the particular kind of gate that is created. Gate creation upon a call to this API goes through the following steps to create the internal representation:

1. the qubit and/or classical register operand indices are checked for validity, i.e. to be in the range of 0 to the number specified in the program creation API minus 1
2. if the configuration file contains a definition for a specialized composite gate matching it, it is taken; the qubit parameter substitution in the gates of the decomposition specification is done; each resulting gate must be available as (specialized or parameterized, and non-composite) custom gate, or as a default gate; the decomposition is applied and all resulting gates are created and added to the circuit
3. otherwise, if a parameterized composite gate is available, take it; the parameter substitution in the gates of the decomposition specification is done; each resulting gate must be available as (specialized or parameterized, and non-composite) custom gate, or as a default gate; the decomposition is applied and all resulting gates are created and added to the circuit
4. otherwise, if a specialized custom gate is available, create it with the attributes specified as parameter of the API call above
5. otherwise, if a parameterized custom gate is available, create it with the attributes specified as parameter of the API call above
6. otherwise, if a default gate (predefined internally in OpenQL) is available, create it with the attributes specified as parameter of the API call above
7. otherwise, it is an error

1.28.4 Output external representation

There are two closely related output external representations supported, both dialects of QASM 1.0:

- sequential QASM
- bundled QASM

In both representations, the QASM representation of a single gate is as defined in the *example in QASM* column in the table above.

When the gate's cycle attribute is still undefined, the sequential QASM representation is the only possible external QASM representation. Gates are specified one by one, each on a separate line. A gate meant to execute after another gate should appear on a later line than the latter gate, i.e. the gates are topologically sorted with respect to their intended execution order. Kernels start with a label which names the kernel and serves as branch target in control transfers.

Once the gate's cycle attribute has been defined (and until it is invalidated), and in addition to the sequential QASM representation above (that ignores the cycle attribute values), the bundled QASM representation can be generated that instead reflects the cycle attribute values.

Each line in the bundled QASM representation represents the gates that start execution in one particular cycle in a curly bracketed list with vertical bar separators. Each subsequent line represents a subsequent cycle. When there isn't a gate that starts execution in a particular cycle, a wait gate is specified instead with as integral argument the number of cycles to wait. As with the sequential QASM representation, kernels start with a label which names the kernel and serves as branch target in control transfers.

1.29 Classical Instructions

Warning: This page has not been revised yet since modularization and refactoring, and may thus be out of date.

OpenQL supports a mix of quantum and classical computing at the gate level. Please recall that classical gates are gates that don't have any qubit as operand, only zero or more classical registers and execute in classical hardware.

Let us first look at some example code (taken from *tests/test_hybrid.py*):

```
num_qubits = 5
num_cregs = 10

p = ql.Program('test_classical', platform, num_qubits, num_cregs)

k1 = ql.Kernel('aKernel1', platform, num_qubits, num_cregs)

# create classical registers
rd = ql.CReg()
rs1 = ql.CReg()
rs2 = ql.CReg()

# add/sub/and/or/xor
k1.classical(rd, ql.Operation(rs1, '+', rs2))

# not
k1.classical(rd, ql.Operation('~', rs2))

# comparison
k1.classical(rd, ql.Operation(rs1, '==', rs2))

# initialize (rd = 2)
k1.classical(rd, ql.Operation(2))

# assign (rd = rs1)
k1.classical(rd, ql.Operation(rs1))

# measure
k1.gate('measure', [0], rs1)

# add kernel
p.add_kernel(k1)
p.compile()
```

In this, we see a few new methods:

- ql.CReg(): Get a free classical register (*creg*) using the classical register constructor. The corresponding destructor would free it again.
- k.classical(creg, operation): Create a classical gate, assigning the value of the *operation* to the specified destination classical register. The destination classical register and any classical registers that are operands to the operation must have indices that are less than the number of classical registers specified with the creation of kernel *k*. The gate is added to kernel *k*'s circuit.
- ql.Operation(value): Create an operation loading the immediate value *value*.
- ql.Operation(creg): Create an operation loading the value of classical register *creg*.

- ql.Operation(operator, creg): Create an operation applying the unary operator *operator* on the value of classical register *creg*.
- ql.Operation(creg1, operator, creg2): Create an operation applying the binary operator *operator* on the values of classical registers *creg1* and *creg2*.

The operators in the calls above are a string with the name of one of the familiar C operators: the binary operators +, -, &, |, ^, ==, !=, <, >, <=, and >=; or the unary ~.

Please note the creation of the quantum measurement gate that takes a classical register as operand to store the result.

1.29.1 Classical gate attributes in the internal representation

A classical gate has all general gate attributes, of which some are not used, and one additional one:

Attribute	kind	example	used by	updated by	C++ type
name	structural	"add"	all passes	never	string
creg_operands		[r0,r1]		scheduler	vector<size_t>
int_operand		3			int
type		__classical_gate__			gate_type_t
duration	semantic	20	schedulers, etc.		size_t
cycle	result	4	code generation		size_t
operands			never		
angle					
mat					

Some further notes on the gate attributes:

- `name`: The internal name. Happens to correspond to the gate name in the output QASM representation.
- `creg_operands`: Please note that for all gates the classical operands are in the *creg_operands* attribute, and the quantum operands are in the *operands* attribute.
- `int_operand`: An immediate integer valued operand is kept here.
- `type`: Is always *__classical_gate__*. Classical gates are distinguished by their name.

  Note That classical gates are distinguished by their name and not by some type, is not as problematic as for quantum gates. The names of classical gates are internal to OpenQL and have no relation to an external representation.

- `duration`: Has a built-in value of 20.

  Note That the value of duration is built-in, is strange. A first better value would be *cycle_time*.

- `operands`, `angle`, and `mat` are not used as attributes by classical gates.

The following classical gates are supported:

name	operands	operation type	inv operation	OpenQL example
“add”	1 dest and 2 src reg indices	ARITHMETIC		k.classical(rd, Operation(rs1, ‘+’, rs2))
“sub”				k.classical(rd, Operation(rs1, ‘-‘, rs2))
“eq”		RELATIONAL	“ne”	k.classical(rd, Operation(rs1, ‘==’, rs2))
“ne”			“eq”	k.classical(rd, Operation(rs1, ‘!=’, rs2))
“lt”			“ge”	k.classical(rd, Operation(rs1, ‘<’, rs2))
“gt”			“le”	k.classical(rd, Operation(rs1, ‘>’, rs2))
“le”			“gt”	k.classical(rd, Operation(rs1, ‘<=’, rs2))
“ge”			“lt”	k.classical(rd, Operation(rs1, ‘>=’, rs2))
“and”		BITWISE		k.classical(rd, Operation(rs1, ‘&’, rs2))
“or”				k.classical(rd, Operation(rs1, ‘|’, rs2))
“xor”				k.classical(rd, Operation(rs1, ‘^’, rs2))
“not”	1 dest and 1 src reg index			k.classical(rd, Operation(‘~’, rs))
“mov”		ARITHMETIC		k.classical(rd, Operation(rs))
“ldi”	1 dest reg index, 1 int_operand			k.classical(rd, Operation(3))
“nop”	none	undefined		k.classical(‘nop’)

In the above:

`Operation()` creates an expression (binary, unary, register, or immediate); apart from in the OpenQL interface as shown above, it is also used as expression in the internal representation of the *br_condition* attribute of a kernel

`operation type` indicates the type of operation which is mainly used for checking

`inv operation` represents the inverse of the operation; it is used in code generation of conditional branching; see *Kernel*

1.29.2 Classical gates in circuits and bundles in the internal representation

In circuits and bundles, no difference is made between classical and quantum gates. Classical gates are scheduled based on their operands and duration. The `cycle` attribute reflects the cycle in which the gate is executed, as usual.

Scheduling of classical instructions is assigning cycle values to these so that the register dependences of these are guaranteed to be met (ordinary scheduler); when resource constraints would be involved, those should be adhered to as well (rcscheduler). The `cycle_time` would have to be the greatest common divider of the `duration` of all gates, classical and quantum.

Classical instructions may depend on quantum gates when they retrieve the result of measurement. Quantum gates may have a control dependence on classical code because of a conditional branch; with immediate feedback, in which a single gate is performed conditionally on the value of a classical register, there also is a dependence of a quantum gate on a classically computed value.

From these dependences, an exact cycle value of the start of execution of each gate can be computed, relative to the start of execution of a kernel/circuit. Any constraints (maximum number of classical instructions to start in one cycle, maximum number of quantum gates to start in one cycle, overlapping resource uses) have to encoded in resources which are then adhered to by the rcscheduler.

1.29.3 Input external representation

OpenQL supports as input external representation currently only the OpenQL program, written in C++ and/or Python. See *Input external representation*.

Classical gates are created using an API of the form as shown above in *Classical Instructions*. The table above shows the correspondence between the input external and internal representation.

> **Note** There is no role for the configuration file in creating classical gates. This is a lost opportunity because it would have harmonized classical and quantum gates more. When defining QASM as input external representation, this might be revised.

1.29.4 Output external representation

There are two closely related output external representations supported, both dialects of QASM 1.0; see *Output external representation*: sequential and bundled QASM. Again, these don't make a difference between classical and quantum gates.

The following table shows the QASM representation of a single classical gate:

name	example operands	QASM representation
"add"	0 as dest reg index, 1 and 2 as source reg indices	add r0, r1, r2
"sub"		sub r0, r1, r2
"and"		and r0, r1, r2
"or"		or r0, r1, r2
"xor"		xor r0, r1, r2
"eq"		eq r0, r1, r2
"ne"		ne r0, r1, r2
"lt"		lt r0, r1, r2
"gt"		gt r0, r1, r2
"le"		le r0, r1, r2
"ge"		ge r0, r1, r2
"not"	0 as dest reg index, 1 as source reg index	not r0, r1
"mov"		mov r0, r1
"ldi"	0 as dest reg index, 3 as int_operand	ldi r0, 3
"nop"	none	nop

1.30 Platforms and architectures

OpenQL supports various target platforms. These platforms can be software simulators or architectures targetting hardware quantum computers.

In principle, platforms are described entirely via a JSON configuration file; as few architecture-dependent things as possible are actually hardcoded in OpenQL. This means, however, that the platform configuration file is quite extensive, making the learning curve for making one from scratch or even adjusting one pretty steep. Furthermore, actually compiling for a particular platform may also require a custom compilation strategy (i.e. which steps/passes the compiler takes/does to actually compile the program), further complicating things. Therefore, OpenQL ships with a bunch of logic to generate sane default settings for particular architectures that we support out of the box. These architectures, and the defaults they provide, are listed in the *Supported architectures* section.

A platform can be created in OpenQL by using one of the various constructors for the `ql.Platform()` class. The most commonly used way to do obtain a platform object is as follows:

```
platform = ql.Platform('<platform_name>', `<platform_config>`)
```

Here, `<platform_name>` is anything you want it to be (it's only used for logging) and `<platform_config>` can be a recognized architecture name, such as *"none"* or *"cc"*, or it can point to a platform configuration file. For example, a platform with the name `CCL_platform` that uses the defaults for CC-light can be created as:

```
platform = ql.Platform('CCL_platform', 'cc_light')
```

For more details, see also the *Python API*, *Configuration*, and *Supported architectures* sections.

1.31 Compiler

Warning: This page has not been revised yet since modularization and refactoring, and may thus be out of date.

To compile a program, the user needs to configure a compiler first. Until version 0.8, this program compilation was done using a monolithic hard-coded sequence of compiler passes inside the program itself when `program.compile()` function was called. This is the legacy operation mode, which is currently described in the *Program* documentation page. However, starting with version 0.8.0.dev1, the programer has the ability to configure its own pass sequence using the Compiler API.

There are two options on how to configure a compiler. The first and the most straightforward is to define a compiler object giving it as the second parameter the name of a json configuration, similar of how the *Platforms and architectures* is defined. The following code line shows an example of a such initialization:

```
... different other program initializations ...

c = ql.Compiler("testCompiler", "cc_compiler_cfg.json")

..... # definition of Platform and Program p .....

c.compile(p)
```

In the above code, the `cc_compiler_cfg.json` compiler configuration file is used. This can be found in the `.\test' folder within the OpenQL installation directory and has the following structure:

```
{
  "CompilerPasses":
  [
    {
        "passName" : "Writer",
        "passAlias": "initialqasmwriter"
        "options":
        [
            {
                "optionName" : "eqasm_compiler_name",
                "optionValue": "eqasm_backend_cc"
            },
            ....
        ]
    },
```

(continues on next page)

(continued from previous page)

```
    ...
  ]
}
```

Furthermore, an additional option to configure a compiler is to use the `compiler::add_pass()` method to manually load compiler passes inside the program itself. To illustrate this interface, consider the following example:

```
from openql import openql as ql

c = ql.Compiler("testCompiler")

c.add_pass_alias("Writer", "outputIR")
c.add_pass("Reader")
c.add_pass("RotationOptimizer")
c.add_pass("DecomposeToffoli")
c.add_pass_alias("CliffordOptimize", "clifford_prescheduler")
c.add_pass("Scheduler")
c.add_pass_alias("CliffordOptimize", "clifford_postscheduler")
c.add_pass_alias("Writer","scheduledqasmwriter")

c.set_pass_option("ALL", "skip", "no");
c.set_pass_option("Reader", "write_qasm_files", "no")
c.set_pass_option("RotationOptimizer", "write_qasm_files", "no")
c.set_pass_option("outputIR", "write_qasm_files", "yes");
c.set_pass_option("scheduledqasmwriter", "write_qasm_files", "yes");
c.set_pass_option("ALL", "write_report_files", "no");

..... # definition of Platform and Program p .....

c.compile(p)
```

Note The code for the platform and the program creation as described earlier (for more information on that, please see *Creating your first program*) has been removed for clarity purposes.

The example code shows that we can add a pass under its real name, which should be the exact pass name as defined in the compiler (for a complete list available pass names, please consult *Compiler Passes*), or under an alias name to be defined by the OpenQL user. This last name can be any string and should be used to set pass specific options. This options setting is shown last, where current pass option choices represent either the "ALL" target or a given pass name (either its alias or its real name). Curently, only the <write_qasm_files>, <write_report_files>, and <skip> options are implemented for individual passes. The other options should be accessed through the global option settings of the program.

Finally, to create and use a new compiler pass, the developer would need to implement three steps:

1) Inherit from the AbstractPass class and implement the following function

```
virtual void runOnProgram(ql::quantum_program *program)
```

2) Register the pass by giving it a pass name in

```
AbstractPass* PassManager::createPass(std::string passName, std::string aliasName)
```

3) Add it in a custom compiler configuration using the Compiler API

Currently, the following passes are available in the compiler class and can be enabled by using the following pass identifiers to map to the existing passes.

Pass Identifier	Compiler Pass
Reader	Program Reading (currently cQASMReader)
Writer	Qasm Printer
RotationOptimizer	Optimizer
DecomposeToffoli	Decompose Toffoli
Scheduler	Scheduling
BackendCompiler	Composite pass calling either CC or CC-Light passes
ReportStatistics	Report Statistics
CCLPrepCodeGeneration	CC-Light dependent code generation preparation
CCLDecomposePreSchedule	Decomposition before scheduling (CC-Light dependent)
WriteQuantumSim	Print QuantumSim program
CliffordOptimize	Clifford Optimization
Map	Mapping
RCSchedule	Resource Constraint Scheduling
LatencyCompensation	Latency Compensation
InsertBufferDelays	Insert Buffer Delays
CCLDecomposePostSchedule	Decomposition before scheduling (CC-Light dependent)
QisaCodeGeneration	QISA generation (CC-Light dependent)

1.32 Compiler Passes

Warning: This page has not been revised yet since modularization and refactoring, and may thus be out of date.

Most of the passes in their function and implementation are platform independent, deriving their platform dependent information from options and/or the configuration file. This holds also for mapping, although one wouldn't think so first, since it is called from the platform dependent part of the compiler now. All passes like this are summarized below first and are described extensively platform independently later in this section.

Note Some passes are called from the platform independent compiler, other ones from the back-end compiler. That is a platform dependent issue and therefore described with the platform.

Their description includes:

- their API, including their name: in general, apart from their name, they take all parameters from the `program` context which includes the configuration file and the platform, the options, and the vector of kernels with their circuits
- the Intermediate Representation (IR) they expect as input and what they update in the IR; in general, they should accept any IR, so all types of gates, quantum as well as classical
- the particular options they listen to; there usually is an option to disable it; also there are ways to dump the IR before and/or after it although this is not generally possible yet
- the function they perform, in terms of the IR and the options

Other passes, of which the implementation (i.e. source code etc.) is platform dependent, can be found with the platforms. An example of the latter passes is QISA (i.e. instruction) generation in CC-Light. In the lists below, these passes are indicated to be platform dependent.

Passes have some general facilities available to them; these are not passes themselves since they don't transform the IR. Examples of such facilities are:

- **`ql::report::report_qasm(prog_name, kernels, platform, relplacename, passname):`** Writing the IR out in an external representation (QASM 1.0) to a file when option `write_qasm_files` has the value `yes`. The file is stored in the default output directory; the name of the file is composed from the program name (`prog_name`), the place relative to the pass (`relplacename`), and the pass name (`passname`), all separated by _, and the result suffixed by `.qasm`. The pass indicates before or after which the IR is written to file. The place relative to the pass indicates e.g. `in` or `out`, meaning before or after the pass, respectively. In this way, multiple qasm files can be written per compile, and be easily related to the point in compilation where the writing was done.
- **`ql::report::report_bundles(prog_name, kernels, platform, relplacename, passname):`** Identical to `report_qasm` but the QASM is written as bundles.
- **`ql::report::report_statistics(prog_name, kernels, platform, relplacename, passname, pr`** Identical to `report_qasm` but the IR itself is not written but a summary of it, e.g. the number of kernels, the numer of one-qubit, two-qubit and more-qubit gates, which qubits were used and which not, the wall clock time that compilation took until this point, etc. This is done for each kernel separately and for the whole program; additional interfaces are available for making the individual reports and adding pass specific lines to the reports. The `prefix` string is prepended to each line in the report file, e.g. to make it qasm comment. Furthermore the suffix is `.report`. And writing the report is only done when option `write_report_files` has the value `yes`.
- **`ql::utils::write_file(filename, contentstring):`** Writing a content string to the file with given `filename` in the default output directory for off-line inspection. An example is writing (in dot format) the gate dependence graph which is a scheduling pass internal data structure. The writing to a file of a string is a general facility but the generation of the string representation of the internal data structure is pass dependent. The options controlling this are also pass specific.

Writing the IR out to a file in a form suitable for a particular subsequent tool such as quantumsim is considered code generation for the quantumsim platform and is therefore considered a pass.

Note A compiler pass is not something defined in OpenQL. It should be. Passes then have a standard API, standard intermediate representation dumpers before and after them, a standard way to include them in the compiler. We could have the list of passes to call be something defined in the configuration file, perhaps with the places where we want to have dumps and reports.

1.32.1 Summary of compiler passes

Compiler passes in OpenQL are the compiler elements that, when called one after the other, gradually transform the OpenQL input program to some platform defined output program. The following passes are available and usually called in this order. More detailed information on each can be found in the sections below.

When it is indicated that a pass is CC-Light (or any other platform) dependent, it means that its implementation with respect to source code is platform dependent. A pass of which the source code is platform independent, can behave platform dependently by its parameterization by the platform configuration file.

- **program reading** not a real pass now; it covers the code that for a particular program sets its options, connects it to a platform, defines its program parameters such as number of qubits, defines its kernels, and defines its gates; in the current OpenQL implementation this is all code upto and including the call to `p.compile()`. See *Input external representation* and *Creating your first program*.
- **optimize** attempts to find contigous sequences of quantum gates that are equivalent to identity (within some small epsilon which currently is 10 to the power -4) and then take those sequences out of the circuit; this relies on the function of each gate to be defined in its `mat` field as a matrix. See optimization.
- **decompose_toffoli** each toffoli gate in the IR is replaced by a gate sequence with at most two-qubit gates; depending on the value of the equally named option; it does this in the Neilsen and Chuang way (`NC`), or in the way as in https://arxiv.org/pdf/1210.0974,pdf (`AM`). See decomposition.

- **unitary decomposition** the unitary decomposition pass is not generally available yet; it is in some private OpenQL branch. See decomposition.
- **scheduling** of each kernel's circuit the gates are scheduled at a particular cycle starting from 0 (by filling in the gate's `cycle` attribute) that matches the gates' dependences, their duration, the constraints imposed by their resource use, the buffer values defined for the platform, and the latency value defined for each gate; multiple gates may start in the same cycle; in the resulting circuits (which are vectors of pointers to gate) the gates are ordered by their cycle value. The schedulers also produce a `bundled` version of each circuit: the circuit is then represented by a vector of bundles in which each bundle lists the gates that are to be started in the same cycle; each bundle further contains sublists that combines gates with the same operation but with different operands. The resource-constrained and non-constrained versions of the scheduler have different entry points (currently). The latter only considers the gates' dependences and their duration, which is sufficient as input to QX. Next to the above necessary constraints, the remaining freedom is defined by a scheduling strategy which is defined by the `scheduler` option value: `ASAP`, `ALAP` and some other options. See scheduling.
- **decomposition before scheduling (CC-Light dependent)** classical non-primitive gates are decomposed to primitives (e.g. `eq` is transformed to `cmp` followed by an empty cycle and an `fbr_eq`); after measurements an `fmr` is inserted provided the measurement had a classical register operand. See decomposition.
- **clifford optimization** dependency chains of one-qubit clifford gates operating on the same qubit are replaced by equivalent sequences of primitive gates when the latter leads to a shorter execution time. Clifford gates are recognized by their name and use is made of the property that clifford gates form a group of 24 elements. Clifford optimization is called before and after the mapping pass. See optimization.
- **mapping** the circuits of all kernels are transformed such that for any two-qubit gate the operand qubits are connected (are NN, Nearest Neighbor) in the platform's topology; this is done by a kernel-level initial placement pass and when it fails, by a subsequent heuristic; the heuristic essentially transforms each circuit from start to end; doing this, it maintains a map from virtual (program) qubits to real qubits (`v2r`); each time that it encounters a two-qubit gate that in the current map is not NN, it inserts swap gates before this gate that gradually make the operand qubits NN; when inserting a swap, it updates the v2r map accordingly. There are many refinements to this algorithm that can be controlled through options and the configuration file. It is not complete in the sense that it ignores transfer of the v2r map between kernels. See mapping.
- **rcscheduler** resource constraints are taken into account; the result reflects the timing required during execution, i.e. also taking into account any further non-OpenQL passes and run-time stages such as (for CC_Light):
  - QISA assembly
  - classical code execution (from here on these passes are executed as run-time stages)
  - quantum microcode generation
  - micro operation to signal and microwave conversion
  - execution unit reprogramming and inter operation reset times
  - signal communication line delays
  - execution time and feed-back delays

  The resulting circuit is stored in the usual manner and as a sequence of bundles. See scheduling.
- **decomposition after scheduling (CC-Light dependent)** two-qubit flux gates are decomposed to a series of one-qubit flux gates of the form `sqf q0` to be executed in the same cycle; this is done only when the `cz_mode` option has the value `auto`; such a gate is generated for each operand and for all qubits that need to be detuned; see the detuned_qubits resource description in the CC-Light platform configuration file for details. See decomposition.
- **opcode and control store file generation (CC-Light dependent)** currently disabled as not used by CC-Light
- **write_quantumsim_program** writes the current IR as a python script that interfaces with quantumsim

- **write_qsoverlay_program** writes the current IR as a python script that interfaces with the qsoverlay module of quantumsim
- QISA generation (CC-Light dependent)
    - bundle to QISA translation
        - deterministic sorting of gates per bundle
        - instruction prefix and wait instruction insertion
        - classical gate to QISA classical instruction translation
        - SOMQ generation and mask to mask register assignment (should include mask instruction generation)
        - insertion of wait states between meas and fmr (should be done by scheduler)
    - mask instruction generation
    - QISA file writing

  See *Platforms and architectures*.

1.32.2 Decomposition

Decomposition of gates [TBD]

Control decomposition

Entry points

The following entry points are supported:

- `entry()` TBD

Input and output intermediate representation

TBD.

Options

The following options are supported:

- `option` TBD

Function

TBD

Unitary decomposition

Unitary decomposition allows a developer of quantum algorithms to specify a quantum gate as a unitary matrix, which is then split into a circuit consisting of `ry`, `rz` and `cnot` gates.

To use it, define a `Unitary` with a name and a (complex) list containing all the values in the unitary matrix in order from the top left to the bottom right. The matrix needs to be unitary to be a valid quantum gate, otherwise an error will be raised by the compilation step.

Name	operands	C++ type	example
Unitary	name	string	"U_name"
	unitary matrix	vector<complex<double>>	[0.5+0.5j,0.5-0.5j,0.5-0.5j,0.5+0.5j]

The unitary is first decomposed, by calling the `.decompose()` function on it. Only then can it be added to the kernel as a normal gate to the number of qubits corresponding to the unitary matrix size. This looks like:

```
u1 = ql.Unitary("U_name", [0.5+0.5j,0.5-0.5j,0.5-0.5j,0.5+0.5j])
u1.decompose()
k.gate(u1, [0])
```

Which generates this circuit:

```
rz q[0], -1.570796
ry q[0], -1.570796
rz q[0], 1.570796
```

The circuit generated might also have different angles, though not different gates, and result in the same effect on the qubits, this is because a matrix can have multiple valid decompositions.

For a two-qubit unitary gate or matrix, it looks like:

```
list_matrix = [1, 0       , 0       , 0,
               0, 0.5+0.5j, 0.5-0.5j, 0,
               0, 0.5-0.5j, 0.5+0.5j, 0,
               0, 0       , 0       , 1]
u1 = ql.Unitary("U_name", list_matrix)
u1.decompose()
k.gate(u1, [0,1])
```

This generates a circuit of 24 gates of which 6 `cnots`, spanning qubits 0 and 1. The rest are `ry` and `rz` gates on both qubits, which looks like this:

```
rz q[0], -0.785398
ry q[0], -1.570796
rz q[0], -3.926991
rz q[1], -0.785398
cnot q[0],q[1]
rz q[1], 1.570796
cnot q[0],q[1]
rz q[0], 2.356194
ry q[0], -1.570796
```

(continues on next page)

(continued from previous page)

```
rz q[0], -3.926991
ry q[1], 0.785398
cnot q[0],q[1]
ry q[1], 0.785398
cnot q[0],q[1]
rz q[0], -0.000000
ry q[0], -1.570796
rz q[0], 3.926991
rz q[1], 0.785398
cnot q[0],q[1]
rz q[1], -1.570796
cnot q[0],q[1]
rz q[0], 3.926991
ry q[0], -1.570796
rz q[0], -2.356194
```

The unitary gate has no limit in how many qubits it can apply to. But the matrix size for an n-qubit gate scales as 2^n*2^n, which means the number of elements in the matrix scales with 4^n. This is also the scaling rate of the execution time of the decomposition algorithm and of the number of gates generated in the circuit. Caution is advised for decomposing large matrices both for compilation time and for the size of the resulting quantum circuit.

More detailed information can be found at http://resolver.tudelft.nl/uuid:9c60d13d-4f42-4d8b-bc23-5de92d7b9600

Decomposition before scheduling

Entry points

The following entry points are supported:

- `entry()` TBD

Input and output intermediate representation

TBD.

Options

The following options are supported:

- `option` TBD

Function

TBD

Decomposition after scheduling

Entry points

The following entry points are supported:

- `entry()` TBD

Input and output intermediate representation

TBD.

Options

The following options are supported:

- `option` TBD

Function

TBD

Decompose_toffoli

Entry points

The following entry points are supported:

- `entry()` TBD

Input and output intermediate representation

TBD.

Options

The following options are supported:

- `option` TBD

Function

TBD

1.32.3 Optimization

Optimization of circuits [TBD]

Optimize

attempts to find contigous sequences of quantum gates that are equivalent to identity (within some small epsilon which currently is 10 to the power -4) and then take those sequences out of the circuit; this relies on the function of each gate to be defined in its `mat` field as a matrix.

Entry points

The following entry points are supported:

- `entry()` TBD

Input and output intermediate representation

TBD.

Options

The following options are supported:

- `option` TBD

Function

TBD

Clifford optimization

dependency chains of one-qubit clifford gates operating on the same qubit are replaced by equivalent sequences of primitive gates when the latter leads to a shorter execution time. Clifford gates are recognized by their name and use is made of the property that clifford gates form a group of 24 elements. Clifford optimization is called before and after the mapping pass.

Entry points

The following entry points are supported:

- `entry()` TBD

Input and output intermediate representation

TBD.

Options

The following options are supported:

- `option` TBD

Function

TBD

1.32.4 Scheduling

Of each kernel's circuit the gates are scheduled at a particular cycle starting from 0 (by filling in the gate's `cycle` attribute) that matches the gates' dependences, their duration, the constraints imposed by their resource use, the buffer values defined for the platform, and the latency value defined for each gate; multiple gates may start in the same cycle; in the resulting circuits (which are vectors of pointers to gate) the gates are ordered by their cycle value. The schedulers also produce a `bundled` version of each circuit; see *Circuits and bundles in the internal representation*.

The resource-constrained and non-constrained versions of the scheduler have different entry points (currently). The latter only considers the gates' dependences and their duration, which is sufficient as input to QX. Next to the above necessary constraints, the remaining freedom is defined by a scheduling strategy which is defined by the `scheduler` option value: `ASAP`, `ALAP` and some other options.

Entry points

The following two entry points are supported, one for the non-constrained and one for the resource-constrained scheduler:

- `p.schedule()` In the context of `program` object `p`, this method schedules the circuits of the kernels of the program, according to a strategy specified by the scheduling options, but without taking resource constraints, buffers and latency compensation of the platform into account.
- `bundles = cc_light_schedule_rc(circuit, platform, num_qubits, num_creg)` In the context of the `cc_light_eqasm_compiler`, a derived class of the `eqasm_compiler` class, in its `compile(prog_name, kernels, platform)` method, inside a loop over the specified kernels, the resource-constrained scheduler is called to schedule the specified circuit, according to a strategy specified by the scheduling options, and taking resource constraints, buffers and latency compensation of the platform into account. It creates a bundled version of the IR and returns it.

  Note These entry points need to be harmonized to fit in the generalized pass model: same class, program-level interface, no result except in IR, buffer and latency compensation split off to separate passes.

The above entry points each create a `sched` object of class `Scheduler` and call a selection of its methods:

- `sched.init(circuit, platform, num_qubits, num_creg)` A dependence graph representation of the specified circuit is constructed. This graph is a Directed Acyclic Graph (DAG). In this graph, the nodes represent the gates and the directed edges the dependences. The top of the graph is a newly created SOURCE gate, the bottom is a newly created SINK gate. With respect to dependences, the SOURCE and SINK gates behave as if they update all qubits and classical registers with 0 duration. Gates are added in the order of presence in the circuit and linked in dependence chains according to their operation and operands.

  The nodes have as attributes (apart from the gate's attributes):

  - `name` with the qasm string representation of the gate (such as `cnot q[1],q[2]`)

  The edges have as attributes:

  - `weight` representing the number of cycles needed from the start of execution of the gate at the source of the edge, to the start of execution of the gate at the target of the edge; this value is initialized from the `duration` attribute of the gate
  - `cause` representing the qubit or classical register causing the dependence
  - `depType` representing the type of the dependence

  The latter two attributes are currently only used internally in the dependence graph construction.

  This `sched.init` method is called by both entry points for each circuit of the program.

- `bundles = sched.schedule_asap(sched_dot)` The cycle attributes of the gates are initialized consistent with an ASAP (i.e. downward) walk over the dependence graph. Subsequently, the gates in the circuit are sorted by their cycle value; and the `bundler` called to produce a bundled version of the IR to return.

  This method is called by `p.schedule()` for each circuit of the program when non-uniform ASAP scheduling.

- `bundles = sched.schedule_alap(sched_dot)` The cycle attributes of the gates are initialized consistent with an ALAP (i.e. upward) walk over the dependence graph. Subsequently, the gates in the circuit are sorted by their cycle value; and the `bundler` called to produce a bundled version of the IR to return.

  This method is called by `p.schedule()` for each circuit of the program when non-uniform ALAP scheduling.

- `bundles = sched.schedule_alap_uniform()` The cycle attributes of the gates are initialized consistent with a uniform ALAP schedule: this modified ALAP schedule aims to have an equal number of gates starting in each non-empty bundle. Subsequently, the gates in the circuit are sorted by their cycle value; and the `bundler` called to produce a bundled version of the IR to return.

  This method is called by `p.schedule()` for each circuit of the program when uniform and ALAP scheduling.

- `bundles = sched.schedule_asap(resource_manager, platform, sched_dot)`

  This method is called by `cc_light_schedule_rc` after calling `sched.init`, and creation of the resource manager for each circuit of the program when non-uniform ASAP scheduling. See scheduling_function for a more extensive description.

- `bundles = sched.schedule_alap(resource_manager, platform, sched_dot)`

  This method is called by `cc_light_schedule_rc` after calling `sched.init`, and creation of the resource manager for each circuit of the program when non-uniform ALAP scheduling. See scheduling_function for a more extensive description.

In the `sched_dot` parameter of the methods above a `dot` representation of the dependence graph of the kernel's circuit is constructed, in which the gates are ordered along a timeline according to their cycle attribute.

Input and output intermediate representation

The schedulers expect kernels with or without a circuit. When with a circuit, the `cycle` attribute need not be valid. Gates that are supported on input are one-qubit `measure`, no-operand `display`, any classical gate, `cnot`, `cz`/`cphase`, and any other quantum and scheduling gate.

They produce a circuit with the same gates (but potentially differently ordered). The `cycle` attribute of each gate has been defined. The gates in the circuit are ordered with non-decreasing cycle value. The cycle values are consistent with all constraints imposed during scheduling and with the scheduling strategy that has been specified through the options or by selection of the entry point.

> **Note** There are no gates for control flow; so these are not defined in the configuration file; these are not scheduled in the usual way; these are not translated to QASM and external representations in the usual way. See *Kernel*.

Options

The following options are supported:

- `scheduler` With the value `ASAP`, the scheduler creates a forward As Soon As Possible schedule of the circuit. With the value `ALAP`, the scheduler creates a backward As Soon As Possible schedule which is equivalent to a forward As Late As Possible schedule of the circuit. Default value is `ALAP`.
- `scheduler_uniform` With the value `yes`, the scheduler creates a uniform schedule of the circuit. With the value `no`, it doesn't. Default value is `no`.
- `scheduler_commute` With the value `yes`, the scheduler exploits commutation rules for `cnot`, and `cz`/`cphase` to have more scheduling freedom to aim for a shorter latency circuit. With the value `no`, it doesn't. Default value is `no`.
- `output_dir` The value is the name of the directory which should be present in the current directory during execution of OpenQL, where all output and report files of OpenQL are created. Default value is `test_output`.
- `write_qasm_files` When it has the value `yes`, `p.schedule` produces in the output directory a bundled QASM (see *Output external representation*) of all kernels in a single file with as name the name of the program followed by `_scheduled.qasm`.
- `print_dot_graphs` When it has the value `yes`, `p.schedule` produces in the output directory in multiple files each with as name the name of the kernel followed by `_dependence_graph.dot` a `dot` representation of the dependence graph of the kernel's circuit. Furthermore it produces in the output directory in multiple files each with as name the name of the kernel followed by the value of the `scheduler` option and `_scheduled.dot` a `dot` representation of the dependence graph of the kernel's circuit, in which the gates are ordered along a timeline according to their cycle attribute.

  > **Note** The options don't discriminate between the prescheduler and the rcscheduler although these could desire different option values. Also there is not an option to skip this pass.

Function

Scheduling of a circuit starts with creation of the dependence graph; see scheduling_entry_points for its definition.

Gates that are supported on input are one-qubit `measure`, no-operand `display`, any classical gate, `cnot`, `cz`/`cphase`, and any other quantum and scheduling gate. With respect to dependence creation, the latter ones are assumed to use and update each of their operands during the operation; and the former ones each have a specific definition regarding the use and update of their operands:

- `measure` also updates its corresponding classical register;

- `display` and the classical gates use/update all qubits and classical registers (so these act as barriers);
- `cnot` uses and doesn't update its control operand, and it commutes with `cnot`/`cz`/`cphase` with equal control operand; `cnot` uses and updates its target operand, it commutes with `cnot` with equal target operand;
- `cz`/`cphase` commutes with `cnot`/`cz`/`cphase` with equal first operand, and it commutes with `cz`/`cphase` with equal second operand. This commutation is exploited to aim for a shorter latency circuit when the `scheduler_commute` option is in effect.

When scheduling without resource constraints the cycle attributes of the gates are initialized consistent with an ASAP (i.e. downward/forward) or ALAP (i.e. upward/backward) walk over the dependence graph. Subsequently, the gates in the circuit are sorted by their cycle value; and the `bundler` called to produce a bundled version of the IR to return.

The remaining part of this subsection describes scheduling with resource constraints.

The implementation of this list scheduler is parameterized on doing a forward or a backward schedule. The former is used to create an ASAP schedule and the latter is used to create an ALAP schedule. We here describe the forward case because that is easier to grasp and later come back on the backward case.

A list scheduler maintains at each moment a list of gates that are available for being scheduled because they are not blocked by dependences on non-scheduled gates. Not all gates that are available (not blocked by dependences on non-scheduled gates) can actually be scheduled. It must be made sure in addition that those scheduled gates that it depends on, actually have completed their execution (using its `duration`) and that the resources are available for it. Furthermore, making a selection from the gates that remain after ignoring these, determines the optimality of the scheduling result. The implemented list scheduler is a critical path scheduler, i.e. it prefers to schedule the most critical gate first. The criticality of a gate estimates the effect that delaying scheduling the gate has on the latency of the resulting circuit, and is determined by computing the length of the longest dependence chain from the gate to the SINK gate; the higher this value, the higher the gate's scheduling priority in the current cycle is.

The scheduler relies on the dependence graph representation of the circuit. At the start only the SOURCE gate is available. Then one by one, according to a criterion, a gate is selected from the list of available ones and added to the schedule. Having scheduled the gate, it is taken out of the available list; after having scheduled a gate, some new gates may become available because they don't depend on non-scheduled gates anymore; those gates are found and put in the available list of gates. This continues, filling cycle by cycle from low to high, until the available list gets empty (which happens after scheduling the last gate, the SINK gate).

Above it was mentioned that a gate can only be scheduled in a particular cycle when the resources are available for it. In this, the scheduler relies on the resource manager of the platform. The latter was created and initialized from the platform configuration file before scheduling started. Please refer to cclplatform for a description of the specification of resources of the CC-Light platform. And furthermore note that only the resources that are specified in the platform configuration file determine the resource constraints that apply to the scheduler; recall that for each resource type, several resources can be specified, each of which typically has some kind of exclusive use. The simplest one is the `qubits` resource type of which there are as many resources as there are qubits. The resource manager maintains a so-called `machine state` that describes the occupation status of each resource. This resource state typically consists of two elements: the operation type that is using this resource; and the occupation period, which is described by a pair of cycle values, representing the first cycle that it is occupied, and the first cycle that it is free again, respectively.

If a gate is to be scheduled at cycle `t`, then all the resources for executing the gate are checked to be available from cycle `t` till (and not including) `t` plus the gate's `duration` in cycles; and when actually committing to scheduling the gate at cycle `t`, all its resources are set to occupied for the duration of its execution. The resource manager offers methods for this check (`bool rm.available()`) and commit (`rm.reserve()`). Doing this check and committing for a particular gate, some additional gate attributes may be required by the resource manager. For the CC-Light resource manager, these additional gate attributes are:

- `operation_name` initialized from the configuration file `cc_light_instr` gate attribute representing the operation of the gate; it is used by the `qwgs` resource type only; two gates having the same `operation_name` are assumed to use the same wave form

- `operation_type` initialized from the configuration file `type` gate attribute representing the kind of operation of the gate: `mw` for rotation gates, `readout` for measurement gates, and `flux` for one and two-qubit flux gates; it is used by each resource type

This concludes the description of the involvement of the resource manager in the scheduling of a gate.

The list scheduler algorithm uses a so-called availability list to represent gates that can be scheduled; see above. When the available list becomes empty, all cycle values were assigned and scheduling is almost done. The gates in the circuit are then first sorted on their cycle value.

Then latency compensation is done: for each gate for which in the platform configuration file a `latency` attribute value is specified, the gate's cycle value is incremented by this latency value converted to cycles; the latter is usually negative. This mechanism allows to start execution of a gate earlier to compensate for a relative delay in the control electronics that is involved in executing the gate. So in theory, in the quantum hardware, gates which before latency compensation had the same cycle value, also execute in the same cycle. After this, the gates in the circuit are again sorted on their cycle value.

After the `bundler` has been called to produce a bundled IR, any buffer delays are inserted. Buffer delays can be specified in the platform configuration file in the `hardware_settings` section. Insertion makes use of the `type` attribute of the gate in the platform configuration file, the one which can have the values `mw`, `readout` and `flux`. For each bundle, it checks for each gate in the bundle, whether there is a non-zero buffer delay specified with a gate in the previous bundle, and if any, takes the maximum of those buffer delays, and adds it (converted to cycles) to the bundle's `start_cycle` attribute. Moreover, when the previous bundle got shifted in time because of earlier bundle delays, the same shift is applied first to the current bundle. In this way, the schedule gets stretched for all qubits at the same time. This is a valid thing to do and doesn't invalidate dependences nor resource constraints.

Note Buffer insertion only has effect on the `start_cycle` attributes of the bundles and not on the `cycle` attributes of the gates. It would be better to do buffer insertion on the circuit and to do bundling afterwards, so that circuit and bundles are consistent.

In the backward case, the scheduler traverses the dependence graph bottom-up, scheduling the SINK gate first. Gates become available for scheduling at a particular cycle when at that cycle plus its duration all its dependent gates have started execution. And scheduling finishes when the available list is empty, after having scheduled the SOURCE gate. In this, cycles are decremented after having scheduled SINK at some very high cycle value, and later, after having scheduled SOURCE, the cycle values of the gates are consistently shifted down so that SOURCE starts at cycle 0. The resource manager's state and methods also are parameterized on the scheduling direction.

Scheduling for software platforms

- Scheduling for qx
- Scheduling for quantumsim

Scheduling for hardware platforms

- Scheduling for CC-Light platform
- Scheduling for CC platform
- Scheduling for CBox platform

1.32.5 Mapping

The circuits of all kernels are transformed such that after mapping for any two-qubit gate the operand qubits are connected (are NN, Nearest Neighbor) in the platform's topology; this is done by a kernel-level initial placement and when it fails, by subsequent heuristic routing and mapping. Both maintain a map from virtual (program) qubits to real qubits (`v2r`) and a map from each real qubit index to its state (`rs`); both are available after each of the two mapping subpasses.

- *initial placement* This module attempts to find a single mapping of the virtual qubits of a circuit to the real qubits (`v2r` map) of the platform's qubit topology, that minimizes the sum of the distances between the two mapped operands of all two-qubit gates in the circuit. The distance between two real qubits is the minimum number of swaps that is required to move the state of one of the two qubits to the other. It employs a Mixed Integer Linear Programming (MIP) algorithm to solve the initial placement that is modelled as a Quadratic Assignment Problem. The module can find a mapping that is optimal for the whole circuit, but because its time-complexity is exponential with respect to the size of the circuit, this may take quite some computer time. Also, the result is only really useful when in the mapping found all mapped operands of two-qubit gates are NN. So, there is no guarantee for success: it may take too long and the result may not be optimal.
- *heuristic routing and mapping* This module essentially transforms each circuit in a linear scan over the circuit, from start to end, maintaining the `v2r` and `rs` maps. Each time that it encounters a two-qubit gate that in the current map is not NN, it inserts `swap` gates before this gate that make the operand qubits NN (this is called *routing* the qubits); when inserting a `swap`, it updates the `v2r` and `rs` maps accordingly. There are many refinements to this algorithm that can be controlled through options and the configuration file. The module will find the minimum number of swaps to make the mapped operands of each two-qubit gate NN in the mapping that applies just before it. In the most basic version, it has a linear time-complexity with respect to circuit size and number of qubits. With advanced search options set, the algorithm may become cubic with respect to number of qubits. So, it is still scalable and is guaranteed to find a solution.

The implementation is not complete:

- In the presence of multiple kernels with control flow among them, the `v2r` at the start of each kernel must match the `v2r` at the end of all predecessor kernels: this is not implemented. Instead, the `v2r` at the start of each kernel is re-initialized freshly, independently of the `v2r` at the end of predecessor kernels. The current implementation thus assumes that at the end of each kernel all qubits don't hold a state that must be preserved for a subsequent kernel.

Entry points

Mapping is implemented by a class `Mapper` with the support of many private other classes among which the scheduler class for obtaining the dependence graph. The following entry points are supported:

- `Mapper()` Constructs a new mapper to be used for the whole program. Initialization is left to the `Init` method.
- `Mapper.Init(platform)` Initialize the mapper for the given platform but independently of a particular kernel and circuit. This includes checking and initializing the mapper's representation of the platform's topology from the platform's configuration file.
- `Mapper.Map(kernel)` Perform mapping on the kernel, i.e. replace the kernel's circuit by an equivalent but *mapped* circuit. Each kernel is mapped independently of any other kernel. Of each gate the `cycle` attribute is assigned, and the resulting circuit is scheduled; which constraints are obeyed in this schedule depends on the mapping strategy (the value of the `mapper` attribute). In the argument `kernel` object, the `qubit_count` attribute is updated from the number of virtual qubits of the kernel to the number of real qubits as specified by the platform; this is done because in the mapped circuit the qubit operands of all gates will be real qubit indices of which the values should be in the range of the valid real qubit indices of the platform.

Furthermore, some reporting of internal mapper statistics is done into attributes of the `Mapper` object. These can be retrieved by the caller of `Map`:

- `nswapsadded` Number of `swap`s and `move`s inserted.
- `nmovesadded` Number of `move`s inserted.
- `v2r_in` Vector with for each virtual qubit index its mapping to a real qubit index (or `UNDEFINED_QUBIT` represented by `INT_MAX`, indicating that the virtual qubit index is not mapped to a real qubit), after initialization and before initial placement and/or heuristic routing and mapping.
- `rs_in` Vector with for each real qubit index its state. This vector shows the state after initialization of the mapper and before initial placement and/or heuristic routing and mapping. State values can be:
  - `rs_nostate`: no statically known quantum state and no dynamically useful quantum state to preserve
  - `rs_wasinited`: known to be in zero base state (`|0>`)
  - `rs_hasstate`: useful but statically unknown quantum state; must be preserved
- `v2r_ip` Vector with for each virtual qubit index its mapping to a real qubit index (or `UNDEFINED_QUBIT` represented by `INT_MAX`, indicating that the virtual qubit index is not mapped to a real qubit), after initial placement but before heuristic routing and mapping.
- `rs_ip` Vector with for each real qubit index its state (see `rs_in` above for the values), after initial placement but before heuristic routing and mapping.
- `v2r_out` Vector with for each virtual qubit index its mapping to a real qubit index (or `UNDEFINED_QUBIT` represented by `INT_MAX`, indicating that the virtual qubit index is not mapped to a real qubit), after heuristic routing and mapping.
- `rs_out` Vector with for each real qubit index its state (see `rs_in` above for the values), after heuristic routing and mapping.

Input and output intermediate representation

The mapper expects kernels with or without a circuit. When with a circuit, the `cycle` attributes of the gates need not be valid. Gates that are supported on input are one-qubit `measure`, no-operand `display`, any classical gate, `cnot`, `cz`/`cphase`, and any other quantum and scheduling gate. The mapper refuses multi-qubit quantum gates as input with more than two quantum operands.

The mapper produces a circuit with the same gates but then *mapped* (see below), with the real qubit operands of two-qubit gates made nearest-neighbor in the platform's topology, and with additional quantum gates inserted to implement the swapping or moving of qubit states. The *mapping* of any (quantum, classical, etc.) gate entails replacing the virtual qubit operand indices by the real qubit operand indices corresponding to the mapping of virtual to real qubit indices applicable at the time of execution of the gate; furthermore the gate itself (when a quantum gate) is optionally replaced at the time of its mapping by one or more gates as specified by the platform's configuration file: if the configuration file contains a definition for a gate with the name of the original gate with `_real` appended, then that one is created and replaces the original gate. Note that when this created gate is defined in the `gate_decomposition` section, the net effect is that the specified decomposition is done. When a `swap` or `move` gate is created to be inserted in the circuit, first a `swap_real` (or `move_real`) is attempted to be created instead before creating a `swap` or `move`; this also allows the gate to be decomposed to more primitive gates during mapping.

When a kernel's circuit has been mapped, an optional final decomposition of the mapped gates is done: each gate is optionally replaced by one or more gates as specified by the platform's configuration file, by creating a gate with the name of the original gate with `_prim` appended, if defined in the configuration file, and replacing the original gate by it. Note that when this created gate is specified in the configuration file in the `gate_decomposition` section, the net effect is that the specified decomposition is done. When in the mapped circuit, `swap` or `move` gates were inserted and `swap_prim` or `move_prim` are specified in the configuration file, these are also used to replace the `swap` or `move` at this time.

The `cycle` attribute of each gate is assigned a valid value. The gates in the circuit are ordered with non-decreasing cycle value. The cycle values are consistent with the constraints that are imposed during mapping; these are specified by the `mapper` option.

The above implies that non-quantum gates are accepted on input and are passed unchanged to output.

Options and Function

The options and corresponding function of the mapper are described.

The options include the proper mapper options and a few scheduler options. The subset of the scheduler options applies because the mapper uses the dependence graph created by the initialization method of the scheduler. Also see scheduling_options.

Most if not all options can be combined to compose a favorite mapping strategy, i.e. the options are largely independent.

With the options, also the effects that they have on the function of the mapper are described.

The options and function are described in the order of their virtual encountering by a particular gate that is mapped. Please remember that heuristic routing and mapping essentially performs a linear scan over the gates of the circuit to route the qubits, map and transform the gates.

Initialization and configuration

The `Init` method initializes the mapper for the given platform but independently of a particular kernel and circuit. This includes sanity checking and initializing the mapper's representation of the platform's topology from the platform's configuration file; see Configuration_file_definitions_for_mapper_control for the description of the platform's topology.

The topology's edges define the neighborhood/connection map of the real qubits. Floyd-Warshall is used to compute a distance matrix that contains for each real qubit pair the shortest distance between them. This makes the mapper applicable to arbitrary formed connection graphs but at the same time less scalable in number of qubits. For NISQ systems this is no problem. For larger and more regular connection grids, the implementation contains a provision to replace this by a distance function.

Subsequently, `Map` is called for each kernel/circuit in the program. It will attempt initial placement and then heuristic routing and mapping. Before anything else, for each kernel again, the `v2r` and `rs` are initialized, each under control of an option:

- `mapinitone2one`: Definition of the initialization of the `v2r` map at the start of the mapping of each kernel; this `v2r` will apply at the start of initial placement.
  - `no`: there is no initial mapping of virtual to real qubits; each virtual qubit is allocated to the first free real qubit on the fly, when it is mapped
  - `yes` (default for back-ward compatibility): the initial mapping is 1 to 1: a virtual qubit with index `qi` is mapped to its real `qi` counterpart (so: same index)
- `mapassumezeroinitstate`: Definition of the initialization of the `rs` map at the start of the mapping of each kernel; this `rs` will apply at the start of initial placement. Values can be: `rs_nostate` (no useful state), `rs_wasinited` (zero state), and `rs_hasstate` (useful but unknown state).
  - `no` (default for back-ward compatibility): each real qubit is assumed not to contain any useful state nor is it known that it is in a particular base state; this corresponds to the state with value `rs_nostate`.

- `yes` (best): each real qubit is assumed to be in a zero state (e.g. `|0>`) that allows a `swap` with it to be replaced by a (cheaper) `move`; this corresponds to the state with value `rs_wasinited`.

Initial Placement

After initialization and configuration, initial placement is started. See the start of mapping of a description of initial placement. Since initial placement may take a lot of computer time, provisions have been implemented to time it out; this comes in use during benchmark runs. Initial placement is run under the control of two options:

- `initialplace`: Definition of initial placement operation. Initial placement, when run, may be 100% successful (all two-qubit gates were made NN); be moderately successful (not all two-qubit gates were made NN, only some) or fail to find a solution:
  - `no` (default): no initial placement is attempted
  - `yes` (best, optimal result): do initial placement starting from the initial `v2r` mapping; since initial placement employs an Integer Linear Programming model as the base of implementation, finding an initial placement may take quite a while.
  - `1s, 10s, 1m, 10m, 1h` (best, limit time, still a result): put a soft time limit on the execution time of initial placement; do initial placement as with `yes` but limit execution time to the indicated maximum (one second, 10 seconds, one minute, etc.); when it is not successfull in this time, it fails, and subsequently heuristic routing and mapping is started, which cannot fail.
  - `1sx, 10sx, 1mx, 10mx, 1hx`: put a hard time limit on the execution time of initial placement; do initial placement as with `yes` but limit execution time to the indicated maximum (one second, 10 seconds, one minute, etc.); when it is not successfull in this time, it fails, and subsequently the compiler fails as well.
- `initialplace2qhorizon`: The initial placement algorithm considers only a specified number of two-qubit gates from the start of the circuit (a `horizon`) to determine a mapping. This limits computer time but also may make a suboptimal result more useful. Option values are:
  - `0` (default, optimal result): When `0` is specified as option value, there is no limit; all two-qubit gates of the circuit are taken into account.
  - `10, 20, 30, 40, 50, 60, 70, 80, 90, 100`: The initial placement algorithm considers only this number of initial two-qubit gates in the circuit to determine a mapping.

Best result would be obtained by running initial placement optionally twice (this is not implemented):

- Once with a modified model in which only the result with all two-qubit gates NN is successful. When it succeeds, mapping has completed. Depending on the resources one wants to spend on this, a soft time limit could be set.
- Otherwise, attempt to get a good starting mapping by running initial placement with a soft time limit (of e.g. 1 minute) and with a two-qubit horizon (of e.g. 10 to 20 gates). What ever the result is, run heuristic routing and mapping afterwards.

This concludes initial placement. The `v2r` and `rs` at this time are stored in attributes for retrieval by the caller of the `Map` method. See mapping_input_and_output_intermediate_representation.

Heuristic Routing and Mapping

Subsequently heuristic routing and mapping starts for the kernel given in the `Map` method call.

- The scheduler's dependence graph is used to feed heuristic routing and mapping with gates to map and to look-ahead: see mapping_dependence_graph.
- To map a non-NN two-qubit gate, various routing alternatives, to be implemented by `swap`/`move` sequences, are generated: see mapping_generating_routing_alternatives.
- Depending on the metric chosen, the alternatives are evaluated: see mapping_comparing_alternatives.
- When minimizing circuit latency extension, ILP is maximized by maintaining a scheduled circuit representation: see mapping_look_back.
- Looking farther ahead beyond the mapping of the current two-qubit gate, the router recurses considering the effects of its mapping on subsequent two-qubit gates: see mapping_looking_farther_ahead.
- Finally, the evaluations of the alternatives are compared, the best one selected and the two-qubit gate routed and mapped: see mapping_deciding_for_the_best.

Dependence Graph and Look-Ahead, Which Gate(s) To Map Next

The mapper optionally uses the dependence graph representation of the circuit to enlarge the number of alternatives it can consider, and to make use of the *criticality* of gates in the decision which one to map next. To this end, it calls the scheduler's `init` method, and sets up the availability list of gates as set of gates to choose from which one to map next: initially it contains just the `SOURCE` gates. See scheduling, and below for more information on the availability list's properties. The mapper listens to the following scheduler options:

- `scheduler_commute`: Because the mapper uses the dependence graph that is also generated for the scheduler, the alternatives that are made available by commutation of `czs`/`cnots`, can be made available to the mapper:
  - `no` (default for backward-compatibility): don't allow two-qubit gates to commute (`cz`/`cnot`) in the dependence graph; they are kept in original circuit order and presented to the mapper in this order
  - `yes` (best): allow commutation of two-qubit `cz`/`cnot` gates; e.g. when one isn't nearest-neighbor but one that comes later in the circuit but commutes with the earlier one is NN now, allow the later one to be mapped before the earlier one
- `print_dot_graphs`: When it has the value `yes`, the mapper produces in the output directory in multiple files each with as name the name of the kernel followed by `_mapper.dot` a `dot` representation of the dependence graph of the kernel's circuit at the start of heuristic routing and mapping, in which the gates are ordered along a timeline according to their cycle attribute.

With the dependence graph available to the mapper, its availability list is used just as in the scheduler:

- the list at each moment contains those gates that have not been mapped but can be mapped now
- the availability list forms a kind of *cut* of the dependence graph: all predecessors of the gates in the list and recursively all their predecessors have been mapped, all other gates have not been mapped (the *cut* is really the set of dependences between the set of mapped and the set of non-mapped gates)
- each moment a gate has been mapped, it is taken out of the availability list; those of its successor dependence gates of which all predecessors have been mapped, become available for being mapped, i.e. are added to the availability list

This dependence graph is used to look-ahead, to find which two-qubit to map next, to make a selection from all that are available or take just the most critical one, to try multiple ones and evaluate each alternative to map it, comparing those alternatives against one of the metrics (see later), and even go into recursion (see later as well), i.e. looking farther ahead to see what the effects on subsequent two-qubit gates are when mapping the current one.

In this context the *criticality* of a gate is an important property of a gate: the *criticality* of a gate is the length of the longest dependence path from the gate to the `SINK` gate and is computed in a single linear backward scan over the dependence graph (Dijkstra's algorithm).

Deciding for the next two-qubit gate to map, is done based on the following option:

- `maplookahead`: How does the mapper exploit the lookahead offered by the dependence graph constructed from the input circuit?
  - `no`: the mapper ignores the dependence graph and takes the gates to be mapped one by one from the input circuit
  - `critical`: gates that by definition do not need routing, are mapped first (and kind of flushed): these include the classical gates, scheduling gates (such as `wait`), and the single qubit quantum gates; and of the remaining (only two qubit) quantum gates the most critical gate is selected first to be routed and mapped next; the rationale of taking the most critical gate is that after that one the most cycles are expected until the end of the circuit, and so a wrong routing decision of a critical gate is likely to have most effect on the mapped circuit's latency; so criticality has higher priority to select the one to be mapped next, than NN (see `noroutingfirst` for the opposite approach)
  - `noroutingfirst` (default, best): gates that by definition do not need routing, are mapped first (and kind of flushed): these include the classical gates, scheduling gates (such as `wait`), and the single qubit quantum gates; in this, this `noroutingfirst` option has the same effect as `critical`; but those two qubit quantum gates of which the operands are neighbors in the current mapping are selected to be mapped first, not needing routing, also when these are not critical; and when none such are left, only then take the most critical one; so NN has higher priority to select the one to be mapped next, than criticality
  - `all` (promising in combination with recursion): as with `noroutingfirst` but don't select the most critical one, select them all; so at each moment gates that do not need routing, are mapped first (and kind of flushed); these thus include the NN two-qubit gates; this mapping and flushing stops when only non-NN two-qubit gates remain; instead of selecting one of these to be routed/mapped next, all of these are selected, the decision is postponed; i.e. for all remaining (two qubit non-NN) gates generate alternatives and find the best from these according to the chosen metric (see the `mapper` option below); and then select that best one to route/map next

Generating Routing Alternatives

Having selected one (or more) two-qubit gates to map next, for each two-qubit gate the routing alternatives are explored. Subsequently, those alternatives will be compared using the selected metric and the best one selected; see further below.

But first the routing alternatives have to be generated. When the mapped operands of a two-qubit gate are not NN, they must be made NN by swapping/moving one or both over nearest-neighbor connections in the target platform's grid topology towards each other. Only then the two-qubit gate can be executed; the mapper will insert those `swaps` and `moves` before the two-qubit gate in the circuit.

There are usually many routes between the qubits. The current implementation only selects the ones with the shortest distance, and these can still be many. In a perfectly rectangular grid, the number of routes is similar to a Fibonaci number depending on the distance decomposed in the x and y directions, and is maximal when the distances in the x and y directions are equal. All shortest paths between two qubits in such a grid stay within a rectangle in the grid with the mapped qubit operands at opposite sides of the diagonal.

A shortest distance leads to a minimal number of `swaps` and `moves`. For each route between qubits at a distance *d*, there are furthermore *d* possible places in the route where to do the two-qubit gate; the other *d-1* places in the route will be a `swap` or a `move`.

The implementation supports an arbitrarily formed connection graph, so not only a rectangular grid. All that matter are the distances between the qubits. Those have been computed using Floyd-Warshall from the qubit neighbor relations during initialization of the mapper. The shortests paths are generated in a brute-force way by only navigating to those neighbor qubits that will not make the total end-to-end distance longer. Unlike other implementations that only minimize the number of swaps and for which the routing details are irrelevant, this implementation explicitly generates all alternative paths to allow the more complicated metrics that are supported, to be computed.

The generation of those alternatives is controlled by the following option:

- `mappathselect`: When generating alternatives of shortest paths between two real qubits:
  - `all` (default, best): select all possible alternatives: those following all possible shortest paths and in each path each possible placement of the two-qubit gate
  - `borders`: only select those alternatives that correspond to following the borders of the rectangle spanning between the two extreme real qubits; so on top of the at most two paths along the borders, there still are all alternatives of the possible placements of the two-qubit gate along each path

It is thus not supported to turn off to generate alternatives for the possible placements of the two-qubit gate along each path.

The alternatives are ordered; this is relevant for the `maptiebreak` option below. The alternatives are ordered:

- first by the *two-qubit gate* for which they are an alternative; the most critical two-qubit gate is first; remember that there can be more than one two-qubit gate when `all` was selected for the `maplookahead` option.
- then by the *followed path*; each path is represented by a sequence of transitions from the mapped first operand qubit to the mapped second operand qubit. The paths are ordered such that of any set of paths with a common prefix these are ordered by a clock-wise order of the successor qubits as seen from the last qubit of the common prefix.
- and then by the *placement* of the two-qubit gate; the placements are ordered from start to end of the path.

So, the first alternative will be the one that clock-wise follows the border and has the two-qubit gate placed directly at the qubit that is the mapped first operand of the gate; the last alternative will be the one that anti-clock-wise follows the border and has the two-qubit gate placed directly at the qubit that is the mapped last operand of the gate.

Comparing Alternatives, Which Metric To Use

With all alternatives available, it is time to compare them using the defined metric. The metric to use is defined by the `strategy` option, called for historic reasons `mapper`. What needs to be done when multiple alternatives compare equal, is specified later.

- `mapper`: The basic mapper strategy (metric of mapper result optimization) that is employed:
  - `no` (default for back-ward compatibility): no mapping is done. The output circuit is identical to the input circuit.
  - `base`: map the circuit: use as metric just the length of the paths between the mapped operands of each two-qubit gate, and minimize this length for each two-qubit gate that is mapped; with only alternatives for one two-qubit gate, all alternatives have the same shortest path, so all alternatives qualify equally; with alternatives for multiple two-qubit gates, those two-qubit gates are preferred that lead to the least `swaps` and `moves`.
  - `minextend` (best): map the circuit: use as metric the extension of the circuit by each of the shortest paths between the mapped operands of each two-qubit gate, and minimize this circuit extension by evaluating all alternatives; the computation of the extension relies on scheduling-in the required swaps and moves in the circuit and just subtracting the depths before and after doing that; the various options controlling this scheduling-in, will be specified later below.

  - `minextendrc`: map the circuit: as in `minextend`, but taking resource constraints into account when scheduling-in the `swaps` and `moves`.

Look-Back, Maximize Instruction-Level Parallelism By Scheduling

To know the circuit's latency extension of an alternative, the mapped gates are represented as a scheduled circuit, i.e. with gates with a defined `cycle` attribute, and the gates ordered in the circuit with non-decreasing `cycle` value. In case the `mapper` option has the `minextendrc` value, also the state of all resources is maintained. When a `swap` or `move` gate is added, it is ASAP scheduled (optionally taking the resource constraints into account) into the circuit and the corresponding cycle value is assigned to the `cycle` attribute of the added gate. Note that when `swap` or `move` is defined by a composite gate, the decomposed sequence is scheduled-in instead.

The objective of this is to maximize the parallel execution of gates and especially of `swaps` and `moves`. Indeed, the smaller the latency extension of a circuit, the more parallelism was created, i.e. the more the ILP was enlarged. When `swaps` and `moves` are not inserted as primitive gates but the equivalent decomposed sequences are inserted, ILP will be improved even more.

This scheduling-in is done separately for each alternative: for each alternative, the `swaps` or `moves` are added and the end-result evaluated.

This scheduling-in is controlled by the following options:

- `mapusemoves`: Use `move` instead of `swap` where possible. In the current implementation, a `move` is implemented as a sequence of two `cnots` while a `swap` is implemented as a sequence of three `cnots`.
  - `no`: don't
  - `yes` (default, best): do, when swapping with an ancillary qubit which is known to be in the zero state (`|0>` for moves with 2 `cnots`); when not in the initial state, insert a `move_init` sequence (when defined in the configuration file, the defined sequence, otherwise a prepz followed by a hadamard) when it doesn't additionally extend the circuit; when a `move_init` sequence would extend the circuit, don't insert the `move`
  - `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20`: yes, and insert a `move_init` sequence to get the ancillary qubit in the initial state, if needed; but only when the number of cycles of circuit extension that this `move_init` causes, is less-equal than 0, 1, . . . 20 cycles.

    Please note that the `mapassumezeroinitstate` option defines whether the implementation of the mapper can assume that each qubit starts off in the initial state; this increases the likelihood that moves are inserted, and makes all these considerations of only inserting a `move` when a `move_init` can bring the ancillary qubit in the initial state somehow without additional circuit extension, of no use.
- `mapprepinitsstate`: Does a `prepz` initialize the state, i.e. leave the state of a qubit in the `|0>` state? When so, this can be reflected in the `rs` map.
  - `no` (default, playing safe): no, it doesn't; a `prepz` during mapping will, as any other quantum gate, set the state of the operand qubits to `rs_hasstate` in the `rs` map
  - `yes` (best): a `prepz` during mapping will set the state of the operand qubits to `rs_wasinited`; any other gate will set the state of the operand qubits to `rs_hasstate`
- `mapselectswaps`: When scheduling-in `swaps` and `moves` at the end for the best alternative found, this option selects that potentially not all required `swaps` and `moves` are inserted. When not all are inserted but only one, the distance of the mapped operand qubits of the two-qubit gate for which the best alternative was generated, will be one less, and after insertion heuristic routing and mapping starts over generating alternatives for the new situation.

Please note that during evaluation of the alternatives, all `swaps` and `moves` are inserted. So the alternatives are compared with all `swaps` and `moves` inserted but only during the final real insertion after having selected the best alternative, just one is inserted.

- `all` (best, default): insert all `swaps` and `moves` as usual
- `one`: insert only one `swap` or `move`; take the one swapping/moving the mapped first operand qubit
- `earliest`: insert only one `swap` or `move`; take the one that can be scheduled earliest from the one swapping/moving the mapped first operand qubit and the one swapping/moving the mapped second operand qubit

- `mapreverseswap`: Since `swap` is symmetrical in effect (the states of the qubits are exchanged) but not in implementation (the gates on the second operand start one cycle earlier and end one cycle later), interchanging the operands may cause a `swap` to be scheduled at different cycles. Reverse operand real qubits of `swap` when beneficial:
  - `no`: don't
  - `yes` (best, default): when scheduling a `swap`, exploiting the knowledge that the execution of a `swap` for one of the qubits starts one cycle later, a reversal of the real qubit operands might allow scheduling it one cycle earlier

Looking Farther Ahead, Recurse To Find Best Alternative

Looking farther ahead beyond the mapping of the current two-qubit gate, the router recurses considering the effects of its mapping on subsequent two-qubit gates.

After having evaluated the metric for each alternative, multiple alternatives may remain, all with the best value. For the `minextend` and `minextendrc` strategies, there are options to select from these by looking ahead farther, i.e. beyond the metric evaluation of this alternative for mapping one two-qubit gate. This *recursion* assumes that the current alternative is selected, its `swaps` and `moves` are added to the circuit the `v2r` map is updated, and the availability set is updated. And then in this new situation the implementation recurses by selecting one or more two-qubit gates to map next, generating alternatives, evaluating these alternatives against the metric, and deciding which alternatives are the best. This recursion can go deeper and deeper until a particular depth has been reached. Then of the resulting tree of alternatives, for all the leaves representing the deepest alternatives, the metric is computed from the root to the leaf and compared to each other. In this way suboptimalities of individual choices can be balanced to a more optimal combination. From these leaves, the best is taken; when multiple alternatives compare equally well from root to leaf, the `maptiebreak` option decides which one to take, as usual; see below there.

The following options control this recursion:

- `mapselectmaxlevel`: Looking farther ahead beyond the mapping of the current two-qubit gate, the router recurses considering the effects of its mapping on subsequent two-qubit gates. The level specifies the recursion depth: how many two-qubits in a row are considered beyond the current one. This generates a tree of alternatives.
  - `0` (default, back-ward compatible): no recursion is done
  - `1, 2, 3, 4, 5, 6, 7, 8, 9, 10`: the indicated number of recursions is done; initial experiments show that a value of `3` produces reasonable results, and that recursion depth of `5` and higher are infeasible because of resource demand explosion
  - `inf`: there is no limit to the number of recursions; this makes the resource demand of heuristic routing and mapping explode
- `mapselectmaxwidth`: Not all alternatives are equally promising, so only some best are selected to recurse on. The width specifies the recursion width: for how many alternatives the recursion is actually done. The specification of the width is done relative to the number of alternatives that came out as best at the current recursion level.

  - `min` (default): only recurse on those alternatives that came out as best at this point
  - `minplusone`: only recurse on those alternatives that came out as best at this point, plus one second-best
  - `minplushalfmin` (best combination of optimality and resources: only recurse on those alternatives that came out as best at this point, plus some number of second-bests: half the number more than the number of best ones
  - `minplusmin`: only recurse on those alternatives that came out as best at this point, plus some number of second-bests: twice the number of best ones
  - `all`: don't put a limit on the recursion width

- `maprecNN2q`: In `maplookahead` with value `all`, as with `noroutingfirst`, two-qubit gates which are already NN, are immediately mapped, kind of flushing them. However, in recursion this creates an imbalance: at each level optionally several more than just one two-qubit gate are mapped and this makes the results of the alternatives largely incomparable. Comparision would be easier to understand when at each level only one two-qubit gate would be mapped. This option specifies independently of the `maplookahead` option that is chosen and that is applied before going into recursion, whether in the recursion this immediate mapping/flushing of NN two-qubit gates is done.
  - `no` (default, best): no, NN two-qubit gates are not immediately mapped and flushed until only non-NN two-qubit gates remain; at each recursion level exactly one two-qubit gate is mapped
  - `yes`: yes, NN two-qubit gates are immediately mapped and flushed until only non-NN two-qubit gates remain; this makes recursion more greedy but makes interpreting the evaluations of the alternatives harder

Deciding For The Best, Committing To The Best

With or without recursion, for the `base` strategy as well as for the `minextend` and `minextendrc` strategies, when at the end multiple alternatives still compare equally well, a decision has to be taken which two-qubit gate to route and map. This selection is made based on the value of the following option:

- `maptiebreak`: When multiple alternatives remain for a particular strategy with the same best evaluation value, decide how to select the best single one:
  - `first`: select the first of the set
  - `last`: select the last of the set
  - `random` (default, best, non-deterministic): select in a random way from the set; when testing and comparing mapping strategies, this option introduces non-determinism and non-reproducibility, which precludes reasoning about the strategies unless many samples are taken and statistically analyzed
  - `critical` (deterministic, second best): select the first of the alternatives generated for the most critical two-qubit gate (when there were more)

Having selected a single best alternative, the decision has been made to route and map its corresponding two-qubit gate. This means, scheduling in the result circuit the `swaps` and `moves` that route the mapped operand qubits, updating the `v2r` and `rs` maps on the fly; see mapping_look_back for the details of this scheduling. And then map the two-qubit gate; see mapping_input_and_output_intermediate_representation for what mapping involves.

After this, in the dependence graph a next gate is looked for to map next and heuristic routing and mapping starts over again.

Configuration file definitions for mapper control

The configuration file contains the following sections that are recognized by the mapper:

- **`hardware_settings`** the number of real qubits in the platform, and the cycle time in nanoseconds to convert instruction duration into cycles used by the various scheduling actions are taken from here
- **`instructions`** the mapper assumes that the OpenQL circuit was read in and that gates were created according to the specifications of these in the configuration file: the name of each encountered gate is looked up in this section and, if not found, in the `gate_decomposition` section; if found, that gate (or those gates) are created; the `duration` field specifies the duration of each gate in nanoseconds; the `type` and various `cc_light` fields of each instruction are used as parameters to select applicable resource constraints in the resource-constrained scheduler
- **`gate_decomposition`** when creating a gate matching an entry in this section, the set of gates specified by the decomposition description of the entry is created instead; the mapper exploits the decomposition support that the configuration file offers by this section in the following way:
  - *reading the circuit* When a gate specified as a composite gate is created in an OpenQL program, its decomposition is created instead. So a `cnot` in the OpenQL program that is specified in the `gate_decomposition` section as e.g. two `hadamard`s with a `cz` in the middle, is input by the mapper as this latter sequence.
  - *swap support* A `swap` is a composite gate, usually consisting of 3 `cnot`s; those `cnot`s usually are decomposed to a sequence of primitive gates itself. The mapper supports generating `swap` as a primitive; or generating its shallow decomposition (e.g. to `cnot`s); or generating its full decomposition (e.g. to the primitive gate set). The former leads to a more readable intermediate qasm file; the latter to more precise evaluation of the mapper selection criteria. Relying on the configuration file, when generating a `swap`, the mapper first attempts to create a gate with the name `swap_real`, and when that fails, create a gate with the name `swap`. The same machinery is used to create a `move`.
  - *making gates real* Each gate input to the mapper is a virtual gate, defined to operate on virtual qubits. After mapping, the output gates are real gates, operating on real qubits. *Making gates real* is the translation from the former to the latter. This is usually done by replacing the virtual qubits by their corresponding real qubits. But support is provided to also replace the gate itself: when a gate is made real, the mapper first tries to create a gate with the same name but with `_real` appended to its name (and using the mapped, real qubits); if that fails, it keeps the original gate and uses that (with the mapped, real qubits) in the result circuit.
  - *ancilliary initialization* For a `move` to be done instead of a `swap`, the target qubit must be in a particular state. For CC-Light this is the `|+>` state. To support other target platforms, the `move_init` gate is defined to prepare a qubit in that state for the particular target platform. It decomposes to a `prepz` followed by a `Hadamard` for CC-Light.
  - *making all gates primitive* After mapping, the output gates will still have to undergo a final schedule with resource constraints before code can be generated for them. Best results are obtained when then all gates are primitive. The mapper supports a decomposition step to make that possible and this is typically used to decompose leftover `swap`s and `move`s to primitives: when a gate is made primitive, the mapper first tries to create a gate with the same name but with `_prim` appended to its name; if that fails, it keeps the original gate and uses that in the result circuit that is input to the scheduler.
- `topology` A qubit grid's topology is defined by the neighbor relation among its qubits. Each qubit has an `id` (its index, used as a gate operand and in the resources descriptions) in the range of `0` to the number of qubits in the platform minus 1. Qubits are connected by directed pairs, called *edge*s. Each edge has an `id` (its index, also used in the resources descriptions) in some contiguous range starting from `0`, a source qubit and a destination qubit. Two grid forms are supported: the `xy` form and the `irregular` form. In grids of the `xy` form, there must be two additional attributes: `x_size` and `y_size`, and the qubits have in addition an X and a Y coordinate: these coordinates in the X (Y) direction are in the range of `0` to `x_size-1` (`y_size-1`).

- `resources` See the scheduler's documentation.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX