
OpenQL

May 27, 2021

1	Overview	3
1.1	This document	3
1.2	OpenQL compiler structure	3
2	Installation	5
2.1	Installing the pre-built package	5
2.2	Compilation from sources	6
3	Creating your first Program	9
3.1	Hello World	9
3.2	Notebooks	10
3.3	Examples	11
3.4	Tests	11
4	Program	13
5	Kernel	15
5.1	Control flow in the internal representation	16
5.2	Control flow in the output external representation	18
5.3	API	19
6	Quantum Gates	21
6.1	Quantum gate attributes in the internal representation	22
6.2	Circuits and bundles in the internal representation	25
6.3	Input external representation	25
6.4	Output external representation	26
7	Classical Instructions	27
7.1	Classical gate attributes in the internal representation	28
7.2	Classical gates in circuits and bundles in the internal representation	29
7.3	Input external representation	30
7.4	Output external representation	30
8	Platform	31
8.1	Platform Configuration File	32
8.2	QX Platform	36
8.3	Quantumsim Platform	36

8.4	CC-Light Platform	37
8.5	Central Controller Platform Configuration	43
8.6	CBox Platform	50
9	Compiler	51
10	Compiler Passes	53
10.1	Summary of compiler passes	54
10.2	Decomposition	56
10.3	Optimization	59
10.4	Scheduling	60
10.5	Mapping	65
11	Change Log	77
11.1	[next] - [TBD]	77
11.2	[0.8.0] - [2019-10-31]	78
11.3	[0.7.1] - [2019-09-02]	78
11.4	[0.7.0] - [2019-06-03]	78
11.5	[0.6] - [2018-10-29]	79
11.6	[0.5.5] - [2018-10-25]	79
11.7	[0.5.4] - [2018-10-17]	80
11.8	[0.5.3] - [2018-10-11]	80
11.9	[0.5.2] - [2018-10-10]	80
11.10	[0.5.1] - [2018-09-12]	81
11.11	[0.5] - [2018-06-26]	81
11.12	[0.4.1] - [2018-05-31]	81
11.13	[0.4] - [2018-05-19]	82
11.14	[0.3] - [2017-10-24]	83
11.15	[0.2] - [2017-08-18]	83
12	Contributors	85
13	openql	89
14	Kernel	99
15	Program	105
16	Compiler	109
17	Platform	111
18	Operation	113
19	CReg	115
20	QX Simulation	117
20.1	OpenQL Program	117
21	DQCsims Simulation	121
21.1	Dependencies	121
21.2	Replicating the QXelarator results	121
21.3	Enabling QX's depolarizing channel error model	122
21.4	Using QuantumSim instead	123
21.5	Further reading	124

22 Developer Documentation	125
Index	127

OpenQL is a framework for high-level quantum programming in C++/Python. The framework provides a compiler for compiling and optimizing quantum code. The compiler produces quantum assembly and instruction-level code for various target platforms. While the instruction-level code is platform-specific, the quantum assembly code (QASM) is hardware-agnostic and can be simulated on one of the simulators.

OpenQL is a framework for high-level quantum programming in C++/Python. The framework provides a compiler for compiling and optimizing quantum code.

1.1 This document

The first three chapters introduce OpenQL, help to install it, and show how to create a first OpenQL program. They are here for people who want to get going with OpenQL as quickly as possible. For people just wanting an overview of OpenQL, these, except for the installation chapter, are a must read.

Further chapters introduce to the basic concepts of OpenQL. They contain a lot of conceptual texts, and inevitable for a good understanding of the system. What is a program, what is a kernel and to which extent are classical instructions supported? What kind of gates does OpenQL support, which are the internal and which are the external representations? Omni-present in OpenQL is the platform, literally in the form of the platform configuration file that parameterizes most passes on the supported platform. And finally the compiler passes, in a summary as well as in an extensive description with functional description and sets of options listened too.

The document concludes with lists of APIs and indices.

1.2 OpenQL compiler structure

An OpenQL compiler reads a quantum program written in some external representation, performs several analysis and transformation passes on it, and prints the result to an external representation again. Internally in the compiler the passes operate on a common internal representation of the program, IR for short, which is equal to all passes.

Understanding this internal representation is key to understanding the operation of an OpenQL compiler. It is structured as an attributed tree of objects.

At the top one finds the (internal representation of the) program. Its main component is the vector of kernels. Each ordinary kernel (object) contains a single circuit which basically is a vector of gates. Gates in OpenQL are the constructs that refer to operations to be executed somehow on the computing platform. These can be quantum gates as well as classical gates; the latter deal with classical arithmetic and measurement results. A circuit of a kernel is always

executed from start to end. There are special kernels without a circuit that take care of control flow between kernels. But for ordinary kernels after the last gate control is transferred to the next kernel.

All passes operate at the program level. Each performs its work on all kernels before it completes and another pass can run. The order of the passes is predefined by OpenQL, but there are ways to enable/disable individual passes. The effect of a pass is to update the internal representation. This can amount to computing attributes, replacing gates by other ones, rearranging gates, and so on.

The objective of an OpenQL compiler is to produce an output external representation of the input program that satisfies the needs of what comes next. What comes next is represented in OpenQL by the (target) platform. These platforms can be software simulators or architectures targetting hardware quantum computers.

To the compiler this platform is described by a *platform configuration file*, a file in JSON format, which contains several sections with descriptions of attributes of the platform. Examples of these are the number of qubits, the supported set of primitive gates with their attributes, the connection graph between the qubits (also called the topology of the grid), and the classical control constraints imposed by the control electronics of the hardware of the platform. It also specifies for which hardware platform family it contains the configuration. These hardware platform families (called architectures) are built-in into the OpenQL compiler, and the compiler, after having executed some platform independent passes, will enter the architecture-specific part of the compiler where it executes several platform dependent passes. When compiling for a hardware quantum computer target, the last ones of these will generate some form of low-level assembly code corresponding to the particular instruction set of the platform.

OpenQL is supported on Linux, Windows and OSX. OpenQL can be installed on these platforms as a pre-built package as well as can be compiled from sources.

- **Pre-built package**
 - python package using pip
 - conda package
- **Compilation from sources**
 - Windows
 - Linux
 - OSX

2.1 Installing the pre-built package

Pre-built packages are available for OpenQL.

2.1.1 Pre-built Wheels

This is perhaps the easiest way to get OpenQL running on your machine.

Pre-built OpenQL wheels are available for 64-bit Windows, Linux and OSX. These wheels are available for Python 3.5, 3.6 and 3.7. OpenQL can be installed by the command:

```
pip install qutechopenql
```

Note: `python` refers to Python 3 and `pip` refers to Pip 3, corresponding to Python version 3.5, 3.6 or 3.7.

2.1.2 Conda package

OpenQL can be installed as a conda package (currently on Linux and Windows only) by:

```
conda install -c qe-lab openql
```

Conda packages can also be built locally by using the recipe available in the conda-recipe directory, by running the following command from the OpenQL directory:

```
conda build conda-recipe/.
```

The generated package can then be installed by:

```
conda install openql --use-local
```

2.2 Compilation from sources

Compiling OpenQL from sources involves:

- Setting-up required packages
- Obtaining OpenQL

2.2.1 Required Packages

The following packages are required to compile OpenQL from sources:

- g++ compiler with C++11 support (Linux)
- MSVC 2015 with update 3 or above (Windows)
- flex (> 2.6)
- bison (> 3.0)
- cmake (>= 3.0)
- swig (Linux: 3.0.12, Windows: 4.0.0)
- Python (3.5, 3.6, 3.7)
- [Optional] pytest used for running tests
- [Optional] Graphviz Dot utility to convert graphs from dot to pdf, png etc
- [Optional] XDot to visualize generated graphs in dot format

2.2.2 Notes for Windows Users

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.

- Use Power Shell for installation
- Set execution policy by:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

- Install [PowerShell Community Extensions] (<https://www.google.com> “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"
```

2.2.3 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
```

2.2.4 Compiling OpenQL as Python Package

Running the following command in the python (virtual) environment in Terminal/Power Shell should install the openql package:

```
cd OpenQL
git submodule update --init --recursive
python setup.py install
```

Or in editable mode by the command:

```
pip install -e .[develop]
```

Running the tests

In order to pass all the python tests, the openql package should be installed in editable mode. Also, *qisa-as* and *libqasm* should be installed first. Follow [qisa-as](#) and [libqasm](#) instructions to install python interfaces of these modules. Once *qisa-as* and *libqasm* are installed, you can run all the tests by:

```
pytest -v
```

or

```
python -m pytest
```

2.2.5 Compiling C++ OpenQL 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/OSX

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.

CHAPTER 3

Creating your first Program

In the OpenQL framework, the quantum program (including kernels and gates) is created by API calls which are contained in a C++ or Python program.

But before this is done, the platform object is created by an API call that takes the name of the platform configuration file as one of its parameters. This platform configuration file is consulted by the APIs creating the program, kernels and gates to generate the matching internal representation of each gate.

After creating the platform, the program and kernels are created. The program creation API takes the program name, the platform object and the number of qubits that are used in the program as parameters. And similarly for each kernel. After this, each kernel can be populated with gates. This is again done by API calls, one per gate.

After having added each kernel to the program, the program can be compiled. This leaves several output files in the *test_output* directory. When compiling for CC-Light which is one of the hardware platforms of OpenQL, one will find there a *.qisa* file which then can be executed on the platform. But one will also find there several *.qasm* files which can be simulated by e.g. QX.

Let us start creating a program.

To begin working with OpenQL, you can start up python however you like. You can open a jupyter notebook (type `jupyter notebook` in your terminal), open an interactive python notebook in your terminal (with `ipython3`), or simply launch python in your terminal (by typing `python3`).

3.1 Hello World

In this section we will run ‘Hello World’ example of OpenQL. The first step is to import `openql` which can be done by:

```
from openql import openql as ql
```

Next, create a platform by:

```
platform = ql.Platform("myPlatform", config_file_name)
```

where, `config_file_name` is the name of the configuration file in JSON format which specifies the platform, e.g. `hardware_configuration_cc_light.json`. For details, refer to [Platform](#).

For this example we will be working on 3 qubits. So let us define a variable so that we can use it at multiple places in our code.

```
nqubits = 3
```

Create a program

```
p = ql.Program("aProgram", platform, nqubits)
```

Create a kernel

```
k = ql.Kernel("aKernel", platform, nqubits)
```

Populate this kernel using default and custom gates

```
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])
k.gate('measure', [1])
```

Add the kernel to the program

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

Compile the program

```
p.compile()
```

This will generate the output files in the `test_output` directory.

A good place to get started with your own programs might be to copy `examples/getting_started.py` to some folder of your choice and start modifying it. For further examples, have a look at the test programs inside the “tests” directory.

Todo: discuss the generated output files

3.2 Notebooks

Following Jupyter notebooks are available in the `<OpenQL Root Dir>/examples/notebooks` directory:

ccLightClassicalDemo.ipynb This notebook provides an introduction to compilation for ccLight with an emphasis on:

- hybrid quantum/classical code generation
- **control-flow in terms of:**
 - if, if-else
 - for loop

- do-while loop
- getting measurement results

3.3 Examples

Following Jupyter notebooks are available in the `<OpenQL Root Dir>/examples` directory:

getting_started.py The Hello World example discussed in *helloworld* section.

rb_single.py Single qubit randomized benchmarking.

3.4 Tests

Various tests are also available in the `<OpenQL Root Dir>/tests` directory which can also be used as examples testing various features of OpenQL.

CHAPTER 4

Program

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_configuration_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])
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]

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(...))
```

(continues on next page)

(continued from previous page)

```

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`.

5.1 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			<code>p.add(ql.kernel(label, ...))</code>
FOR_START	<code>body.name+'for_start'</code>			loopcount	<code>p.add_for(body, loopcount)</code>
FOR_END	<code>body.name+'for_end'</code>				
DO_WHILE_START	<code>body.name+'do_while_start'</code>		loopcond		<code>p.add_do_while(body, loopcond)</code>
DO_WHILE_END	<code>body.name+'do_while'</code>				
IF_START	<code>then.name+'if'</code>		thencond		<code>p.add_if(then, thencond)</code>
IF_END	<code>then.name+'if_end'</code>				
ELSE_START	<code>else.name+'else'</code>				<code>p.add_if_else(then, else, thencond)</code>
ELSE_END	<code>else.name+'else_end'</code>				

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.

5.2 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`:

<code>br_condition</code>	<code>operands</code>	<code>target label</code>	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

5.3 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]

Quantum Gates

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

6.1 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. `mr \times 90`, `m \times 90`, and `Rm \times 90` 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

6.2 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.

6.3 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

6.4 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.

Classical Instructions

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)
```

(continues on next page)

(continued from previous page)

```
# 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.

7.1 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 scheduler	string
creg_operands		[r0,r1]			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	ARITHMETIC		k.classical(rd, Operation(‘~’, rs))
“mov”				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](#)

7.2 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 be encoded in resources which are then adhered to by the rcscheduler.

7.3 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.

7.4 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

OpenQL supports various target platforms. These platforms can be software simulators or architectures targeting hardware quantum computers. The following platforms are supported by OpenQL:

- **software simulator platforms**
 - *QX Platform*
 - *Quantumsim Platform*
- **hardware platforms**
 - *CC-Light Platform*
 - *Central Controller Platform Configuration*
 - *CBox Platform*

Note Quantumsim and QX are not really platforms. They are means to simulate a particular (hardware) platform. Qasm files for use by QX and python scripts to interface to quantumsim are generated for any hardware platform under the control of options. See the descriptions of the QX and Quantumsim platforms referred to above.

Note We are planning to use DQCs, a platform to connect to simulators. In that context, software simulator platforms are connected to by DQCs, and OpenQL just provides compilation support to a particular hardware platform.

A platform can be created in OpenQL by using the `Platform()` API as shown below:

```
platform = ql.Platform('<platform_name>', <path_to_json_config_file>)
```

For example, a platform with the name `CCL_platform` and using `hardware_config_cc_light.json` as platform configuration file can be created as:

```
platform = ql.Platform('CCL_platform', 'hardware_config_cc_light.json')
```

8.1 Platform Configuration File

The platform configuration file describes the target platform in JSON format. The information in this file is used by all platform independent compiler passes. The parameterization by this information makes these platform independent compiler passes in source code independent of the platform but in effective function dependent on the platform.

A platform configuration file consists of several sections (in arbitrary order) which are described below. Most of these are mandatory; the specification of the `topology` and the `resources` sections are optional.

For example (the `...` contains the specification of the respective section):

```
1 {
2   "eqasm_compiler" : "cc_light_compiler",
3
4   "hardware_settings":
5   {
6     "qubit_number": 7,
7     "cycle_time" : 20,
8     ...
9   },
10
11  "topology":
12  {
13    ...
14  },
15
16  "resources":
17  {
18    ...
19  },
20
21  "instructions":
22  {
23    ...
24  },
25
26  "gate_decomposition":
27  {
28    ...
29  }
30 }
```

The platform configuration file for its structure is platform independent. It can be extended at will with more sections and more attributes for platform dependent purposes.

The sections below describe sections and attributes that are used by platform independent compiler passes.

Please refer to the sections of the specific platforms for full examples and for the description of any additional attributes.

8.1.1 Attribute `eqasm_compiler`

The `eqasm_compiler` attribute specifies the backend compiler to be used for this platform. After the passes of the platform independent compiler have been called, the platform independent compiler switches out to the backend compiler to run the platform dependent passes. The specification of this attribute is mandatory. The `eqasm_compiler` attribute can take the following values; these correspond to the platforms that are supported:

- `none`: no backend compiler is called

- `qx`: no backend compiler is called; see above how to generate a qasm file for QX
- `cc_light_compiler`: backend compiler for CC_Light
- `eqasm_backend_cc`: backend compiler for CC
- `qumis_compiler`: backend compiler to CBOX

8.1.2 Section `hardware_settings`

The `hardware_settings` section specifies various parameters describing the platform. These include the `qubit_number` and `cycle_time` which are generally used, and the buffer delays, only used by the rescheduler, which are related to control electronics in the experiments (for hardware backends). The specification of this section is mandatory.

For example:

```

1 "hardware_settings":
2 {
3     "qubit_number": 7,
4     "cycle_time" : 20,
5
6     "mw_mw_buffer": 0,
7     "mw_flux_buffer": 0,
8     "mw_readout_buffer": 0,
9     "flux_mw_buffer": 0,
10    "flux_flux_buffer": 0,
11    "flux_readout_buffer": 0,
12    "readout_mw_buffer": 0,
13    "readout_flux_buffer": 0,
14    "readout_readout_buffer": 0
15 }
```

In this:

- `qubit_number` indicates the number of (real) qubits available in the platform. Gates and instructions that address qubits do this by providing a qubit index in the range of 0 to `qubit_number-1`. Using an index outside this range will raise an error.
- `cycle_time` is the clock cycle time. As all other timing specifications in the configuration file it is specified in nanoseconds. Only at multiples of this cycle time, instructions can start executing. The schedulers assign a cycle value to each gate, which means that that gate can start executing a number of nanoseconds after program execution start that equals that cycle value multiplied by the clock `cycle_time` value.
- The other entries of the `hardware_settings` section specify various buffer times to be inserted between various operations due to control electronics setup. For example, `mw_mw_buffer` can be used to specify time to be inserted between a microwave operation followed by another microwave operation. See [Scheduling](#) for details.

8.1.3 Section `topology`

Specifies the qubit topology as the connection graph of the qubits of the platform. This is primarily used by the mapping pass; this section is optional. It specifies the mapping of qubit indices to qubit positions in the platform, as well as the mapping of connection indices to connections in the platform. A connection is a directed connection in the platform between a pair of qubits that supports qubit interaction. It is directed to distinguish the control and target qubits of two-qubit gates. In a platform topology's connection graph, qubits are the nodes, and connection are the edges.

It looks like (the . . . contains further specifications):

```
1 "topology" :
2 {
3     "x_size": 5,
4     "y_size": 3,
5     "qubits":
6     [
7         { "id": 0,  "x": 1, "y": 2 },
8         ...
9     ],
10    "edges":
11    [
12        { "id": 0,  "src": 2, "dst": 0 },
13        ...
14    ]
15 },
```

The `topology` section starts with the specification of the two dimensions of a rectangular qubit grid by specifying `x_size` and `y_size`. The positions of the real qubits of the platform are defined relative to this (artificial) grid. The coordinates in the X direction are 0 to `x_size-1`. In the Y direction they are 0 to `y_size-1`. Next, for each available qubit in the platform, its position in the grid is specified: the `id` specifies the particular qubit's index, and `x` and `y` specify its position in the grid, as coordinates in the X and Y direction, respectively. Please note that not every position in the `x_size` by `y_size` grid needs to correspond to a qubit.

Qubits are connected in directed pairs, called edges. Edge indices form a contiguous range starting from 0. Each edge in the topology is given an `id` which denotes its index, and a source (control) and destination (target) qubit index by `src` and `dst`, respectively. This means that there can be edges between the same pair of qubits but in opposite directions. The qubit indices specified here must correspond to available qubits in the platform.

For a full example of this section, please refer to [CC-Light Platform](#).

8.1.4 Section resources

Specify the classical control constraints of the platform. This section is optional. These constraints are used by the resource manager, that on its turn is used by the scheduling and mapping passes. These classical control constraints are described as restrictions on concurrent access to resources of predefined resource types. Specification of these resources affects scheduling and mapping of gates.

The `resources` section specifies zero or more resource types that are predefined by the mandatory platform dependent resource manager. For CC-Light, these resource types are `qubits`, `qwgs`, `meas_units`, `edges`, and `detuned_qubits`. The presence of one in the configuration file indicates that the resource-constrained scheduler should take it into account when trying to schedule operations in parallel, i.e. with overlapping executions. Although their names suggest otherwise, they are just vehicles to configure the scheduler and need not correspond to real resources present in the hardware. This also implies that they can be easily reused for other platforms.

For a full example of this section, including an extensive description of the various resource types, please refer to [CC-Light Platform](#). For a description of their use by the scheduler, please refer to [Scheduling](#).

8.1.5 Section instructions

Specifies the list of primitive gates supported by the platform. Creation of a primitive custom gate takes its parameters from this specification to initialize the gate's attributes.

Examples of a 1-qubit and a 2-qubit instruction are shown below:


```

1  "instructions": {
2      "x q0": {
3          "duration": 40,
4          "latency": 0,
5          "qubits": ["q0"],
6          "matrix": [ [0.0,0.0], [1.0,0.0],
7                      [1.0,0.0], [0.0,0.0]
8                      ],
9          "disable_optimization": false,
10         "type": "mw"
11     },
12     "cnot q2,q0": {
13         "duration": 80,
14         "latency": 0,
15         "qubits": ["q2","q0"],
16         "matrix": [ [0.1,0.0], [0.0,0.0], [0.0,0.0], [0.0,0.0],
17                     [0.0,0.0], [1.0,0.0], [0.0,0.0], [0.0,0.
18 ↪0],
19                     [0.0,0.0], [0.0,0.0], [0.0,0.0], [1.0,0.
20 ↪0],
21                     [0.0,0.0], [0.0,0.0], [1.0,0.0], [0.0,0.
22 ↪0],
23                     ],
24         "disable_optimization": true,
25         "type": "flux"
26     },
27     ...
28 }

```

`x q0` is the name of the instruction which will be used to refer to this instruction inside the OpenQL program. `x` would also be allowed as name. The former defines a specialized gate, the latter defines a generalized gate; please refer to [Quantum Gates](#) for the definitions of these terms and to [Input external representation](#) for the use of these two forms of instruction definitions.

- `duration` specifies the time duration required to complete this instruction.
- `latency`; due to control electronics, it is sometimes required to add a positive or negative latency to an instruction. This can be specified by the `latency` field. This field is divided by cycle time and rounded up to obtain an integer number of cycles. After scheduling is performed, an instruction is shifted back or forth in time depending upon the calculated cycles corresponding to the latency field.
- `qubits` refer to the list of qubit operands.

Note This field has to match the operands in the name of the instruction, if specified there. This is checked. Otherwise there is no use of this field. So there is redundancy here.

- `matrix` specifies the process matrix representing this instruction. If optimization is enabled, this matrix will be used by the optimizer to fuse operations together, as discussed in [Optimization](#). This can be left un-specified if optimization is disabled.
- `disable_optimization` is used to enable/disable optimization of this instruction. Setting `disable_optimization` to `true` will mean that this instruction cannot be compiled away during optimization.

Note This is not implemented. Propose to do so. Then have to define what is exactly means: compiling away is interpreted as the gate with this flag `true` will never be deleted from a circuit once created, nor that the circuit that contains it will be deleted.

- `type` indicates whether the instruction is a microwave (`mw`), flux (`flux`) or readout (`readout`). This is used in CC-Light by the resource manager to select the resources of a gate for scheduling.

8.1.6 Section gate_decomposition

Specifies a list of gates defined by decomposition into primitive gates.

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.

```
1  "gate_decomposition": {
2      "rx180 %0" : ["x %0"],
3      "cnot %0,%1" : ["ry90 %1", "cz %0,%1", "ry90 %1"]
4  }
```

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. OpenQL can also perform Control and Unitary decompositions which are discussed in [Decomposition](#).

8.2 QX Platform

Details of the configuration file for the QX simulator platform. [TBD]

The OpenQL compiler is able to generate a qasm file to interface to QX. This qasm file generation is controlled by option *write_qasm_files*:

- yes: Qasm files are generated before and after most of the passes.
- no: No qasm files are generated.

The qasm files are generated in the default output directory. The qasm file to be used for QX is named by the program name suffixed with .qasm.

Unlike in previous releases, quantumsim is not considered a platform. It is a means to simulate a particular (hardware) platform without timing.

[TBD]

8.3 Quantumsim Platform

The OpenQL compiler is able to generate a python script to interface to quantumsim. This script generation is controlled by option *quantumsim*:

- no: No script to interface to quantumsim is generated.
- yes: A python script is generated to interface with a standard version of quantumsim.
- qsoverlay: A python script is generated to interface with the qsoverlay module on top of quantumsim.

The scripts are generated in the default output directory.

Unlike in previous releases, quantumsim is not considered a platform. It is a means to simulate a particular (hardware) platform.

The quantumsim option is checked in the CC-Light backend in two places:

- just when entering the backend after *decomposition before scheduling*
- just before generating QISA, i.e. after *mapping*, *rcscheduling* and *decomposition after scheduling*.

[TBD]

8.4 CC-Light Platform

The file `hardware_configuration_cc_light.json` available inside the `tests` directory is an example configuration file for the CC-Light platform with 7 qubits.

This file consists of several sections (in arbitrary order) which are described below.

`eqasm_compiler` specifies the backend compiler to be used for this CC-Light platform, which in this case has the name `cc_light_compiler`. The backend compiler is called after the platform independent passes, and calls several private passes by itself. This backend compiler and its passes are described in detail in *Compiler Passes*. One of these is the code generation pass.

```
"eqasm_compiler" : "cc_light_compiler",
```

`hardware_settings` is used to configure various hardware settings of the platform as shown below. These settings affect the scheduling of instructions. Please refer to *Platform* for a full description and an example.

`topology` specifies the mapping of qubit indices to qubit positions in the platform, as well as the mapping of connection indices to connections in the platform. A connection is a directed connection in the platform between a pair of qubits that supports qubit interaction. It is directed to distinguish the control and target qubits of two-qubit gates. In a platform topology's connection graph, qubits are the nodes, and connection are the edges.

Figure Fig. 8.1 shows these numberings in the 7 qubit CC-Light platform.

The `topology` section starts with the specification of the two dimensions of a rectangular qubit grid by specifying `x_size` and `y_size`. The positions of the real qubits of the platform are defined relative to this (artificial) grid. The coordinates in the X direction are 0 to `x_size`-1. In the Y direction they are 0 to `y_size`-1. Next, for each available qubit in the platform, its position in the grid is specified: the `id` specifies the particular qubit's index, and `x` and `y` specify its position in the grid, as coordinates in the X and Y direction, respectively. Please note that not every position in the `x_size` by `y_size` grid needs to correspond to a qubit.

Qubits are connected in directed pairs, called edges. Edge indices form a contiguous range starting from 0. Each edge in the topology is given an `id` which denotes its index, and a source (control) and destination (target) qubit index by `src` and `dst`, respectively. This means that although Edge 0 and Edge 8 are between qubit 0 and qubit 2, they are different as these edges are in opposite directions. The qubit indices specified here must correspond to available qubits in the platform.

```
1  "topology" : {
2      "x_size": 5,
3      "y_size": 3,
4      "qubits":
5      [
6          { "id": 0,  "x": 1, "y": 2 },
7          { "id": 1,  "x": 3, "y": 2 },
8          { "id": 2,  "x": 0, "y": 1 },
9          { "id": 3,  "x": 2, "y": 1 },
10         { "id": 4,  "x": 4, "y": 1 },
11         { "id": 5,  "x": 1, "y": 0 },
12         { "id": 6,  "x": 3, "y": 0 }
13     ],
14     "edges":
15     [
16         { "id": 0,  "src": 2, "dst": 0 },
17         { "id": 1,  "src": 0, "dst": 3 },
```

(continues on next page)

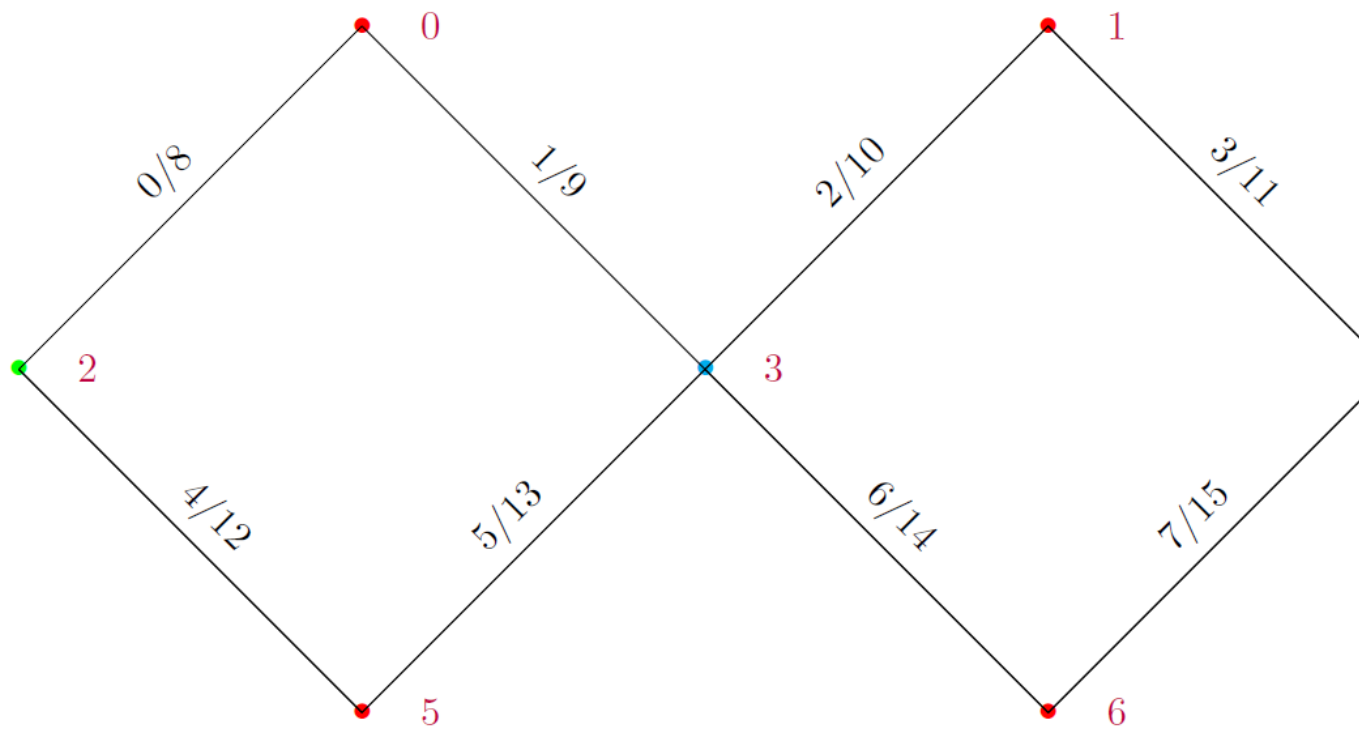


Fig. 8.1: Connection graph with qubit and connection (edge) numbering in the 7 qubits CC-Light Platform

(continued from previous page)

```

18         { "id": 2, "src": 3, "dst": 1 },
19         { "id": 3, "src": 1, "dst": 4 },
20         { "id": 4, "src": 2, "dst": 5 },
21         { "id": 5, "src": 5, "dst": 3 },
22         { "id": 6, "src": 3, "dst": 6 },
23         { "id": 7, "src": 6, "dst": 4 },
24         { "id": 8, "src": 0, "dst": 2 },
25         { "id": 9, "src": 3, "dst": 0 },
26         { "id": 10, "src": 1, "dst": 3 },
27         { "id": 11, "src": 4, "dst": 1 },
28         { "id": 12, "src": 5, "dst": 2 },
29         { "id": 13, "src": 3, "dst": 5 },
30         { "id": 14, "src": 6, "dst": 3 },
31         { "id": 15, "src": 4, "dst": 6 }
32     ],
33 },

```

These mappings are used in:

- the QISA, the instruction set of the platform, notably in the instructions that set the masks stored in the mask registers that are used in the instructions of two-qubit gates to address the operands.
- the mapper pass that maps virtual qubit indices to real qubit indices. It is described in detail in [Mapping](#).
- the postdecomposition pass that maps two-qubit flux instructions to sets of one-qubit flux instructions.

`resources` is the section that is used to specify/configure various resource types available in the platform as discussed below. Specification of these resource types affects scheduling and mapping of gates. The configuration of the various resource types in [hardware_configuration_cc_light.json](#) assumes that the CC-Light architecture has the following relations between devices, connections, qubits and operations:

Device Name	DIO connector	Target qubits	Operation Type
UHFQC-0	DIO1	0, 2, 3, 5, 6	measurement
UHFQC-1	DIO2	1, 4	measurement
AWG-8 0, channel 0~6	DIO3	0~6	flux
AWG-8 1, channel 0	DIO4	0,1	microwave
AWG-8 1, channel 1	DIO4	5,6	microwave
AWG-8 2, channel 0	DIO5	2,3,4	microwave
VSM	—	0~6	microwave masking

The `resources` section specifies zero or more resource types. Each of these must be predefined by the platform's resource manager. For CC-Light, these resource types are `qubits`, `qwgs`, `meas_units`, `edges` and `detuned_qubits`. The presence of one in the configuration file indicates that the resource-constrained scheduler should take it into account when trying to schedule operations in parallel, i.e. with overlapping executions; absence of one in the configuration file thus indicates that this resource is ignored by the scheduler. Although their names suggest otherwise, they are just vehicles to configure the scheduler and need not correspond to real resources present in the hardware.

`qubits`: That one qubit can only be involved in one operation at each particular cycle, is specified by the `qubits` resource type, as shown below. `count` needs to be at least the number of available qubits.

```

1     "qubits":
2     {
3         "count": 7
4     },

```

So, when this resource type is included in the configuration in this way, it will guarantee that the resource-constrained scheduler will never schedule two operations in parallel when these share a qubit index in the range of 0 to count-1 as operand.

`qwgs`: This resource type specifies, when configured, several sets of qubit indices. For each set it specifies that when one of the qubits in the set is in use in a particular cycle by an instruction of ‘mw’ type (single-qubit rotation gates usually), that when one of the other qubits in the set is in use by an instruction of ‘mw’ type, that instruction must be doing the same operation. In CC-light, this models QWG wave generators that only can generate one type of wave at the same time, and in which each wave generator is connected through a switch to a subset of the qubits.

```

1  "qwgs" :
2  {
3      "count": 3,
4      "connection_map":
5      {
6          "0" : [0, 1],
7          "1" : [2, 3, 4],
8          "2" : [5, 6]
9      }
10 },

```

The number of sets (waveform generators) is specified by the `count` field. In the `connection_map` it is specified which waveform generator is connected to which qubits. Each qubit that can be used by an instruction of ‘mw’ type, should be specified at most once in the combination of sets of connected qubits. For instance, the line with "0" specifies that `qwgs 0` is connected to qubits 0 and 1. This is based on the AWG-8 1, channel 0 entry in Table 8.4. This information is utilized by the scheduler to perform resource-constraint aware scheduling of gates.

`meas_units`: This resource type is similar to `qwgs`; the difference is that it is not constraining on the operations to be equal but on the start cycle of measurement to be equal. It specifies, when configured, several sets of qubit indices. For each set it specifies that when one of the qubits in the set is in use in a particular cycle by an instruction of ‘readout’ type (measurement gates usually) that when one of the other qubits in the set is in use by an instruction of ‘readout’ type the latter must also have started in that cycle. In CC-light, this models measurement units that each can only measure multiple qubits at the same time when the measurements of those qubits start in the same cycle.

There are `count` number of sets (measurement units). For each measurement unit it is described which set of qubits it controls. Each qubit that can be used by an instruction of ‘readout’ type, should be specified at most once in the combination of sets of connected qubits.

```

1  "meas_units" :
2  {
3      "count": 2,
4      "connection_map":
5      {
6          "0" : [0, 2, 3, 5, 6],
7          "1" : [1, 4]
8      }
9  },

```

`edges`: This resource type specifies, when present, for each directed qubit pair corresponding to a directed connection in the platform (edge), which set of other edges cannot execute a two-qubit gate in parallel.

Two-qubit flux gates (instructions of `flux` type) are controlled by qubit-selective frequency detuning. Frequency-detuning may cause neighbor qubits (qubits connected by an edge) to inadvertently engage in a two-qubit flux gate as well. This happens when two connected qubits are both executing a two-qubit flux gate. Therefore, for each edge executing a two-qubit gate, certain other edges should not execute a two-qubit gate.

Edges and the constraints imposed by these edges are specified in the `edges` section. `count` specifies at least the number of edges between qubits in the platform. `connection_map` specifies connections. For example, the entry

with “0” specifies for Edge 0 a constraint on Edge 2 and Edge 10. This means, if Edge 0 is in use by a two-qubit flux gate, a two-qubit flux gate on Edge 2 and Edge 10 will not be scheduled, until the one on Edge 0 completes.

When `edges` is present as a resource type, each edge of the platform must appear in the `connection_map`. Providing an empty list for an edge in the `connection_map` will result in not applying any edge constraint during scheduling.

```

1  "edges":
2  {
3    "count": 16,
4    "connection_map":
5    {
6      "0": [2, 10],
7      "1": [3, 11],
8      "2": [0, 8],
9      "3": [1, 9],
10     "4": [6, 14],
11     "5": [7, 15],
12     "6": [4, 12],
13     "7": [5, 13],
14     "8": [2, 10],
15     "9": [3, 11],
16     "10": [0, 8],
17     "11": [1, 9],
18     "12": [6, 14],
19     "13": [7, 15],
20     "14": [4, 12],
21     "15": [5, 13]
22   }
23 },

```

`detuned_qubits`: Constraints on executing two-qubit gates in parallel to other gates, are specified in this `detuned_qubits` section, when present. For each edge, the set of qubits is specified that cannot execute a gate when on the particular edge a two-qubit gate is executed; at the same time, this resource type specifies implicitly for each qubit when it would be executing a gate, on which edges a two-qubit gate cannot execute in parallel.

There are at least `count` number of qubits involved. When `detuned_qubits` is present as a resource type, each edge of the platform must appear in the `connection_map`. Providing an empty set of qubits for an edge in the `connection_map` will result in not applying the `detuned_qubits` constraint related to this edge during scheduling. Not all qubits need to be involved in this type of constraint with some edge. In the example below, Qubit 0 and Qubit 1 are examples of qubits executing a gate on which can be in parallel to executing a two-qubit gate on any pair of qubits.

A two-qubit flux gate lowers the frequency of its source qubit to get near the frequency of its target qubit. Any two qubits which have near frequencies execute a two-qubit flux gate. To prevent any neighbor qubit of the source qubit that has the same frequency as the target qubit to interact as well, those neighbors must have their frequency detuned (lowered out of the way). A detuned qubit cannot execute a single-qubit rotation (an instruction of ‘mw’ type).

```

1  "detuned_qubits":
2  {
3    "count": 7,
4    "connection_map":
5    {
6      "0": [3],
7      "1": [2],
8      "2": [4],
9      "3": [3],
10     "4": [],

```

(continues on next page)

(continued from previous page)

```

11     "5": [6],
12     "6": [5],
13     "7": [],
14     "8": [3],
15     "9": [2],
16     "10": [4],
17     "11": [3],
18     "12": [],
19     "13": [6],
20     "14": [5],
21     "15": []
22 }
23 }

```

instructions: Instructions can be specified/configured in the `instructions` section. Examples of a 1-qubit and a 2-qubit instruction are shown below:

```

1  "instructions": {
2      "x q0": {
3          "duration": 40,
4          "latency": 0,
5          "qubits": ["q0"],
6          "matrix": [ [0.0,0.0], [1.0,0.0],
7                      [1.0,0.0], [0.0,0.0]
8                      ],
9          "disable_optimization": false,
10         "type": "mw",
11         "cc_light_instr_type": "single_qubit_gate",
12         "cc_light_instr": "x",
13         "cc_light_codeword": 60,
14         "cc_light_opcode": 6
15     },
16     "cnot q2,q0": {
17         "duration": 80,
18         "latency": 0,
19         "qubits": ["q2","q0"],
20         "matrix": [ [0.1,0.0], [0.0,0.0], [0.0,0.0], [0.0,0.0],
21                     [0.0,0.0], [1.0,0.0], [0.0,0.0], [0.0,0.
22 ↪0],
23                     [0.0,0.0], [0.0,0.0], [0.0,0.0], [1.0,0.
24 ↪0],
25                     [0.0,0.0], [0.0,0.0], [1.0,0.0], [0.0,0.
26 ↪0],
27                     [0.0,0.0], [0.0,0.0], [1.0,0.0], [0.0,0.
28 ↪0],
29                     ],
30         "disable_optimization": true,
31         "type": "flux",
32         "cc_light_instr_type": "two_qubit_gate",
33         "cc_light_instr": "cnot",
34         "cc_light_right_codeword": 127,
35         "cc_light_left_codeword": 135,
36         "cc_light_opcode": 128
37     },
38     ...
39 }

```

Please refer to [Platform](#) for a description of the CC-Light independent attributes. The CC-Light dependent attributes are:

`cc_light_instr_type` is used to specify the type of instruction based on the number of expected qubits. Please refer to [Scheduling](#) for its use by the rcscheduler.

`cc_light_instr` specifies the name of this instruction used in CC-Light architecture. This name is used in the generation of the output code and in the implementation of the checking of the `qwg` resource. Please refer to [Scheduling](#) for its use by the rcscheduler.

`cc_light_codeword`, `cc_light_right_codeword`, `cc_light_left_codeword` and `cc_light_opcode` are used in the generation of the control store file for CC-Light platform. For single qubit instructions, `cc_light_codeword` refers to the codeword to be used for this instruction. Recall that the quantum pipeline contains a VLIW front end with two VLIW lanes, each lane processing one quantum operation. `cc_light_right_codeword` and `cc_light_left_codeword` are used to specify the codewords used for the left and right operation in two-qubit instruction. `cc_light_opcode` specifies the opcode used for this instruction.

Warning: At the moment, generation of the control-store file is disabled in the compiler as this was not being used in experiments.

`gate_decomposition` Gate decompositions can also be specified in the configuration file in the `gate_decomposition` section. Please refer to [Platform](#) for a description and full example of this section.

8.5 Central Controller Platform Configuration

8.5.1 CC configuration file

This section describes the JSON configuration file format for OpenQL in conjunction with the Central Controller (CC) backend. Note that for the CC - contrary to the CC-light - the final hardware output is entirely *determined* by the contents of the configuration file, there is no built-in knowledge of instrument connectivity or codeword organization.

The CC configuration file consists of several sections described below.

To select the CC backend, the following is required:

```
"eqasm_compiler" : "eqasm_backend_cc",
```

`resources` unused for the CC backend, section may be empty

`topology` unused for the CC backend, section may be empty

`alias` unused by OpenQL

`hardware_settings` is used to configure various hardware settings of the platform as shown below. These settings affect the scheduling of instructions. Please refer to [Platform](#) for a full description and an example.

The following settings are not used by the CC backend:

- `hardware_settings/mw_mw_buffer`
- `hardware_settings/mw_flux_buffer`
- `hardware_settings/mw_readout_buffer`
- `hardware_settings/flux_mw_buffer`
- `hardware_settings/flux_flux_buffer`
- `hardware_settings/flux_readout_buffer`

- hardware_settings/readout_mw_buffer
- hardware_settings/readout_flux_buffer
- hardware_settings/readout_readout_buffer

All settings related to the CC backend are in section `hardware_settings/eqasm_backend_cc` of the configuration file. This section is divided into several subsections as shown below.

Instrument definitions

Subsection `instrument_definitions` defines immutable properties of instruments, i.e. independent of the actual control setup:

```

1 "instrument_definitions": {
2   "qutech-qwg": {
3     "channels": 4,
4     "control_group_sizes": [1, 4],
5   },
6   "zi-hdawg": {
7     "channels": 8,
8     "control_group_sizes": [1, 2, 4, 8], // NB: size=1 needs special treatment of
↳ waveforms because one AWG unit drives 2 channels
9   },
10  "qutech-vsm": {
11    "channels": 32,
12    "control_group_sizes": [1],
13  },
14  "zi-uhfqa": {
15    "channels": 9,
16    "control_group_sizes": [1],
17  }
18 }, // instrument_definitions

```

Where:

- `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

Subsection `control_modes` defines modes to control instruments. These define which bits are used to control groups of channels and/or get back measurement results:

```

1 "control_modes": {
2   "awg8-mw-vsm-hack": { // ZI_HDAWG8.py::cfg_codeword_
↳ protocol() == 'microwave'. Old hack to skip DIO[8]
3     "control_bits": [
4       [7, 6, 5, 4, 3, 2, 1, 0], // group 0
5       [16, 15, 14, 13, 12, 11, 10, 9] // group 1
6     ],
7     "trigger_bits": [31]
8   },
9   "awg8-mw-vsm": { // the way the mode above should have
↳ been

```

(continues on next page)

(continued from previous page)

```

10     "control_bits": [
11         [7,6,5,4,3,2,1,0],           // group 0
12         [15,14,13,12,11,10,9,8]      // group 1
13     ],
14     "trigger_bits": [31]
15 },
16     "awg8-mw-direct-iq": {           // just I&Q to generate microwave
→without VSM. HDAWG8: "new_novsm_microwave"
17         "control_bits": [
18             [6,5,4,3,2,1,0],         // group 0
19             [13,12,11,10,9,8,7],     // group 1
20             [22,21,20,19,18,17,16],  // group 2. NB: starts at bit 16 so
→twin-QWG can also support it
21             [29,28,27,26,25,24,23]   // group 4
22         ],
23         "trigger_bits": [31]
24     },
25     "awg8-flux": {                  // ZI_HDAWG8.py::cfg_codeword_
→protocol() == 'flux'
26         // NB: please note that internally one AWG unit handles 2 channels, which
→requires special handling of the waveforms
27         "control_bits": [
28             [2,1,0],                 // group 0
29             [5,4,3],
30             [8,7,6],
31             [11,10,9],
32             [18,17,16],              // group 4. NB: starts at bit 16 so
→twin-QWG can also support it
33             [21,20,19],
34             [24,23,22],
35             [27,26,25]              // group 7
36         ],
37         "trigger_bits": [31]
38     },
39     "awg8-flux-vector-8": {         // single code word for 8 flux
→channels.
40         "control_bits": [
41             [7,6,5,4,3,2,1,0]
42         ],
43         "trigger_bits": [31]
44     },
45     "uhfqa-9ch": {
46         "control_bits": [[17],[18],[19],[20],[21],[22],[23],[24],[25]], //
→group[0:8]
47         "trigger_bits": [16],
48         "result_bits": [[1],[2],[3],[4],[5],[6],[7],[8],[9]], //
→group[0:8]
49         "data_valid_bits": [0]
50     },
51     "vsm-32ch":{
52         "control_bits": [
53             [0],[1],[2],[3],[4],[5],[6],[7],           // group[0:7]
54             [8],[9],[10],[11],[12],[13],[14],[15],      // group[8:15]
55             [16],[17],[18],[19],[20],[21],[22],[23],    // group[16:23]
56             [24],[25],[26],[27],[28],[28],[30],[31]     // group[24:31]
57         ],
58         "trigger_bits": [] // no trigger

```

(continues on next page)

(continued from previous page)

```

59     }
60 }, // control_modes

```

Where:

- <key> is a name which can be referred to from key ‘instruments/[]/ref_control_mode’
- control_bits defines G groups of B bits, with:
 - G determines which the ‘instrument_definitions/<key>/control_group_sizes’ used
 - B is an ordered list of bits (MSB to LSB) used for the code word
- trigger_bits vector of bits used to trigger the instrument. Must either be size 1 (common trigger) or size G (separate trigger per group)

FIXME: examples * result_bits reserved for future use * data_valid_bits reserved for future use

Signals

Subsection signals provides a signal library that gate definitions can refer to:

```

1  "signals": {
2      "single-qubit-mw": [
3          { "type": "mw",
4            "operand_idx": 0,
5            "value": [
6                "{gateName}-{instrumentName}:{instrumentGroup}-gi",
7                "{gateName}-{instrumentName}:{instrumentGroup}-gq",
8                "{gateName}-{instrumentName}:{instrumentGroup}-di",
9                "{gateName}-{instrumentName}:{instrumentGroup}-dq"
10           ]
11        },
12        { "type": "switch",
13          "operand_idx": 0,
14          "value": ["dummy"] // NB: no actual_
15        }
16    ],
17    "two-qubit-flux": [
18        { "type": "flux",
19          "operand_idx": 0, // control
20          "value": ["flux-0-{qubit}"]
21        },
22        { "type": "flux",
23          "operand_idx": 1, // target
24          "value": ["flux-1-{qubit}"]
25        }
26    ]
27 }, // signals

```

Where:

- <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:

Instruments

Subsection `instruments` defines instruments used in this setup, their configuration and connectivity.

```

1  "instruments": [
2      // readout.
3      {
4          "name": "ro_0",
5          "qubits": [[6], [11], [], [], [], [], [], [], []],
6          "signal_type": "measure",
7          "ref_instrument_definition": "zi-uhfqa",
8          "ref_control_mode": "uhfqa-9ch",
9          "controller": {
10             "name": "cc",
11             "slot": 0,
12             "io_module": "CC-CONN-DIO"
13         }
14     },
15     // ...
16
17     // microwave.
18     {
19         "name": "mw_0",
20         "qubits": [                                // data qubits:
21             [2, 8, 14],                            // [freq L]
22             [1, 4, 6, 10, 12, 15]                  // [freq H]
23         ],
24         "signal_type": "mw",
25         "ref_instrument_definition": "zi-hdawg",
26         "ref_control_mode": "awg8-mw-vsm-hack",
27         "controller": {
28             "name": "cc",
29             "slot": 3,
30             "io_module": "CC-CONN-DIO-DIFF"
31         }
32     },
33     // ...
34
35     // VSM
36     {
37         "name": "vsm_0",
38         "qubits": [
39             [2], [8], [14], [], [], [], [], [], // [freq L]
40             [1], [4], [6], [10], [12], [15], [], // [freq H]
41             [0], [5], [9], [13], [], [], [], // [freq Mg]
42             [3], [7], [11], [16], [], [], [], // [freq My]
43         ],
44         "signal_type": "switch",
45         "ref_instrument_definition": "qutech-vsm",

```

(continues on next page)

(continued from previous page)

```

46     "ref_control_mode": "vsm-32ch",
47     "controller": {
48         "name": "cc",
49         "slot": 5,
50         "io_module": "cc-conn-vsm"
51     }
52 },
53
54 // flux
55 {
56     "name": "flux_0",
57     "qubits": [[0], [1], [2], [3], [4], [5], [6], [7]],
58     "signal_type": "flux",
59     "ref_instrument_definition": "zi-hdawg",
60     "ref_control_mode": "awg8-flux",
61     "controller": {
62         "name": "cc",
63         "slot": 6,
64         "io_module": "CC-CONN-DIO-DIFF"
65     }
66 },
67 // ...
68 ] // instruments

```

Where:

- name a friendly name for the instrument
- `ref_instrument_definition` selects record under ‘instrument_definitions’, which must exist or an error is raised
- `ref_control_mode` selects 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
- `controller/slot` the slot number of the CC this instrument is connected to
- `controller/name` reserved for future use
- `controller/io_module` reserved for future use

Additions to section ‘instructions’

The CC backend extends section “instructions/<key>” with a subsection “cc” as shown in the example below:

```

1  "ryl80": {
2      "duration": 20,
3      "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
4      "type": "mw",
5      "cc_light_instr": "y",
6      "cc": {
7          "ref_signal": "single-qubit-mw",
8          "static_codeword_override": 2

```

(continues on next page)

(continued from previous page)

```

9      }
10    },
11    "cz_park": {
12      "duration": 40,
13      "matrix": [ [0.0,1.0], [1.0,0.0], [1.0,0.0], [0.0,0.0] ],
14      "type": "flux",
15      "cc_light_instr": "cz",
16      "cc": {
17        "signal": [
18          { "type": "flux",
19            "operand_idx": 0,
20            "value": ["flux-0-{qubit}"] // control
21          },
22          { "type": "flux",
23            "operand_idx": 1,
24            "value": ["flux-1-{qubit}"] // target
25          },
26          { "type": "flux",
27            "operand_idx": 2,
28            "value": ["park_cz-{qubit}"] // park
29          }
30        ],
31        "static_codeword_override": 1
32      }
33    }

```

Where:

- `cc/ref_signal` points to a signal definition in `hardware_settings/eqasm_backend_cc/signals`, which must exist or an error is raised
- `cc/signal` defines a signal in place, in an identical fashion as `hardware_settings/eqasm_backend_cc/signals`
- `cc/static_codeword_override` provides a user defined codeword for this instruction. Currently, this key is compulsory, but in the future, codewords will be assigned automatically to make better use of limited codeword space

The following standard OpenQL fields are used:

- `<key>` name for the instruction. The following syntaxes can be used for instruction names:
 - “<name>”
 - “<name><qubits>”
- `duration` duration in [ns]
- `matrix` the process matrix. Required, but only used if optimization is enabled
- `type` instruction type used by scheduler, one of the builtin names “mw”, “flux” or “measure”. Has no relation with signal type definition of CC backend, even though we use the same string values there
- `cc_light_instr` required by scheduler.
- `latency` optional instruction latency in [ns], used by scheduler
- `qubits` optional

The following fields in ‘instructions’ are not used by the CC backend:

- `cc_light_instr_type` FIXME: is used in scheduler.h

- `cc_light_cond`
- `cc_light_opcode`
- `cc_light_codeword`
- `cc_light_left_codeword`
- `cc_light_right_codeword`
- `disable_optimization` not implemented in OpenQL

Converting quantum gates to instrument codewords

FIXME: TBW

8.5.2 Compiler options

FIXME: TBW

8.5.3 CC backend output files

FIXME: TBW: `.vqlasm`, `.vcd`

8.5.4 Standard OpenQL features

FIXME: just refer to relevant section. Kept here until we're sure this has been absorbed elsewhere

Parametrized gate-decomposition

Parametrized gate decompositions can be specified in `gate_decomposition` section, as shown below:

```
"rx180 %0" : ["x %0"]
```

Based on this, `k.gate('rx180', 3)` will be decomposed to `x(q3)`. Similarly, multi-qubit gate-decompositions can be specified as:

```
"cnot %0,%1" : ["ry90 %0", "cz %0,%1", "ry90 %1"]
```

Specialized gate-decomposition

Specialized gate decompositions can be specified in `gate_decomposition` section, as shown below:

```
"rx180 q0" : ["x q0"] "cz_park q0,q1" : ["cz q0,q1", "park q3"]
```

8.6 CBox Platform

Details of configuration file for CBox hardware platform. [TBD]

Compiler

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 program has the ability to configure its own pass sequence using the [Compiler API](#). 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). Currently, 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)

CHAPTER 10

Compiler Passes

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, prefix):**
Identical to `report_qasm` but the IR itself is not written but a summary of it, e.g. the number of kernels, the number 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.

10.1 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 contiguous sequences of quantum gates that are equivalent to identity (within some small epsilon which currently is 10^{-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 *Platform*.

10.2 Decomposition

Decomposition of gates [TBD]

10.2.1 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

10.2.2 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
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 $2^n \times 2^n$, 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>

10.2.3 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

10.2.4 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

10.2.5 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

10.3 Optimization

Optimization of circuits [TBD]

10.3.1 Optimize

attempts to find contiguous 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

10.3.2 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

10.4 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.

10.4.1 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 [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 [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.

10.4.2 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](#).

10.4.3 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.

10.4.4 Function

Scheduling of a circuit starts with creation of the dependence graph; see *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 *CC-Light Platform* 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 `qwg`s 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.

10.4.5 Scheduling for software platforms

- Scheduling for `qx`
- Scheduling for `quantumsim`

10.4.6 Scheduling for hardware platforms

- Scheduling for CC-Light platform
- Scheduling for CC platform
- Scheduling for CBox platform

This section will document how OpenQL schedules gates for the CC-Light Platform. It will also highlight how constraints mentioned in *CC-Light Platform* affect scheduling.

This section will document how OpenQL schedules gates for the CC Platform. It will also highlighted how constraints mentioned in the platform configuration file affect scheduling.

10.5 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.

10.5.1 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 swaps and moves inserted.
- `nmovesadded` Number of moves 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 ($|0\rangle$)
 - * `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.

10.5.2 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.

10.5.3 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 [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. $|0\rangle$) 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 successful 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 successful 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 *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 *Dependence Graph and Look-Ahead, Which Gate(s) To Map Next*.
- To map a non-NN two-qubit gate, various routing alternatives, to be implemented by swap/move sequences, are generated: see *Generating Routing Alternatives*.
- Depending on the metric chosen, the alternatives are evaluated: see *Comparing Alternatives, Which Metric To Use*.
- When minimizing circuit latency extension, ILP is maximized by maintaining a scheduled circuit representation: see *Look-Back, Maximize Instruction-Level Parallelism By Scheduling*.
- 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 *Looking Farther Ahead, Recurse To Find Best Alternative*.
- Finally, the evaluations of the alternatives are compared, the best one selected and the two-qubit gate routed and mapped: see *Deciding For The Best, Committing To 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 Fibonacci 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 shortest 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 ($|0\rangle$ 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.

- `maprepinitstate`: Does a `prepz` initialize the state, i.e. leave the state of a qubit in the $|0\rangle$ 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 suboptimality 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. Comparison 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 [Look-Back, Maximize Instruction-Level Parallelism By Scheduling](#) for the details of this scheduling. And then map the two-qubit gate; see [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.

10.5.4 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 hadamards 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 `cnots`; those `cnots` 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 `cnots`); 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.
 - *ancillary 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 $|+\rangle$ 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 `swaps` and `moves` 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 *edges*. 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 11

Change Log

All notable changes to this project will be documented in this file. This project adheres to [Semantic Versioning](#).

11.1 [next] - [TBD]

11.1.1 Added

- interface (C++ and Python) to compile cQASM 1.0

11.1.2 Changed

- CC backend:
 - renamed JSON field “signal_ref” to “ref_signal”
 - renamed JSON field “ref_signals_type” to “signal_type”
 - added option for new seq_bar semantics (cc firmware from 20191219 onwards)
 - improved reporting on JSON semantic errors
 - implemented option to output scheduled QASM files

11.1.3 Removed

11.1.4 Fixed

- changed register used for FOR loop, so it doesn't clash with delay setting
- fixed documentation for python setup and running tests

11.2 [0.8.0] - [2019-10-31]

11.2.1 Added

- support for CC backend

11.2.2 Changed

11.2.3 Removed

11.2.4 Fixed

- fixed issue with duplicate kernel names
- updated json library to fix osx builds

11.3 [0.7.1] - [2019-09-02]

11.3.1 Added

11.3.2 Changed

- re-factored folders

11.3.3 Removed

11.3.4 Fixed

- fixed issue with correct python library picking on tud win systems

11.4 [0.7.0] - [2019-06-03]

11.4.1 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)

11.4.2 Changed

- openql is now public
- improved resource-constrained scheduling
- sweep point array is now optional
- support for barrier/wait on all qubits

11.4.3 Removed

- set_sweep_points(sweep_points list, num of sweep points)

11.4.4 Fixed

- resource-constrained qasm is generated by same scheduler for cc-light as is used to generate qisa
- Illegal parameter in gate_decomposition

11.5 [0.6] - [2018-10-29]

11.5.1 Added

- generated qasm code conforms to cQASM v1.0 specification
- added libqasm to pytest to test conformance of generated qasm

11.5.2 Changed

- ALAP scheduler is the default option (Issue #193)
- compiling an empty program raises error (Issue #164)

11.5.3 Removed

11.5.4 Fixed

- tests are added to test option setting/getting (Issue #190)

11.6 [0.5.5] - [2018-10-25]

11.6.1 Added

11.6.2 Changed

- simplified interface of Program.set_sweep_points (Issue #184)

11.6.3 Removed

11.6.4 Fixed

- instruction ordering to generate consistent qisa (Issue #190)
- stateful behaviour in OpenQL (Issue #171)

11.7 [0.5.4] - [2018-10-17]

11.7.1 Added

11.7.2 Changed

11.7.3 Removed

11.7.4 Fixed

- qubit ordering in SMIS and SMIT instructions

11.8 [0.5.3] - [2018-10-11]

11.8.1 Added

- added detuning constraints for cclight

11.8.2 Changed

11.8.3 Removed

11.8.4 Fixed

- alap scheduling for cclight

11.9 [0.5.2] - [2018-10-10]

11.9.1 Added

11.9.2 Changed

11.9.3 Removed

11.9.4 Fixed

- wrong target qubits in the configuration files

- Jenkins test build profile to test against assembler

11.10 [0.5.1] - [2018-09-12]

11.10.1 Added

- API to obtain version number

11.10.2 Changed

11.10.3 Removed

11.10.4 Fixed

- qisa format (removed comma)

11.11 [0.5] - [2018-06-26]

11.11.1 Added

- support for classical instructions
- support for flow control (selection and repetition)
- classical register manager implementation

11.11.2 Changed

- measure instruction updated to support classical target register
- kernels are not any more fused to generate a single qisa program

11.11.3 Removed

- kernel does not receive iteration count, deprecated in favor of for-loop

11.11.4 Fixed

- qisa format (pre-interval syntax updated)

11.12 [0.4.1] - [2018-05-31]

11.12.1 Added

-

11.12.2 Changed

-

11.12.3 Removed

-

11.12.4 Fixed

- getting started example

11.13 [0.4] - [2018-05-19]

11.13.1 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
- verbosity levels

11.13.2 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

11.13.3 Removed

- qisa-as is not a part of openql
- prog.compile() does not get optimiz/schedule/verbose options

11.13.4 Fixed

- static iteration count for scheduled qasm
- rotation angle printing

11.14 [0.3] - [2017-10-24]

11.14.1 Added

- CCLight eQASM compiler
- unittests using qisa-as
- simplified gate decompositions
- wait/barrier instructions

11.14.2 Changed

-

11.14.3 Removed

-

11.14.4 Fixed

- varying prepz duration
- M_PI issue in windows install

11.15 [0.2] - [2017-08-18]

11.15.1 Added

- CBox eQASM compiler
- Python and C++ interface
- Configuration file specification
- trace support for qumis code
- cmake based builds

11.15.2 Changed

-

11.15.3 Removed

-

11.15.4 Fixed

CHAPTER 12

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 .. code-block:: guess
 - * 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 recipes 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
- [Jeroen van Straten](#)
 - tutorial on DQCsims + OpenQL interoperability
 - doxygen documentation

OpenQL is a C++/Python framework for high-level quantum programming. The framework provides a compiler for compiling and optimizing quantum code. The compiler produces the intermediate quantum assembly language in cQASM (Common QASM) and the compiled eQASM (executable QASM) for various target platforms. While the eQASM is platform-specific, the quantum assembly code (QASM) is hardware-agnostic and can be simulated on the QX simulator.

Functions

<code>get_option(option_name)</code>	Returns value of any of the following OpenQL options:
<code>get_version()</code>	Returns OpenQL version
<code>print_options()</code>	Prints a list of available OpenQL options with their values.
<code>set_option(option_name, option_value)</code>	Sets any of the following OpenQL options:

Classes

<code>CReg()</code>	Classical register class.
<code>Kernel(*args)</code>	Kernel class which contains various quantum instructions.
<code>Operation(*args)</code>	Operation class representing classical operations.
<code>Platform(*args)</code>	Platform class specifying the target platform to be used for compilation.
<code>Program(*args)</code>	Program class which contains one or more kernels.
<code>Compiler(name)</code>	Compiler class which contains one or more compiler passes.

```
class openql.openql.CReg
    Classical register class.
```

```
class openql.openql.Compiler(name)
```

Compiler class which contains one or more compiler passes.

add_pass (*realPassName*)

Adds a compiler pass under its real name

Parameters **arg1** (*str*) – name of the real pass to be added.

add_pass_alias (*realPassName*, *symbolicPassName*)

Adds a compiler pass under an alias name

Parameters

- **arg1** (*str*) – name of the real pass to be added.
- **arg2** (*str*) – alias name of the pass to be added.

compile (*program*)

Compiles the program

Parameters **arg1** (*Program*) – program object to be compiled.

set_pass_option (*passName*, *optionName*, *optionValue*)

Sets a compiler pass option

Parameters

- **arg1** (*str*) – name (real or alias) of the compiler pass to be added.
- **arg2** (*str*) – option name of the option to be configured.
- **arg3** (*str*) – value of the option.

class openql.openql.**Kernel** (*args)

Kernel class which contains various quantum instructions.

barrier (*args)

inserts explicit barrier on specified qubits.

wait with duration '0' is also equivalent to applying barrier on specified list of qubits. If no qubits are specified, then barrier is applied on all the qubits.

Parameters **arg1** ([]) – list of qubits

classical (*args)

adds classical operation kernel.

Parameters

- **arg1** (*CReg*) – destination register for classical operation.
- **arg2** (*Operation*) – classical operation.

clifford (*id*, *q0*)

Applies clifford operation of the specified id on the qubit.

The ids and the corresponding operations are:

id	Operations
0	['I']
1	['Y90', 'X90']
2	['mX90', 'mY90']
3	['X180']
4	['mY90', 'mX90']
5	['X90', 'mY90']
6	['Y180']
7	['mY90', 'X90']
8	['X90', 'Y90']
9	['X180', 'Y180']
10	['Y90', 'mX90']
11	['mX90', 'Y90']
12	['Y90', 'X180']
13	['mX90']
14	['X90', 'mY90', 'mX90']
15	['mY90']
16	['X90']
17	['X90', 'Y90', 'X90']
18	['mY90', 'X180']
19	['X90', 'Y180']
20	['X90', 'mY90', 'X90']
21	['Y90']
22	['mX90', 'Y180']
23	['X90', 'Y90', 'mX90']

Parameters

- **arg1** (*int*) – clifford operation id
- **arg2** (*int*) – target qubit

cnot (*q0, q1*)

Applies controlled-not operation.

Parameters

- **arg1** (*int*) – control qubit
- **arg2** (*int*) – target qubit

conjugate (*k*)

generates conjugate version of the kernel from the input kernel.

Parameters **arg1** (*q1 :: Kernel*) – input kernel. Except measure, Kernel to be conjugated.

Returns

Return type None

controlled (*k, control_qubits, ancilla_qubits*)

generates controlled version of the kernel from the input kernel.

Parameters

- **arg1** (*ql : Kernel*) – input kernel. Except measure, Kernel to be controlled may contain any of the default gates as well custom gates which are not specialized for a specific qubits.
- **arg2** (*[]*) – list of control qubits.
- **arg3** (*[]*) – list of ancilla qubits. Number of ancilla qubits should be equal to number of control qubits.

Returns**Return type** None**cphase** (*q0, q1*)

Applies controlled-phase operation.

Parameters

- **arg1** (*int*) – control qubit
- **arg2** (*int*) – target qubit

display ()

inserts QX display instruction (so QX specific).

Parameters None –**Returns****Return type** None**gate** (**args*)

adds unitary to kernel.

Parameters

- **arg1** (*Unitary*) – unitary matrix
- **arg2** (*[]*) – list of qubits

get_custom_instructions ()

Returns list of available custom instructions.

Parameters None –**Returns** List of available custom instructions**Return type** []**hadamard** (*q0*)

Applies hadamard on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit**identity** (*q0*)

Applies identity on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit**measure** (*q0*)

measures input qubit.

Parameters **arg1** (*int*) – input qubit

mr_x90 (*q0*)

Applies mr_x90 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

rx180 (*q0*)

Applies rx180 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

rx90 (*q0*)

Applies rx90 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

ry180 (*q0*)

Applies ry180 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

s (*q0*)

Applies x on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

sdag (*q0*)

Applies sdag on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

toffoli (*q0, q1, q2*)

Applies controlled-controlled-not operation.

Parameters

- **arg1** (*int*) – control qubit
- **arg2** (*int*) – control qubit
- **arg3** (*int*) – target qubit

wait (*qubits, duration*)

inserts explicit wait of specified duration on specified qubits.

wait with duration '0' is equivalent to barrier on specified list of qubits. If no qubits are specified, then wait/barrier is applied on all the qubits.

Parameters

- **arg1** (*[]*) – list of qubits
- **arg2** (*int*) – duration in ns

y (*q0*)

Applies y on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

z (*q0*)

Applies z on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

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

Operation class representing classical operations.

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

Platform class specifying the target platform to be used for compilation.

get_qubit_number ()

returns number of qubits in the platform.

Parameters **None** –

Returns number of qubits

Return type int

class openql.openql.**Program** (*args)

Program class which contains one or more kernels.

add_do_while (*args)

Adds specified sub-program to a program which will be repeatedly executed while specified condition is true.

Parameters

- **arg1** (*Program*) – program to be executed repeatedly
- **arg2** (*Operation*) – classical relational operation (<, >, <=, >=, ==, !=)

add_for (*args)

Adds specified sub-program to a program which will be executed for specified iterations.

Parameters

- **arg1** (*Program*) – sub-program to be executed repeatedly
- **arg2** (*int*) – iteration count

add_if (*args)

Adds specified sub-program to a program which will be executed if specified condition is true. This allows nesting of operations.

Parameters

- **arg1** (*Program*) – program to be executed
- **arg2** (*Operation*) – classical relational operation (<, >, <=, >=, ==, !=)

add_if_else (*args)

Adds specified sub-programs to a program. First sub-program will be executed if specified condition is true. Second sub-program will be executed if specified condition is false.

Parameters

- **arg1** (*Program*) – program to be executed when specified condition is true (if part).
- **arg2** (*Program*) – program to be executed when specified condition is false (else part).
- **arg3** (*Operation*) – classical relational operation (<, >, <=, >=, ==, !=)

add_kernel (*k*)

Adds specified kernel to program.

Parameters **arg1** (*kernel*) – kernel to be added

compile ()

Compiles the program

Parameters **None** –

get_sweep_points ()

Returns sweep points for an experiment.

Parameters **None** –

Returns list of sweep points

Return type []

microcode ()

Returns program microcode

Parameters **None** –

Returns microcode

Return type str

set_sweep_points (*sweep_points*)

Sets sweep points for an experiment.

Parameters **arg1** ([]) – list of sweep points

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

Unitary class to hold the matrix and its decomposition

decompose ()

Decomposes the unitary matrix

Parameters **None** –

Returns

Return type None

class openql.openql.**cQasmReader** (*q_platform, q_program*)

cQasmReader class specifies an interface to add cqasm programs to a program.

file2circuit (*cqasm_file_path*)

Adds a cqasm program read from a file.

Parameters **arg1** (*str*) – File path to the file specifying the cqasm that is added to the program.

string2circuit (*cqasm_str*)

Adds a cqasm program defined in a string.

Parameters **arg1** (*str*) – The cqasm that is added to the program.

`openql.openql.get_option(option_name)`

Returns value of any of the following OpenQL options:

Opt. Name	Defaults	Possible values
log_level	LOG_NOTHING	LOG_{NOTHING/CRITICAL/ERROR/WARNING/INFO/DEBUG}
output_dir	test_output	<output directory>
optimize	no	yes/no
use_default_gates	yes	yes/no
decompose_toffoli	no	yes/no
scheduler	ASAP	ASAP/ALAP
scheduler_uniform	no	yes/no
sched- uler_commute	no	yes/no
scheduler_post179	yes	yes/no
cz_mode	manual	auto/manual

Parameters **arg1** (*str*) – Option name

Returns Option value

Return type str

`openql.openql.get_version()`

Returns OpenQL version

Parameters **None** –

Returns version number as a string

Return type str

`openql.openql.print_options()`

Prints a list of available OpenQL options with their values.

`openql.openql.set_option(option_name, option_value)`

Sets any of the following OpenQL options:

Opt. Name	Defaults	Possible values
log_level	LOG_NOTHING	LOG_{NOTHING/CRITICAL/ERROR/WARNING/INFO/DEBUG}
output_dir	test_output	<output directory>
optimize	no	yes/no
use_default_gates	yes	yes/no
decompose_toffoli	no	yes/no
scheduler	ASAP	ASAP/ALAP
scheduler_uniform	no	yes/no
sched- uler_commute	no	yes/no
scheduler_post179	yes	yes/no
cz_mode	manual	auto/manual

Parameters

- **arg1** (*str*) – Option name
- **arg2** (*str*) – Option value

class openql.openql.**Kernel** (*args)
Kernel class which contains various quantum instructions.

__init__ (*args)

Constructs a Kernel object.

Parameters

- **arg1** (*str*) – name of the Kernel
- **arg2** (*Platform*) – target platform for which the kernel will be compiled
- **arg3** (*int*) – qubit count
- **arg4** (*int*) – classical register count

Methods

<i>__init__</i> (*args)	Constructs a Kernel object.
<i>barrier</i> (*args)	inserts explicit barrier on specified qubits.
<i>classical</i> (*args)	adds classical operation kernel.
<i>clifford</i> (id, q0)	Applies clifford operation of the specified id on the qubit.
<i>cnot</i> (q0, q1)	Applies controlled-not operation.
<i>conjugate</i> (k)	generates conjugate version of the kernel from the input kernel.
<i>controlled</i> (k, control_qubits, ancilla_qubits)	generates controlled version of the kernel from the input kernel.
<i>cphase</i> (q0, q1)	Applies controlled-phase operation.
<i>cz</i> (q0, q1)	
<i>display</i> ()	inserts QX display instruction (so QX specific).

Continued on next page

Table 14.1 – continued from previous page

<i>gate</i> (*args)	adds unitary to kernel.
<i>get_custom_instructions</i> ()	Returns list of available custom instructions.
<i>hadamard</i> (q0)	Applies hadamard on the qubit specified in argument.
<i>identity</i> (q0)	Applies identity on the qubit specified in argument.
<i>measure</i> (q0)	measures input qubit.
<i>mrx90</i> (q0)	Applies mrx90 on the qubit specified in argument.
<i>mry90</i> (q0)	
<i>prepz</i> (q0)	
<i>rx</i> (q0, angle)	
<i>rx180</i> (q0)	Applies rx180 on the qubit specified in argument.
<i>rx90</i> (q0)	Applies rx90 on the qubit specified in argument.
<i>ry</i> (q0, angle)	
<i>ry180</i> (q0)	Applies ry180 on the qubit specified in argument.
<i>ry90</i> (q0)	
<i>rz</i> (q0, angle)	
<i>s</i> (q0)	Applies x on the qubit specified in argument.
<i>sdag</i> (q0)	Applies sdag on the qubit specified in argument.
<i>t</i> (q0)	
<i>tdag</i> (q0)	
<i>toffoli</i> (q0, q1, q2)	Applies controlled-controlled-not operation.
<i>wait</i> (qubits, duration)	inserts explicit wait of specified duration on specified qubits.
<i>x</i> (q0)	
<i>y</i> (q0)	Applies y on the qubit specified in argument.
<i>z</i> (q0)	Applies z on the qubit specified in argument.

Attributes

<i>creg_count</i>
<i>kernel</i>
<i>name</i>
<i>platform</i>
<i>qubit_count</i>

barrier (*args)

inserts explicit barrier on specified qubits.

wait with duration '0' is also equivalent to applying barrier on specified list of qubits. If no qubits are specified, then barrier is applied on all the qubits.

Parameters *arg1* ([]) – list of qubits

classical (*args)

adds classical operation kernel.

Parameters

- **arg1** (CReg) – destination register for classical operation.
- **arg2** (Operation) – classical operation.

clifford (*id*, *q0*)

Applies clifford operation of the specified id on the qubit.

The ids and the corresponding operations are:

id	Operations
0	['I']
1	['Y90', 'X90']
2	['mX90', 'mY90']
3	['X180']
4	['mY90', 'mX90']
5	['X90', 'mY90']
6	['Y180']
7	['mY90', 'X90']
8	['X90', 'Y90']
9	['X180', 'Y180']
10	['Y90', 'mX90']
11	['mX90', 'Y90']
12	['Y90', 'X180']
13	['mX90']
14	['X90', 'mY90', 'mX90']
15	['mY90']
16	['X90']
17	['X90', 'Y90', 'X90']
18	['mY90', 'X180']
19	['X90', 'Y180']
20	['X90', 'mY90', 'X90']
21	['Y90']
22	['mX90', 'Y180']
23	['X90', 'Y90', 'mX90']

Parameters

- **arg1** (*int*) – clifford operation id
- **arg2** (*int*) – target qubit

cnot (*q0*, *q1*)

Applies controlled-not operation.

Parameters

- **arg1** (*int*) – control qubit
- **arg2** (*int*) – target qubit

conjugate (*k*)

generates conjugate version of the kernel from the input kernel.

Parameters **arg1** (*q1* : *Kernel*) – input kernel. Except measure, Kernel to be conjugated.

Returns

Return type None

controlled (*k*, *control_qubits*, *ancilla_qubits*)

generates controlled version of the kernel from the input kernel.

Parameters

- **arg1** (*ql* : *Kernel*) – input kernel. Except measure, Kernel to be controlled may contain any of the default gates as well custom gates which are not specialized for a specific qubits.
- **arg2** (*[]*) – list of control qubits.
- **arg3** (*[]*) – list of ancilla qubits. Number of ancilla qubits should be equal to number of control qubits.

Returns

Return type None

cphase (*q0*, *q1*)

Applies controlled-phase operation.

Parameters

- **arg1** (*int*) – control qubit
- **arg2** (*int*) – target qubit

display ()

inserts QX display instruction (so QX specific).

Parameters None –

Returns

Return type None

gate (**args*)

adds unitary to kernel.

Parameters

- **arg1** (*Unitary*) – unitary matrix
- **arg2** (*[]*) – list of qubits

get_custom_instructions ()

Returns list of available custom instructions.

Parameters None –

Returns List of available custom instructions

Return type []

hadamard (*q0*)

Applies hadamard on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

identity (*q0*)

Applies identity on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

measure (*q0*)

measures input qubit.

Parameters **arg1** (*int*) – input qubit

mrx90 (*q0*)

Applies mrx90 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

rx180 (*q0*)

Applies rx180 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

rx90 (*q0*)

Applies rx90 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

ry180 (*q0*)

Applies ry180 on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

s (*q0*)

Applies x on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

sdag (*q0*)

Applies sdag on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

toffoli (*q0, q1, q2*)

Applies controlled-controlled-not operation.

Parameters

- **arg1** (*int*) – control qubit
- **arg2** (*int*) – control qubit

- **arg3** (*int*) – target qubit

wait (*qubits, duration*)

inserts explicit wait of specified duration on specified qubits.

wait with duration '0' is equivalent to barrier on specified list of qubits. If no qubits are specified, then wait/barrier is applied on all the qubits.

Parameters

- **arg1** (*[]*) – list of qubits
- **arg2** (*int*) – duration in ns

y (*q0*)

Applies y on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

z (*q0*)

Applies z on the qubit specified in argument.

Parameters **arg1** (*int*) – target qubit

Program

class openql.openql.**Program**(*args)
Program class which contains one or more kernels.

__init__(*args)
Constructs a program object.

Parameters

- **arg1** (*str*) – name of the program
- **arg2** (*Platform*) – instance of an OpenQL Platform
- **arg3** (*int*) – number of qubits the program will use
- **arg4** (*int*) – number of classical registers the program will use (default: 0)

Methods

<code>__init__(*args)</code>	Constructs a program object.
<code>add_do_while(*args)</code>	Adds specified sub-program to a program which will be repeatedly executed while specified condition is true.
<code>add_for(*args)</code>	Adds specified sub-program to a program which will be executed for specified iterations.
<code>add_if(*args)</code>	Adds specified sub-program to a program which will be executed if specified condition is true. This allows nesting of operations.
<code>add_if_else(*args)</code>	Adds specified sub-programs to a program. First sub-program will be executed if specified condition is true. Second sub-program will be executed if specified condition is false.

Continued on next page

Table 15.1 – continued from previous page

<code>add_kernel(k)</code>	Adds specified kernel to program.
<code>add_program(p)</code>	
<code>compile()</code>	Compiles the program
<code>get_sweep_points()</code>	Returns sweep points for an experiment.
<code>microcode()</code>	Returns program microcode
<code>print_interaction_matrix()</code>	
<code>Program.qasm</code>	
<code>set_sweep_points(sweep_points)</code>	Sets sweep points for an experiment.
<code>write_interaction_matrix()</code>	

Attributes

<code>creg_count</code>
<code>name</code>
<code>platform</code>
<code>program</code>
<code>qubit_count</code>

add_do_while (*args)

Adds specified sub-program to a program which will be repeatedly executed while specified condition is true.

Parameters

- **arg1** (*Program*) – program to be executed repeatedly
- **arg2** (*Operation*) – classical relational operation (<, >, <=, >=, ==, !=)

add_for (*args)

Adds specified sub-program to a program which will be executed for specified iterations.

Parameters

- **arg1** (*Program*) – sub-program to be executed repeatedly
- **arg2** (*int*) – iteration count

add_if (*args)

Adds specified sub-program to a program which will be executed if specified condition is true. This allows nesting of operations.

Parameters

- **arg1** (*Program*) – program to be executed
- **arg2** (*Operation*) – classical relational operation (<, >, <=, >=, ==, !=)

add_if_else (*args)

Adds specified sub-programs to a program. First sub-program will be executed if specified condition is true. Second sub-program will be executed if specified condition is false.

Parameters

- **arg1** (*Program*) – program to be executed when specified condition is true (if part).
- **arg2** (*Program*) – program to be executed when specified condition is false (else part).
- **arg3** (*Operation*) – classical relational operation (<, >, <=, >=, ==, !=)

add_kernel (*k*)

Adds specified kernel to program.

Parameters **arg1** (*kernel*) – kernel to be added

compile ()

Compiles the program

Parameters **None** –

get_sweep_points ()

Returns sweep points for an experiment.

Parameters **None** –

Returns list of sweep points

Return type []

microcode ()

Returns program microcode

Parameters **None** –

Returns microcode

Return type str

set_sweep_points (*sweep_points*)

Sets sweep points for an experiment.

Parameters **arg1** ([]) – list of sweep points

class `openql.openql.Compiler` (*name*)
 Compiler class which contains one or more compiler passes.

__init__ (*name*)
 Constructs a compiler object.

Parameters **arg1** (*str*) – name of the compiler

Methods

<code>__init__(name)</code>	Constructs a compiler object.
<code>compile(program)</code>	Compiles the program
<code>add_pass(realPassName)</code>	Adds a compiler pass under its real name
<code>add_pass_alias(realPassName, symbolicPass-Name)</code>	Adds a compiler pass under an alias name
<code>set_pass_option(passName, optionName, ...)</code>	Sets a compiler pass option

Attributes

<code>name</code>

add_pass (*realPassName*)
 Adds a compiler pass under its real name

Parameters **arg1** (*str*) – name of the real pass to be added.

add_pass_alias (*realPassName, symbolicPassName*)
 Adds a compiler pass under an alias name

Parameters

- **arg1** (*str*) – name of the real pass to be added.
- **arg2** (*str*) – alias name of the pass to be added.

compile (*program*)

Compiles the program

Parameters **arg1** (*Program*) – program object to be compiled.**set_pass_option** (*passName, optionName, optionValue*)

Sets a compiler pass option

Parameters

- **arg1** (*str*) – name (real or alias) of the compiler pass to be added.
- **arg2** (*str*) – option name of the option to be configured.
- **arg3** (*str*) – value of the option.

Platform

class openql.openql.**Platform**(*args)
Platform class specifying the target platform to be used for compilation.

__init__(*args)
Constructs a Platform object.

Parameters

- **arg1** (*str*) – name of the Platform
- **arg2** (*str*) – name of the configuration file specifying the platform

Methods

<code>__init__</code> (*args)	Constructs a Platform object.
<code>get_qubit_number</code> ()	returns number of qubits in the platform.

Attributes

<code>config_file</code>
<code>name</code>
<code>platform</code>

get_qubit_number()
returns number of qubits in the platform.

Parameters None –

Returns number of qubits

Return type int

Operation

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

Operation class representing classical operations.

__init__(*args)

Constructs an Operation object (used for initializing with immediate values).

Parameters **arg1** (*int*) – immediate value

Methods

__init__(*args)

Constructs an Operation object (used for initializing with immediate values).

Attributes

operation

CReg

class openql.openql.CReg

Classical register class.

__init__()

Constructs a classical register which can be source/destination for classical operations.

Parameters **None** –

Returns classical register object

Return type *CReg*

Methods

__init__()

Constructs a classical register which can be source/destination for classical operations.

Attributes

creg

CHAPTER 20

QX Simulation

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`



20.1 OpenQL Program

We start by importing `openql`, `qxelator` 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 qxelator
```

(continues on next page)

(continued from previous page)

```

from functools import reduce
import os
import matplotlib.pyplot as plt

curdir = os.path.dirname(__file__)
output_dir = os.path.join(curdir, 'test_output')

ql.set_option('output_dir', output_dir)
ql.set_option('write_qasm_files', 'yes')
ql.set_option('scheduler', 'ASAP')
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():
    print('compiling 8-face dice program by openql')
    config = os.path.join(curdir, '../tests/hardware_config_qx.json')

    platform = ql.Platform("myPlatform", config)
    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 v1.0** format:

- `test_output/dice.qasm` which is the generated un-scheduled qasm code
- `test_output/dice_scheduled.qasm` which is the generated qasm 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('test_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
```

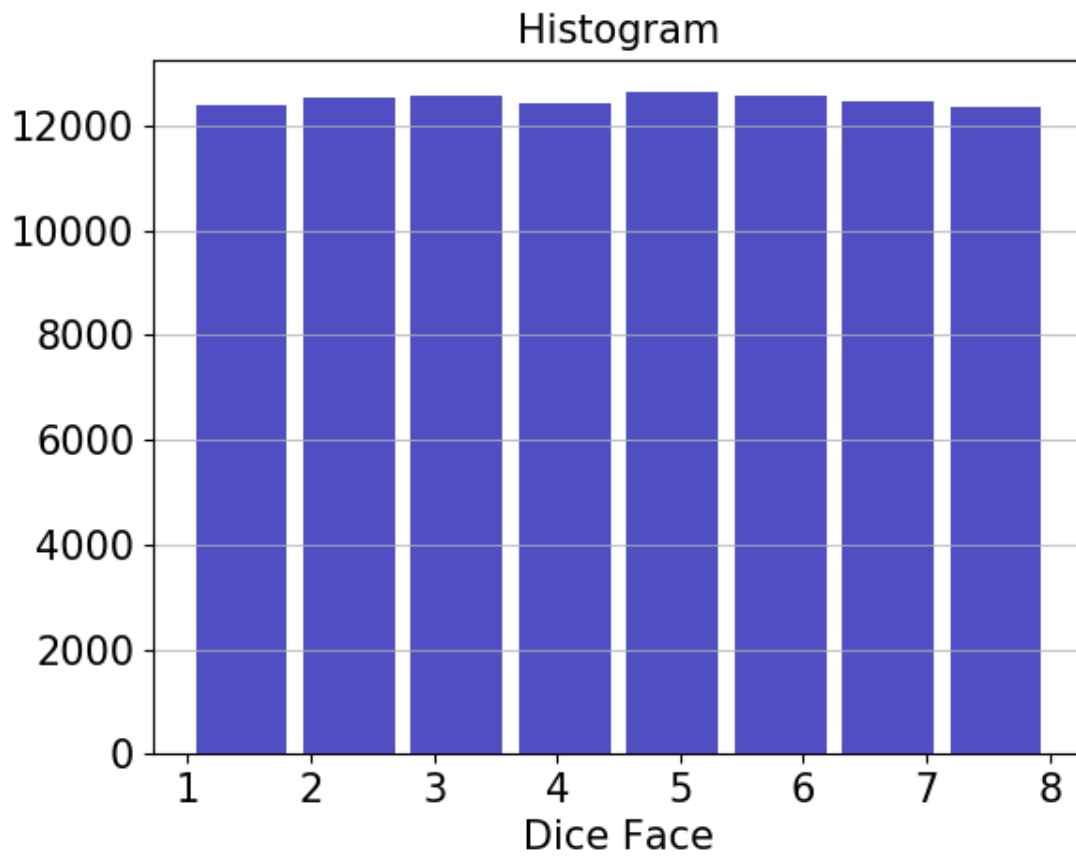
where, the Dice face can be any number between 1 and 8.

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

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



CHAPTER 21

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.

21.1 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
```

21.2 Replicating the QXelator 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('test_output/dice.qasm', 'test_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 through run()
    with Simulator('test_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.

21.3 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.

21.4 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('test_output/dice_scheduled.qasm', 'test_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.

21.5 Further reading

A more extensive Python tutorial for DQCsims 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 DQCsims.

Developer Documentation

Documentation for the C++ source code is [generated by Doxygen](#).

This is not intended as an API reference, as it includes internal stuff. If you just want to *use* OpenQL, it's strongly recommended to use it from Python, because OpenQL's internals change a lot. Nevertheless, you *can* use it straight from C++. To do that, first make sure you're familiar with the Python API, then look at the examples folder of the repository to see more or less how the API calls map, and only then look in the Doxygen documentation for details.

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

Symbols

`__init__()` (*openql.openql.CReg method*), 115
`__init__()` (*openql.openql.Compiler method*), 109
`__init__()` (*openql.openql.Kernel method*), 99
`__init__()` (*openql.openql.Operation method*), 113
`__init__()` (*openql.openql.Platform method*), 111
`__init__()` (*openql.openql.Program method*), 105

A

`add_do_while()` (*openql.openql.Program method*), 106
`add_for()` (*openql.openql.Program method*), 106
`add_if()` (*openql.openql.Program method*), 106
`add_if_else()` (*openql.openql.Program method*), 106
`add_kernel()` (*openql.openql.Program method*), 107
`add_pass()` (*openql.openql.Compiler method*), 109
`add_pass_alias()` (*openql.openql.Compiler method*), 109

B

`barrier()` (*openql.openql.Kernel method*), 100

C

`classical()` (*openql.openql.Kernel method*), 100
`clifford()` (*openql.openql.Kernel method*), 101
`cnot()` (*openql.openql.Kernel method*), 101
`compile()` (*openql.openql.Compiler method*), 110
`compile()` (*openql.openql.Program method*), 107
`Compiler` (*class in openql.openql*), 109
`conjugate()` (*openql.openql.Kernel method*), 101
`controlled()` (*openql.openql.Kernel method*), 101
`cphase()` (*openql.openql.Kernel method*), 102
`CReg` (*class in openql.openql*), 115

D

`display()` (*openql.openql.Kernel method*), 102

G

`gate()` (*openql.openql.Kernel method*), 102
`get_custom_instructions()` (*openql.openql.Kernel method*), 102
`get_qubit_number()` (*openql.openql.Platform method*), 111
`get_sweep_points()` (*openql.openql.Program method*), 107

H

`hadamard()` (*openql.openql.Kernel method*), 102

I

`identity()` (*openql.openql.Kernel method*), 103

K

`Kernel` (*class in openql.openql*), 99

M

`measure()` (*openql.openql.Kernel method*), 103
`microcode()` (*openql.openql.Program method*), 107
`mr90()` (*openql.openql.Kernel method*), 103

O

`Operation` (*class in openql.openql*), 113

P

`Platform` (*class in openql.openql*), 111
`Program` (*class in openql.openql*), 105

R

`rx180()` (*openql.openql.Kernel method*), 103
`rx90()` (*openql.openql.Kernel method*), 103
`ry180()` (*openql.openql.Kernel method*), 103

S

`s()` (*openql.openql.Kernel method*), 103
`sdag()` (*openql.openql.Kernel method*), 103

`set_pass_option()` (*openql.openql.Compiler method*), [110](#)
`set_sweep_points()` (*openql.openql.Program method*), [107](#)

T

`toffoli()` (*openql.openql.Kernel method*), [103](#)

W

`wait()` (*openql.openql.Kernel method*), [104](#)

Y

`y()` (*openql.openql.Kernel method*), [104](#)

Z

`z()` (*openql.openql.Kernel method*), [104](#)