OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators

N. KHAMMASSI, I. ASHRAF, J. V. SOMEREN, R. NANE, and A. M. KROL, Quantum & Computer Engineering Dept., Delft University of Technology, The Netherlands
M. A. ROL, Kavli Institute of Nanoscience, Delft University of Technology, The Netherlands
L. LAO, K. BERTELS, and C. G. ALMUDEVER, Quantum & Computer Engineering Dept., Delft University of Technology, The Netherlands

With the potential of quantum algorithms to solve intractable classical problems, quantum computing is rapidly evolving, and more algorithms are being developed and optimized. Expressing these quantum algorithms using a high-level language and making them executable on a quantum processor while abstracting away hardware details is a challenging task. First, a quantum programming language should provide an intuitive programming interface to describe those algorithms. Then a compiler has to transform the program into a quantum circuit, optimize it, and map it to the target quantum processor respecting the hardware constraints such as the supported quantum operations, the qubit connectivity, and the control electronics limitations. In this article, we propose a quantum programming framework named OpenQL, which includes a high-level quantum programming language and its associated quantum compiler. We present the programming interface of OpenQL, we describe the different layers of the compiler and how we can provide portability over different qubit technologies. Our experiments show that OpenQL allows the execution of the same high-level algorithm on two different qubit technologies, namely superconducting qubits and Si-Spin qubits. Besides the executable code, OpenQL also produces an intermediate quantum assembly code, which is technology independent and can be simulated using the QX simulator.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **Compilers;**

Additional Key Words and Phrases: Quantum compiler, quantum computing, quantum circuit, quantum processor

This project is funded by Intel Corporation.

N. Khammassi, Currently affiliated with the Intel Labs, Intel Corporation, Oregon, USA. I. Ashraf, Currently affiliated with the Computer Engineering Department, HITEC University, Taxila, Pakistan. L. Lao, Currently affiliated with the Department of Physics and Astronomy, University College London, UK.
Authors' addresses: N. Khammassi, Intel Labs, JF2-2-82-53 Pole J2, 2111 NE 25th Ave, Hillsboro, OR 97124, United States; email: nader.khammassi@intel.com; I. Ashraf, HITEC University, Khanpur Road, Taxila, Pakistan; email: i.ashraf@tudelf.nl; J. V. Someren, Quantum & Computer Engineering Dept., Mekelweg 5, 2628 CD Delft, Netherlands; email: J.Vansomeren-1@tudelft.nl; R. Nane, Quantum & Computer Engineering Dept., Mekelweg 5, 2628 CD Delft, Netherlands; email: R.Nane@tudelft.nl; A. M. Krol, Quantum & Computer Engineering Dept., Mekelweg 5, 2628 CD Delft, Netherlands; email: annerietkrol@gmail.com; M. A. Rol, Kavli Institute of Nanoscience, Lorentzweg 1, 2628 CJ Delft, Netherlands; email: M.A.Rol@tudelft.nl; L. Lao, Department of Physics and Astronomy, University College, Gower St, London WC1E 6BT, United Kingdom; email: l.lao@ucl.ac.uk; K. Bertels, Quantum & Computer Engineering Dept., Mekelweg 5, 2628 CD Delft, Netherlands; email: K.L.M.Bertels@tudelft.nl; C. G. Almudever, Quantum & Computer Engineering Dept., Mekelweg 5, 2628 CD Delft, Netherlands; email: c.garciaalmudever-1@tudelft.nl.

1550-4832/2021/12-ART13 $15.00
https://doi.org/10.1145/3474222

ACM Reference format:
N. Khammassi, I. Ashraf, J. V. Someren, R. Nane, A. M. Krol, M. A. Rol, L. Lao, K. Bertels, and C. G. Almudever. 2021. OpenQL: A Portable Quantum Programming Framework for Quantum Accelerators. *J. Emerg. Technol. Comput. Syst.* 18, 1, Article 13 (December 2021), 24 pages.
https://doi.org/10.1145/3474222

1 INTRODUCTION

Since the early formulation of the foundations of quantum computing, several quantum algorithms have been designed for solving intractable classical problems in different application domains. For instance, the introduction of Shor's algorithm [45] outlined the significant potential of quantum computing in speeding up prime factorization. Later, Grover's search algorithm [17] demonstrated quadratic speedup over its classical implementation counterpart. The discovery of these algorithms boosted the development of different physical qubit implementations such as superconducting qubits [52], trapped ions [32], and semiconducting qubits [53].

In the absence of a fully programmable quantum computer, the implementation of these algorithms on real quantum processors is a tedious task for the algorithm designer, especially in the absence of deep expertise in qubit control electronics. To make a quantum computer programmable and more accessible to quantum algorithm designers similarly to classical computers, several software and hardware layers are required [4]: At the highest level, an intuitive quantum programming language is needed to allow the programmer to express the quantum algorithm without worrying about the hardware details. Then, a compiler transforms the algorithm into a quantum circuit and maps and optimizes it for a given quantum processor. Ultimately, the compiler produces an executable code that can be executed on the target micro-architecture controlling the qubits. A modular quantum compiler would ideally not expose low-level hardware details and its constraints to the programmer to allow portability of the algorithm over a wide range of quantum processors and qubit technologies.

In this article, we introduce OpenQL,[1] an open source[2] high-level quantum programming framework. OpenQL is mainly composed of a quantum programming interface for implementing quantum algorithms independently from the target platform, and a compiler that can compile the algorithm into executable code for various target platforms and qubit technologies such as superconducting qubits and semiconducting qubits.

The rest of the article is organized as follows. Section 2 provides a brief account of the related work. The necessary background for the quantum accelerator model is given in Section 3. OpenQL architecture is detailed in Section 4, followed by a discussion of the quantum programming interface provided by OpenQL in Section 5. OpenQL compilation passes are presented in Section 6, where it is shown how the quantum code is decomposed, optimized, scheduled, and mapped on the target platform. Some of the works in which we utilized OpenQL to compile quantum algorithms on different quantum processors using different qubit technologies, are briefly mentioned in Section 7. Finally, Section 8 concludes the article.

2 RELATED WORK

Some of the initial work in the field of quantum compilation has been theoretical [9, 36, 37, 43, 56, 57]. Now that quantum computers are a reality, various compilation and simulation software frameworks have been developed. A list of open source compilation projects is available at [12], and a list of quantum simulators is available in Reference [30]. In the following, we provide a brief

[1]OpenQL documentation: https://openql.readthedocs.io.
[2]OpenQL source code: https://github.com/QE-Lab/OpenQL.

list of recent active works in the field of quantum compilation in chronological order. The reader is referred to a recent overview and comparison of gate-level quantum software platforms [27].

- ScaffCC has been presented as a scalable compilation and analysis tool for quantum programs [2, 19]. It is based on the LLVM compilation framework. ScaffCC compiles the Scaffold language [1], which is a pure quantum language embedded into the classical *C* language.
- Microsoft proposed a domain-specific language Q# [49] and **Quantum Development Kit (QDK)** to compile and simulate quantum programs. At the moment, QDK does not target a real quantum computer, however, programs can be executed on the provided software backend.
- ProjectQ [48] is an open source software framework that allows the expression of a quantum program targeting IBM backend computers as well as simulators. ProjectQ allows programmers to express their programs in a language embedded in Python. Apart from low-level gate descriptions, meta-instructions are provided to add conditional control, compute, uncompute, and repeating sections of code a certain number of times.
- IBM's Qiskit [3] is an open source quantum software framework that allows users to express their programs in Python and compiles them to OpenQASM targeting the IBM Q Experience [18]. Qiskit allows users to explicitly allocate quantum and classical registers. Quantum operations are performed on quantum registers, and after measurement, classical results are stored in classical registers.
- Quilc [47] is an open source quantum compiler for compiling Rigetti's Quil language [46]. The focus of the authors is on the noisy intermediate scale quantum programs, allowing the programmers to compile quantum programs to byte code, which can be interpreted by control electronics. This allows programmers to execute programs not only on a software simulator but also on a real quantum processor.
- Amazon provides Braket [5] service to allow users to perform quantum computing in the cloud. Amazon Braket provides a development environment that includes Amazon Braket SDK [6], which is an open source library. This library helps developers express a quantum program, compile it and run it on Braket-enabled software simulators and various quantum computers.

OpenQL has some common characteristics with the compilers above, such as being an open source, modular quantum compilation framework that is capable of targeting different hardware backends. However, the distinctive and, at the same time, the primary motivation behind OpenQL is that it is a generic and flexible compiler framework. These requirements directly translated into the OpenQL design to support multiple *configurable* backends through its platform configuration file (Section 5.3). Finally, OpenQL is one of the engines behind QuTech's Quantum Inspire [38] platform, where the user can gain access to various technologies to perform quantum experiments enabled through the use of OpenQL's plugin-able backends and its ability to generate executable code (Section 6.5).

3 QUANTUM ACCELERATOR MODEL

Accelerators are used in classical computers to speed up specific types of computation that can take advantage of the execution capabilities of the accelerator such as massive parallelism, vectorization or fast digital signal processing. OpenQL adopts this heterogeneous computing model while using the quantum processor as an accelerator and provides a programming interface for implementing quantum algorithms involving both classical computation and quantum computation.

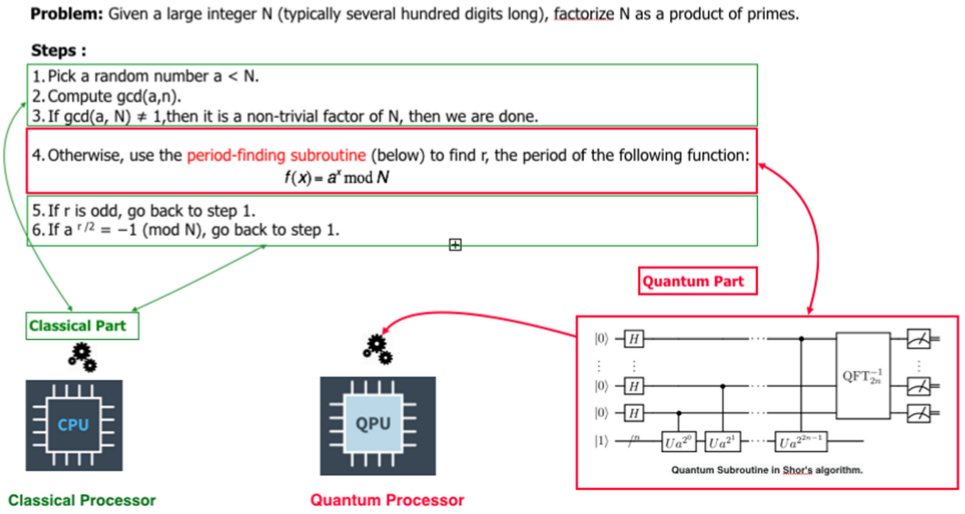

Fig. 1. Shor's algorithm is composed of both classical computations and quantum computations.

3.1 Heterogeneous Computing

Heterogeneous computing [40, 55] is a computing model where a program is executed jointly on a general-purpose processor or host processor and an accelerator or co-processor. The general-purpose processor is capable of executing not only general computations such as arithmetic, logic or floating point operations, but also controlling various accelerators or co-processors. The accelerators or co-processors are specialized processors designed to accelerate specific types of computation such as graphics processing, digital signal processing and other workloads that can take advantage of vectorization or massive thread-level parallelism. Therefore the accelerator can speedup a part of the computation traditionally executed on a general purpose processor. The computation is then offloaded to the accelerator to speed up the overall execution of the target program. Examples of accelerators are the Intel Xeon Phi co-processor [20], Digital Signal Processors [50], Field Programmable Gate Array [51, 54] that can be also utilized as accelerators to parallelize computations and speed up their execution. Finally General-Purpose Computation on Graphics Processing Units uses a GPU as an accelerator [29] to speed up certain types of computations.

3.2 Quantum Processors as Accelerators

The OpenQL programming framework follows a heterogeneous programming model that aims to use the quantum processor as a co-processor to accelerate the part of the computation that can benefit from the quantum speedup. A quantum algorithm is generally composed of classical and quantum computations. For instance Shor's algorithm is a famous quantum algorithm for prime number factoring. As shown in Figure 1 the algorithm includes classical computations such as the Greatest Common Divisor computation that can be executed efficiently on a traditional processor, and a quantum part such as the Quantum Fourier Transform, which should be executed on a quantum processor.

OpenQL uses traditional host languages, namely C++ and Python, to define a programming interface that allows the expression of the quantum computation and the communication with the quantum accelerator: The quantum operations are executed on the quantum processor using a dedicated micro-architecture and the measurement results are collected and sent back to the host

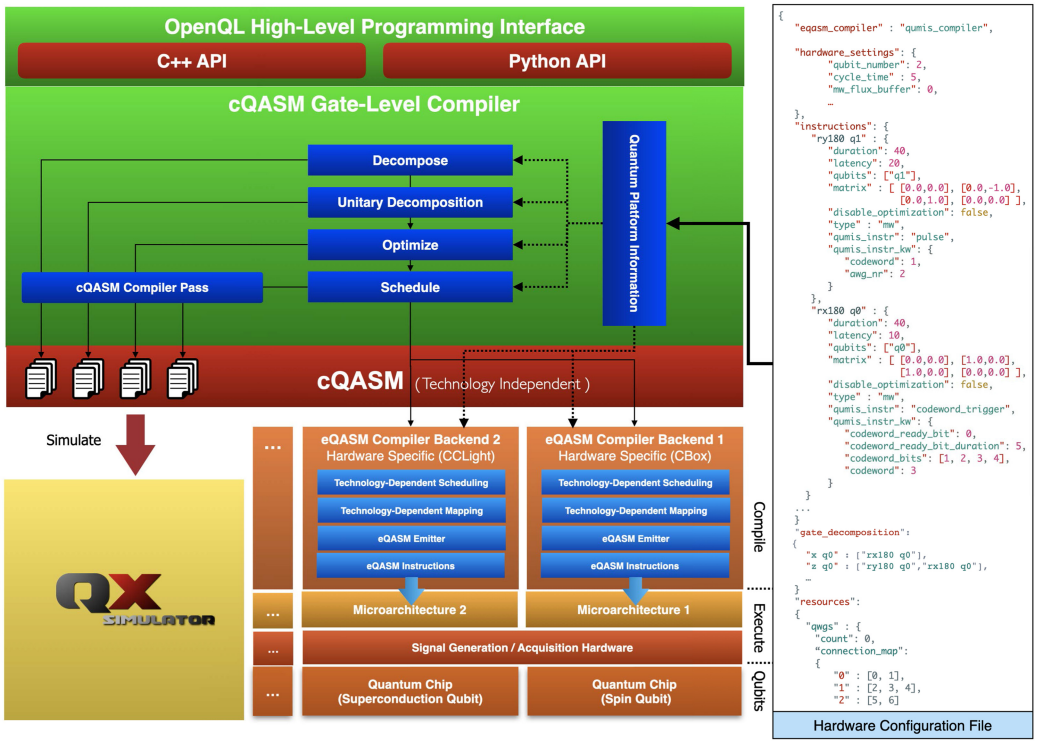

Fig. 2. OpenQL compiler architecture.

program running on the classical processor. While non time-critical classical operations can be executed on the host processor, time-critical classical operations that need to be executed within the coherence time of the qubits, such as in error correction quantum circuits, can be offloaded to the accelerator to provide fast reaction time and avoid communication overhead between the host PC and the accelerator.

4 OPENQL ARCHITECTURE

Figure 2 depicts the OpenQL framework, which presents a high-level programming interface to the user at the top. The compiler implements a layered architecture that is composed mainly of two parts: a set of hardware-agnostic compilation passes that operate at the quantum gate level and a set of low-level technology-specific backends that can target different quantum processors with specific control hardware. The goal of those backends is to enable compiling the same quantum algorithm for a specific qubit technology without any change in the high-level code and making the hardware details transparent to the programmer. Moreover, this architecture allows the implementation of new backends to extend the support to other qubit technologies and new control hardware whenever needed. As the qubit control hardware is constantly evolving in the last years, this flexibility and portability over a wide range of hardware is crucial. This enhances the productivity and ensures the continuity of the research efforts toward a full-stack quantum computer integration.

The **Quantum Assembly Code (QASM)** is the intermediate layer that draws the abstraction line between the high-level hardware-agnostic layers (gate-level compilation stages) and the low-level hardware-specific layers. The low-level layers are implemented inside a set of interchangeable backends each targeting a different micro-architecture and/or a different qubit technology.

The OpenQL framework is composed mainly of the following layers:

- A High-level programming interface using a standard host language namely C++ or Python to express the target quantum algorithm as a quantum program.
- A quantum gate-level compiler that transforms the quantum program into a quantum circuit, optimizes it, schedules it and maps it to the target quantum processor to comply to the different hardware constraints such as limited qubit connectivity.
- The last stage of the gate-level compilation produces a technology-independent **Common Quantum Assembly code (cQASM)** [24], which describes the final quantum circuit while abstracting away the low-level hardware details such as the target instruction set architecture, or the quantum gate implementation, which differs across the different qubit technologies For now, our compiler targets Superconducting qubits and Si-Spin qubits but can be easily extended to other qubit technologies. The produced QASM code complies with the cQASM 1.0 syntax and can be simulated in our QX simulator [22] to debug the quantum algorithm and evaluate its performance for different quantum error rates.
- At the lowest level, different **executable QASM (eQASM)** [14] backends can be used to compile the QASM code into instructions that can be executed on a specific micro-architecture, e.g., the QuMA micro-architecture described in Reference [16]. At this compilation level, very detailed information about the target hardware setup, stored in a hardware configuration file, is used to generate an executable code that takes into account various hardware details such as the implementation of the quantum gates, the connectivity between the qubits and the control instruments, the hardware resource dependencies, the quantum operation latencies and the operational constraints.

5 QUANTUM PROGRAMMING INTERFACE

OpenQL provides three main interfaces to the developer, namely Quantum Kernel, Quantum Program, and Quantum Platform.

5.1 Quantum Kernel

A *Quantum Kernel* is a quantum functional block that consists of a set of quantum or classical instructions and performs a specific quantum operation. For instance, the kernel could be dedicated to creating a bell pair while another could be dedicated to teleportation or decoding. In OpenQL a *Quantum Kernel* can be created using Python, as shown in Code Example 1 where three kernels are created: (i) the "init" kernel for initializing the qubits, (ii) the "epr" kernel to create a Bell pair, and (iii) the "measure" kernel to measure the qubits. These kernels are then added to the main program, and compiled while enabling the compiler optimizations and the **As Late As Possible (ALAP)** scheduling scheme. In code Example 2, the same code is written in the C++ programming language. Note that the programming API of C++ is identical to the Python API.

OpenQL supports standard quantum operations as listed in Table 1. For multi-qubit gates, first qubit is the target qubit. To allow for further flexibility in implementing the quantum algorithms, custom operations can also be defined in a hardware configuration file. These operations can either be independent physical quantum operations supported by the target hardware or a composition of a set of physical operations. Once defined in the configuration file of the platform, the new operation can be used in composing a kernel as any other predefined standard operation. This allows for more flexibility when designing a quantum algorithm or a standard experiment used for calibration or other purposes.

5.2 Quantum Program

As the quantum kernels implement functional blocks of a given quantum algorithm, a "*quantum_program*" is the container holding those quantum kernels and implementing the complete

quantum algorithm. For instance, if our target algorithm is a quantum error correction circuit that includes the encoding of the logical qubit, then the error syndrome measurement, the error correction and finally the decoding, we can create four distinct kernels that implement these four blocks, and we can add these kernels to our program. The program can then be compiled and executed on the target platform.

```
import openql as ql

# load the hardware config of the target platform
transmon = ql.quantum_platform('transmon', 'hardware_config.json');

# we create the main quantum program
prog = program('bell_pair',2,transmon)

# create init kernel to prepare q0 and q1 in zero state
k1 = kernel('init');
k1.prepz(0);
k1.prepz(1);

# create a bell pair kernel
k2 = kernel('epr');
k2.hadamard(0); # H q0
k2.cnot(0,1); # CNOT q0,q1

# create measure kernel
k3 = kernel('measure');
k3.measure(0);
k3.measure(1);

# add kernel to the quantum program
prog.add_kernel(k1);
prog.add_kernel(k2);
prog.add_kernel(k3);

# compile and optimize the program
prog.compile(optimize=true,schedule='ALAP');
```

Code Example 1. OpenQL Python code creating a Bell pair.

5.3 Quantum Platform

A "*quantum_platform*" is a specification of the target hardware setup including the quantum processor and its control electronics. The specification includes the description of the supported quantum operations and their attributes such as the duration, the built-in latency of each operation and the mathematical description of the supported quantum operation such as its associated unitary matrix.

6 QUANTUM GATE-LEVEL COMPILATION

The first compilation stages of OpenQL are performed at the quantum gate-level while abstracting the low-level hardware implementation on the target device as much as possible. The high-level compilation stages include the decomposition of the quantum operations, the optimization and the scheduling of the decomposed quantum circuit. The gate-level compilation layers can produce

a technology-agnostic quantum assembly code called cQASM that can be simulated using the QX Simulator [23].

```
#include <ql/openql.h>

// load the hardware config of the target platform
ql::quantum_platform transmon('transmon', 'hardware_config.json');

// create quantum program
ql::program prog('prog',2,transmon);

// create init kernel to prepare q0 and q1 in zero state
ql::kernel k1('init');
k1.prepz(0);
k1.prepz(1);

// create a bell pair kernel
ql::kernel k2('epr');
k2.hadamard(0); // H q0
k2.cnot(0,1); // CNOT q0,q1

// create measure kernel
ql::kernel k3('measure');
k3.measure(0);
k3.measure(1);

// add kernels to the quantum program
prog.add_kernel(k1);
prog.add_kernel(k2);
prog.add_kernel(k3);

// compile and optimize the program
prog.compile(optimize=true,schedule='ALAP');
```

Code Example 2. OpenQL C++ code creating a Bell pair.

6.1 Gate Decomposition

OpenQL supports decomposition of multi-qubit gates to 1 and 2 qubit gates, as well as control decomposition of multiple gates that are controlled by 1 or more qubits. Gates that are expressed as unitary matrices can also be decomposed to rotation and controlled-not gates.

6.1.1 Multi-qubit Gate Decomposition. In the first step, quantum gates are decomposed into a set of elementary operations from a universal gate set. For instance, as shown in Figure 3, the Toffoli gate can be decomposed into a set of single and two-qubit gates using different schemes such as in Reference [34], Reference [8], or Reference [7].

The decomposition of gates with more than two qubit operands is necessary to enable the later mapping stage, which can only deal with single and two-qubit gates that are available in the target physical implementation. Furthermore, this decomposition allows us to perform fine-grain optimization through fusing operations and extracting parallelism using gate dependency analysis. When a physical target platform and its supported physical operations are specified in the configuration file, by doing this decomposition the compiler makes sure that the remaining operations are the target primitive operations that are supported by the target platform. The hardware

Table 1. Supported Quantum Gates

Gate Name	Description	Example
I	Identity	kernel.identity(3)
H	Hadamard	kernel.hadamard(0)
X	Pauli-X	kernel.x(1)
Y	Pauli-Y	kernel.y(3)
Z	Pauli-Z	kernel.z(7)
Rx	Arbitrary x-rotation	kernel.rx(0, 3.14)
Ry	Arbitrary y-rotation	kernel.ry(5, 1.75)
Rz	Arbitrary z-rotation	kernel.rz(2, 0.5)
X90	R_x($\pi/2$)	kernel.x90(7)
Y90	R_y($\pi/2$)	kernel.y90(5)
mX90	R_x($-\pi/2$)	kernel.mx90(2)
mY90	R_y($-\pi/2$)	kernel.my90(1)
S	Phase	kernel.s(3)
Sdag	Phase dagger	kernel.sdag(13)
T	T	kernel.t(2)
Tdag	T dagger	kernel.tdag(12)
CNOT	CNOT	kernel.cnot(3,5)
Toffoli	Toffoli	kernel.toffoli(3,5,7)
CZ	CPHASE	kernel.cz(1,2)
SWAP	Swap	kernel.swap(0,3)
Custom	Custom gate	kernel.gate("name",2)

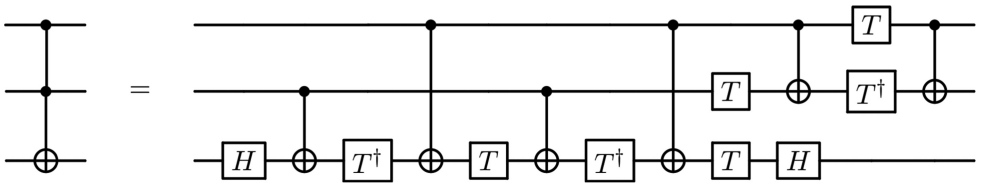

Fig. 3. Toffoli gate decomposition.

configuration specification is detailed in Section 6.7. We note that we can disable this decomposition stage when the QX simulator backend [22] is targeted as QX can simulate composite gates such as the Toffoli gate or arbitrary controlled rotations that are not necessarily available in many physical devices.

Multi-qubit controlled gates can also be decomposed to 2-qubit controlled gates as discussed in Reference [34] based on the network implementing $C^n(U)$ operation shown in Figure 4. For more details, reader can refer to Figure 4.10 in Reference [34]. This generalized decomposition scheme requires as many ancilla qubits as many control qubits. It is the responsibility of the user to specify the ancilla qubits to be used for this control decomposition.

OpenQL further extends the facility of control decomposition to multiple gates (kernel). This is achieved by generating a controlled version of a kernel by using the *controlled*() API as depicted in Code Example 3 and then applying decomposition.

6.1.2 Unitary Gate Decomposition. It has been demonstrated that a universal quantum computer can simulate any Turing machine [11] and any local quantum system [28]. A set of gates

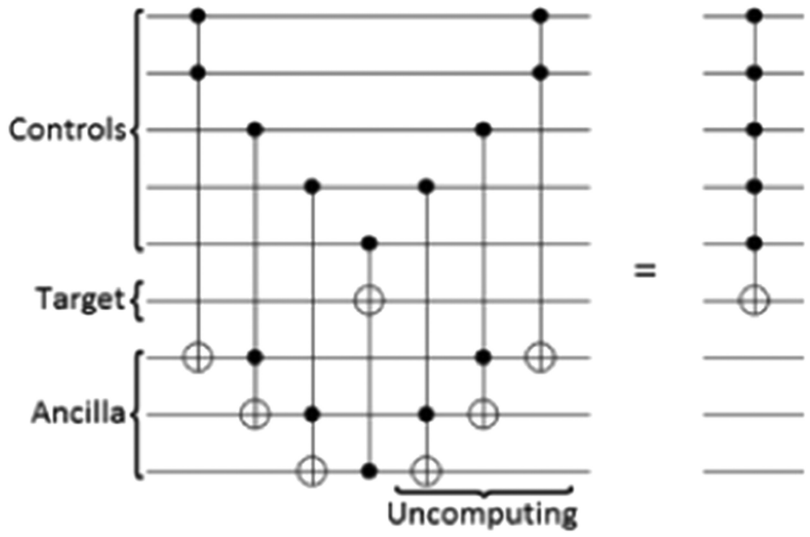

Fig. 4. Multi-qubit controlled decomposition.

```
...
k.gate("x", [0])
k.gate("y", [0])
k.gate("h", [0])
...

# generate controlled version of k.
# qubit 1 is used as control qubit
# qubit 2 is used as ancilla qubit
ck.controlled(k, [1], [2])
```

Code Example 3. OpenQL Multi-qubit Controlled kernel.

is called universal if they can be used to construct a quantum circuit that can approximate any unitary operation to arbitrary accuracy,

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \quad T = \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}, \tag{1}$$

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \quad Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} \quad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}, \tag{2}$$

$$CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}. \tag{3}$$

It has been proven that any unitary operation can be approximated to arbitrary accuracy by using only single qubit gates such as given in Equations (1) and (2) and the CNOT gate, as given in Equation (3), which belongs to the Clifford+T library [34].

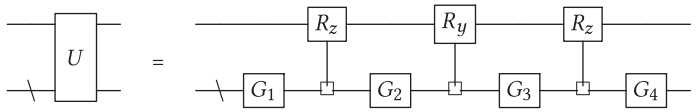

Fig. 5. Quantum shannon decomposition [44].

A unitary matrix is used to represent each quantum operation of our quantum circuit to enable decomposition and fusing of quantum operations. The unitary matrix representation of gates is a useful mathematical tool that allows the compiler to efficiently fuse quantum operations using simple matrix multiplications and Kronecker product computations. Combining quantum gates is particularly useful for reducing the number of quantum operations and thus the overall execution time of a quantum algorithm to perform the largest possible number of quantum operations within the coherence time of the qubits. For instance, combining a set of single qubit rotations can be cancelled out if their fusion is equivalent to an identity operation that can be removed from the quantum circuit.

Any quantum gate can be fully specified using a unitary matrix, and any unitary matrix can be decomposed into a finite number of gates from some universal set. In OpenQL, this is achieved using Quantum Shannon Decomposition [44] as show in Figure 5, which has been implemented using the C++ Eigen library [13]. The universal set of gates used are the arbitrary y-rotation, the arbitrary z-rotation and the controlled-not gate. The matrices for these are shown in Equations (3) and (4),

$$R_y(\theta) = \begin{bmatrix} cos\theta/2 & sin\theta/2 \\ -sin\theta/2 & cos\theta/2 \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}. \tag{4}$$

At each level of the recursion, a unitary gate U is decomposed into four unitary gates spanning one less qubit, and three uniformly controlled rotation gates. The latter are decomposed using the technique from Reference [33], and the algorithm is called again on the smaller unitary gates. This recursion continues until the one-qubit unitary gates can be implemented using ZYZ-decomposition [8].

For an n-qubit unitary, the decomposition results in $U(n) = 3/2 * 4^n - 3/2 * 2^n$ rotation gates and $C(n) = 3/4 * 4^n - 3/2 * 2^n$ controlled-not gates. These gates are added to the circuit and passed on to the next stages in the compilation.

6.2 Gate-Level Optimization

6.2.1 Gate Dependency Analysis. Once the quantum operations have been decomposed into a sequence of elementary operations, the gate dependency is analyzed and represented in the form of a Direct Acyclic Graph where the nodes represent the quantum gates and the edges the dependency between them. We refer to this graph as the **Gate Dependency Graph (GDG)**. Beside extracting any parallelism from the quantum circuit, the GDG allows reordering the gates with respect to their dependencies and helps with extracting local gate sequences that can be fused into smaller sequence of operations or even cancelled out if equivalent to an identity gate. This allows reducing the overall circuit depth and thus the algorithm execution time. The fidelity can also be greatly improved as more operations can be executed within the qubit coherence time.

Figure 6 shows the gate dependency graph of a quantum circuit and a potential gate sequence optimization. We note that without gate dependency analysis, some optimization opportunities can be missed as those gate sequences may be split into small scattered chunks that are not necessarily specified back-to-back in the original algorithm.

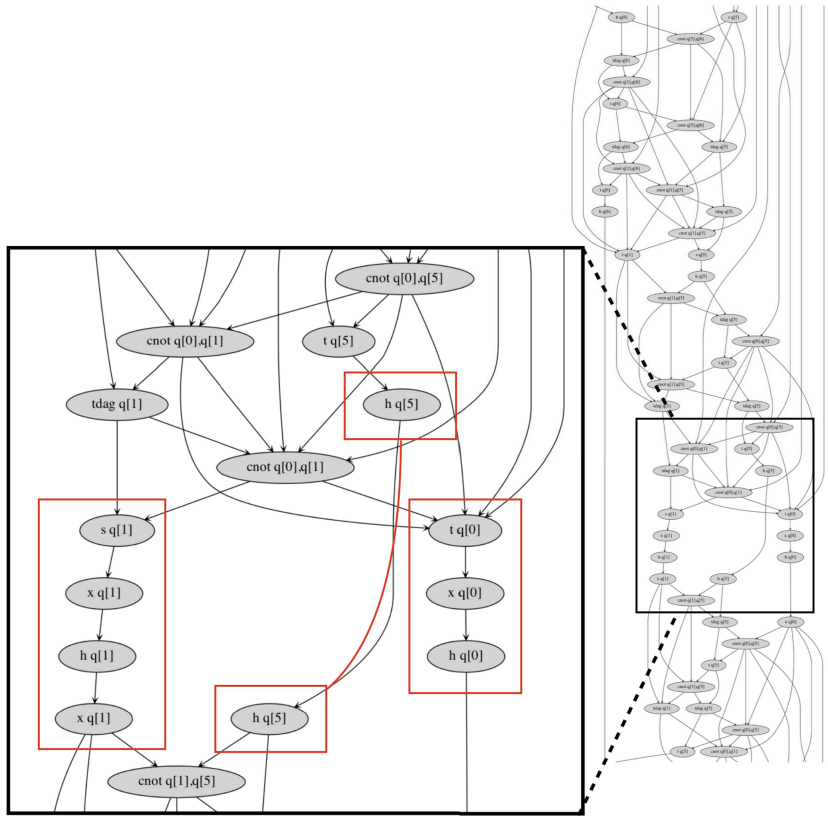

Fig. 6. Local optimization in gate-dependency graph: Local sequences of single qubit operations can be merged into smaller sequences of elementary operations or cancelled-out when equivalent to an identity gate. This can be done by using the unitary matrix representations of the gate and multiplying them to obtain a unitary matrix that is equivalent to the combined gates. The later could be an identity and thus removed or could be equivalent to a single gate that can replace the sequence of operations with that single gate or a shorter equivalent sequence.

6.2.2 Gate Sequence Optimization. Gate sequence optimization uses the unitary representation of quantum gates to approximate the overall unitary operation. For instance, the equivalent unitary operation of a sequence of quantum gates operating on the same qubit can be obtained through matrix multiplication. The equivalent operation could be (i) an identity that can be taken out of the circuit, (ii) an operation that can be implemented using a shorter sequence of elementary gates, and (iii) an operation that can be approximated using a shorter sequence of elementary operations. To control the accuracy of the compilation process, the compiler computes the distance between the target sequence of operation and the new set of elementary operations. The optimization will take place if that distance is smaller than the allowed error, which is specified as a compilation parameter that can be controlled by the user to achieve the desired accuracy.

OpenQL uses a sliding window over each sequence of gates to fuse locally quantum operations whenever possible. The size of the sliding window is critical to the compilation complexity, which grows linearly with the number of gates.

6.2.3 Gate Scheduling. Gate scheduling aims to use gate-dependency analysis to extract parallelism and schedule the operations in parallel while respecting dependencies. It uses the knowledge

of the duration of each gate as specified in the platform's configuration file to determine the exact time at which each gate can potentially start its execution.

The duration of each gate in the configuration file is specified in terms of a unit less number that is required to execute that gate on the specific target. This and all other times are divided by *cycle_time* and rounded up to convert these numbers in to clock cycles. A Clock cycle or a "click" is the basic unit of time for a quantum computer. In short, from the point of view of the quantum program developer, nothing happens in less than one clock cycle. The job of a compiler is to find the exact cycle in which an operation can be carried out. This is done in such away that all the dependencies are respected. Furthermore, as an optimization step, most of the time, compiler also tries to reduce the number of cycles required to execute all the quantum operations to reduce the total execution-time of the quantum program.

OpenQL gate scheduling can perform three types of scheduling: an **As Soon As Possible (ASAP)**, an ALAP or a Uniform ALAP.

- In an ASAP schedule, the cycle values are minimal but it may result in many gates being executed at the start of the circuit and thus longer cycles between successive gates operating on the same qubit, and thus a lower fidelity.
- At the other extreme, in an ALAP schedule the cycle values are maximal under the constraint that the total execution time of the circuit is equal to that of an ASAP schedule of the same circuit. But while at the start of the circuit relatively few gates are executed per cycle, at the end many gates will get executed on average. That they are executed as late as possible is good to get a higher fidelity but executing many gates per cycle may be more than the control electronics of the quantum computer was designed for, potentially leading to buffer overflows in that area and therefore to the requirement of a local feedback system to hold more gates off, effectively making execution time of a circuit longer.
- The Uniform ALAP schedule aims to produce an ALAP schedule with a balanced number of gates per cycle over the whole execution of the circuit. This scheduling scheme is based on Reference [21]. It starts by creating an ASAP schedule and then performs a backward pass over the circuit in an ALAP fashion: filling cycles with gates by moving them toward the end while respecting the dependencies.

Each of these three types of schedulers, dependencies and gate duration primarily determine the result. However, scheduler may need to respect more constraints, especially for the real targets. These constraints are mainly hardware constraints, for example those of control electronics, that limit the parallelism [25].

Using resource descriptions of those control electronics in the hardware configuration file, the gate scheduler optionally produces an ASAP, an ALAP or a Uniform ALAP schedule that respects these resource constraints. The main, and from a hardware design perspective, a crucial property of the resulting schedules is that hardware can execute gates in the cycles determined by the scheduler as in a Very Long Instruction Word processor, without the need of maintaining whether gates are ready, and so on. This significantly reduces the complexity and size of the hardware.

6.3 Mapping of Quantum Circuits

The OpenQL compiler also includes the Qmap mapper [25] that is responsible for creating a version of the circuit that respects the processor contraints. The main constraints include the elementary gate set, the qubit topology that usually limits the interaction between qubits to only nearest-neighbour and the control electronics contraints, e.g., a single Arbitrary Waveform Generator is used to operate in a group of qubits.

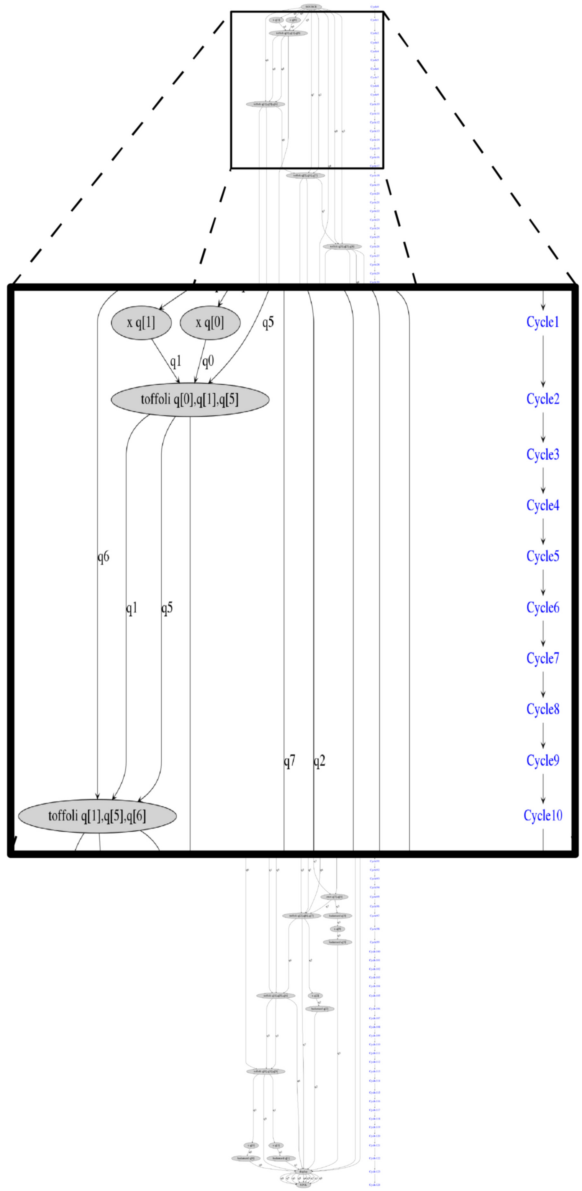

Fig. 7. Example of an ASAP scheduling of the 3-qubit grover algorithm.

To adapt the circuit to these quantum hardware characteristics, the Qmap mapper: (i) performs an initial placement of the qubits in which virtual qubits (qubits in the circuit) are mapped to the hardware qubits (physical qubits in the chip); (ii) it will move non-neighbouring qubits to adjacent positions to perform a two-qubit gate; and (iii) it will re-schedule the quantum operations respecting their dependencies and all hardware constraints. Note that it uses the hardware properties that are described in the configuration file.

The mapper aims to find the best qubit placement. Ideally, qubits can be placed in a way that all two-qubit interactions (two-qubit gates) present in the quantum program are allowed without need

of any movement. However, this is rarely the case when the program is designed without considering the placement beforehand. Often qubit routing is required to perform two-qubit operations between non-neighbouring qubits when the optimal placement does not allow direct interaction between them. From this perspective, qubit routing can be considered as a critical component of the qubit mapping, which allows OpenQL to resolve such conflicts.
OpenQL supports this by two algorithms, in sequence:

- Initial Placement: This first pass aims to find the optimal qubit placement in the target physical device to enable performing two-qubits operations at the lowest possible cost. Currently, OpenQL can detect where constraints violations and thus illegal operations on such two-qubit gates between non-neighbouring qubits appear. It tries to find a map of the qubits that minimizes the overhead and enables qubit interactions. The mapper does this by using an Integer Linear Programming algorithm as explained in Reference [26]. Such an approach works perfectly on smaller circuits but takes too much execution time on longer circuits because of exponential scaling.
- Qubit router: The second pass guarantees that two-qubit gate operations on non-neighbouring qubits can be performed by inserting a series of gates, e.g., SWAP gates that move qubits to neighbouring places. For each of such two-qubit gate operations, it determines the distance of those qubits and when too far apart, it evaluates all possible ways to make those qubits nearest neighbour. To do so, it evaluates all possible shortest paths and chooses the one that, for instance, results in the minimum increase of the circuit depth (number of cycles). Then, the corresponding 'move' operations are inserted in the program.

Note that after mapping the number of gates and the circuit depth will increase, increasing the failure rate thus reducing the algorithm's reliability.

6.4 Technology-Independent Compilation: cQASM

After gate decomposition, quantum circuit optimization or gate scheduling, a cQASM compiler is responsible for producing a technology-independent common quantum assembly code called cQASM. Currently cQASM 1.0 [24] is used to describe the circuit at the gate level and allows the user to simulate the execution of the quantum algorithm using the QX Simulator [22]. The simulation allows the programmer to verify the correctness of the quantum algorithm or to simulate and evaluate its behaviour on noisy quantum computing devices.

cQASM 1.0 aims to enable the description of a quantum circuit while abstracting away the hardware details, for instance, `H q[1]` describes a Hadamard gate on qubit `q[1]` without specifying the low level implementation of that quantum operation on a specific qubit technology. Besides the description of common quantum operations, cQASM 1.0 allows the specification of parallelism in the quantum circuit in the form of *bundles* (lists of gates starting in the same cycle) and *SIMD operations* (a gate operating on a range of qubits). This allows the OpenQL scheduler to express the parallelism that it found in cQASM 1.0.

cQASM 1.0 allows the naming of quantum circuit sections or "sub-circuits"; these sub-circuits correspond to the names of the quantum kernels and allow the user to relate the produced cQASM to its high-level algorithm written in Python or C++.

In the cQASM code Example 4, we see the scheduled code produced for the Grover search algorithm.

6.5 Technology-Dependent Compilation: eQASM

After compiling the technology-independent QASM code, the compiler generates the eQASM, which targets specific control hardware. The compiler uses different eQASM compilation

```
version 1.0

# define a quantum register of 9 qubits
qubits 7

# sub-circuit for state initialization
.init
    prep_z q[0:6]  # prepare all qubits in |0>
    x q[4]         # oracle qubit
    h q[0:4]       # parallel hadamard gates on qubits 0,1,2,3 and 4

# core step of Grover's algorithm
# loop with 3 iterations
.grover(3)

    # search for |x> = |1011>

    # oracle implementation
    x q[2]
    { toffoli q[0],q[1],q[5] | toffoli q[2],q[3],q[6] }
    toffoli q[5],q[6],q[4]
    { toffoli q[2],q[3],q[6] | toffoli q[0],q[1],q[5] }
    x q[2]

    # Grover diffusion operator
    { h q[0] | h q[1] | h q[2] | h q[3] } # parallel gates
    { x q[0] | x q[1] | x q[2] | x q[3] }
    h q[3]
    toffoli q[0],q[1],q[5]
    toffoli q[2],q[5],q[6]
    cnot q[6],q[3]
    toffoli q[2],q[5],q[6]
    toffoli q[0],q[1],q[5]
    h q[3]
    { x q[0] | x q[1] | x q[2] | x q[3] }
    { h q[0] | h q[1] | h q[2] | h q[3] }
    # display


# final measurement
.final_state
    h q[4]
    display   # display the quantum state when simulating in QX
.measurement_result
    measure_all
    display
```

Code Example 4. Grover Algorithm showing several gates scheduled to execute in parallel.

backends depending on the target platform specified in the hardware configuration file. The eQASM compiler can reschedule the quantum operations to exploit the available parallelism on the target micro-architecture and map the quantum circuit based on the topology of the target qubit chip and the connectivity of the control hardware.

6.6 Quantum Computer Micro-Architecture

OpenQL currently has several backends capable of generating eQASM for two different microarchitectures discussed in References [14, 16]. The backends convert the compiled cQASM code to

a specific eQASM code for the target microarchitecture with respect to the hardware constraints such as the available parallelism and the timing constraints.

6.6.1 Temporal Transformation: Low-level Scheduling. While the QASM-level scheduler pass extracts all the available gate-level parallelism, the target platform can have limited parallelism due the control electronic constraints. After analyzing the quantum gate dependencies, the compiler schedules the instructions either ALAP or ASAP with respect to the gate dependencies and cycle-accurate durations of the different gates.

6.6.2 Spatial Transformation : Connectivity-Aware Mapping. The OpenQL compiler maps the qubits with respect to the qubit plane topology which specifies the operation constraints such as nearest neighbour interactions or operation parallelism limitations. The current version of OpenQL relies on the two-qubit instruction specification in the hardware configuration file to extract the constraints, but the mapping task is being shifted to the mapping layer at the gate level, which will use a dedicated mapping specification in the hardware configuration file and more advanced mapping techniques. The targeted topology specification should provide enough abstraction to preserve the technology independence of the gate level compilation layers, and provide support for various technologies with different topologies.

6.6.3 eQASM Execution Monitoring. Tracing the various instruction executions and timings of the signals controlling the qubits is critical for debugging and monitoring the hardware. The OpenQL compiler generates auxiliary outputs for tracing purposes such as timed instructions and a graphical timing diagram as shown in Figure 8. In this timing diagram, both the digital and analog signals are shown with their respective starting time and duration. Each signal refers to both its originating eQASM instruction and the originating cQASM instruction with the precise execution clock cycle. When the compiler compensates for latencies in a given channel, both the original and the compensated timing are shown.

6.7 Hardware Configuration Specification: Control Electronics

To compile the produced QASM instructions into executable instructions (e.g., eQASM), the compiler needs to know not only the instruction set supported by the target microarchitecture but also the specification of all the constraints related the hardware resource usage, the operations timing and the qubits connectivity, and so on.

The hardware specification file aims to provide this information in an abstract way to allow describing different architectures and enable the compiler to adapt to their constraints and requirements when producing the executable code. This allows extending the compiler support to many architectures without fundamental changes in its upper technology-independent layers.

The hardware configuration file is specified in JSON format. A simplified example is shown in Listing 5 that depicts the hardware setup and lists all the supported operations. These operations are also accompanied with their settings such as the number of qubits, the time scale, the operations dependencies, their timing parameters, mathematical description, and associated instruction set.

The sections of the hardware configuration file are organized as follows:

- **eqasm_compiler**: This section specifies the eQASM compiler backend that should be used to generate the executable code. The allows the compiler to target different microarchitectures using the appropriate backend.
- **instructions**: In this section, the quantum operations supported by the target platform are described by their duration, their latency in the control system, their unitary matrix

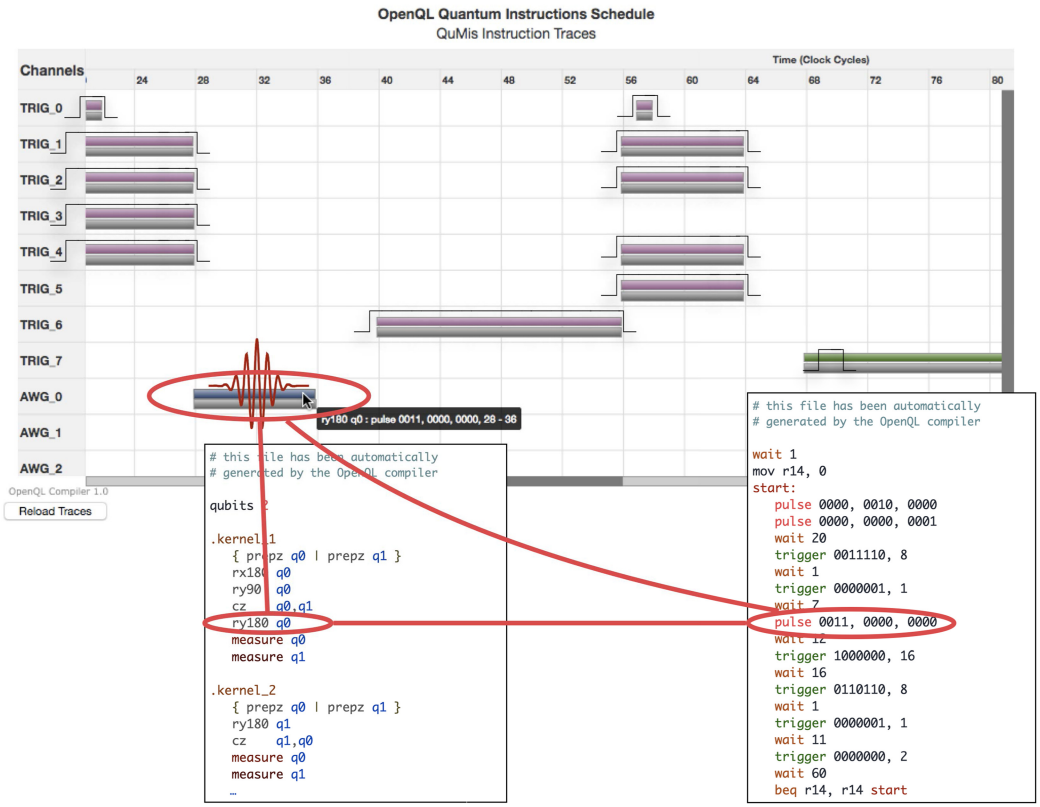

Fig. 8. Instruction timing diagram generated by OpenQL.

representation, their type (microwave, flux or readout) and finally microarchitecture-specific information to enable the compiler to generate the executable code.

- **Instruction Properties**
  - *duration* (int): duration of the operation in ns
  - *latency* (int): latency of operation in ns
  - *qubits* (list): list of affected qubits by this operation (this includes the qubits that are directly used or made inaccessible by this operation).
  - *matrix* (matrix): the unitary matrix representation of the quantum operation.
  - *disable_optimization* (bool): setting this field to True prevent the compiler from compiling away or optimizing the operation.
  - *type* (str): one of either 'mw' (microwave), 'flux', 'readout', or 'none'.
- **Microarchitecture Specific Properties**
  - *qumis_instr* (str): one of wait, pulse, trigger, CW_trigger, dummy, measure.
  - *qumis_instr_kw* (dict): dictionary containing keyword arguments for the qumis instruction.

- **gate_decomposition**: The gate decomposition section aims to describe the decomposition of coarse grain quantum operations into the elementary operations defined in the previous section. Each composite instruction in this section is defined by its equivalent quantum gate sequence. For instance, a CNOT gate can be described as: "ry90 q1","cz q0,q1","ry90 q1".
- **resources**: Describes the various hardware constraints that are used by the hardware constrained scheduling algorithm

- **topology**: Describes the qubit grid topology, i.e., qubits and their connnections for performing two-qubit gates

```
{
  "eqasm_compiler" : "qumis_compiler",

  "hardware_settings": {
        "qubit_number": 2,
        "cycle_time" : 5,
        "mw_mw_buffer": 0,
        "mw_flux_buffer": 0,
         ...
  },
  "instructions": {
     "rx180 q1" : {
        "duration": 40,
        "latency": 20,
        "qubits": ["q1"],
        "matrix" : [ [0.0,0.0], [1.0,0.0],
                     [1.0,0.0], [0.0,0.0] ],
        "disable_optimization": false,
        "type" : "mw",
        "qumis_instr": "pulse",
        "qumis_instr_kw": {
           "codeword": 1,
           "awg_nr": 2
        }
     },
     "rx180 q0" : {
        "duration": 40,
        "latency": 10,
        "qubits": ["q0"],
        "matrix" : [ [0.0,0.0], [1.0,0.0],
                     [1.0,0.0], [0.0,0.0] ],
        "disable_optimization": false,
        "type" : "mw",
        "qumis_instr": "codeword_trigger",
        "qumis_instr_kw": {
           "codeword_ready_bit": 0,
           "codeword_ready_bit_duration" : 5,
           "codeword_bits": [1, 2, 3, 4],
           "codeword": 1
        }
    },
     "prepz q0" : {
        "duration": 100,
        "latency": 0,
        "qubits": ["q0"],
        "matrix" : [ [1.0,0.0], [0.0,0.0],
                     [0.0,0.0], [1.0,0.0] ],
        "disable_optimization": true,
        "type" : "mw",
        "qumis_instr": "trigger_sequence",
        "qumis_instr_kw": {
           "trigger_channel": 4,
           "trigger_width": 0
        }
```

```
    }
  },

  "gate_decomposition": {
    "x q0" : ["rx180 q0"],
    "y q0" : ["ry180 q0"],
    "z q0" : ["ry180 q0","rx180 q0"],
    "h q0" : ["ry90 q0"],
    "cnot q0,q1" : ["ry90 q1","cz q0,q1","ry90 q1"]
  },

  "resources" : {
  },

  "topology" : {
  }
  ...
}
```

The operation duration, latency and the target qubits are used by the eQASM backend to analyze the dependencies of the instructions. This information is critical for different compilation stages, for instance the duration of an instruction and its qubit dependency is crucial for the low-level hardware-dependent scheduling stage that use these information to schedule the instructions.

The latency field is used by the backend compiler to compensate for the instruction latency by adjusting the instructions starting times to synchronize different channels with different latencies. Different latencies could exist in different control channels due to propagation delays through different cables, control latencies in waveform generators or readout hardware.

7 OPENQL APPLICATION

OpenQL has been used to program several experiments and algorithms on various quantum computer architectures and also on different qubit technologies, namely superconducting and semiconducting qubits.

7.1 Superconducting Qubit Experiments

We used OpenQL to compile quantum code and implement various experiments on several quantum chips with 2, 5, and 7 qubits using two different microarchitectures, namely QuMA 1.0 [16] and QuMA 2.0 [15], for controlling the qubits using two different instruction sets. Figure 9 shows the flow used to demonstrate OpenQL capabilities on a superconducting qubit platform located at QuTech in Delft. In particular, we implemented several standard experiments such as Clifford-based Randomized Benchmarking *RB* [31], *AllXY* [39] and other calibration routines, such as Rabi oscillation [39]. For each experiment, the same high-level OpenQL code has been reused on different setups and devices without changes, only the hardware configuration file has been changed to specify each target hardware setup and its constraints to instruct the compiler how to generate the appropriate code for each platform. Apart from the above basic experiments, OpenQL has also been used to compile code for the following applications:

(1) Net-zero two qubit gate [41]
(2) 3 qubit repeated parity checks [10]
(3) Variational quantum eigen solver [42]
(4) Calculating energy derivatives in quantum chemistry [35]

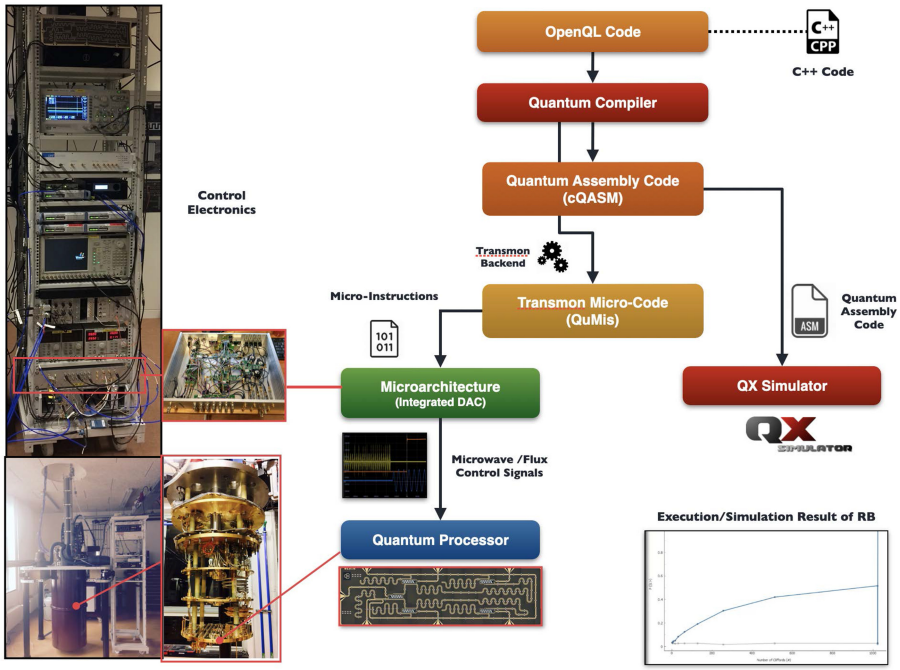

Fig. 9. OpenQL demonstration flow on a superconducting qubit processor in QuTech, TU delft.

7.2 Semiconducting Qubit

To evaluate the portability of OpenQL over different qubit technologies, the AllXY experiment has been reproduced on both superconducting qubit and semiconducting qubit devices using the same code and different configuration files. We used a Si-Spin qubit device [53] controlled by different control electronics, the hardware configuration file was changed to reflect the control setup and enable the compiler to automatically adapt the generated code to the target system: the compiler took into account the latencies of the different signal generators and measurement units involved in the setup and rescheduled all the quantum operations accordingly to compensate for those latencies and provide coherent qubit control.

8 CONCLUSION

In this article we presented the OpenQL quantum programming framework that includes a high-level quantum programming language and its compiler. A quantum program can be expressed using a C++ or Python interface and compiler translates this high-level program into a cQASM to target simulators. This program can further be compiled for a specific architecture targeting physical quantum computer. OpenQL has been used for implementing several experiments and quantum algorithms on several quantum computer architectures targeting both superconducting and semiconducting qubit technologies.

ACKNOWLEDGMENTS

The authors thank Prof. L. DiCarlo and his team for giving us the opportunity to test OpenQL on various Superconducting qubit systems and Pr. L. Vandersypen and his team for allowing us to test OpenQL on a Si-Spin qubit device. The authors thank all members of the Quantum Computer Architecture Lab at TU Delft for their valuable feedback and suggestions.

REFERENCES

[1] Ali Javadi Abhari, Arvin Faruque, et al. 2012. Scaffold: Quantum programming language. Princeton University TR-934-12, Princeton, NJ.

[2] A. J. Abhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi. 2015. ScaffCC: Scalable compilation and analysis of quantum programs. *Parallel Comput.* 45, C (June 2015), 2–17. (2015). https://doi.org/10.1016/j.parco.2014.12.001 arXiv:arXiv:1507.01902

[3] Héctor Abraham et al. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[4] Carmen G. Almudever, Lingling Lao, Xiang Fu, Nader Khammassi, Imran Ashraf, Dan Iorga, Savvas Varsamopoulos, Christopher Eichler, Andreas Wallraff, Lotte Geck, et al. 2017. The engineering challenges in quantum computing. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 836–845. https://doi.org/10.23919/DATE.2017.7927104

[5] Amazon. [n.d.]. Amazon Braket. Retrieved from https://aws.amazon.com/braket/.

[6] Amazon. [n.d.]. Amazon Braket SDK. Retrieved from https://github.com/aws/amazon-braket-sdk-python.

[7] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 32 (2013), 818–830.

[8] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. 1995. Elementary gates for quantum computation. *Phys. Rev. A* 52, 5, (Nov. 1995), 3457–3467. https://doi.org/10.1103/PhysRevA.52.3457

[9] S. Bettelli, T. Calarco, and L. Serafini. 2003. Toward an architecture for quantum programming. *Eur. Phys. J.* D 25 (2003), 181–200. https://doi.org/10.1140/epjd/e2003-00242-2

[10] C. Bultink, T. E. O'Brien, R. Vollmer, N. Muthusubramanian, M. W. Beekman, M. A. Rol, X. Fu, B. Tarasinski, V. Ostroukh, B. Varbanov, A. Bruno, and L. DiCarlo. 2020. Protecting quantum entanglement from qubit errors and leakage via repetitive parity measurements. *Science Advances* 6, 12, eaay3050 (2020). https://doi.org/10.1126/sciadv.aay3050

[11] David Deutsch. 1985. Quantum theory, the Church-Turing principle and the universal quantum computer. 400 (1985), 97–117.

[12] M. Fingerhuth. Open-Source Quantum Software Projects. Retrieved August 1, 2020 from https://github.com/qosf/awesome-quantum-software#quantum-compilers.

[13] Benoît Jacob (founder), Gaël Guennebaud (guru), and many more. 2019. The Eigen Documentation. Retrieved Aril 9, 2019 from http://eigen.tuxfamily.org/index.php?title=Main_Page.

[14] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. 2018. eQASM: An executable quantum instruction set architecture. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA'19)*.

[15] X. Fu, L. Riesebos, M. A. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, V. Newsum, K. K. L. Loh, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. [n.d.]. eQASM: An Executable Quantum Instruction Set Architecture.

[16] X. Fu, M. A. Rol, C. C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. F. L. Vermeulen, J. C. de Sterke, W. J. Vlothuizen, R. N. Schouten, C. G. Almudever, L. DiCarlo, and K. Bertels. 2017. An experimental microarchitecture for a superconducting quantum processor. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50'17)*. Association for Computing Machinery, New York, NY, USA, 813–825. https://doi.org/10.1145/3123939.3123952

[17] Lov K. Grover. 1997. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.* 79, 2 (1997), 325–328. http://link.aps.org/doi/10.1103/PhysRevLett.79.325.

[18] IBM. [n.d.]. IBM Quantum Experience. Retrieved from https://www.research.ibm.com/ibm-q/.

[19] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: A framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF'14)*. ACM, New York, NY, Article 1, 10 pages. https://doi.org/10.1145/2597917.2597939

[20] James Jeffers and James Reinders. 2013. *Intel Xeon Phi Coprocessor High Performance Programming* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA.

[21] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. ACM, New York, NY, 127–136. https://doi.org/10.1145/3174243.3174264

[22] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. L. M. Bertels. 2017. QX: A high-performance quantum computer simulation platform. *Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE'17)* (March 2017), 464–469.

[23] Nader Khammassi, Imran Ashraf, Xiang Fu, Carmen G. Almudéver, and Koen Bertels. 2017. QX: A high-performance quantum computer simulation platform. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*. IEEE, 464–469.

[24] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. 2018. cQASM v1. 0: Towards a common quantum assembly language. arXiv:1805.09607. Retrieved from https://arxiv.orb/abs/1805.09607.

[25] Lingling Lao, Daniel M. Manzano, Hans van Someren, Imran Ashraf, and Carmen G. Almudever. 2019. Mapping of quantum circuits onto NISQ superconducting processors. arXiv:1908.04226. Retrieved from https://arxiv.org/abs/1908.04226.

[26] L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. G. Almudever. 2019. Mapping of lattice surgery-based quantum circuits on surface code architectures. *Quant. Sci. Technol.* 4 (2019), 015005.

[27] Ryan LaRose. 2019. Overview and comparison of gate level quantum software platforms. *Quantum* 3 (Mar. 2019), 130. https://doi.org/10.22331/q-2019-03-25-130

[28] Seth Lloyd. 1996. Universal quantum simulators. *Science* 273, 5278 (1996), 1073–1078.

[29] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. 2006. GPGPU: General-purpose computation on graphics hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'06)*. Association for Computing Machinery, New York, NY, 208–es. https://doi.org/10.1145/1188455.1188672

[30] M. Fingerhuth. [n.d.]. Open-Source Quantum Software Projects. Retrieved from https://github.com/qosf/awesome-quantum-software#quantum-simulators.

[31] Easwar Magesan, Jay M. Gambetta, and Joseph Emerson. 2011. Scalable and robust randomized benchmarking of quantum processes. *Phys. Rev. Lett.* 106 (2011), 180504.

[32] Christopher Monroe and Jungsang Kim. 2013. Scaling the ion trap quantum processor. *Science* 339, 6124 (2013), 1164–1169.

[33] Mikko Möttönen, Juha J. Vartiainen, Ville Bergholm, and Martti M. Salomaa. 2004. Quantum circuits for general multiqubit gates. *Phys. Rev. Lett.* 93, 13 (Sep. 2004), 130502.

[34] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press, New York, NY.

[35] T. E. O'Brien, B. Senjean, R. Sagastizabal, X. Bonet-Monroig, A. Dutkiewicz, F. Buda, L. DiCarlo, and L. Visscher. 2019. Calculating energy derivatives for quantum chemistry on a quantum computer. arXiv:1905.03742. Retrieved from http://arxiv.org/abs/1905.03742.

[36] Selinger P. and Valiron B. 2005. A lambda calculus for quantum computation with classical control. In *Typed Lambda Calculi and Applications*. P. Urzyczyn (Ed.), Lecture Notes in Computer Science, Vol. 3461 (2005).

[37] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. Association for Computing Machinery, New York, NY, 846–858. https://doi.org/10.1145/3009837.3009894

[38] QuTech. [n.d.]. Quantum Inspire: The Multi Hardware Quantum Technology Platform. Retrieved from https://www.quantum-inspire.com/.

[39] Matthew David Reed. 2013. *Entanglement and Quantum Error Correction with Superconducting Qubits*. Ph.D. Dissertation. Yale University.

[40] P. Rogers. 2016. HSA overview. In *Heterogeneous System Architecture*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 7–18. https://doi.org/10.1016/B978-0-12-800386-2.00001-8

[41] M. A. Rol, F. Battistel, F. K. Malinowski, C. C. Bultink, B. M. Tarasinski, R. Vollmer, N. Haider, N. Muthusubramanian, A. Bruno, B. M. Terhal, and L. DiCarlo. 2019. A fast, low-leakage, high-fidelity two-qubit gate for a programmable superconducting quantum computer. arXiv:1903.02492. Retrieved from http://arxiv.org/abs/1903.02492.

[42] R. Sagastizabal, X. Bonet-Monroig, M. Singh, M. A. Rol, C. C. Bultink, X. Fu, C. H. Price, V. P. Ostroukh, N. Muthusubramanian, A. Bruno, M. Beekman, N. Haider, T. E. O'Brien, and L. DiCarlo. 2019. Error mitigation by symmetry verification on a variational quantum eigensolver. arXiv:1902.11258. Retrieved from http://arxiv.org/abs/1902.11258.

[43] Peter Selinger. 2004. Towards a quantum programming language. *Math. Struct. Comput. Sci.* 14, 4 (2004), 527–586. https://doi.org/10.1017/S0960129504004256

[44] V. Shende, S. S. Bullock, and I. Markov. 2006. Synthesis of quantum logic circuits. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 25 (July 2006), 1000–1010. https://doi.org/10.1109/TCAD.2005.855930

[45] Peter W. Shor. 1997. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.* 26, 5 (Oct. 1997), 1484–1509. https://doi.org/10.1137/S0097539795293172

[46] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A practical quantum instruction set architecture. arXiv:1608.03355 [quant-ph]. Retrieved from https://arxiv.org/abs/1608.03355.

[47] Robert S. Smith, Eric C. Peterson, Mark G. Skilbeck, and Erik J. Davis. 2020. An spen-source, industrial-strength optimizing compiler for quantum programs. arXiv:2003.13961 [quant-ph]. Retrieved from https://arxiv.org/abs/2003.13961.

[48] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *Quantum* 2 (Jan. 2018), 49. https://doi.org/10.22331/q-2018-01-31-49

[49] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Workshop on Real World Domain Specific Languages (RWDSL'18)*. https://doi.org/10.1145/3183895.3183901

[50] Texas Instruments. [n.d.]. OMAP3530 Application Processors. Retrieved from http://www.ti.com/product/omap3530.

[51] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. K. Kuzmanov, and E. Moscu Panainte. 2004. The molen polymorphic processor. *IEEE Trans. Comput.* 53 (2004), 1363–1375.

[52] Richard Versluis, Stefano Poletto, Nader Khammassi, Brian Tarasinski, Nadia Haider, David J. Michalak, Alessandro Bruno, Koen Bertels, and Leonardo DiCarlo. 2017. Scalable quantum circuit and control for a superconducting surface code. *Phys. Rev. Appl.* 8, 3 (2017), 034021.

[53] T. Watson, Stephan Philips, E. Kawakami, D. Ward, P. Scarlino, M. Veldhorst, D. Savage, M. Lagally, Mark Friesen, Susan Coppersmith, M. Eriksson, and L. Vandersypen. 2018. A programmable two-qubit quantum processor in silicon. *Nature* 555 (03 2018). https://doi.org/10.1038/nature25766

[54] Xilinx. [n.d.]. Zynq-7000 All Programmable SoC. Retrieved from http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.

[55] Mohamed Zahran. 2016. Heterogeneous computing: Here to stay. *Queue* 14, 6 (Dec. 2016), 31–42. https://doi.org/10.1145/3028687.3038873

[56] Margherita Zorzi. 2016. On quantum lambda calculi: A foundational perspective. *Math. Struct. Comput. Sci.* 26, 7 (2016), 1107–1195. https://doi.org/10.1017/S0960129514000425

[57] Bernhard Ömer. 1998. *A Procedural Formalism for Quantum Computing*. Technical Report.

Received November 2020; revised March 2021; accepted July 2021