On the Impact of Affine Loop Transformations in Qubit Allocation

MARTIN KONG, The University of Oklahoma, USA

Most quantum compiler transformations and qubit allocation techniques to date are either peep-hole focused or rely on sliding windows that depend on a number of external parameters including the topology of the quantum processor. Thus, global optimization criteria are still lacking.

In this article, we explore the synergies and impact of affine loop transformations in the context of qubit allocation and mapping. With this goal in mind, we designed and implemented **AXL**, a domain specific language and source-to-source compiler for quantum circuits that can be directly described with affine relations. We conduct an extensive evaluation spanning circuits from the recently introduced QUEKO benchmark suite, eight quantum circuits taken from the literature, three distinct coupling graphs, four affine transformations (including the Pluto dependence distance minimization and Feautrier's minimum latency algorithms), four qubit allocators, and two back-end quantum compilers. Our results demonstrate that affine transformations using global optimization criteria can cooperate effectively in several scenarios with quantum qubit mapping algorithms to reduce the circuit depth, size and allocation time.

CCS Concepts: • **Hardware** → **Emerging tools and methodologies**; **Emerging languages and compilers**; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Polyhedral model, affine transformations, quantum computing, qubit allocation

ACM Reference format:

Martin Kong. 2021. On the Impact of Affine Loop Transformations in Qubit Allocation. *ACM Trans. Quantum Comput.* 2, 3, Article 9 (September 2021), 40 pages.
<https://doi.org/10.1145/3465409>

1 INTRODUCTION

The field of **Quantum Computing (QC)** has made tremendous advances in the past two decades at the hardware (e.g., ion trap and superconducting QC), algorithmic (QFT [7, 20, 22, 41], Grover's Search [15, 40, 60], Shor's algorithm [10, 62, 68, 84, 85]), and software levels [24, 69, 79]. Known quantum algorithms already provide us with a glimpse of their expected exceptional complexity. Thus, it is imperative for a programming language to be a vehicle for algorithmic specification rather than an obstacle in the path to progress. To address and bridge the semantic gap between algorithm specification and quantum architectures, several languages, compilers, and frameworks have been proposed. Examples of these are ProjectQ [94], Scaffold and the ScaffCC compiler [49],

Author's address: M. Kong, The University of Oklahoma, School of Computer Science, 110 W. Boyd St., Devon Energy Hall, Rm. 150, Norman, OK 73019; email: mkong@ou.edu.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike International 4.0 License.

© 2021 Copyright held by the owner/author(s).

2643-6817/2021/09-ART9 \$15.00

<https://doi.org/10.1145/3465409>

Quipper [36], Microsoft's Q# DSL [95], and SIMD approaches such as Reference [44], or approaches focused on safe uncomputation such as SILQ [11].

Ultimately, the high-level programming language produces a stream of quantum assembly operations [12, 24], at which point *Qubit Allocation*, a technique akin to classic register allocation [16, 75], is applied to find a space-time mapping of the quantum gate operations in the program to the quantum device. Qubit allocation techniques typically decompose the input program into (network) layers and generally suffer from limitations such as approximating the global solution from local optima [46], use relative small sliding windows [65], leverage random initial mappings [46, 89], or incur in high time and space complexity due to exponentially large search spaces [43, 110].

The main goal of this work is to explore and understand potential synergistic interactions between affine loop transformations and qubit allocation techniques to find scenarios where the power of a global optimization criteria can effectively improve the quality of the qubit allocation. Our evaluation shows that even state-of-the-art allocators such as **sabre** [65] can improve by as much as 34% with classical polyhedral loop transformations, while other techniques less computationally demanding (i.e., **wpm** [89]) can improve by up to 60%.¹

In summary, this article makes the following contributions:

- Inspired in the Omega calculator [51] and ISCC's [105] notation, we introduce **AXL** (the Affine Circuit Specification Language), a simple domain specific language based on polyhedral abstractions to enable the description, manipulation and composition of quantum circuits.²
- We discuss how the polyhedral model can be used as an efficient intermediate representation for the optimization of affine quantum circuits. In particular, we highlight how we can use it to represent quantum networks.
- A consolidated set of quantum circuits collected from the literature, implemented in **AXL**, and made available as a new benchmark suite, **Ax1Bench**.
- We conduct an extensive study to understand the interactions of affine loop transformations with qubit allocation techniques. Our evaluation encompasses quantum circuits from **Ax1Bench** and from the QUEKO benchmark suite [97], four qubit allocators, three topologies, three affine transformations (in addition to the pass through code generation mode), and two quantum compilers (Qiskit and Enfield).
- The evaluation in itself represents a methodology that future studies exploring loop transformations and qubit allocation can apply.

The rest of this article is organized as follows: Section 2 develops the motivations for this study. Section 3 recaps the necessary quantum terminology and background. In particular, we briefly recap several quantum allocators recently introduced and which we use in our evaluation. Section 4 presents an overview of our language, **AXL**. Section 5 revises the polyhedral background while introducing a simple domain-specific language for affine quantum circuits. Section 6 discusses our extensive evaluation, general results, individual analyses, scalability, and density tests. We conclude this article with the related work (Section 7) and the final remarks in Section 8.

2 MOTIVATION

To motivate the relevance and impact of high-level affine frameworks, both their representational advantages and their transformation power, consider the `parity_247` circuit of Figure 1 [80] and its execution on a quantum device with a linear coupling graph with six physical qubits such as the one in Figure 2.

¹**sabre**'s average improvement over **jku** for large circuits is 14%, reported in Reference [65].

²An earlier iteration of this article is available in Reference [54].

Fig. 1. parity_247 circuit [80].

Fig. 2. Linear coupling graph.

Topological restrictions on current quantum devices impose that two-qubit gates must be applied to physically adjacent qubits [65, 89, 97, 110]. To meet these constraints, quantum SWAP operations can be introduced to interchange the state of qubits with the goal of moving closer the qubits holding the state of gate operands. Depending on the underlying device, SWAP operations can be natively supported or implemented by a sequence of three back-to-back CNOT.

In this example, deciding where to place the parity qubit (logical qubit q_0 in the figure) is the key, as it directly affects where will the control qubits be placed. For instance, if the target qubit is assigned to either q_1 or q_6 in Figure 2, then the NOT operations will have to be mapped to the remaining 5 qubits, causing 4 of the 5 CNOT to require SWAP operations (Case A). As a result, if the SWAP operation is not natively supported, 12 additional CNOT gates (3 for each non-adjacent CNOT) would be required, increasing the circuit's gate count from 10 gates to 22 gates. If, however, the parity qubit is placed on any physical qubit other than q_1 and q_6 , then two CNOT would meet the adjacency constraints, reducing the number of SWAPs to 3, and requiring only 9 additional CNOT (Case B).

Several other logical-to-physical qubit mappings are also possible, as well as variations of the initial assignment of quantum gates to qubits, and which can also affect the circuit depth (i.e., the number of gates in the critical path). For example, consider a third scenario (Case C), where all the CNOT gates are assigned to physical qubits q_5 and q_6 (q_5 being the control qubit); it would be possible to map all the NOT operations (being dependences to the control qubit) to physical qubit q_5 as long as the NOT gates on logical qubits $q_{2:5}$ are preceded by a SWAP operation between their original state and logical qubit q_1 . This scenario will execute all gates in a serial fashion, including the SWAP operations. Consequently, the circuit depth will be either 22 (if a built-in SWAP is not supported) or 14 gates (as 5 CNOT, 5 NOT, and 4 SWAP) if a native SWAP is available. The reader can observe that Cases A and B still preserve the inherent parallelism of the NOT gates, which can be executed in parallel when mapped to distinct qubits. In these two cases, the SWAP operation will be applied after each NOT gate.

This sort of decision can be better made with a better picture of the dependences among quantum operations in a steady state. For instance, consider the 3 legal gate stream variations of the parity_247 circuit shown in Listing 1:

```
Stream#1: X 1; X 2; X 3; X 4; X 5; CX 1,0; CX 2,0; CX 3,0; CX 4,0; CX 5,0;
Stream#2: X 1; CX 1,0; X 2; CX 2,0; X 3; CX 3,0; X 4; CX 4,0; X 5; CX 5,0;
Stream#3: X 5; X 4; X 3; X 1; CX 1,0; X 2; CX 2,0; CX 3,0; CX 4,0; CX 5,0;
```

Listing 1. Three equivalent gate streams for parity_247 circuit; X and CX denote NOT and CNOT gates, respectively. First argument of CX is the control qubit index, while the second argument is the target qubit.

```

1  param M;
2  statement S1, S2, S3;
3  S1 := {t:1 <= t <= M (%) #NOT(t) (+) #CNOT(t,0)};
4  S2 := {t:1 <= t <= M (%) #NOT(t) };
5  S3 := {t:1 <= t <= M (%) #CNOT(t,0)};
6  codegen {S1} with {M=8} apply {plutomax};
7  codegen {S2 (+) S3} with {M=8} apply {plutomin};

```

Fig. 3. **AXL** implementations of circuit parity₂₄₇.

The difference among the three gate streams shown resides in the degree of regularity, parallelism exposure, and the proximity to dependences. One can easily observe that Stream#1 exposes early gate parallelism (X operations applied to different qubits), but also delays the most the first CX operation; Stream#2 sacrifices some of the obvious early parallelism in exchange for earlier dependency exposure, information that can be leveraged by the underlying qubit allocator to decide on better logical-to-physical qubit mappings as well as better initial assignment of gates to qubits; Stream#3 provides an irregular view of the circuit, with three dependences having the longest possible delay between their production and consumption. This can significantly impact early decisions made by the underlying qubit allocator and scheduler if relatively small search windows are used.

The relevance of polyhedral abstractions, our proposed language (**AXL**), and affine transformations is their ability to compress quantum gate traces and optimizing them (scheduling in this case) at a coarser granularity. Our definition of coarser granularity is essentially “quantum operations prescribed by the same set of affine relations.” Our new language, **AXL**, permits to directly describe groups of gates that behave similarly, effectively lifting groups of gates to have “statement semantics” surrounded by one or more loops. Such groupings are equivalent to network layer decompositions performed by qubit allocators [89, 110], but with the addition of them being denser and more regular. Consequently, they can be used as *pre-transformations* that produce structured information for the qubit and gate mappers.

To illustrate our point, we show in Figure 3 two **AXL** implementations of the parity₂₄₇ circuit. The first one, statement 1 (S1) uses a single statement to model the entire circuit, embedding both gates in its body. The (+) *time concatenation* operator synchronizes the NOT and CNOT operations for every $t \in \mathcal{D}^{S1} = \{1 \leq t \leq M\}$. In contrast, the second implementation uses two statements (S2 and S3), and puts each gate in its own statement. Lines 6–7 perform code generation from each **SCoP (Static Control Part)** while providing non-default parameters and choosing a pre-determined transformation of choice. The second implementation imposes the synchronization of both statements also by the *time concatenation* operator (+) and internally creates the pertaining polyhedral abstractions.

Describing quantum programs in **AXL** and with polyhedral abstractions permits us to schedule and optimize gates that behave similarly and that are “close enough.” This reduces the burden of the underlying qubit allocator and mapper, specifically when decomposing the program into network layers. Moreover, as only a small subset of layers and operations are used to make the initial mapping decisions, facilitating a representative view of the dependences allows the mapper to make better decisions. For example, applying the **pluto-minfuse** heuristic would schedule all the NOT operations before the CNOT ones. Therefore, an allocator may only decide on an initial mapping based on detecting NOT operations that do not necessarily affect later communication and ignoring their impact on subsequent CNOTs. In contrast, if the **pluto-maxfuse** heuristic is used, then each CNOT operation will be scheduled immediately after the NOT operation on which it depends. This gives a clearer picture and more representative behavior of the circuit in steady state.

Similarly, all (irregular) variations where dependent CNOTs are scheduled arbitrarily far from their NOT dependence will also fail to provide a reliable view of the circuit's steady state. It is not hard to envision this scenario repeating multiple times throughout large enough circuits.

The relevance of loop fusion heuristics, such as **pluto-minfuse** and **pluto-maxfuse**, as well as the complete spectrum of loop fusion/distribution structural variants in between, is that they affect the initial layout mapping produced by the qubit allocator. The assumption here is that the initial mapping will use some fraction of the input QASM (e.g., a few network layers) to derive the necessary information and decide on a good mapping that will help to minimize communication (SWAP operations) while reducing the circuit's depth and other metrics. If, however, the allocator utilizes a fully random initial mapping, then the potential high-level information will be immediately lost. Similarly, if the allocator devotes a significant computational effort to produce the initial mapping, perhaps by doing exhaustive searches, then the high-level information provided by the polyhedral compiler could be rendered redundant.

Consider, for instance, the `parity_247` circuit, Figure 3. When applying **pluto-minfuse** to this benchmark, all the NOT operations will precede all of the CNOT. Then, for large enough circuits or for short enough search windows, an allocator could decide to map all the NOT operations to non-adjacent physically distant qubits, without knowledge that their location will also affect the communication required for the subsequent CNOT operations. In contrast, when leveraging **pluto-maxfuse**, NOT and CNOT operations will be interleaved, granting the allocator a better view of the computational pattern and seeing that every NOT operation is a dependence of a CNOT. Deciding which of the heuristics will be most beneficial is, of course, dependent of the computational pattern.

To further motivate this study, we focus our attention on the `sum5` circuit, part of a quantum adder taken from Reference [96]. Figure 4 shows the gate count produced and the mapping time required by four qubit allocator algorithms recently introduced: the **ibm-mapper** [46], the **jku** algorithm [110], **wpm** [89], and **sabre** [65]. We evaluate three coupling graphs (multi-ring, tiled, and grid) that describe and vary the connectivity among qubits in a quantum processor (see Section 6 for their description), and four semantically equivalent circuits produced with our domain-specific language (see Section 5) by changing the high-level affine transformation applied to the circuit. The four circuit variants generated with our compiler are **base**, **Feautrier**, **pluto-minfuse**, and **pluto-maxfuse**. The four variants exhibit different properties, such as min/max-imize the dependence distance among operations in a producer-consumer relation, eagerly satisfying dependences as early as possible, or in *pass-through mode* but still exploiting the efficient code generation with minimal control overhead of polyhedral compilers. Metrics were collected with the Enfield compiler [86, 87], which implements several qubit allocators. In the three sub-plots, the x-axis represents different configurations of {topologies x qubit allocator x affine transformation}. In particular, in the two top plots, the percentages shown at the top of each cluster represent the maximum percentual variation among affine transformations within the cluster. Error bars are included where appropriate.

The relevance of the topology in the allocation process is that the in/out degree of qubits (nodes in the graph) constrains where can multi-qubit gate operations be placed. Allocators search for suitable space-time mappings, decide when and where to place gate operations on the coupling graph [89] by using expensive search heuristics and narrow window sizes. Affine transformations can support this task by finding global optima solutions with various properties (e.g., maximal fusion of loops).

We can observe that **jku** and **sabre** typically fare best, as they are two state-of-the-art qubit allocators. However, their computational cost (in time) can vary both with the affine transformation applied and the topology of the quantum graph. This is evidenced in the bottom plot of Figure 4. For

Fig. 4. Impact of Affine Loop Transformations on sum₅ circuit: circuit depth (top), gates added during the mapping process (middle), and mapping time in seconds (bottom) on topologies grid, multi-ring, and tiled.

instance, these two allocators are nearly an order of magnitude faster than the **ibm-mapper** implementation in Enfield. Similarly, the quality of the mapping produced by a qubit allocator can be measured with two metrics: the *circuit depth* and the *number of gates introduced* by the allocator. In both cases, lower is better. Figure 4 shows that **jku** produces circuits 20% shallower than **sabre** on the tiled topology with the **base** and **pluto-minfuse** variants and roughly the same on the grid and multi-ring topologies. From the perspective of the computational cost, **jku** is 2.5×–3× faster than **sabre** on tiled and grid topologies. We also note that both **jku** and **wpm** are deterministic in nature, producing always the same mapping for the same input. In contrast, the **ibm-mapper** and **sabre** use randomized initializations that can affect, in some cases substantially, the quality of the qubit mapping. It is also interesting to observe that the {grid, **wpm**, **pluto-maxfuse**} configuration yields a circuit depth 35% larger than the average in {grid, **sabre**} at an extremely low computational cost. In the longer term, this tradeoff will be even more valuable as qubit allocators reach their scaling limit. (NOTE: The total number of gates required by a circuit varies with the number of gate SWAP operations introduced by the allocator to meet physical qubit constraints such as gates using physically connected qubits.) We can also observe that the configuration {multi-ring, **wpm**, **base**} is nearly as good as most **jku** and **sabre** mappings in terms of circuit depth while being over an order of magnitude faster. This is an example of how affine transformations in tandem with less computationally demanding qubit allocators can compete with more powerful ones.

Thus, there are essentially three motivations for performing this study. First, it is of interest to the community to explore high-level loop transformations that can work in tandem with the underlying qubit allocator and quantum compiler to produce better mappings. Second, in the scenarios where better mappings are not possible, to improve the computational cost by narrowing the search window of qubit allocators. Third, while having an impact on qubit mapping, allocation and gate scheduling, representing quantum circuits with polyhedral abstractions has the extra benefit of compressing quantum gate traces (straight line quantum assembly operations) [52, 81], making computationally cheap qubit allocators almost as good as more demanding ones. In particular, our findings demonstrate that, in some scenarios, maximizing parallelism is more beneficial, while in others minimizing the communication distance is more relevant. Both of these traits are properties heavily studied in loop transformations for HPC [14, 17, 26, 70, 77].

3 BACKGROUND IN QUANTUM COMPUTING

The appeal of QC is its ability to accelerate calculations by leveraging properties of quantum physics. QC is fundamentally based on *qubits*, quantum units of information akin to bits in the classical RAM model. In contrast to classical bits that are either 0 or 1, qubit states are linear combinations of $|0\rangle$ and $|1\rangle$ states, i.e., an n -qubit vector can simultaneously represent 2^n states. This is referred to as *superposition*. Classical answers are produced in quantum systems by the act of *measurement* of qubits, which causes qubits to lose their superposition state and collapse probabilistically into one of the component states. Another relevant property for quantum systems is *entanglement*; namely, the state of entangled qubits are inter-dependent, which allows for rich mechanisms for computing and communicating state. Finally, the *no-cloning* property says that no qubit state can be perfectly copied, so QC systems do not have an analog for classical “fanout” or accurate data copies [21, 42].

Qubits. Quantum bits are the basic unit of information of quantum programs and are analogous to classical bits. However, unlike their classical counterpart, which can only take the values in the set $\{0,1\}$, qubit values take the form of linear combinations of two basis states ($|0\rangle$ and $|1\rangle$).

Gates and Measurement. Are the elementary operations applied to qubits. Their role is to evolve the state of qubits. More generally, quantum gates can be seen as unitary matrix operations applied to vectors representing quantum states. Current quantum technologies utilize gate operations with 1 and up to 3 qubits. An example of a single qubit operation is the NOT (X) gate, which negates the state of a single qubit. An example of a two-qubit gate is the CNOT (Controlled-NOT or CX) operation, which reverses the state of the second qubit operand when the first one is 1. An example of a 3-qubit gate is the **CCNOT (Controlled-Controlled-NOT or Toffoli)**, which utilizes two control and one target qubits.

Coupling Graphs. A quantum processor can be conceptually conceived as a graph/network where the vertices are the qubits and the edges the communication links between them. Computational steps (gates) performed in the network are synchronized in time. Qubits serve as inputs and outputs to each quantum operation. A current limitation of quantum computing technology is that multi-qubit gates can only be applied to qubits directly connected by a link [61, 109].

Quantum Circuits. Quantum programs can be graphically represented in the form of circuits [99]. We show an example of a simple circuit obtained from the Revlib online repository [80] in Figure 1. The x-axis signifies time, while the y-axis are the qubits available in the quantum processor. It uses 6 qubits, 5 NOT gates (left-most), and 5 CNOT gates. Each NOT gate is synchronized with the control qubit of a CNOT gate. Operations on the same qubit lane, from left-to-right execute one-after another, and embody classic data-flow input/output dependences.

The actual semantics of the specific gate determine if the state of a qubit q_i is read, written, or both. The *depth* of a quantum circuit is the maximum number of gates scheduled on any single qubit lane, whereas the total number of gates is the *circuit size*. The quantum circuit ultimately defines the unitary evolution of the input (initial) state into the final/output state.

Qubit Allocation. Is the space-time mapping (assignment) of quantum operations to qubits in the coupling graph and is very similar in spirit to the classical register allocation problem. Recently several qubit allocators have been proposed. We briefly summarize a few of the most relevant techniques.

The **ibm-mapper** [46] (available in Qiskit) divides the input into a sequence of layers using disjoint sets of qubits; qubits within layers are mapped by minimizing the sum of squared distances among vertices and potentially inserting SWAP operations. In addition, each distance term is scaled by a factor $1 + r$, where r is a random number between 0 and 1. The algorithm defaults to one gate per layer if a valid mapping is not found.

wpm [89] is a heuristic that finds first an initial allocation maximizing the number of control dependences, followed by a second pass that completes any remaining ordering constraint, potentially inserting SWAP operations. Siraichi et al. [89] also proposed an exact solution to the qubit allocation problem as a dynamic program with state memoization, and showing that it leads to a $O(|Q|^{1/2} \cdot |Q| \cdot |D|)$, where Q is the set of (physical) qubits, and D the list of dependences. The heuristic proposed (**wpm**) was shown to achieve $O(|Q| \cdot \lg(|Q|) + |E| + |D| + |D| \cdot (|Q| + |E|))$.

jku [110] uses the circuit size as the main optimizing metric, decomposing the input network into layers, and attempting to minimize the number of gates within each layer. To avoid falling into local optima, **jku** uses an A* search algorithm [43], a family of graph traversal algorithms known to incur in $O(b^d)$ space complexity, where b is the branching factor in a tree and d its depth. The heuristic used to drive the search attempts to convert the mapping of each layer into the subsequent one by inserting SWAP operations.

sabre [65] is an efficient algorithm with $O(N^{2.5}g)$ time complexity that attempts to minimize both the circuit depth and size, usually exploring a tradeoff between them. As previous techniques, it divides the input program into layers, but performs two additional passes (a back-traversal and a second forward traversal) to improve the initial (random) mapping.

(De)Coherence. One of the main challenges facing QC is the decoherence problem, where the state of a qubit decays over time. Each quantum gate has a specific decoherence time, which for state-of-the-art quantum machines using superconducting technology is approximately $100 \mu s$ [79]. In addition, gate operations can also introduce errors at rates varying between $O(10^{-3})$ for single-qubit gates and $O(10^{-2})$ for two-qubit gates. The execution time of a quantum circuit results from the aggregated time needed to run all the gates along the circuit's depth. Thus, minimizing this metric is one of the main optimizing criteria. Likewise, minimizing the total number of operations in circuits also equates to reducing the compounded error.

4 AXL OVERVIEW

To illustrate our language and compiler, we show in Figure 5 a quantum circuit taken from the literature [33]. Quantum circuit diagrams depict the structure and relations among quantum gates and operations; the x-axis is time while the y-axis represents the various qubit (including *ancilla* or temporary qubits). By convention, each quantum gate operation is represented by different node shapes, and involving one or more qubit lines. If the gate only “touches” a single qubit, then it is a single-qubit operation. Typical quantum operations can require reading and updating two and even three qubits. For instance, the solid circle means that the gate is reading a particular qubit, while the $\oplus$ represents a NOT gate; combined, this is the Controlled-NOT gate operation, which

Fig. 5. **AXL** input code for Multi-CNOT benchmark (left) and corresponding circuit diagram (right).

negates the state on the $\oplus$ qubit when the control qubit is 1. More complex gates can be constructed via a set of more elementary ones.

AXL leverages the concept of *affine quantum circuits*; multiple gate instances can be grouped into *circuit statements*, first-class citizens in our language. Programs can be parameterized by symbols (e.g., N), and manipulate an implicit quantum register. The actual size of the quantum register is determined by the lexicographic maximum value among all the arguments passed to the quantum gate operators. Our example defines five affine circuit statements and uses some of **AXL**'s unique features, circuit reversal. We show in the figure how the gates are grouped into statements. In particular, the gates surrounded by blue solid and dotted lines (statements $S3$ and $S5$) are the reversals of statements $S1$ and $S4$ (surrounded by red lines), respectively. Statement $S2$ (green) consists of a single gate, but is used twice during the circuit composition phase. A thing to note here is how are statement circuits composed to construct larger circuits. The codegen construct allows the user to specify the lexicographic order of the sub-circuits. As the circuit in our example consist of only 1-dimensional iteration domains, it suffices to list the statements from left-to-right to establish the ordering constraints. Intuitively, this makes all statement instances of $S1$ to be executed before the single instance of statement $S2$, all instances of $S3$ to execute before all instances of $S5$, and so on. More complex shapes are possible by simply parenthesizing the composition expression.

Once the **circuit statements** and their bodies are defined, **AXL**'s **codegen** primitive assembles the quantum program (in polyhedral terminology, it constructs the SCoP) and assigns specific parameter values. In addition, it allows the user to apply individual transformations and algorithms to the chosen circuit. For this particular example, we show the application of the Pluto algorithm [13] with the *minimum fusion* (nofuse) heuristic. Although we do not use it in this example, the Feautrier scheduling algorithm (provided via ISL [104]) can be used instead of Pluto in the apply clause.

This program is then compiled with **AXL** to produce a transformed C-code, which is later compiled into a binary, and executed. The output is straight-line code consisting uniquely of quantum gate operations. This stream of quantum gates is fed to a back-end quantum compiler (e.g., Qiskit) to perform a simulation, to render a circuit diagram, or to compute a qubit allocation/mapping for the quantum circuit.

5 THE POLYHEDRAL MODEL IN QUANTUM COMPUTING

In this section, we quickly revise the four polyhedral abstractions in the context of (affine) quantum circuits. We then describe our domain-specific language, **AXL**, heavily inspired in the Omega Calculator [51] and ISCC [105] notation. However, unlike its predecessors, one of its main goals is to facilitate and capture the identity schedule of the quantum circuit. We explain how the circuit structure and dependences are mapped to the polyhedral abstractions and quickly recap well-known affine loop transformations previously proposed.

5.1 Polyhedral Abstractions

Polyhedral compilers focus on fragments of programs that exhibit static control parts (SCoPs) [28–30]. From these, four abstractions are extracted: iteration domains, access functions, dependence polyhedra, and scattering/scheduling functions [38].

Iteration Domains: Each syntactic statement S is associated to the set of points $\mathcal{D}^S$ in $\mathbb{Z}^+$ comprised by the dynamic instances of the statement. In the context of QC, an iteration domain can group several operations of the same type or of different ones, but that behave in the same fashion. Consider for instance the `parity_247` circuit shown in Figure 1, which can be represented with two iteration domains, $\mathcal{D}^{S^1} = \{[i] : 0 \leq i < N\}$ and $\mathcal{D}^{S^2} = \{[i] : 0 \leq i < N\}$, one for the NOT operations and another for the CNOTs; where N is an unknown but fixed value that parameterizes the circuit. We also note that several techniques have been developed to model and extend the applicability of the polyhedral model to different forms of irregular computations [6, 100, 101]. The remaining three abstractions are essentially functions applied to the iteration domains.

Program schedules: The execution order of statement instances is defined in the polyhedral model with the program schedule, a transformation matrix or an affine map that assigns to each statement instance an execution date. Schedules can be seen as multi-dimensional time-stamps $\Theta^S(\vec{x}) = \langle \theta_1^S, \theta_2^S, \dots, \theta_d^S \rangle$, $\vec{x} \in \mathcal{D}^S$, where each θ_i^S is a one-dimensional affine function, and d is the number of dimensions in Θ^S . Schedules can be lexicographically compared and are used in the polyhedral scanning process [9] to generate the loop structure that will visit each statement instance in the order established by the schedule Θ^S . Continuing with our ongoing example, the execution order of `parity_247` circuit can be represented with the schedules $\Theta^{S^1}(i) = \langle 0, i \rangle$, $\Theta^{S^2}(i) = \langle 1, i \rangle$. We note, however, that loop fusion can be legally applied to this circuit. In this case, another potential schedule could be $\Theta^{S^1}(i) = \langle 0, i, 0 \rangle$, $\Theta^{S^2}(i) = \langle 0, i, 1 \rangle$.

Access Relations: Are an abstraction that permit to model memory accesses. Access relations map points in an iteration domain to a data-space. The motivation is two-fold. First, to later be able to identify program statements accessing the same memory location. Second, to update the array subscript functions post-transformation. The construction of access relations in quantum computing differs in two fundamental ways from its classical counterpart. First, in classical computing the usage of multi-dimensional arrays is the norm, whereas the current practice in quantum computing is to operate on a single, large, one-dimensional array that represents a quantum register. The second difference involves what constitutes a *read* and *write* access. In effect, in classical computing, where polyhedral compilation has been predominantly applied to imperative programming languages such as C/C++ and Fortran, there is, practically always, a single write reference and zero or more read references. This is not the case in quantum computing, where the program's state evolves as specific entries of the register are input to gate operations. Moreover, the type of gate determines if a specific register entry is *read* or *updated*. We make this distinction for two reasons: First, some gates take a *control argument*, which do not modify the contents of a register entry. Second, every gate operation performing a *write* also reads the input register entry, i.e., the same entry is both read and written. Resuming our example, if we assume that the top qubit has

Fig. 6. parity_247 dependence graph.

```
// Dep.1: NOT to CNOT dependencies
{S[i]->T[i]: 1 <= i <= 6} and

// Dep.2: CNOT to CNOT dependencies
{T[i]->T[j]: 1 <= i, j <= 6 and j>i}
```

Listing 2. parity_247 dependence constraints

```
S(1); // NOT(1)
for (i = 2; i <= 6; ++i) {
  T(i-1); // CNOT i-1, 0
  S(i); // NOT i
}
T(6); // CNOT 6, 0
```

Listing 3. Loop structure with partial fusion parity_247–semantically equivalent to **pluto-maxfuse**.

index 0, increasing downwards, then statement S2 would have three access relations, a read and write relation $\{[i] \rightarrow [0]\}$ for the target qubit and a read-only relation $\{[i] \rightarrow [i+1]\}$ for the control qubit. We note, however, that to model some of the quantum gates, we fix the read/write semantics of gate operands. As a result, a commuting gate such as the Controlled-Z could potentially miss transformation opportunities (since target and control qubits commute). This phenomenon can be partially addressed in several ways. For example, a first approach would be to account for two dependences with sources at every CZ gate, one for each semantically equivalent variant, but enforcing the satisfaction of only one of them during the scheduling phase. A second approach would be to explicitly differentiate CZ gates where the control qubit is the first argument (i.e., CZ_1) or second argument (CZ_2). Regardless, both of the proposed alternatives will still lose some transformation opportunities. Such semantics, fixed to statically determine the program dependences, will apply to complete statements and not only to individual gate instances.

Dependence Polyhedra embody the semantic orderings of the program. Every program dependence in a SCoP is represented by one or more dependence polyhedra $\mathcal{D}^{R \rightarrow S}$. These polyhedra define the ordering among points $\vec{x}^R$ and $\vec{y}^S$ from the iteration domains $\mathcal{D}^R$ and $\mathcal{D}^S$, respectively. This critical pass is necessary to perform aggressive loop optimizations and compute reordering transformations via one or more **integer linear problems (ILPs)**, which must preserve the legality of the transformations applied. Essentially every scheduling technique [2, 23, 27, 34, 98] embeds the program semantic constraints in the form of dependence polyhedra (possibly linearized by the application of the Farkas Lemma) into one or more ILP systems. These polyhedra are usually extracted with “classical” data-flow dependence analysis [28].

As an illustration, we show in Figure 6 a depiction of the instancewise-dependence graph for the parity_247 circuit; NOT gates are the bottom nodes, while the CNOT gates are positioned at the top. Dependences arise among instances of the two sets of gates according to the constraints shown in Listing 2. Figure 7 presents an excerpt of a single ILP problem constructed to minimize the maximal dependence distance (U_l variables); θ variables are the scheduling coefficients to find, and δ variables denote the different l levels where a program dependence can be satisfied. The result obtained, shown in Listing 3, exhibits a loop structure with partial fusion that is semantically equivalent to the **pluto-maxfuse** heuristic.

(Parameterized) Affine Quantum Circuits (PAQCs). We define PAQCs as a subclass of quantum programs that can be expressed with affine relations, or a union of them. We have two

$$\begin{aligned}
\Theta^S &= \begin{pmatrix} \theta_{0,0}^S & \theta_{0,1}^S \\ \theta_{1,0}^S & \theta_{1,1}^S \\ \theta_{2,0}^S & \theta_{2,1}^S \end{pmatrix}, \quad \Theta^T = \begin{pmatrix} \theta_{0,0}^T & \theta_{0,1}^T \\ \theta_{1,0}^T & \theta_{1,1}^T \\ \theta_{2,0}^T & \theta_{2,1}^T \end{pmatrix} \\
\text{Objective :} & \quad -\theta_{1,0}^T + U_1 \geq 0 \wedge \\
& \quad -5\theta_{1,0}^T + U_1 \geq 0 \wedge \\
& \quad -\theta_{1,0}^S + \theta_{1,0}^T - \theta_{1,1}^S + \theta_{1,1}^T + 40\delta_0^0 - \delta_1^0 \geq 0 \wedge \\
& \quad -6\theta_{1,0}^S + 6\theta_{1,0}^T - \theta_{1,1}^S + \theta_{1,1}^T + 40\delta_0^0 - \delta_1^0 \geq 0 \wedge \\
& \quad \theta_{1,0}^T + 40\delta_0^1 - \delta_1^1 \geq 0 \wedge \\
& \quad 5\theta_{1,0}^T + 40\delta_0^1 - \delta_1^1 \geq 0 \wedge \\
& \quad -\theta_{2,1}^S + \theta_{2,1}^T + 40\delta_0^0 + 40\delta_1^0 - \delta_2^0 \geq 0 \wedge \\
& \quad 40\delta_0^1 + 40\delta_1^1 - \delta_2^1 \geq 0 \\
\min\{U_0, U_1, U_2, \theta_0^S, \theta_0^T, \theta_1^S, \theta_1^T, \dots, \delta_2^0, \delta_2^1\} & \quad \text{Sol : } \Theta^S = \langle 0, i, 0 \rangle; \Theta^T = \langle 0, i + 1, 0 \rangle; \\
& \quad \delta_2^0 = \delta_2^0 = 1 \\
& \quad U_0 = 0, U_1 = 5, U_2 = 0
\end{aligned}$$

Fig. 7. Excerpt of Integer Linear Program (ILP) optimization problem for parity₂₄₇ circuit.

Table 1. Modeling of Quantum Gates with Read/Write Semantics

Gate	Description	No. Inputs	No. Outputs	Integer Map Representation
$X(rw_1)$	Pauli-X (NOT) gate	1	1	$S[\vec{x}] \rightarrow q[rw_1(\vec{x})]$
$Y(rw_1)$	Pauli-Y gate	1	1	$S[\vec{x}] \rightarrow q[rw_1(\vec{x})]$
$Z(rw_1)$	Pauli-Z gate	1	1	$S[\vec{x}] \rightarrow q[rw_1(\vec{x})]$
$H(rw_1)$	Hadamard (H) gate	1	1	$S[\vec{x}] \rightarrow q[rw_1(\vec{x})]$
$Measure(rw_1, w_2)$	Measure qubit to classical bit	1	2	$S[\vec{x}] \rightarrow q[rw_1(\vec{x})], S[\vec{x}] \rightarrow c[w_2(\vec{x})]$
$CNOT(r_1, rw_2)$	Controlled-NOT (CX)	2	1	$S[\vec{x}] \rightarrow q[r_1(\vec{x})], S[\vec{x}] \rightarrow q[rw_2(\vec{x})]$
$CY(r_1, rw_2)$	Controlled-Y	2	1	$S[\vec{x}] \rightarrow q[r_1(\vec{x})], S[\vec{x}] \rightarrow q[rw_2(\vec{x})]$
$CZ(r_1, rw_2)$	Controlled-Z	2	1	$S[\vec{x}] \rightarrow q[r_1(\vec{x})], S[\vec{x}] \rightarrow q[rw_2(\vec{x})]$
$Swap(rw_1, rw_2)$	Exchange state	2	2	$S[\vec{x}] \rightarrow q[rw_1(\vec{x})], S[\vec{x}] \rightarrow q[rw_2(\vec{x})]$
$Tofoli(r_1, r_2, rw_3)$	Controlled-Controlled-Not	2	1	$S[\vec{x}] \rightarrow q[r_1(\vec{x})], S[\vec{x}] \rightarrow q[r_2(\vec{x})],$ $S[\vec{x}] \rightarrow q[rw_3(\vec{x})]$

requirements: First, the instances of gate operations must be groupable by affine expressions; Second, the indices of qubits being accessed must be representable via affine functions.

5.2 AXL: A Simple Domain-specific Language for Parameterized Affine Quantum Circuits

Enter **AXL**, a declarative language with operations that enable the specification, manipulation and composition of quantum circuits. **AXL**'s syntax is simple and straightforward. **AXL** provides four datatypes: *gate*, *circ* (constant circuits), *statement* circuits (our link to the polyhedral abstractions), and program parameters. The former three define a hierarchy of types: $gates \subset circ \subset statement$. A gate is any of the operations shown in Table 1 and represents a single quantum assembly instance (or a point in $\mathbb{Z}^+$). The *circ* type essentially allows for the composition of gates. For example, the NOT gate operating on qubit 1 can be written as NOT(1), and a CNOT gate controlled by qubit 0 and targeting qubit 1 can be expressed as CNOT(0,1). These operations can then be synchronized (in time) via the *time concatenation* operator (+). Internally, **AXL** will manipulate the program schedule to properly represent this ordering constraint. Climbing the hierarchy, we have a *circuit statement*, an enriched *circ* with polyhedral abstractions (iteration domains, access relations, gate relations, schedules).

Parameterized Affine Quantum Circuits (PAQC). A unique capability of **AXL** is to represent *affine quantum circuits* in a parameterized fashion, that is, circuits can be modeled with symbolic values representing arbitrary, unknown, constant values. PAQCs are built by defining an iteration domain for the circuit and by attaching a circuit body to it. The circuit body can be: (i) a basic gate; (ii) a fixed, possibly composite expression of type **circ**; or (iii) a parameterized, variable **circ** expression. The last of these body types leverages the full power of the polyhedral model to represent individual instances of the computation. PAQCs are implemented in **AXL** via the **statement** type, making parameterized and variable sized quantum circuits first-class citizens in our language.

We show in Table 1 a subset of the gates supported by **AXL**. The arguments to each gate represent indices in an implicit one-dimensional space; depending on the type of the gate, some arguments are either *read-only* (i.e., any r_i argument), *read-write* (arguments rw_j), or *write-only* (arguments w_k). These semantics allow us to determine the number of input and output relations, which we show in the third and fourth columns. The last column shows the extracted access relations for each of the accessed register entries, modeled as a single affine relation mapping points of the iteration domain to the data-space of the quantum register (register **q**) or of the classical register (register **c**). We follow the same naming convention to indicate when an access function is of type read, read-write, or write. We note here that, although we list the main quantum gate operations, these can be quickly added into the **AXL** language under the category defined by the number of qubits, and the access type to each of its arguments.

Gate Call Relations. In classical (scientific) computing, where multi-dimensional arrays are pervasive in imperative programming languages, each array reference is either read or written. This naturally follows the semantics of languages with explicit assignment operations; only the reference on the left-hand-side is written, while any array or scalar variable on the right-hand-side is read-only (except when passed to some function with side-effects). In our DSL, quantum programs operate in a data-space that is implicitly one-dimensional. This results from manipulating specific entries of a quantum or classical register. Thus, we add a new abstraction representing *quantum gate calls*. These are very similar in spirit to classical access relations that map points in iteration domains to the various data spaces of the program. While quantum gates are effectively functions that could change the state of a register, they do come equipped with domain semantics that must be correctly mapped. More precisely, each quantum gate operation has a set of *read* and *write* index sets that establish the register entries accessed in each of these modes. Later, during the polyhedral scanning process (phase-1 of code generation) these relations are used to produce the exact quantum gate operation specified by the end-user.

Program Assembly. We define Program Assembly as the process of constructing the final circuit schedule establishing the complete ordering among statement circuit instances in a single SCoP. It takes place at the start of the transformation and code generation process (see clause *codegen* in Figure 3). Building the final schedule happens in two phases. First, when defining circuit statements, a local identity schedule is created. For instance, in the same example, the local schedule $\Theta = \langle i \rangle$ is created for statements S1-S3 (Lines 3–5 in the same figure). **AXL** will then split S1 into sub-statements S1₁ and S1₂, effectively making each gate its own statement. The usage of the (+) operator for S1 will introduce a suffix scalar dimension, producing the schedules $\Theta^{S1_1} = \langle i, 0 \rangle$ and $\Theta^{S1_2} = \langle i, 1 \rangle$. The second phase of program assembly synchronizes statement circuits in a similar fashion, but with the distinction of inserting scalar dimensions as prefixes. For example, the local schedules of S2 and S3 will be modified to $\Theta^{S2} = \langle 0, i \rangle$ and $\Theta^{S3} = \langle 1, i \rangle$. Along the process, multiple scalar dimensions might be added to faithfully represent the ordering among various circuit patterns, which can be controlled by proper parenthesization.

```

1  <program>          ::= <parameter-list>   <statement-decl>
2                      <statement-list>   <scop-assembly>
3  <parameter-list>   ::= param <identifier-list> ';' <parameter-list> | ε
4  <statement-decl>   ::= <statement-decl> statement <identifier-list> ';' |
5                      statement <identifier-list> ';'
6  <statement-list>   ::= <statement-list> <circuit-statement>
7                      | <circuit-statement>
8  <circuit-statement> ::= <identifier> "!=" '{' <circuit-tuple> ':'
9                      <affine-contraint-list> "(%" <gate-expression> '}' ';
10 <scop-assembly>    ::= codegen '{' <stmt-expr> '}' with '{'
11                  <parameter-values> '}' <transformation>
12 <gate-expr>        ::= <gate-expr> "[+]" <gate-term> | <gate-term>
13 <gate-term>        ::= <gate-term> "(+)" <gate-fact> | <gate-fact>
14 <gate-fact>        ::= '(' <gate-expr> ')' | #NOT '(' <ae> ')'
15                  | #CNOT '(' <ae> ',' <ae> ')' | ...
16 <stmt-expr>        ::= <stmt-expr> "[+]" <stmt-term> | <stmt-term>
17 <stmt-term>        ::= <stmt-term> "(+)" <stmt-fact> | <stmt-fact>
18 <stmt-fact>        ::= '(' <stmt-expr> ')' | <identifier>
19 <transformation>   ::= apply '{' <schedule-name> '}' | ε
20 <schedule-name>    ::= "feautrier" | "plutomax" | "plutomix" | ...
21 <ae>               ::= /* arithmetic expression */

```

Listing 4. Main fragments of **AXL** grammar in Backus-Naur Form.

Formally, an **AXL** program consists of four parts: an optional list of program parameters, a list of circuit statement declarations, a list of circuit statement definitions, and the SCoP assembly operation. Of these four, *circuit statements* permit **AXL** to group tightly related operations into a single logical statement with its associated polyhedral abstractions (i.e., iterations domains, access relations, and schedule). Thus, *circuit statements* are **AXL**'s main building block.

Conceptually, a circuit statement permits to surround a *gate expression* by a loop-nest. Hence, the definition of a circuit statement requires three components: (i) The circuit's tuple, where the number of tuple dimensions equates to the number of surrounding loops that will be generated; (ii) The affine inequalities defining the circuit's shape; (iii) A *gate expression* as a function of the circuit's tuple. The syntax of a *circuit statement* is shown in Line 8 of Listing 4. *Gate expressions* are single instances of quantum gates applied to specific qubit entries. Qubit indices denoting operands to a gate can be written as affine functions on the dimension variables defined in the *circuit's tuple*.

A key aspect of **AXL** is the construction of a circuit's schedule. Each circuit statement is created together with a *local schedule* produced as an *identity map*, that is, if $\langle \vec{i} \rangle$ represents the circuit's tuple, then the local schedule is constructed as: $\theta_{local}^S = \{S[\vec{i}] \rightarrow [\vec{i}]\}$. Local schedules are complemented by the semantics of two operators, *time concatenation* denoted by (+) and *space separation* denoted by [+]. The former operator enforces a sequential order of gates fused under a common set of loops within a circuit statement (likely because the user intends to specify a qubit-to-qubit dependence pattern), whereas the latter permits their separation. The motivation for having *space separation* is to avoid the introduction of spurious dependences in the event of overlapping qubit indices. As an illustration, Line 3 in Figure 3 uses (+). Hence, when in pass-through mode, **AXL** would produce a single 1-dimensional loop nest, as shown in Lines 9–13 of Figure 8 (bottom half). The loop structure would result from the schedule $\langle 0, i, 0 \rangle$ for the X gates, and the schedule $\langle 0, i, 1 \rangle$ for the CX gates. In contrast, had we used the *space separation* operator [+], the schedules for these two gates would instead be $\langle 0, i \rangle$ and $\langle 1, i \rangle$. The corresponding code would exhibit two disjoint 1-dimensional loops, as shown in Figure 8, Lines 1–7.

While *local schedules* define the behavior of a single *circuit statement*, a much larger quantum program requires connecting several circuit statements and establishing their temporal relations.

```

1 // Plutomin (min.fusion) heuristic: Loop fission
2 for (int c1 = 1; c1 <= 8; c1 += 1) {
3   X[c1];
4 }
5 for (int c1 = 1; c1 <= 8; c1 += 1) {
6   CX[c1][0];
7 }
8 // *****
9 // Plutomax (max.fusion) heuristic: Loop fusion
10 for (int c0 = 1; c0 <= 8; c0 += 1) {
11   X[c0];
12   CX[c0][0];
13 }

```

Fig. 8. Generated loop structure for circuit parity₂₄₇ using Pluto’s Minfuse and Maxfuse loop fusion heuristics.

We therefore leverage the two introduced operators, (+) and [+], with the same semantics, and permit to create *statement expressions* (Lines 16–18 of the **AXL** grammar, Listing 4). Statement expressions allow us to define the visiting order among circuit statements, with parenthesization effectively adding back-to-back scalar dimensions. For example, in Figure 5, we arrange the six circuit statements with the expression S1 (+) S2 (+) S3 (+) S4 (+) S2 (+) S5, which append a leading scalar dimension to the schedule of each statement, 0 to S1, 1 to S2, 2 to S3, and so on.

$$\begin{aligned}
 \text{Identity schedule : } & \frac{\langle \vec{i}, \sigma \rangle \Downarrow_{\text{local}} \{\theta^{S_g} \equiv S_g[\vec{i}] \rightarrow [\vec{i}]\}}{(1)} \\
 \text{Gate time concatenation : } & \frac{\langle g1, \sigma \rangle \Downarrow_{\text{local}} \theta^{g1} \quad \langle g2, \sigma \rangle \Downarrow_{\text{local}} \theta^{g2}}{\langle g1 (+) g2, \sigma \rangle \Downarrow_{\text{local}} \{\theta'^{g1} \equiv [0|\theta^{g1}|0], \theta'^{g2} \equiv [0|\theta^{g2}|1]\}} (2) \\
 \text{Gate space separation : } & \frac{\langle g1, \sigma \rangle \Downarrow_{\text{local}} \theta^{g1} \quad \langle g2, \sigma \rangle \Downarrow_{\text{local}} \theta^{g2}}{\langle g1 [+] g2, \sigma \rangle \Downarrow_{\text{local}} \{\theta'^{g1} \equiv [0|\theta^{g1}], \theta'^{g2} \equiv [1|\theta^{g2}]\}} (3) \\
 \text{Statement time concatenation : } & \frac{\langle s1, \sigma \rangle \Downarrow_{\text{scop}} \theta^{s1} \quad \langle s2, \sigma \rangle \Downarrow_{\text{scop}} \theta^{s2}}{\langle s1 (+) s2, \sigma \rangle \Downarrow_{\text{scop}} \{\theta'^{s1} \equiv [0|\theta^{s1}|0], \theta'^{s2} \equiv [0|\theta^{s2}|1]\}} (4) \\
 \text{Statement space separation : } & \frac{\langle s1, \sigma \rangle \Downarrow_{\text{scop}} \theta^{s1} \quad \langle s2, \sigma \rangle \Downarrow_{\text{scop}} \theta^{s2}}{\langle s1 [+] s2, \sigma \rangle \Downarrow_{\text{scop}} \{\theta'^{s1} \equiv [0|\theta^{s1}], \theta'^{s2} \equiv [1|\theta^{s2}]\}} (5)
 \end{aligned}$$

The semantics of **AXL**’s schedule operations are formalized in Equations (1)–(5) using Big-Step Operational Semantics. We consider input and output schedules as values, and distinguish between *local schedule* operations and *scop-level schedule* operations, denoted by $\Downarrow_{\text{local}}$ and $\Downarrow_{\text{scop}}$, respectively. The notation used is as follows: $\langle g, \sigma \rangle$ and $\langle S, \sigma \rangle$ denote the state of gate g and statement S in the SCoP σ ; $\vec{i}$ denotes a tuple in the iteration domain of some statement S , $g1$ and $g2$ represent instance-wise gates, $s1$ and $s2$ denote circuit statements, σ represents the program’s state

(i.e., statement attributes not changed such as the iteration domains and the access relations). The θ variables are multi-dimensional affine schedules, and the $|$ symbol is used to express concatenation of affine functions on the co-domain of a schedule θ . We also assume that the SCoP's state σ treats the iteration domains and access relations as constants. Under these definitions, an *identity schedule* is created for each gate in a statement (Equation (1)); *gate time concatenation* (Equation (2)) effectively wraps two gates with the same set of surrounding loops while still preserving their relative order in the innermost loop; *gate space separation* (Equation (3)) establishes order at a coarser granularity by generating disjoint loop-nests for each gate; *statement time concatenation* (Equation (4)) is similar to *gate time concatenation* but with statement operands; Last, *statement space separation* (Equation (5)) resembles *gate space separation*. From a practical view, most users would prefer to use *gate time concatenation* for defining the intra-statement schedule (the local schedule) and the *statement space separation* to construct the full SCoP schedule.

5.3 Dependence Analysis

We leverage ISL's capabilities to compute the dependence polyhedra from the access functions (read and write relations) as well as from the statement schedules. The union of the former are used to define the sources and sinks of the dependences under the provided program schedule. Hence, the importance of properly composing statements within the codegen call. The inherent semantics of each quantum gate operator (e.g., a Controlled-NOT gate) are used to build the **read relations** and **write relations** for each circuit statement. This information is readily available from the definition of the quantum gate operator. For instance, the CNOT(control,target) gate operates on two quantum qubits, the *control* index is *read-only* while the *target* qubit is read and written, i.e., an update.

Two QASM operations are in a *dependence relation* if they access the same qubit and at least one of them is an update operation. From AXL's point of view, QASM gates are instances of points in iteration domains. However, one of the benefits of polyhedral techniques is the ability to model program dependences in a closed form, via *dependence polyhedra* [27, 28, 37].

Definition 5.1 (Dependence Polyhedron). Given two circuit statements S and T , a *dependence polyhedron* $\mathcal{D}_{S,T}$ is a subset of the Cartesian product of iteration domains $\mathcal{D}_S$ and $\mathcal{D}_T$. Then each point $\langle \vec{x}, \vec{y} \rangle$, with $\vec{x} \in \mathcal{D}_S$ and $\vec{y} \in \mathcal{D}_T$, defines a dependence pair such that the operation identified by $\vec{y}$ depends on one executed by $\vec{x}$.

Given Definition 5.1, computed transformations (denoted by Θ) then preserve the original program semantics iff:

$$\Theta^S(\vec{x}) < \Theta^T(\vec{y}).$$

The above constraint essentially states that the execution dates, $\Theta^S(\vec{x})$ and $\Theta^T(\vec{y})$, assigned to instances in the iteration domains of a dependence relation must be preserved by the computed transformation, where the $<$ operator is the lexicographic ordering, defined as:

$$\langle x_1, x_2, \dots, x_m \rangle < \langle y_1, y_2, \dots, y_n \rangle$$

iff an integer i exists s.t. $1 \leq i \leq \min(m, n)$, $\forall k \in \{1, \dots, i-1\} x_k = y_k$, and $x_i < y_i$.

Having the ability to model inter- and intra-statement dependences permits us to preserve the original program semantics, possibly requiring multiple dependence polyhedra to precisely characterize the complete program's behavior.

Nonetheless, while dependence polyhedra capture semantic constraints among instances of statements, they do so under the input (original) execution order. What is necessary is a mechanism to project the ordering constraints directly onto the scheduling coefficients of Θ^S (for all

statements). To address this, Feautrier [29] proposed to leverage the Affine form of Farkas Lemma [82] to perform a linearization of the dependence constraints on the (new) scheduling space.

LEMMA 5.2 (AFFINE FORM OF FARKAS LEMMA). *Given $\mathcal{D}$ a non-empty polyhedron with the inequality system $A\vec{i} + \vec{b} \geq \vec{0}$, a function $f(\vec{i})$ is non-negative everywhere in $\mathcal{D}$ iff it is an affine positive combination:*

$$f(\vec{i}) = \mu_0 + \vec{\mu}^T (A\vec{i} + \vec{b}), \mu_0 \geq 0, \vec{\mu}^T \geq \vec{0},$$

where μ_0 and $\vec{\mu}^T$ are Farkas multipliers.

Relying on this key result [3, 13, 30, 71], we can then establish the satisfaction of dependences in a number of ways. For example, the below equation (used in References [58, 77, 98])

$$\forall \mathcal{D}_{S,T}, \forall \langle \vec{x}, \vec{y} \rangle \in \mathcal{D}_{S,T} : \Theta_k^T(\vec{y}) - \Theta_k^S(\vec{x}) \geq \delta_k^{\mathcal{D}_{S,T}}, \delta_k^{\mathcal{D}_{S,T}} \in \{0, 1\}$$

imposes scheduling constraints of dependence $\mathcal{D}_{S,T}$ by modeling a *time delay* at time dimension k , forcing the difference between affine functions to be non-negative. For instance, two one-dimensional schedules satisfying this property would be $\Theta^S = \langle i + 2 \rangle$ and $\Theta^T = \langle 2i + 5 \rangle$. If this difference is zero at level k , then the dependence is *weakly satisfied*. In this case, the direction of the dependence is not inverted, but left to be satisfied at some level $p > k$. However, when $\delta_k^{\mathcal{D}_{S,T}} = 1$, then the dependence is strongly satisfied. In practice, this permits to nullify the constraints of $\mathcal{D}_{S,T}$ for later scheduling dimensions, or not embedding anymore the legality constraints in subsequent ILP (e.g. in level-by-level scheduling approaches).

5.4 Transformations

We next recap two well-known formulations that have been extensively used at the heart of several scheduling algorithms, the Feautrier minimum latency schedules [29, 30], and the minimization of the maximal dependence distance used in the Pluto compiler [13]. In particular, Pluto has been extremely successful in generalizing the tileability of imperfectly nested loops and using the aforementioned cost function to produce high-quality transformations. Similarly, the Feautrier algorithm has been used in Reference [59] and is still the fall-back scheduling strategy in ISL [104].

The Pluto Algorithm. The Pluto Tiling Hyper-plane algorithm [13] was introduced as a general greedy algorithm to find affine transformations to make a program tileable. At its core, it uses a cost function that bounds and minimizes the distance among statements in dependence relations. As in this work, we do not apply loop tiling to quantum programs, we only recap Pluto's main cost function formulation below:

$$\begin{aligned} \delta_e(\vec{x}) &= \phi_{S_i}(\vec{x}) - \phi_{S_j}(h_e(\vec{x})), \vec{x} \in P_e \\ \phi_{S_i}(\vec{x}) - \phi_{S_j}(h_e(\vec{x})) &\leq v(\vec{p}), \vec{x} \in P_e, \forall e \in E \\ v(\vec{p}) &= u \cdot \vec{p} + w \\ v(\vec{p}) - \delta_e(\vec{x}) &\geq 0, \vec{x} \in P_e, \forall e \in E \\ \text{minimize}_{<} \{u &= (u_1, u_2, u_3, \dots, u_n), \dots\} \end{aligned}$$

The above equations essentially perform the following: (i) define the constraints to satisfy a dependence edge e of the dependence set E , for every point $\vec{x}$ in the dependence polyhedron P_e ; (ii) introduce an affine function v on the vector of program parameters, $\vec{p}$; (iii) bound the distance between the target and source of the dependence via a function $v(\vec{p})$; (iv) minimize the u coefficients that bound the dependence distance. (NOTE: We omit here the application of the Farkas Lemma.)

The rest of Pluto’s algorithm proceeds level-by-level, from the outermost to the innermost, finding one-dimensional affine transforms for all statements in the program. By default, *splitters* (scalar dimensions) are only introduced by the algorithm when no solutions are found for a new hyperplane. However, doing this whenever is legal produces loop structures maximally distributed. We show in Figure 8 the result of applying both heuristics, labeled as *plutomin* for **pluto-minfuse** (i.e., maximal loop distribution/fission yielding each program statement in its own loop nest) and *plutomax* for **pluto-maxfuse** (resulting in the smallest number of disjoint loop nests produced by maximal loop fusion), to the **AXL** implementation of the `parity_247` circuit.

The Feautrier Scheduling Algorithm. Feautrier’s seminal papers [29, 30] on one-dimensional and multi-dimensional affine schedules have been a common test for several applications and domains. The main property of Feautrier’s scheduling approach is to greedily satisfy as many dependences as early as possible (“early” referring to the scheduling dimension being computed and appended next), going from the outermost (linear) dimension to the innermost. This greedy approach produces schedules with the minimum number of dimensions, minimum latency schedules. More importantly, this approach yields the maximum freedom to reorder operations in the innermost loop dimensions. The overall algorithm builds a single system with the legality constraints derived from all the dependence polyhedra and associating Boolean decision variables to each of them. These variables model the satisfaction of a dependence at the current scheduling level l . At each level, the number of dependences being satisfied is maximized. Prior to proceeding to the next scheduling level, $l + 1$, the ILP system is updated by removing the already satisfied dependences. The solution to each problem at level l represents a new schedule row θ_l that is appended to the global solution θ . An obvious limitation of this scheduling algorithm is that it implicitly assumes that all program dependences are equally relevant. For further details, we refer the reader to Section 3 and Section 4 of Reference [30]. This algorithm is readily available in the **Integer Set Library (ISL)** [104] and is also used as the fallback algorithm to its Pluto’s implementation.

Circuit Reversal. Another transformation very applicable to the quantum computing domain is **loop reversal** [70]. This language construct is **not uncomputation** of circuits. Instead, it is a simple mechanism to improve the user productivity. This language feature is motivated by the fact that several of the circuits from the literature exhibit repetitive patterns that are symmetric either to some time or space hyperplane.

Circuit reversal is implemented using ISL by traversing every linear dimension of the schedule map, i.e., any dimension that is not a fixed scalar value, and negating the affine function of the given dimension; scalar dimensions are left unmodified:

$$\begin{aligned}\Theta^S &= S[\vec{x}] \rightarrow [\theta_1(\vec{x}), \theta_2(\vec{x}), \dots, \theta_n(\vec{x})] \wedge \\ \Theta_{rev}^S &= S[\vec{x}] \rightarrow [\theta_1^{rev}(\vec{x}), \theta_2^{rev}(\vec{x}), \dots, \theta_n^{rev}(\vec{x})] : \\ \forall i \in \{1, \dots, n\}, \theta_i^{rev}(\vec{x}) &= -\theta_i(\vec{x}), \text{ if } \theta_i(\vec{x}) \neq \text{constant}; \\ \theta_i^{rev}(\vec{x}) &= \theta_i(\vec{x}), \text{ otherwise.}\end{aligned}$$

As an illustration, we continue the example from Section 4, the **multi-cnot** circuit shown in Figure 5, which is represented by five **AXL** statements, three “original” (S1, S2, and S4) and two statement reversals (S3 and S5). The final circuit shape is obtained by reusing statement S2. In this example, S3 is the reversal of S1 and is used to build the descending circuit pattern (w.r.t. to the qubit index accessed) of the left-V (S1 and S3). Similarly, the smaller V-pattern on the right is built from S4 and S5. Statement S2 is a single CNOT gate used twice as the vertex of both V-shapes.

Fig. 9. AXL code generation flow.

Table 2. Quantum Circuits Evaluated with Default Parameters

Benchmark Name	Source	No.AXL State-ments	No.Poly-hedral Depen-dences	Para-meters	No. Qubits	No. QASM ops	Output Lines
adder-maj-uma	[25]	3	11	N = 9	20	55	13–15
cuccaro-adder-6bit	[25]	6	50	N = 6	14	46	26–36
sum ₅	[96]	7	41	N = 5	11	36	22–32
init-G ₅	[96]	8	33	N = 5	11	35	24–34
cheung	[19]	1	1	N = 6	18	21	5
pipelined-swap	[32]	13	28	N = 6	14	75	37–43
cnt3-5_179	[108]	2	14	N = 5	19	30	14–20
rd84_142	[108]	3	36	M = 2, N = 4	15	28	16–32

5.5 Code Generation

The code generation process in **AXL** follows the flow depicted in Figure 9. Once the polyhedral program representation (SCoP) has been extracted, the polyhedral scanning process generates the loop and control structure necessary to visit all the quantum gate operations in the order defined by the scattering functions (schedules). When possible, we leverage the ISL [104] capabilities to perform unrolling of the non-degenerate loops in the AST representation of the program. The next stage builds a binary executable of the program and prints out all the gate operations together with the quantum register entries being accessed. The output is then used to assemble the source program for a quantum back-end compiler. For the purpose of this work, we target the **OpenQASM (Open Quantum Assembly)** language [24] to produce the experimental data, Qiskit [45] (for our experimental evaluation) and ProjectQ [94] to visually render the circuits for debugging purposes. (NOTE: Circuit diagrams of evaluated benchmarks are included in the Appendix.)

6 EXPERIMENTAL EVALUATION

We have implemented the **AXL** language and the analyses and transformations described in Section 5 as a source-to-source compiler toolchain [56] using the Integer Set Library [104]. Quantum circuits are written and compiled with **AXL** to produce the loop structure, which is then post-processed to construct a compilable C-program. This is then lowered into an executable binary to finally generate the stream of quantum assembly operations, targeting either the Qiskit compiler [45] or producing a compliant OpenQASM 2.0 [24] circuit. The **AXL** benchmarks were compiled on an 8-core AMD Ryzen 7 2700X—2.1 GHz with 128 GB DRAM, 32 KB L1, 512 KB L2, and 8 MB L3 cache.

Table 3. Geometric Mean Circuit Depth and Added Gates across Topologies, Loop Transformations, and Qubit Allocators

Topology x Transformation	Circuit Depth (No.Gates)				No. of Added Gates			
	jku	ibm	sabre	wpm	jku	ibm	sabre	wpm
multi-ring x base	204.79	314.63	208.17	358.02	317.25	568.47	290.30	723.47
multi-ring x feautrier	211.31	311.54	203.04	358.47	327.48	556.27	275.04	747.09
multi-ring x plutomin	203.85	312.80	204.96	367.19	316.87	562.75	269.52	769.92
multi-ring x plutomax	206.43	309.15	205.16	356.87	321.21	550.15	284.55	735.11
grid x base	162.31	228.26	169.33	307.15	225.17	419.21	203.31	584.08
grid x feautrier	167.22	225.93	169.20	302.71	249.83	417.37	226.35	626.60
grid x plutomin	161.93	228.69	165.52	308.05	225.12	425.11	208.55	626.09
grid x plutomax	161.61	228.71	170.60	298.12	224.40	415.32	223.36	590.60
tilted x base	191.55	250.50	198.37	319.59	336.20	469.33	288.63	370.72
tilted x feautrier	196.23	254.10	192.31	321.07	344.30	473.35	294.37	385.64
tilted x plutomin	190.22	249.75	188.44	315.98	330.93	473.35	266.59	382.62
tilted x plutomax	195.73	246.92	194.65	329.98	333.47	461.80	287.61	379.69

The overall goal of our experimental evaluation is to demonstrate that even “classical” high-level loop transformations, i.e., not yet tailored to the quantum computing domain, can synergistically work with back-end compiler optimization such as Qubit Allocation. More concretely, we wish to determine where (in which experimental configurations), why (interactions of loop transformations with the qubit allocator), and when (in some specific stage of a circuit) can impact the qubit allocation result, either for good (improved quality, i.e., shallower circuits) or bad (bigger circuits).

Testbed and Protocol: We use in our evaluation the Enfield compiler [86, 87] for the experiments discussed in Sections 6.1–6.3 and IBM’s Qiskit in Section 6.4, both of which implement several qubit allocation techniques. In particular, we consider four of the qubit allocation algorithms covered in Section 3: *sabre* [65], *jku* [110], *wpm* [89], and *ibm* [46]. Both compilers allow to collect statistics such as allocation time, circuit depth (number of gates in critical path), circuit size (total number of gates), among many others. The methodology we follow is the same as that of *sabre* [65] when comparing against *jku*, while *wpm* and *jku* compare against the *ibm* mapper in their respective publications. As context, the *jku* paper reported a 23% improvement (number of elementary gates added) w.r.t. to *ibm*, while *sabre* reports an average improvement of 14% over *jku*, considering their **Quantum Fourier Transform (QFT)** and large benchmark categories.

We use the circuit depth and size as primary quality metrics, which vary with the allocator due to the number of *quantum SWAP* and *REVERSE* operations introduced to advance the program state. Each of these operations is implemented with elementary gates. Each SWAP is implemented with three CNOT, while each reverse operation requires four Hadamard (H) and one CNOT gates.

Our experimental testbed consists of eight quantum circuits taken from the literature³ and circuits taken from the QUEKO benchmark suite [97]. We generate four different variants for each benchmark: pass-through mode performing only code generation (**base**); the transformed code obtained by applying the Feautrier scheduling algorithm (**Feautrier**) [29, 30]; and the two well-known Pluto fusion heuristics, maximal loop fusion (**pluto-maxfuse**) and maximal loop distribution (**pluto-minfuse**). We note that the **base** variant, while only involving code generation, can already be considered an optimized program, as the polyhedral scanning process will produce minimal control overhead that can vastly differ from hand-written loop-based code [9]. The latter two variants represent ends of the classical locality spectrum. Although Enfield already repeats the

³Please refer to the Appendix to view their source and properties.

allocation process for the algorithms with random properties (5 for **sabre** and 20 for **ibm**), we still found substantial variation in the results. Thus, for each allocator, we repeat the allocation process 10 times and report their arithmetic mean. We also include error bars showing the standard deviation.

Circuit Implementation and Validation: The eight quantum circuits used in our evaluation have been manually validated. The initial porting process from a circuit diagram to an **AXL** program consists on first identifying groups of quantum gate operations (potentially of different type) that behave in the same (regular) form, e.g., applying NOT gates on all even positions of a qubit register in the interval $1 \dots 15$. When possible, the initial implementation will describe these operations in a fused form, using as criterion that groups of operations of different type must perform the same number of operations while not altering the original semantics of the program. We also note that the gate grouping choice when writing an **AXL** program can affect, for good or for bad, the variety of applicable loop transformations, just as in classical computing. Each group of operations becomes a **circuit statement**. The second step is to identify the affine relations describing the qubit index expressions, the indices of a qubit register being accessed, among gate operations. The last step consists on assembling the SCoP by using the (+) and [+] operators and parenthesizing the building expression to correctly represent the topological structure among circuit statements. Each **AXL** program is then compiled, producing a loop-based C program, which is then lowered to machine code and compiled using either the ProjectQ or Qiskit compilers. This version of the circuit is rendered to a circuit diagram and compared to the original circuit. This is the process followed for the **base** variants. While this process is straightforward for small circuits, it becomes challenging for larger ones. In this scenario an incremental approach is taken. We split the circuit along the time dimension into some reasonable number of stages, apply the previous steps in each subcircuit, and then assemble a single program from all the identified circuit statements. Visual inspection is not meant to replace a correct implementation of a compiler, but it is simple enough for the circuits evaluated. We resort to using ProjectQ, since its rendering system also takes into consideration quantum properties such as two back-to-back CNOT gates being executed on the same qubit cancel each other. Thus, if some affine loop transformation changes the semantics of a program, then the circuit rendered by ProjectQ and Qiskit will differ, sometimes making it rather obvious by producing circuits with fewer gates. The implementations of the evaluated circuits (and a few others) are publicly available as a new benchmark suite, **AxlBench** [55].

Evaluated Topologies: Just as in classical computing, the underlying architecture (e.g., multi-core CPU, many-core CPU, GPUs, etc.) can have a substantial impact on the program's performance; in quantum computing, the quality of the resulting qubit allocation can depend of the underlying architecture (topology). We thus evaluate three coupling graphs with the same number of qubits (36) and slightly vary the graph's connectivity to produce different properties. The three coupling graphs used are shown in Figure 10 and include a 6×6 grid (grid), a graph with three doubly linked concentric rings (multi-ring), and a four 3×3 tiled array of qubits (tiled). All graphs have the same diameter (10), but differ in their bisection bandwidth, which is 6 for multi-ring and grid, and only 2 for tiled. In addition, having more qubits with higher degree enhances the architectural parallelism. For instance, if a given qubit has degree d , then completing an update (write) on this qubit can enable up to $d - 1$ gate-to-gate dependences. For multi-ring, tiled, and grid the number of gates with degree 3 or greater are 8, 24, and 32, respectively. This design decision differs from the IBM QX20 (Tokyo) coupling graph, which varies over time and which typically has several qubits with degree 5 or 6.

On the Applicability of the Polyhedral Model: The configuration used in our experimental testbed involves a wide range of circuit sizes and shapes. In particular, our evaluation accounts for circuits written directly in **AXL** and also for larger circuits taken from the QUEKO suite, used as

Fig. 10. Evaluated coupling graphs (topologies).

input to a prototype tool that reconstructs **AXL**'s **intermediate representation (IR)**. We believe that even larger circuits than those evaluated here can further benefit from polyhedral abstractions, including **Quantum Fourier Transform (QFT)** circuits, realizations of the **Variational Quantum Eigensolver (VQE)** [50], and **Quantum Approximate Optimization Algorithms (QAOA)** [102]. However, for the latter, and whenever part of the quantum circuit is subject to change (as is between different iterations or phases separated by measurements), the corresponding fragment in the polyhedral IR will need to be updated.

The rest of this section describes our results on the impact of loop transformations on our testbed. In Section 6.1, we discuss general trends on the circuit depth and effect on gate count in the summary (see Table 3 and the corresponding discussion) using default values for the parameters. Next, we delve into the individual circuits focusing exclusively on the circuit depth, followed by parallel analysis on the gate count for a subset of benchmarks (Section 6.2). After this, in Section 6.3, we conduct a scalability experiment on three of the most regular circuits to further understand the effect of resource constraints (e.g., qubits with degree 3 or higher), and conclude our evaluation with a large scale study on the QUEKO benchmark suite [97] (Section 6.4).

6.1 Results Summary and General Trends

Overall, we observe that quantum qubit allocation algorithms and quantum programs can benefit from the power of global cost functions offered by polyhedral optimization techniques. Our evaluation shows that even the most advanced allocators (**jku** and **sabre**) can improve up to 34%, while achieving 60% on **wpm**. We also observe and confirm general trends such as allocation variability due to the inherent data- and pipelined-parallelism, distance among qubit operands, or the available hardware parallelism.

We summarize in Table 3 the geometric mean of the circuit depth (number of gates) and number of added gates across all combinations of topologies, affine transformations, and qubit allocators. The first trend to observe is that the depth of the circuit tends to increase with lower architectural parallelism (e.g., number of qubits with degree 3 or greater), as expected. Next, we observe that even the state-of-the-art allocators, **jku** and **sabre**, can greatly benefit from affine transformations. In particular, we observe up to a 10% gap of added gates ($\max \Delta_{transform} / \max(added)$) due to loop transformations in the **{grid,jku}**, **{grid,sabre}**, and **{tiled,sabre}** configurations. In regard to **jku**, this qubit allocator produces the shortest circuits (depth-wise) in all three topologies when used in combination with **pluto-minfuse**, and producing its worst circuit depth when combining it with **Feautrier**. This trend is nearly the same for **sabre**. Last, we do not observe any obvious trend involving the **ibm** allocator.

In regard to the number of gates added by each allocator (last four columns), we observe that this metric also correlates with the architectural parallelism of the topologies used. We also observe that an overall increase of gate count does not necessarily correlate with better circuit depth. This phenomenon can be seen when comparing the **{grid,jku}** and **{grid,sabre}** with **{multi-ring,jku}** and **{multi-ring,sabre}**.

Fig. 11. Impact of Affine Loop Transformations on circuit depth for grid, multi-ring, and tiled.

6.2 Individual Analysis with Default Parameters

We next dissect the behavior of the remaining seven quantum circuits (sum₅ was discussed in Section 2), and show the achieved circuit depth in Figure 11. For each circuit, we cluster the results by topology $\times$ allocator. Each bar represents the mean of 10 repetitions and includes their corresponding standard deviation. In addition, each cluster is tagged with the highest percentual variation between the highest and lowest depth ($(depth_{max} - depth_{min})/depth_{max}$) among the loop transformation for the same topology and allocator. In general, we expect lower variation for the grid topology than for the multi-ring and tiled topologies, given the higher number of qubits with higher connectivity for the former coupling graph.

Turning our attention to the pipelined-swap circuit, this benchmark represents the swap between two distant qubits (e.g., q_0 and q_{35}). The circuit initially exhibits two-way parallelism, simultaneously starting with two CNOT, and morphing into six-way parallelism in its steady-state (see Appendix for circuit diagram). Each CNOT operation being the input dependence to the operations on two adjacent qubits. The interesting result here is that **wpm**, which normally finds lower quality mappings, consistently produces shallower circuits. We also observe up to a 23% gap for **sabre** on the multi-ring topology yielded by the **Feautrier** transformation.

The cheung circuit exhibits N-way data-parallelism, N being the number of distinct target (updated) qubits by the **CCNOT (Controlled-Controlled NOT)** gates found. The first CCNOT of each qubit can be mapped to a distinct qubit to maximize parallelism and reduce the circuit depth. Intuitively, both the **Feautrier** and **pluto-maxfuse** should yield the highest benefit in terms of circuit depth, since maximizing the number of satisfied dependences per schedule dimension equates (for this benchmark) to minimizing the maximal-dependence distance. The circuit's depth drastically increases with the quality of the qubit allocator. This is due to the fact that the control qubits

Fig. 12. Impact of Affine Loop Transformations on circuit size for grid, multi-ring, and tiled.

for each operation start the closest and separate as the state evolves. This, in turn, requires more swap operations to make the qubit operands adjacent.

The adder-maj-uma circuit [25], an arithmetic adder using the **majority (MAJ)** and **unmajority (UNM)** circuit pattern, exhibits a highly serial form in its steady-state. The only opportunities for reordering transformations arise from the potential fusion/distribution of the CNOT operations with the sequence of CCNOTs following later. In general terms, we expect the **pluto-maxfuse** heuristic to have the highest impact on the circuit depth. The same observations also hold for circuit cuccaro-adder-6bit, which differs from the former in the degree of data-parallelism. This translates to much shallower circuits, roughly half the depth of the adder-maj-uma counterparts. It also differs in that **base** and **pluto-minfuse** typically achieve the best circuit depth due to the already available parallelism. All four qubit allocators also benefit from the higher scheduling and placement flexibility of cuccaro-adder-6bit due to the lower number of CCNOTs (three-operand gates) as well as the presence of CNOT and NOT gates. We also note that circuits using CCNOT (Toffoli) gates—which are typically synthesized—could benefit from first inlining the expanded gates, permitting a better detection of parallelism opportunities.

Benchmarks sum_5 and init-G_5 [96] share similar properties. As both exhibit distinct and noticeable parallel and serial phases. For instance, sum_5 's initial and init-G_5 ending phase offer a lot of reordering freedom in the scheduling and placement of the CNOT operations. In contrast, the middle and ending phases of sum_5 and initial and middle ones of init-G_5 are mostly serial with stints of pipelined parallelism. In addition, the large and varying separation between the gate operands induce a higher number of required SWAP operations. This combination of factors makes harder the prediction of the most suitable affine transformation, making the outcome highly variable w.r.t. the coupling graph and the qubit allocator. So the overall topological trends remain.

Circuit `cnt3-5_179` also shares some features with `cuccaro-adder-6bit`. They both have a mix of data- and pipelined-parallelism, with short constant distance (3 or lower) among the gate operands; they both contrast with benchmark `rd84_142`, which exhibits much larger, but constant separation (5). The distance between qubit operands is important, because affine circuit statements with constant distance in their access relations and non-overlapping values are easily parallelizable. Naturally, the assumption here is that the logical mapping of qubit indices corresponds (or is similar enough) to the physical one. `cnt3-5_179` benefits from a slighter higher degree of data parallelism (left-most CNOTs). Given the combination of characteristics, we expect **pluto-maxfuse** to be more beneficial for `rd84_142`, while **Feautrier** and **pluto-minfuse** to perform best (or close to) for `cnt3-5_179`, as loop fission will effectively isolate the parallel operations from the more serial/pipelined ones. Empirically, both of these observations and hypotheses hold.

Impact on Circuit Size Next, we show in Figure 12 the gap in circuit size (number of gates added by the qubit allocator) for three of the previous circuits. (NOTE: The impact on `sum5` was already discussed in Section 2.) While this metric is second to the circuit depth, it is still relevant due to it affecting the coherence time of the circuit, albeit at a different scale. In particular, we highlight that **wpm** effectively detected that the **pipelined-swap** was a swap circuit in itself and did not attempt to introduce SWAP operations. In opposition, all other allocators still attempted to optimize the circuit. The `init-G5` circuit is interesting, because it represents a scenario where the circuit depth produced by **jku** and **sabre** are the same, while yielding different numbers of added gates across topologies, with **jku** taking the upper hand on the tiled topology, drawing on the grid topology and **sabre** having the advantage on `multi-ring`. Last, while **wpm**'s mappings are of significantly lower quality across all topologies, we observe a noticeable gap between the best and worst generated variant, producing up to 24% difference. We emphasize that this variation is exclusively attributed to our compiler reformulating the problem for the qubit allocator.

6.3 Impact of Circuit Scaling

To complement our study, we perform a scaling evaluation of circuits `cheung`, `pipelined-swap` and `cnt3-5_179`. We focus on these benchmarks due to their high regularity and lack of singleton quantum statements that require more specific scheduling. We vary the SCoP parameter $N \in \{2..12\}$. Beyond this, the quantum register size is exceeded. The purpose of this experiment is to highlight the topological restrictions of circuits that require a higher number of qubits of degree 2 or 3, due to these embodying physical mapping constraints for the allocator. It also allows us to determine if the particular effects of high-level loop transformations persist across problem sizes and topologies. Before proceeding, we note that `pipelined-swap` under **jku** on the grid topology only scaled up to $N = 6$ (14 qubits), at which point each run started taking above an hour to complete. Furthermore, at $N = 10$, memory was being exhausted in our benchmarking server. The time limit was also exceeded for the **base** variant of `pipelined-swap` using **jku** on the other two topologies, for $N \geq 8$, a consequence of the A^* search used by **jku**. We omit summary clusters with the arithmetic or geometric mean, since these results would be skewed by larger problem sizes and possibly missing data points.

We next pivot our attention to the `cheung` benchmark (Figure 13), which only utilizes Toffoli gates. Most realizations of this 3-qubit gate would require the three accessed qubits to conform a strongly connected component—making them mutually accessible—to avoid the need of SWAP insertions. This is not the case in our topologies. The impact of this requirement is notorious on the `multi-ring` (8) graph, which for $N = 4$ requires a total of 8 (4 producing and 4 consuming) gates. Past this point the number of swap operations introduced by the **wpm** allocator nearly doubles for every increment of 2 on N . Given these resource restrictions, we now refine our initial assessment on this circuit, and expect the **Feautrier** transformation to fare best, as it was

Fig. 13. Impact of problem size scaling on circuit depth for cheung circuit on topologies grid (top), multi-ring (middle), and tiled (bottom).

conceived as a minimum latency, resource-conscious transformation. Equivalently, we also now expect **pluto-maxfuse** to induce typically higher circuit depth. The intuition here (in classical loop transformation terms) is that the **pluto-maxfuse** heuristic would amount to an outer parallel loop and an inner serial loop, while the **Feautrier** transformation would make the outer loop serial, thereby exposing more inner parallelism. The general effect of this tradeoff for most qubit allocators is to expose earlier the gate-to-gate dependences. In particular, this is most beneficial when the allocator makes local decisions using sliding windows. In more general scaling trends, we observe that the **ibm-mapper** achieves, on average, the best circuit depth on the two less regular topologies, multi-ring and tiled.

Changing our focus to the pipelined-swap circuit in Figure 14, we first note that, unlike the previous benchmark, this one only consists of CNOT gates. A quick inspection of this benchmark reveals that at most six CNOT operations can be executed in parallel. This degree of parallelism is independent of the problem size and provides more freedom to the qubit allocators, since all of our topologies satisfy this requisite. In general, the **jku** allocator produces the circuits with the shallowest depth, outperforming **sabre** in every topology. Nonetheless, we highlight that **wpm** in tandem with **pluto-maxfuse** yields results comparable, and at times better, than **sabre**, e.g., for $N = 12$. This is relevant, because **wpm** is $10\times$ faster than **sabre** and $10\times$ to $15\times$ faster than **jku**. The much improved circuit depth w.r.t. **wpm**'s impact on other benchmarks is due to a specific feature

Fig. 14. Impact of problem size scaling on circuit depth for pipelined-swap circuit on topologies grid (top), multi-ring (middle), and tiled (bottom).

of pipelined-swap. If we divide the circuit into four quadrants, the **north-east (NE)**, **north-west (NW)**, **south-east (SE)** and **south-west (SW)**, and consider them as the four statements to schedule, then we can notice that the NW and SW statements converge in the middle of the circuit. This means that the underlying qubit allocator would benefit from first finding those operations, the ones in the middle of the four quadrants, concentrated into a small window of operations. That is precisely the effect of using **pluto-maxfuse**, and leads up to a 60% circuit depth improvement when using **wpm** on the tiled topology.

We show in Figure 15 the scaling results on the cnt3-5_179 circuit. First, we observe that the **Feautrier** and **pluto-minfuse** transformations have a consistent impact on this benchmark when using the **wpm** allocator across all topologies, improving the circuit depth by up to 37% w.r.t. the **base** and **pluto-maxfuse** variants. As previously stated, while in some cases the quality of the **wpm** mapping is vastly outperformed by **jku** and **sabre**, in other cases (i.e., when enhanced with a high-level transformation) the lower quality can be acceptable when taking into consideration its scalability and computational complexity. Last, we also observe up to a 17% gap between the best and worst mapping produced with **sabre**. A competitive improvement owed solely to the trade-offs in transformations, and in which the **Feautrier** transformation typically yields the highest quality for **sabre** across all topologies.

Fig. 15. Impact of problem size scaling on circuit depth for cnt3-5_179 circuit on topologies grid (top), multi-ring (middle), and tiled (bottom).

6.4 Impact on QUEKO B_{IGD}

To conclude, we evaluated the impact of applying the **pluto-maxfuse** fusion heuristic to a subset of the QUEKO benchmark suite [97], focusing on the B_{IGD} class (which stands for “Benchmarks for Impact of Gate Density”). The density vector of a circuit implemented with single and two-qubit gates is controlled by four parameters: the number of single qubit gates (M_{1q}), the number of two-qubit gates in the circuit (M_{2q}), the number of logical qubits (Q), and the length of the critical path in the circuit (L). Then, the density vector $\langle d_1, d_2 \rangle$ is defined with $d_1 = M_{1q}/(Q \cdot L)$ and $d_2 = 2 \cdot M_{2q}/(Q \cdot L)$ [97]. We evaluate the circuits with the three lower single qubit density values, $d_1 = \langle 0.0, 0.1, 0.2 \rangle$ and all corresponding values for d_2 . This amounts to an exploration space with 197 circuits (13 circuit instances failed to reconstruct). Within this space, the number of original gates varied between 45 and 675.

As with previous experiments, we focus our evaluation on the number of added gates and circuit depth (critical path). Given the size of these circuits and the number of instances in the benchmark suite, we compare the original circuits (implemented in QASM) against a reconstructed polyhedral representation of the same benchmarks transformed with the **pluto-maxfuse** heuristic. The reconstruction process of QASM files thus directly instantiates **AXL**’s IR. In contrast to the previous experiments, where the program was described with **AXL**, this approach permits us to establish a native QASM baseline. Both the original and **pluto-maxfuse** transformed circuit variants are

Table 4. Formulas of Computed Statistics for QUEKO B_{IGD}

Metric Description (Column)	Formula Used
Base (original) number of gates in circuit i	g_i (given)
Metric value with Qiskit + pluto-maxfuse	δ_i^+ (measured)
Metric value with Qiskit	δ_i^- (measured)
Geomean Improvement Ratio with pluto-maxfuse ([A])	$1 - gmean(\{r_i : r_i < 1 \wedge r_i = \delta_i^+ / \delta_i^-\})$
Geomean Deterioration Ratio with pluto-maxfuse ([B])	$1 - gmean(\{r_i : r_i > 1 \wedge r_i = \delta_i^+ / \delta_i^-\})$
Percent of improved circuits ([C])	$ \{r_i : r_i < 1 \wedge r_i = \delta_i^+ / \delta_i^-\} $
Overall Metric Variation ([D])	$(\sum \delta_i^- - \sum \delta_i^+) / \{r_i : r_i \neq 1 \wedge r_i = \delta_i^+ / \delta_i^-\} $
Best Improvement relative to number of base q-operations ([E])	$\max\{(\delta_i^- - \delta_i^+) / g_i\}$
Worst Deterioration relative to number of base q-operations ([F])	$\min\{(\delta_i^- - \delta_i^+) / g_i\}$
Worst case obtained with pluto-maxfuse w.r.t. base q-ops ([G])	$\max\{(g_i + \delta_i^+) / g_i\}$
Worst case obtained without pluto-maxfuse w.r.t. base q-ops ([H])	$\max\{(g_i + \delta_i^-) / g_i\}$

Table 5. Impact of **pluto-maxfuse** Fusion Heuristic on the **Gate Count** of QUEKO B_{IGD} Benchmarks

Topology	D1	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]
grid	0.0	13.3%	-12.6%	38%	-28.7	1.3	-1.3	5.9	6.0
	0.1	10.8%	-10.3%	46%	-1.2	0.7	-0.7	4.4	4.5
	0.2	9.9%	-13.5%	44%	-11.3	0.6	-0.6	3.8	3.7
	all	11.4%	-12.1%	42%	-14.6	1.3	-1.3	5.9	6.0
multi-ring	0.0	10.2%	-11.4%	58%	45.4	2.2	-1.6	7.9	8.1
	0.1	10.5%	-8.1%	44%	6.1	1.0	-1.5	5.8	5.8
	0.2	8.9%	-8.1%	61%	35.6	0.9	-0.4	4.7	4.8
	all	9.9%	-9.2%	54%	29.5	2.2	-1.6	7.9	8.1
tiled	0.0	9.8%	-8.9%	47%	7.0	1.0	-1.0	6.1	6.1
	0.1	8.7%	-10.3%	54%	2.6	0.8	-0.6	4.9	5.1
	0.2	12.6%	-14.3%	59%	-3.7	0.9	-0.7	3.9	4.0
	all	10.3%	-10.7%	53%	2.4	1.0	-1.0	6.1	6.1

Breakdown by D1-density; all circuits compiled with Qiskit at -O3 level.

compiled with Qiskit [45] (version 0.23.6) at level -O3, and using Python 3.9. We note that, internally, Qiskit will resort to a fixed-point computation, applying several layouts, scheduling, and routing methods. So the comparison is not relative to a single layout or synthesis method, but likely a collection of them. Also, as the topologies used in our evaluation do not have pre-defined quality and performance weights associated to qubits and edges, each native SWAP operation is considered as 3 CNOT gates. Example outputs produced with Qiskit can be found in Reference [57].

Tables 5–6 show the summaries describing the impact of **pluto-maxfuse** on the QUEKO B_{IGD} benchmarks w.r.t. the gate count and circuit depth. Columns following the topologies, from left to right, are computed with the formulas of Table 4: Column D1 shows the three density gate values for single-qubit or “all”; Columns [A] and [B] are the geometric mean of the ratios between Qiskit + **pluto-maxfuse** and Qiskit alone, but separated by metric improvement and deterioration; that is, columns [A] and [B] are obtained from mutually exclusive subsets of circuits where **pluto-maxfuse** improves or deteriorates the metric. Column [C] is the percentage of evaluated benchmarks that exhibit any sort of improvement with Qiskit + **pluto-maxfuse**, while Column [D] is the average of the difference between all pairs of evaluated instances (i.e., Qiskit with and without **pluto-maxfuse**). Thus, a negative mean value translates to an overall metric increase when using **pluto-maxfuse**, while a positive one equates to a circuit being optimized. Columns

Table 6. Impact of **pluto-maxfuse** Fusion Heuristic on the **Circuit Depth** of QUEKO B_{IGD} Benchmarks

Topology	D1	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]
grid	0.0	10.9%	-14.4%	38%	-13.8	1.0	-0.7	3.6	4.0
	0.1	7.9%	-11.1%	36%	-10.3	0.2	-0.3	2.0	2.0
	0.2	7.4%	-12.5%	42%	-10.6	0.1	-0.2	1.8	1.8
	all	8.9%	-12.8%	38%	-11.8	1.0	-0.7	3.6	4.0
multi-ring	0.0	10.4%	-11.3%	45%	-2.7	0.8	-0.6	4.4	4.6
	0.1	9.6%	-9.4%	43%	-7.1	0.5	-0.4	2.5	2.4
	0.2	8.6%	-9.0%	47%	-1.1	0.3	-0.3	2.2	2.2
	all	9.6%	-10.0%	45%	-3.7	0.8	-0.6	4.4	4.6
tiled	0.0	9.6%	-10.6%	44%	-5.5	0.4	-0.4	3.3	3.5
	0.1	9.2%	-8.2%	42%	-3.5	0.3	-0.2	2.2	2.2
	0.2	8.4%	-10.6%	52%	-4.4	0.2	-0.2	1.9	1.9
	all	9.1%	-9.7%	46%	-4.5	0.4	-0.4	3.3	3.5

Breakdown by D1-density; all circuits compiled with Qiskit at -O3 level.

[E]–[H] are extreme cases of the spectrum normalized to the base number of quantum operations; Specifically, columns [E] and [F] represent the best achieved improvement with **pluto-maxfuse** and the highest worsening of the metric, respectively. Therefore, all values in column [E] will always be positive, while the ones in [F] are all negative. Columns [G] and [H] are absolute **Worst Case (WC)** values for each strategy ([G]:Qiskit + **pluto-maxfuse**, [H]:only Qiskit) relative to the number of base quantum operations. We include the last four columns to express how much does the original circuit grow with each of the evaluated strategies.

Overall, we observe a much greater impact of **pluto-maxfuse** on the multi-ring topology, followed by the tiled structure, and see the worst improvement with the grid coupling graph due to its higher connectivity. We also observe that reducing the maximal dependence distance, i.e., scheduling dependent operations as close as possible, has a much stronger impact on the total added gates, Column [C]–Table 5, than on the circuit’s depth, Column [C]–Table 6. In general, the original circuit can shrink or grow between 8.9% and 12.8% (Columns [A] and [B]). Nonetheless, even though a smaller fraction of the evaluated benchmarks are subject to non-negligible improvements, we notice an interesting effect, where Columns [G] and [H] exhibit up to 10% reduction in their *worst case*, for both metrics. The implication is that the evaluated benchmarks are individually worse, but better in the aggregate. This observation is in line with the relatively low depth variation obtained across the three coupling graphs, Column [D].

As our experiments involve three different density gate levels available in the QUEKO B_{IGD} class, we show in Tables 5–6 their breakdown by single gate density values, D1, ranging from 0.0 to 0.2. One can see from the breakdown shown in these tables that the trends previously noticed still hold, i.e., the impact on the worst-case scenarios (Columns [G] and [H]), fraction of improved circuits (Column [C]), the overall relative variation window when leveraging **pluto-maxfuse** (Columns [A] and [B]), and the maximum improvement and deterioration of a circuit’s metrics relative to the original number of quantum operations (Columns [E] and [F]). It is interesting to see, however, that some of the phenomena are enhanced. For instance, the range of geometric-mean ratios (Columns [A] and [B]) is widened to 7.4% and 14.4%, while we can now see up to 61% of circuit instances exhibiting improvement (Table 5–Column [C]). Last, these results are consistent with that of Reference [97], where critical communication and performance metrics naturally float down as the density of single qubit gates increases. This is justified by having fewer two-qubit operations that induce communication.

Fig. 16. Impact of **pluto-maxfuse** fusion heuristic gate count of QUEKO benchmarks - BigD class for grid, multi-ring, and tiled topologies.

Finally, to conclude this section, we show in Figures 16–17 plots comparing the achieved gate counts and circuit depth for all circuits across the three topologies. In particular, in Figure 16, we present the number of added gates introduced by Qiskit and Qiskit + **pluto-maxfuse**; at the bottom of each topology plot, we include a curve representing the base number of operations, which conveys the scale at which circuits will grow after the layout, allocation, synthesis, and mapping optimizations. In each plot one can observe the region corresponding to different values of single qubit gate density (near the sharp falls).

7 RELATED WORK

We now quickly review some of the most relevant work in the area of quantum compilation frameworks. As we will see, the vast majority of this research has been conducted in the past five to seven years and has largely concentrated on extending classical imperative languages (based in control-flow constructs) to enable the manipulation of quantum gates. Modeling affine quantum circuits shares some similarities with **polyhedral process networks (PPN)** [73, 103], independent processes that communicate with unbounded FIFOs. However, most of this work has focused on translating serial programs into parallel hardware. Polyhedral and affine transformations have also successfully targeted several architectures, among them CPUs [47, 66], GPUs [1, 8, 39, 64, 106], and FPGAs [4, 78]. Reference [42] discusses a quantum compiler architecture with stages from the quantum program up to its mapping to hardware.

Knill’s work [53] provided some of the first known quantum pseudo codes, examples, and use cases of quantum computing. Gay [31] and Selinger [83] surveyed the existing quantum programming languages in the mid 2000s. Functional languages have been proposed as viable candidates for the specification of quantum programs: Reference [63] introduced a statically typed functional DSL; QML [5] focused on allowing the specification of reversible and irreversible quantum computations and combining it with first-order strict linear logic. The Scaffold language and the ScaffoldC

Fig. 17. Impact of **pluto-maxfuse** fusion heuristic on critical path of QUEKO benchmarks - BigD class for grid, multi-ring, and tiled topologies.

compiler [48, 49] allow for a modular organization of quantum programs and are equipped with control-flow constructs that allow to manipulate quantum gates. Loke, Wang, and Chen [18, 67] developed the Qcompiler and OptQC, which focused on the optimization of circuits by determining permutation matrices that minimized the number of required swap gates. Their algorithm used simulated annealing to determine near optimal number of swap gates. Svore et al. developed Q# [95], a DSL with a rich type system, modular definitions, reversible operations, control-flow constructs, and qubit management. Similarly, quantum instructions sets [90] and assembly languages such as OpenQASM [24] and cQASM [12] (the common Quantum Assembly language) have also been proposed. Qiskit [46] is an open-sourced quantum toolkit, available as a Python package, which enables users to write programs with OpenQASM and run them in the IBM Quantum Experience [79], a cloud service. These efforts embody important software building blocks that new quantum compiler infrastructures can build upon to develop more scalable and high-level frameworks. RevKit [91, 92] has also been used to support fully automatic synthesis of quantum circuits [93]. Quipper [35, 36], an embedded quantum programming language for circuit specification developed by Green et al., introduced several features such as ancilla scope and reuse, classical-to-quantum circuit lifting for automatic generation of application specific oracles, basic data types, boxed/procedural definition, and reversing operators for defined circuits. LIQUi> [107], a DSL for quantum computing, proposed language features such as static typing, opaque types for qubit and kets representation, and introspection functionality that uses Microsoft's F# language and .NET support. QWIRE [74], a language designed for the specification of quantum circuits, comes with a strong type system and safe properties for well-defined circuits. QWIRE treats circuits as first-class citizens, and provides boxing and unboxing functionality that enables the composition of circuits. In addition, it leverages dynamic lifting to convert a quantum circuit to its classical equivalent. The problem of qubit allocation, similar to classical register allocation, attempts to schedule and assign quantum

operations while considering physical and topological constraints. These impose restrictions on the type of operations that can be executed on particular qubits, for instance, by considering the connectivity (neighbors) of a qubit in a coupling graph. Various approaches have been recently proposed including depth-based partitioning and A* search with look-ahead [110], the SWAP-based **BidiREctional** heuristic (*sabre*) [65], heuristics based on weighted graph coupling that traverse the list of dependences by out-degree [89], and techniques based on sub-graph isomorphism [88]. Last, efforts to produce more robust quantum program mappings for **NISQ (Noisy Intermediate-Scale Quantum)** architectures by exploiting calibration parameters, scalability, and routing options are also being explored [72].

8 CONCLUSION AND FUTURE DIRECTIONS

In this article, we have introduced the first polyhedral quantum specification language and compiler, **AXL**. We have demonstrated how off-the-shelf polyhedral analyses and transformations from the classical HPC world can be applied and be beneficial to quantum computing. In addition, **AXL** raises the level of abstraction in quantum circuit design and specification by defining the concept of affine quantum circuits, groupings of one or more quantum gates that exhibit regularity and affine relations. We believe that such a high-level specification is the missing link between previously introduced quantum languages and lower-level implementations such as quantum computers and simulators. We have found that even not-yet-tailored transformations can improve state-of-the-art qubit allocators such as **jku** and **sabre** by as much as 36%, and others such as **wpm** by 60%. The natural next step is to devise different model-driven optimizations that embed parallelism constraints, as well as considering the underlying machine topology, possibly relying on iterative optimization techniques [76].

APPENDICES

A –AXL BENCHMARKS

We show in this Appendix the set of evaluated circuits, which are summarized in Table 2. These should be viewed as small computational building blocks used in larger applications, in similar spirit to modern and classic computational kernels such as DGEMM.

Here, we quickly summarize their overall role/goal:

- **sum₅,init-G₅**: Sub-circuits used in Reference [96] corresponding to the computation of initial values and sum of an adder. The circuits do not use ancillary qubits and have depth $O(n)$. The corresponding circuit diagrams shown in Figure 19(e) and Figure 19(a).
- **pipelined-swap**: Performs a qubit SWAP operation between two qubits $2N$ lanes apart. We show in Figure 18 its implementation in **AXL**, and its circuit diagram in Figure 19(g).
- **cnt3-5_179**: A five-digit binary coded ternary counter with count control input (cnt) [108]. Circuit diagram shown in Figure 19(c).
- **cheung**: sub-circuit used in computing the $\vec{d}$ component of the Elliptic Curve Discrete Logarithm problem (ECDLP) [19]. Circuit diagram shown in Figure 19(f).
- **rd84_142**: Counts the number of ones in the input [108]. Circuit diagram shown in Figure 19(d).
- **adder-maj-uma**: is a ripple-carry adder using the in-place “MAJority” (MAJ) and “UnMAjority and Add” (UMA) patterns [25]. Circuit diagram shown in Figure 19(b).
- **cuccaro-adder-6bit**: Depth optimized ripple-carry adder of depth $2n + 4$, with $2n - 1$ time slices and 5 CNOT time-slices [25]. Circuit diagram shown in Figure 19(h).

```

1  param N;
2  statement S1a, S1b, S1c;
3  statement S2a, S2b, S2c;
4  statement S3;
5  statement S4a, S4b, S4c;
6  statement S5a, S5b, S5c;
7
8  S1a := { i : 0 <= i < N (%) #CNOT(i, i+1) };
9  S1b := { i : 0 <= i < N (%) #CNOT(i+1, i) };
10 S1c := { i : 0 <= i < N (%) #CNOT(i, i+1) };
11
12 S2a := { i : 0 <= i < N (%) #CNOT(2*N-i+1, 2*N-i) };
13 S2b := { i : 0 <= i < N (%) #CNOT(2*N-i, 2*N-i+1) };
14 S2c := { i : 0 <= i < N (%) #CNOT(2*N-i+1, 2*N-i) };
15
16 S3 := { i : i = N (%) #CNOT(i, i+1) (+)
17       #CNOT(i+1, i) (+) #CNOT(i, i+1) };
18
19 S4a := { i : 0 <= i < N (%) #CNOT(N-i-1, N-i) };
20 S4b := { i : 0 <= i < N (%) #CNOT(N-i, N-i-1) };
21 S4c := { i : 0 <= i < N (%) #CNOT(N-i-1, N-i) };
22
23 S5a := { i : 0 < i <= N (%) #CNOT(N+i+1, N+i) };
24 S5b := { i : 0 < i <= N (%) #CNOT(N+i, N+i+1) };
25 S5c := { i : 0 < i <= N (%) #CNOT(N+i+1, N+i) };
26
27 codegen { S1a (+) S1b (+) S1c (+) S2a (+) S2b (+)
28          S2c (+) S3 (+) S4a (+) S4b (+) S4c (+) S5a (+)
29          S5b (+) S5c } with { N=6};

```

Fig. 18. AXL implementation of pipelined-swap circuit.

B – AXL CIRCUIT DIAGRAMS

This Appendix shows the circuit diagrams of the evaluated benchmarks.

Fig. 19. Evaluated Quantum Circuits (rendered with ProjecQ; barriers only included for rendering purposes).

REFERENCES

- [1] Khaled Abdelaal and Martin Kong. 2021. Tile size selection of affine programs for GPGPUs using polyhedral cross-compilation. In *Proceedings of the International Conference on Supercomputing (ICS'21)*. ACM, New York, NY, 13. DOI: <https://doi.org/10.1145/3447818.3460369>
- [2] Aravind Acharya, Uday Bondhugula, and Albert Cohen. 2018. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, 529–542. DOI: <https://doi.org/10.1145/3192366.3192401>

- [3] Christophe Alias. 2019. Fkcc: The Farkas calculator. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part II (Lecture Notes in Computer Science)*, Vol. 12233. Springer, 526–536. DOI: https://doi.org/10.1007/978-3-030-54997-8_32
- [4] Christophe Alias, Alain Darté, and Alexandru Plesco. 2013. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'13)*. 575–580.
- [5] Thorsten Altenkirch and Jonathan Grattage. 2005. A functional quantum programming language. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science*. IEEE, 249–258.
- [6] Travis Augustine, Janarthanan Sarma, Louis-Noël Pouchet, and Gabriel Rodriguez. 2019. Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. Association for Computing Machinery, New York, NY, 625–639. DOI: <https://doi.org/10.1145/3314221.3314615>
- [7] Adriano Barenco, Artur Ekert, Kalle-Antti Suominen, and Päivi Törmä. 1996. Approximate quantum Fourier transform and decoherence. *Phys. Rev. A* 54, 1 (1996), 139.
- [8] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd International Conference on Supercomputing (ICS'08)*. ACM, New York, NY, 225–234. DOI: <https://doi.org/10.1145/1375527.1375562>
- [9] Cedric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*. 7–16. DOI: <https://doi.org/10.1109/PACT.2004.1342537>
- [10] Stephane Beauregard. 2002. Circuit for Shor's algorithm using $2n+3$ qubits. *arXiv preprint quant-ph/0205095* (2002).
- [11] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. Association for Computing Machinery, New York, NY, 286–300. DOI: <https://doi.org/10.1145/3385412.3386007>
- [12] Lev S. Bishop. 2017. QASM 2.0: A quantum circuit intermediate representation. In *Proceedings of the APS Meeting Abstracts*.
- [13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 101–113. DOI: <https://doi.org/10.1145/1375581.1375595>
- [14] Pierre Boulet, Alain Darté, Georges-André Silber, and Frédéric Vivien. 1998. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Comput.* 24, 3–4 (1998), 421–444.
- [15] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Tight bounds on quantum searching. *Fortsch. Phys.: Prog. Phys.* 46, 4–5 (1998), 493–505.
- [16] Gregory J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*. Association for Computing Machinery, New York, NY, 98–105. DOI: <https://doi.org/10.1145/800230.806984>
- [17] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'20)*. Association for Computing Machinery, New York, NY, 83–96. DOI: <https://doi.org/10.1145/3410463.3414635>
- [18] Y. G. Chen and J. B. Wang. 2013. Qcompiler: Quantum compilation with the CSD method. *Comput. Phys. Commun.* 184, 3 (2013), 853–865.
- [19] Donny Cheung, Dmitri Maslov, Jimson Mathew, and Dhiraj K. Pradhan. 2008. On the design and optimization of a quantum polynomial-time attack on elliptic curve cryptography. In *Proceedings of the Workshop on Quantum Computation, Communication, and Cryptography*. Springer, 96–104.
- [20] Je a Chilverini, J. Britton, D. Leibfried, E. Knill, Murray D. Barrett, R. B. Blakestad, Wayne M. Itano, John D. Jost, C. Langer, R. Ozeri, et al. 2005. Implementation of the semiclassical quantum fourier transform in a scalable system. *Science* 308, 5724 (2005), 997–1000.
- [21] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. 2017. Programming languages and compiler design for realistic quantum hardware. *Nature* 549, 7671 (2017), 180–187.
- [22] Richard Cleve and John Watrous. 2000. Fast parallel circuits for the quantum Fourier transform. In *Proceedings of the 41st Symposium on Foundations of Computer Science*. IEEE, 526–536.
- [23] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. 2005. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th International Conference on Supercomputing*. ACM, 151–160.

- [24] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).
- [25] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. 2004. A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184* (2004).
- [26] Alain Darte. 1999. On the complexity of loop fusion. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 149–157.
- [27] Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Oper. Res.* 22, 3 (1988), 243–268.
- [28] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Prog.* 20, 1 (01 Feb. 1991), 23–53. DOI: <https://doi.org/10.1007/BF01407931>
- [29] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *Int. J. Parallel Prog.* 21, 5 (01 Oct. 1992), 313–347. DOI: <https://doi.org/10.1007/BF01407835>
- [30] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Prog.* 21, 6 (1992), 389–420.
- [31] Simon J. Gay. 2006. Quantum programming languages: Survey and bibliography. *Math. Struct. Comput. Sci.* 16, 4 (2006), 581–600.
- [32] Craig Gidney. 2017. Algorithmic Assertions: Breaking Down the Quantum Swap. Retrieved from <https://algassert.com/post/1717>.
- [33] Craig Gidney. 2017. Algorithmic Assertions: CnNot from O(n) basic gates and n-2 borrowed ancillas. Retrieved from <https://algassert.com/post/1701>.
- [34] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Prog.* 34, 3 (2006), 261–317.
- [35] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. An introduction to quantum programming in quipper. In *Proceedings of the International Conference on Reversible Computation*. Springer, 110–124.
- [36] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: A scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’13)*. ACM, New York, NY, 333–342. DOI: <https://doi.org/10.1145/2491956.2462177>
- [37] Martin Griebl. 2004. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau.
- [38] Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index set splitting. *Int. J. Parallel Prog.* 28, 6 (2000), 607–631. DOI: <https://doi.org/10.1023/A:1007516818651>
- [39] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the International Conference on Supercomputing (ICS’16)*. ACM, New York, NY. DOI: <https://doi.org/10.1145/2925426.2926286>
- [40] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th ACM Symposium on Theory of Computing*. ACM, 212–219.
- [41] Lisa Hales and Sean Hallgren. 2000. An improved quantum Fourier transform algorithm and applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science*. IEEE, 515–525.
- [42] Thomas Häner, Damian S. Steiger, Krysta Svore, and Matthias Troyer. 2018. A software methodology for compiling quantum programs. *Quant. Sci. Technol.* 3, 2 (2018).
- [43] P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cyber.* 4, 2 (1968), 100–107.
- [44] Jeff Heckey, Shruti Patil, Ali JavadiAbhari, Adam Holmes, Daniel Kudrow, Kenneth R. Brown, Diana Franklin, Frederic T. Chong, and Margaret Martonosi. 2015. Compiler management of communication and parallelism for quantum computation. *ACM SIGARCH Comput. Archit. News* 43, 1 (2015), 445–456.
- [45] IBM. 2018. Qiskit. Retrieved from <https://qiskit.org>.
- [46] IBM-Research, Alexandre Blais, Robin Blume Kohout, Francesco Buscemi, Jonas Bylander, Isaac Chuang, Vincent Dwyer, Mark Everitt, Michael Geller, Thomas Haener, Andrew Houck, Thomas Ihn, Sabre Kais, Miguel Ángel Martín-Delgado, Rod Van Meter, Harumichi Nishimura, Russell Rundle, Enrique Solano, Damian Steiger, Todd Tilma, Christophe Vuillot, Martin Weides, James Wootton, and Shigeru Yamashita. 2018. Open Source Quantum Information Science Kit. Retrieved from <https://qiskit.org>.
- [47] François Irigoien and Rémy Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’88)*. ACM, New York, NY, 319–329. DOI: <https://doi.org/10.1145/73560.73588>

- [48] Ali JavadiAbhari, Arvin Faruque, Mohammad J. Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. 2012. *Scaffold: Quantum Programming Language*. Technical Report. Princeton University Department of Computer Science, Princeton, NJ.
- [49] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2015. Scaffold: Scalable compilation and analysis of quantum programs. *Parallel Comput.* 45 (2015), 2–17.
- [50] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549, 7671 (2017), 242–246.
- [51] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.
- [52] Alain Ketterlin and Philippe Clauss. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*. Association for Computing Machinery, New York, NY, 94–103. DOI : <https://doi.org/10.1145/1356058.1356071>
- [53] Emmanuel Knill. 1996. *Conventions for Quantum Pseudocode*. Technical Report. Los Alamos National Lab., NM.
- [54] Martin Kong. 2020. Exploring the impact of affine loop transformations in qubit allocation. *arXiv preprint arXiv:2010.11999* (2020).
- [55] Martin Kong. 2021. Axl-Bench: A Benchmark Suite of Affine Quantum Circuits. Retrieved from <https://github.com/mkongiv/axl-bench.git>.
- [56] Martin Kong. 2021. AXL: The Affine Quantum Circuit Specification Language. Retrieved from <https://github.com/mkongiv/axl.git>.
- [57] Martin Kong. 2021. Example of OpenQASM code produced with Qiskit and Pluto-Maxfuse. Retrieved from <https://github.com/mkongiv/plutomax-qiskit.git>.
- [58] Martin Kong and Louis-Noël Pouchet. 2019. Model-driven transformations for multi- and many-core CPUs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, NY, 469–484. DOI : <https://doi.org/10.1145/3314221.3314653>
- [59] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 127–138. DOI : <https://doi.org/10.1145/2491956.2462187>
- [60] P. G. Kwiat, J. R. Mitchell, P. D. D. Schwindt, and A. G. White. 2000. Grover's search algorithm: An optical approach. *J. Mod. Opt.* 47, 2–3 (2000), 257–266.
- [61] Thaddeus D. Ladd, Fedor Jelezko, Raymond Laflamme, Yasunobu Nakamura, Christopher Monroe, and Jeremy Lloyd O'Brien. 2010. Quantum computers. *Nature* 464, 7285 (2010), 45.
- [62] Ben P. Lanyon, Till J. Weinhold, Nathan K. Langford, Marco Barbieri, Daniel F. V. James, Alexei Gilchrist, and Andrew G. White. 2007. Experimental demonstration of a compiled version of Shor's algorithm with quantum entanglement. *Phys. Rev. Lett.* 99, 25 (2007), 250505.
- [63] Andrei Lapets, Marcus P. da Silva, Mike Thome, Aaron Adler, Jacob Beal, and Martin Rötteler. 2013. QuaFL: A typed DSL for quantum programming. In *Proceedings of the 1st Workshop on Functional Programming Concepts in Domain-specific Languages*. ACM, 19–26.
- [64] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. 2010. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-purpose Computation on Graphics Processing Units (GPGPU-3)*. ACM, New York, NY, 51–61. DOI : <https://doi.org/10.1145/1735688.1735698>
- [65] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, NY, 1001–1014. DOI : <https://doi.org/10.1145/3297858.3304023>
- [66] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99)*. Association for Computing Machinery, New York, NY, 228–237. DOI : <https://doi.org/10.1145/305138.305197>
- [67] Thomas Loke and Jingbo B. Wang. 2016. OptQC v1.3: An (updated) optimized parallel quantum compiler. *Comput. Phys. Commun.* 207 (2016), 531–532.
- [68] Enrique Martin-Lopez, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L. O'Brien. 2012. Experimental realization of Shor's quantum factoring algorithm using qubit recycling. *Nat. Photon.* 6, 11 (2012), 773.
- [69] Jarrod R. McClean, Ian D. Kivlichan, Damian S. Steiger, Yudong Cao, E. Schuyler Fried, Craig Gidney, Thomas Häner, Vojtěch Havlíček, Zhang Jiang, Matthew Neeley, et al. 2017. OpenFermion: The electronic structure package for quantum computers. *arXiv preprint arXiv:1710.07629* (2017).

- [70] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving data locality with loop transformations. *ACM Trans. Prog. Lang. Syst.* 18, 4 (July 1996), 424–453. DOI : <https://doi.org/10.1145/233561.233564>
- [71] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. R-stream compiler. In *Encyclopedia of Parallel Computing*. Springer, 1756–1765.
- [72] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, NY, 1015–1029. DOI : <https://doi.org/10.1145/3297858.3304075>
- [73] Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov. 2013. Automated generation of polyhedral process networks from affine nested-loop programs with dynamic loop bounds. *ACM Trans. Embed. Comput. Syst.* 13, 1s (Dec. 2013). DOI : <https://doi.org/10.1145/2536747.2536750>
- [74] 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)*. ACM, New York, NY, 846–858. DOI : <https://doi.org/10.1145/3009837.3009894>
- [75] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.* 21, 5 (Sept. 1999), 895–913. DOI : <https://doi.org/10.1145/330249.330250>
- [76] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. *SIGPLAN Not.* 43, 6 (June 2008), 90–100. DOI : <https://doi.org/10.1145/1379022.1375594>
- [77] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 549–562. DOI : <https://doi.org/10.1145/1926385.1926449>
- [78] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA'13)*. ACM, New York, NY, 29–38. DOI : <https://doi.org/10.1145/2435264.2435273>
- [79] IBM Research. 2017. IBM Q Experience. Retrieved from <https://www.research.ibm.com/ibm-q/>.
- [80] Revlib. 2020. Revlib: An Online Resource for Reversible Functions and Circuits. Retrieved from <http://www.revlib.org/functions.php>.
- [81] Gabriel Rodríguez, José M. Andión, Mahmut T. Kandemir, and Juan Touriño. 2016. Trace-based affine reconstruction of codes. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'16)*. ACM, New York, NY, 139–149. DOI : <https://doi.org/10.1145/2854038.2854056>
- [82] Alexander Schrijver. 1998. Theory of linear and integer programming. In *Wiley-Interscience Series In Discrete Mathematics And Optimization*. Wiley.
- [83] Peter Selinger. 2004. A brief survey of quantum programming languages. In *Proceedings of the International Symposium on Functional and Logic Programming*. Springer, 1–6.
- [84] Peter W. Shor. 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Symposium on Foundations of Computer Science*. IEEE, 124–134.
- [85] Peter W. Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.* 41, 2 (1999), 303–332.
- [86] Marcos Yukio Siraichi. 2018. Enfield. Retrieved from <https://github.com/ysiraichi/enfield.git>.
- [87] Marcos Yukio Siraichi. 2020. Qubit Allocation: The Enfield Project. Retrieved from <http://cuda.dcc.ufmg.br/enfield/>.
- [88] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. 2019. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proc. ACM Prog. Lang.* 3, OOPSLA (Oct. 2019). DOI : <https://doi.org/10.1145/3360546>
- [89] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. 2018. Qubit allocation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'18)*. ACM, New York, NY, 113–125. DOI : <https://doi.org/10.1145/3168822>
- [90] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355* (2016).
- [91] Mathias Soeken, Stefan Frehse, Robert Wille, and Rolf Drechsler. 2011. RevKit: An open source toolkit for the design of reversible circuits. In *Proceedings of the International Workshop on Reversible Computation*. Springer, 64–76.
- [92] Mathias Soeken, Stefan Frehse, Robert Wille, and Rolf Drechsler. 2012. RevKit: A toolkit for reversible circuit design. *Mult.-val. Logic Soft Comput.* 18, 1 (2012), 55–65.
- [93] Mathias Soeken, Thomas Haener, and Martin Roetteler. 2018. Programming quantum computers using design automation. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. IEEE, 137–146.

- [94] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An open source software framework for quantum computing. *Quantum* 2 (2018), 49.
- [95] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop*. ACM, 7.
- [96] Yasuhiro Takahashi and Noboru Kunihiro. 2008. A fast quantum circuit for addition with few qubits. *Quantum Info. Comput.* 8, 6 (July 2008), 636–649. Retrieved from <http://dl.acm.org/citation.cfm?id=2016976.2016981>.
- [97] Bochen Tan and Jason Cong. 2020. Optimality study of existing quantum computing layout synthesis tools. *IEEE Trans. Comput.* (2020), 1–12. DOI: <https://doi.org/10.1109/TC.2020.3009140>
- [98] Nicolas Vasilache. 2007. *Scalable Program Optimization Techniques in the Polyhedra Model*. Ph.D. Dissertation. University of Paris-Sud 11.
- [99] Vlatko Vedral, Adriano Barenco, and Artur Ekert. 1996. Quantum networks for elementary arithmetic operations. *Phys. Rev. A* 54, 1 (July 1996), 147–153. DOI: <https://doi.org/10.1103/physreva.54.147>
- [100] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, New York, NY, 521–532. DOI: <https://doi.org/10.1145/2737924.2738003>
- [101] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. 2014. Non-affine extensions to polyhedral code generation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, NY. DOI: <https://doi.org/10.1145/2581122.2544141>
- [102] Davide Venturelli, Minh Do, Eleanor Gilbert Rieffel, and Jeremy Frank. 2017. Temporal planning for compilation of quantum approximate optimization circuits. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'17)*. 4440–4446.
- [103] Sven Verdoolaege. 2009. Polyhedral process networks. Retrieved from <https://lirias.kuleuven.be/retrieve/91003>.
- [104] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *Proceedings of the International Congress on Mathematical Software*, Vol. 6327. Springer, 299–302.
- [105] Sven Verdoolaege. 2011. Counting affine calculator and applications. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*.
- [106] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013). DOI: <https://doi.org/10.1145/2400682.2400713>
- [107] Dave Wecker and Krysta M. Svore. 2014. LIQUi|>: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467* (2014).
- [108] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W. Dueck, and Rolf Drechsler. 2008. RevLib: An online resource for reversible functions and reversible circuits. In *Proceedings of the International Symposium on Multi-valued Logic*. 220–225. Retrieved from <http://www.revlib.org>.
- [109] Ze-Liang Xiang, Sahel Ashhab, J. Q. You, and Franco Nori. 2013. Hybrid quantum circuits: Superconducting circuits interacting with other quantum systems. *Rev. Mod. Phys.* 85, 2 (2013), 623.
- [110] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. Efficient mapping of quantum circuits to the IBM QX architectures. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'18)*. 1135–1138. DOI: <https://doi.org/10.23919/DATE.2018.8342181>

Received October 2020; revised March 2021; accepted May 2021