Exploiting the Regular Structure of Modern Quantum Architectures for Compiling and Optimizing Programs with Permutable Operators

Yuwei Jin
yj243@scarletmail.rutgers.edu
Rutgers University
USA

Fei Hua
huafei90@gmail.com
Rutgers University
USA

Yanhao Chen
chenyh64@gmail.com
Rutgers University
USA

Ari Hayes
arihayes@gmail.com
Rutgers University
USA

Chi Zhang
raymond.chizhang@gmail.com
University of Pittsburgh
USA

Eddy Z. Zhang
eddy.zhengzhang@gmail.com
Rutgers University
USA

ABSTRACT

A critical feature in today's quantum circuit is that they have permutable two-qubit operators. The flexibility in ordering the permutable two-qubit gates leads to more compiler optimization opportunities. However, it also imposes significant challenges due to the additional degree of freedom. Our Contributions are two-fold. We first propose a general methodology that can find structured solutions for scalable quantum hardware. It breaks down the complex compilation problem into two sub-problems that can be solved at small scale. Second, we show how such a structured method can be adapted to practical cases that handle sparsity of the input problem graphs and the noise variability in real hardware. Our evaluation evaluates our method on IBM and Google architecture coupling graphs for up to 1,024 qubits and demonstrate better result in both depth and gate count – by up to 72% reduction in depth, and 66% reduction in gate count. Our real experiments on IBM Mumbai show that we can find better expected minimal energy than the state-of-the-art baseline.

CCS CONCEPTS

• **Computer Systems Organization** → **Quantum Computing**.

KEYWORDS

Quantum Circuit Compilation, QAOA, Circuit Fidelity

1 INTRODUCTION

Quantum computing is gaining traction due to significant progress made in quantum hardware. The coherence time of a qubit, analogue to a qubit's lifetime, has increased exponentially in the past two decades [23], much like Moore's law for classical computers. Notably, IBM is projected to launch quantum computers with over 4,000 qubits in the near future [7], for gate-based universal quantum computing. The rapid advancement in quantum hardware has made it possible for quantum computers to soon solve problems that are currently beyond reach of the best classical computers.

Regular patterns start to emerge in large-scale quantum computers. In the past, the industry has explored various small-scale architectures with different connectivity structures [12, 26]. During this time, it was necessary to design compilers for arbitrary architectures. However, to build large scalable architectures, it needs well-structured building blocks. For instance, IBM has adopted the heavy-hex [2] structure for its large-scale architecture, as shown in Fig. 1 (b). Google adopted a rotated square lattice for its Sycamore architecture, as shown in Fig. 1 (a). Either Google or IBM has chosen their building blocks for specific reasons. Since IBM uses fixed-frequency qubits and cross-resonance gates, it is necessary to have heavy-hex architecture as its building block, in order to mitigate crosstalk. Since Google uses tunable transmon qubits, the crosstalk is less severe. Hence, it can create a square lattice, thereby facilitating the deployment of surface code QEC, one of the most powerful QECs. But in either case, the scalable architecture chosen by the hardware vendors start to show recurring patterns (building blocks).

Implications for Compiler Builders Previous compilation methods [18, 21, 25, 27–30, 35, 37] are designed for quantum architecture with arbitrary connectivity. In this paper, we find that it is time to exploit the regularity in quantum architecture. Through extensive experimentation, we discover that leveraging the regularity can significantly improve circuit performance and fidelity.

In particular, we focus on exploring the advantage of hardware regularity in a specific class of applications – QAOA and 2-Local Hamiltonian Simulation. This class of applications are unique in that their main circuit consists of permutable two-qubit operators.

We focus on the QAOA and 2-Local Hamiltonian Simulation also because they have broad applications. QAOA represents quantum approximate optimization algorithms (QAOA) [9]. QAOA are used to solve combinatorial optimization problems [8]. 2-local Hamiltonian simulations are useful for many physical systems [16] [10]. Both QAOA and 2-local Hamiltonian simulation allow one to flexibly permutate two-qubit operators in the circuit without changing the execution result. An example is in Fig. 2, where permuting the gates reduces the original logical circuit depth by 25%.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0394-2/23/03…$15.00
https://doi.org/10.1145/3623278.3624751

The importance of the QAOA and 2-local Hamiltonian simulation applications has led to the development of domain-specific compilers [3, 4, 16, 19]. But none of the studies leverages the regularity in the hardware. Tan *et al.* [30] provides a constrained SAT solver and combine it with heuristics to optimize QAOA programs. Alam *et al.* [3] use connectivity strength to improve compilation efficiency. The Paulihedral framework [19] includes new intermediate representation and optimizations for arbitrary Pauli-strings in Hamiltonian simulation, where 2-local simulation is a special case. 2QAN [16] tackles the problem with a quadratic-time solver and uses unitary unifying to reduce the final gate count.

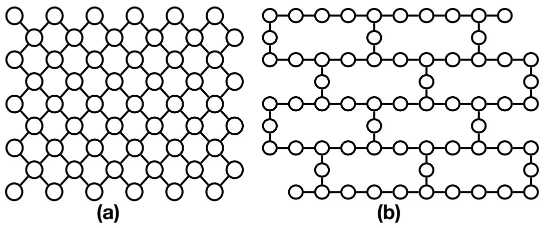


Figure 1: (a) Google Sycamore (b) IBM Heavy-hex

Our Main Contributions Our work is the first **architecture-regularity-aware** design for compiling quantum circuits with permutable 2-qubit operators. Our main contributions are two-fold: (1) We use the depth-optimal solution derived from small-cases to generalize solutions for large-cases; (2) We adapt the large-case method for practical scenarios while considering realistic factors: noise, sparsity, and circuit depth.

Insight 1: From Small Architectures to Large Architectures. Our first main contribution is the idea of using a depth-optimal solver to find solutions for small-scale architectures, and if the obtained solution is generalizable, we generalize it to larger cases. In particular, we solve for the special case where all pairs of qubits need to interact with each other, which is a challenging problem even for small instances, as the SWAP insertion problem is NP-hard in general [28, 30].

To overcome this challenge, we propose a divide-and-conquer method to convert a quadratic problem to a near-linear problem. Using this method, we successfully derived generalizable **linear-depth** solutions for various architectures including Google Sycamore, 2D grid, and hexagon.

Insight 2: Taking the Best of Both Worlds. After solving the special all-to-all interaction case, it is necessary to handle practical scenarios. The two-qubit gate interaction graph typically does not form a clique. However, it is a subgraph of a clique. We can simply adopt the compiled circuit for that of a clique interaction graph, and skip the gates that are not in the practical circuit. This method guarantees linear depth, since the clique-circuit has linear depth. But it may lead to idle cycles and unnecessary SWAPs.

Our second main contribution is the idea of combining greedy heuristics with the structured solution from the clique-circuit. It does not rigidly follow the compilation pattern for the clique circuit. Instead, it performs greedy SWAP insertion when necessary for the sparse (sub-)problem.

But just using greedy heuristic method do not guarantee the worst-case bound of linear depth. Our method combines the benefits of both in our compilation framework such that it always produces a compiled circuit that is better or at least the same good as the one that would otherwise be produced by following the patterns in the clique-circuit. Hence, it guarantees linear depth and in the meantime can reduce idle cycles.

Last but not least, we can take noise and qubit variability into consideration in our heuristic framework, which is not possible in the approach that rigidly sticks to the compiled clique circuit.

With all of this above, we are able to obtain much improved circuit duration and gate-count on large-scale circuits. We are also able to demonstrate end-to-end application improvement for small-scale circuits on real hardware. Our detailed contributions are summarized as follows:

- **Hardware regularity exploitation:** We exploit the hardware regularity for achieving the all-to-all interaction in a multi-dimensional architectures. We show how to come up with structured solutions for IBM Heavy-hex, Google Sycamore, 2D grid, and an imaginary hexagon architecture.
- **Worst case linear-depth bound**: Since our all-to-all architecture solution has linear-depth bound and any circuit's 2-qubit gate interaction graph is a subgraph of clique, we ensure linear-depth bound for worst case scenario.
- **A depth-optimal solver for small sized circuit**: We find the all-to-all solution using an A* admissible-cost function developed in this paper. We open-source our solver here [1].
- **Scalability**: We build a compiler that takes advantage of both the all-to-all structured solution and the greedy method. Our compilation is scalable. We can compile circuits of 1,024 qubits. The compilation time almost scales linearly with the number of qubits in quantum hardware.
- **Extensive Evaluation**: Overall, we did experiments with QAOA circuits and 2-local Hamiltonian circuits. Compared with three state-of-the-art baselines [3, 16, 19], our method has up to 72% of depth reduction, 66% of CNOT gate count reduction over baseline. For large circuits, our method performs better in both gate count and depth, indicating better overall fidelity of the circuit.
- **Real Machine Experiments**: We further perform the experiments on a real quantum architecture Mumbai for end-to-end experiment results. Using our compiled circuit, we obtain better expected minimal energy within the same number of rounds with the default classical optimizer COBYLA.

Section 2 is background. We describe how to go from small cases to large cases in Section 3. In Section 4, we describe our depth-optimal solver. We provide the full-fledged compiler design in Section 5 and 6, and experiment results in Section 7.

2 BACKGROUND

2.1 Input-dependent Circuit

Unlike other quantum applications, the structure of a QAOA or 2-Local Hamiltonian circuit is not known until an input problem graph is specified. For instance, in a QAOA-Maxcut circuit, a qubit corresponds to one vertex in the input problem graph. Each CPHASE gate corresponds to an edge in the problem graph, as

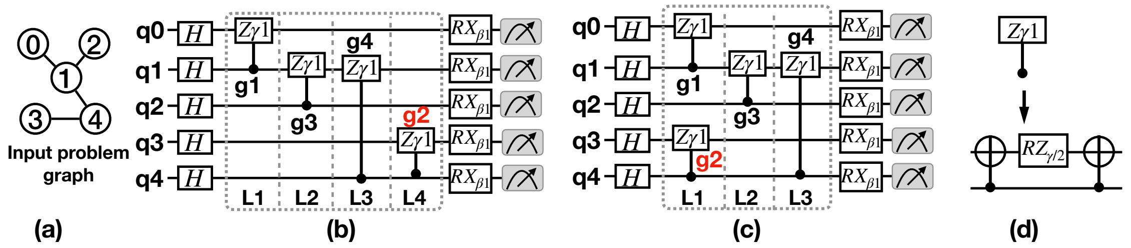


Figure 2: A five-qubit QAOA-maxcut circuit example: (a) the input problem graph where MAX-CUT is applied to. (b) and (c) are two valid circuits corresponding to the input problem graph. Note that each edge in the problem graph correspond to a gate in the logical circuit; Each node corresponds to a qubit; The gates can flexibly commute without affecting the circuit outcome [3, 17]. (d) One CPHASE gate decomposition.

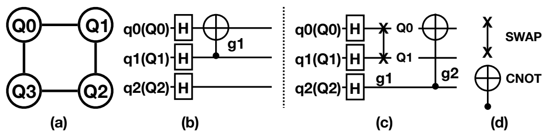


Figure 3: An example of inserting swaps for the quantum circuit. (a) Hardware coupling; (b) A circuit that does not need any swap operation; (c) A circuit needs one swap insertion; (d) SWAP and CNOT gates.

shown in Fig. 2, the edges in (a) correspond to the two-qubit gates in (b) and (c).

The two-qubit operators commute in these two types of applications. For instance, in Fig. 2, both (b) and (c) are valid instances of the QAOA circuit for the input problem graph in (a). Such flexibility can be exploited to optimize compilation, for instance, reduce the gate count and depth during the hardware mapping phase, described below in Section 2.2.

2.2 Compilation objective

SWAP Insertion Near-term quantum computers have a native gate set comprising single-qubit and two-qubit gates. A two-qubit gate can only be performed between *two connected* hardware qubits. The connectivity between qubits on a quantum chip is often limited. To overcome this limitation, the locations of two logical qubits in a two-qubit gate need to be remapped via SWAP gates until they reside on two connected physical qubits. The SWAP gates move logical qubits around to overcome the connectivity constraints.

Here is an example. Consider the hardware coupling graph shown in Fig. 3 (a). To run the circuit in Fig. 3 (b), it does not need any SWAP. But to run the circuit in Fig. 3 (c), a SWAP gate must be inserted, because q0 and q2 are not physically connected.
Commutativity To minimize the number of swap insertions, we can utilize the commutativity property in quantum computing. Commutativity arises frequently in quantum applications. Commutativity means that two quantum gates can be applied in any order without affecting the final outcome of the circuit.

To better illustrate the advantage of commutativity, an example is shown in Fig. 4. One valid circuit corresponding to the input program graph is shown in Fig. 4 (b). This circuit requires eight gates and seven layers to compile, as illustrated in Fig. 4 (c). However, permuting gates allows us to construct circuits in more ways without affecting the final measurement results. This adds more degree of freedom and makes it easier to find the best circuit with the minimum number of swap insertions. Instead of the circuit in Fig. 4 (b), we can construct another circuit in Fig. 4 (d), which only requires six gates and five layers, as shown in Fig. 4 (e). In summary, exploiting commutativity helps us find a compiled circuit with fewer layers and gate counts, which can improve efficiency and reduce errors in quantum computing.

3 FROM SMALL CASES TO LARGE CASES

Our design for the depth-optimal solver in the small cases relies on the following implications.

- IMPLICATION 1: Every QAOA program can be described using a graph, where a vertex is a qubit, and an edge is a gate, since all the 2-qubit gates are the same. Every n-vertex graph is a subgraph of a n-clique. If we can solve a n-clique QAOA circuit, we can solve any sub-circuit of the n-clique circuit. Hence we focus on the **clique graph** as the special case for which we solve for optimal solutions.
- IMPLICATION 2: In the literature of classical computing literature, there exist well-structured solutions, when the input to a problem is a clique graph, and when the underlying architecture is regular. For instance, the famous permutation network problem by Abraham Waksman [32]. The setting of the permutation network problem is different from that of our problem. We aim to work on multi-dimensional architectures, rather than linear architectures. Nevertheless, such findings in classical computing literature, inspire us to develop well-structured solutions for our QAOA mapping problems defined in this paper.
- IMPLICATION 3: It is challenging to find optimal solutions for a large-sized problem, for instance, a large clique on a large regular architecture. But it is possible to find that for a small-sized problem. If the solution for the small-sized problem is generalizable, we can generalize it for the large problems. This motivates our study for algorithmic engineering, where we design optimal solver that can handle small cases in special scenarios, and we generalize the well-structured solutions, if any, to large cases.

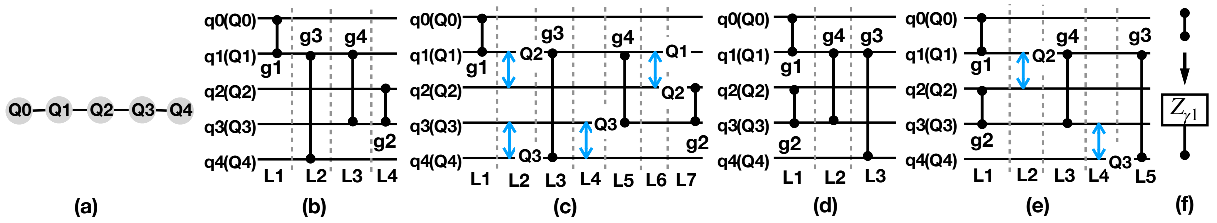


Figure 4: Different circuits result in different compilation results. (a) Hardware coupling; (b) One possible valid qaoa circuit; (c) One possible compiled circuit of (b) with four SWAP gates; (d) Another valid circuit corresponding to the same circuit in (b); (e) Depth-optimal compiled circuit of (d); (f) Control-Z gate abstracted.

With all of the implications above, we now define our special problem of compiling clique-circuit.

Definition 1. *A clique-circuit and its compilation: A clique-circuit is a circuit with exactly one 2-qubit between any two qubits. Compiling a clique-circuit is the problem of moving qubits around in the given architecture, such that each qubit is neighbor to every other qubit at least once for completing a 2-qubit gate.*

We next describe how to compile a clique-circuit on 2D grid architecture and other architectures.

3.1 A Case Study – The 2D Grid Architecture

For an NxN grid, we do not have to solve directly for a small instance of such an architecture, i.e., a 4x4 for NxN grid. Rather we take a different approach. It turns out we only need to solve the 1x4 grid and 2x4 problem in order to get the solution for 4x4, and then generalize it to NxN.

First, we divide an NxN grid into multiple rows. We call each row *a unit*. It can be seen that each unit is a line. Across the units they are connected, as shown in Fig. 5.

We now claim that if the following two-sub problems can be solved efficiently, then the NxN grid problem can be solved efficiently. The two sub-problems are the following: (1) Solve the all-to-all interaction for the line architecture, that is, for all the nodes within a unit, and (2) Solve the bipartite all-to-all interaction between 2 connected units, denoted as 2xUnit problem.

We reason about it as follows. If sub-problem (1) is solved, that means we can move around nodes by SWAPs on a line such that each node can be neighbor to every other node once. Then if we treat each unit as if it is a “node", and the unit exchange (exchange two nodes in the same column in the 2xUnit case) shown in Fig. 5 (b) as if it is a "SWAP", then we can move the units by *exchanges* such that each unit is neighbor to every other node once. Once two units are next to each other, it forms a 2xUnit architecture as shown in Fig. 5 (b). Now since we have the second sub-problem solution of bipartite all-to-all interaction between two units, this solves the bipartite all-to-all between all units. Lastly, for all-to-all interaction within each unit, we have the solution to the first sub-problem, we can conclude that all-to-all interaction for all nodes in the architecture is possible.

Now the problem of solving NxN all-to-all interaction through SWAP has been reduced to the sub-problems of 2xN and 1xN. Using our depth-optimal solver (the implementation is described in Section 4), we successfully found optimal solutions for these sub-problems. We describe each solution below.

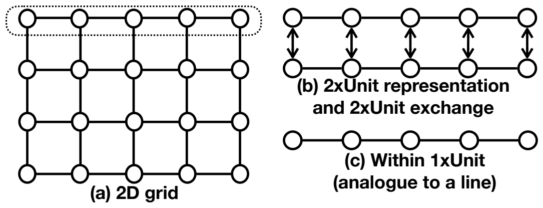

(a) 2D grid
(b) 2xUnit representation and 2xUnit exchange
(c) Within 1xUnit (analogue to a line)

Figure 5: 2D Grid Motivation 2D grid Breakdown. (a) a 4x5 grid; (b) 2xUnits representation; (c) 1xUnit representation.

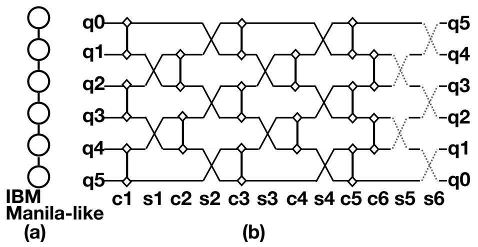


Figure 6: The solution for the 6-qubit line architecture.

Our discovery for the 1xUnit case. We denote the nodes with respect to the parity of its index, i.e., odd-index and even-index. We discovered that if we repeat odd-even and then even-odd SWAP pairs as shown in Fig. 6, we will be able to let all qubits be neighbor to every other qubit once. The corresponding loop description is shown in Fig. 7. After each SWAP layer, we let 2-qubit gates take place immediately on the same pair of qubits that just perform SWAP. After $n-2$ (n is the number of qubits) rounds of SWAP layers and n CPHASE gate layers in total, we can finish all-to-all interaction between all qubits. It is interesting to see that by adding two additional SWAP layers, the qubit locations are as if reversed, as shown in the dotted SWAPs in Fig. 6 (b).

We found such a pattern using the problem instance of 1x6 1xUnit case (raw data also in our GitHub link [1]).

Our discovery for the 2xUnit Case. For the 2xUnit bipartite all-to-all interaction, the discovery is quite interesting. It shows the pattern that, for two rows, if one row performs odd-even swaps within itself, and the other row performs even-odd swaps within

```
For cycle  = 0;  cycle += 4; cycle <= 2N-2:
    CPHASE( Q_i, Q_i+1 ); parallel for i = [0, N), step = 2
    SWAP( Q_i, Q_i+1 );    parallel for i = [1, N), step = 2
    CPHASE( Q_i, Q_i+1 ); parallel for i = [1, N), step = 2
    SWAP( Q_i, Q_i+1 );    parallel for i = [0, N), step = 2
```

Figure 7: Loop description for the linear pattern in Fig. 6. N is the number of physical qubits in the line architecture.

itself, or vice versa. In total $\leq n$ steps, we can let each node on the top row be neighbor to each node in the bottom row once and only once. Hence following each SWAP layer, we can immediately perform a computation layer. An example is shown in Fig. 8, and the corresponding loop is shown in Fig. 9. We found such solution by running our depth-optimal solver on the 2x4 problem instance.

A loop description of the generalized solution the 2-unit solution is in Fig. 9. From the loop, it can be seen easily, this solution guarantees linear depth.

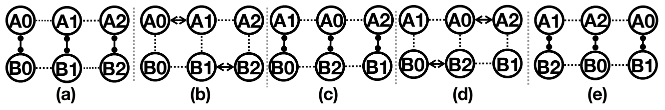


Figure 8: (a),(c), and (e) are three computation cycles. (b) and (d) are two swap cycles

```
For cycle  = 0;  cycle += 2; cycle <= 2N-1:
    start = (cycle/2) % 2;
    CPHASE( A_i, B_i );   parallel for i = [0, N), step = 1
Parallel ┌SWAP( A_i, A_i+1 );   parallel for i = [start, N), step = 2
         └SWAP( B_i, B_i+1 );   parallel for i = [1-start, N), step = 2
```

Figure 9: A loop description for bipartite all-to-all interactions in Fig. 8 . N is the size of one unit. A & B represent two adjacent units. The two parallel SWAP layers run concurrently too.

Note that our solution for 1xUnit happens to be the same as the one that is manually found in the physics literature [13, 22]. Our solution for 2xUnit happens to be the same as that found by an arXiv paper in 2022 February [33]. However, even though our paper is not formally published. The discovery of 2xUnit is in our manuscript in December 2021 (time can be verified), which is before the 2022 February arXiv paper.

Moreover, these studies did not describe how they find the solutions (we assume it is manually), but we found them by running our automatic solver, which is open-sourced and can be used by other researchers in the community. Moreover, our high level idea of reducing the full-sized problem to 1xUnit and 2xUnit sub-problems have not been proposed by any prior study. We not only give the solutions but also the methodology.

Another difference is that we find solutions for Google Sycamore (2 solutions – one automatically and the other one manually) for the first time. We also find the solution to hexagon architecture automatically for the first time. We describe our discoveries in the next section.

3.2 Other Architectures

3.2.1 *Google Sycamore.*

Now we apply the same high level approach for Google Sycamore.

First, an unit is defined for the nodes in the same horizontal line, as shown in Fig. 10 (a).

Next we reduce the full-sized problem to the following two-sub problems. The 2xUnit structure is shown in Fig. 10 (b), where two units are connected using a zig-zag shaped linked line. The edges between two units make it possible to implement the unit exchange operation in one step.

Our discovery for the 2xUnit sub-problem of Sycamore. We find structured solutions using our depth-optimal solver. Just to get an idea of how it works, we show that in Fig. 11. We first highlight the triangle areas, where each triangle is formed by three qubits. There are three possible ways to decompose the architecture into consecutive triangles, as shown in Fig. 11 (a)-(c). For each triangle organized architecture graph, we repeat three sets of operations: (1) perform operations from the first inter-unit link and its parallel other links, (2) perform operations from the second inter-unit link and its parallel ones, and (3) repeat one. For each such operation, if an inter-unit link is on a triangle, we perform a SWAP, otherwise, we perform a computational gate. This is shown in Fig. 11 (d) to (f). This is the solution we obtained by automatically solving the 2xUnit problem. We do not have a formal proof for this, but we have verified its correctness by programming this pattern, and running it with a very large number of qubits. By programming it, we find that it takes $2n-2$ steps to complete the bipartite all-to-all interaction. We find this pattern by solving the 2xUnit problem of 7 qubits (3 in one unit and 4 in the other unit).

1xUnit for Sycamore. For the 1xUnit sub-problem, we can indirectly solve that. Note that for every two neighboring units, we can connect a line that cover all nodes in these two units, as shown in Fig. 10 (c). Hence, we can apply the 1xUnit solution for 2D grid to solve the 1xUnit problem of Sycamore.

One may have noticed that by this point, why not just using the line-architecture solution to indirectly solve the 2xUnit solution in Sycamore. It is true. One caveat is that by directly applying the line-architecture solution, the nodes may or may not end up being in their original unit. But our automatically found solution somehow ensures this. Moreover, the line solution has un-necessary interaction for intra-unit in this case. Last but not least, we use this to show the usefulness of our optimal solver tool.

In addition to this automatic 2xUnit solution, we also have a manual 2xUnit solution with similar latency, which has not been found before. We show details in Appendix B. The full solution for N by N sycamore is also shown in Appendix.

3.2.2 *Hexagon.*

Moreover, we apply our tool to the hypothetical hexagon architecture in Fig. 12.

We define one unit as the nodes in the same vertical line, as shown in Fig. 12 (a) and (b). Exchanges can be done using the direct (horizontal) links between two units.

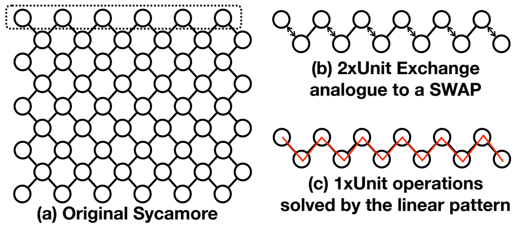

(a) Original Sycamore
(b) 2xUnit Exchange analogue to a SWAP
(c) 1xUnit operations solved by the linear pattern

Figure 10: Breakdown of the Sycamore Architecture

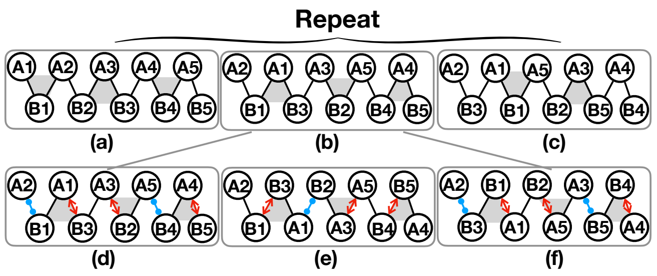


Figure 11: Sycamore pattern discovered by the optimal solver

Our discovery for the 2xUnit. The solution we have discovered for 2xUnit by our optimal solver also surprisingly follows a regular pattern. Each computation layer is followed by two SWAP layers, where the first SWAP layer is between two units, and the second SWAP layer is within each unit, as shown in Fig. 12 (c).

For 1xUnit solution, similarly, we can connect a line for all nodes in every two adjacent units. We can then apply the line architecture solution as 1xUnit case of 2D grid to indirectly solve the 1xUnit case of Hexagon architecture.

3.2.3 *IBM Heavy-hex.* For this architecture, we manually adapt it from the 1xUnit solution of 2D grid, instead of partitioning it into more coarser grained units. We find it is better this way. It is discussed in Section 5.1.

Discussion. Note that our high level idea applies to a multi-dimensional architecture beyond two-dimensional. As Fig. 13 shows for a three-dimensional lattice. It is first divided into planes. Each plane is derived into rows, and each row is divided into nodes. If we can handle all-to-all interaction between the planes as well as all-to-all interaction within each plane, we can then solve the mapping problem for a clique graph on the three-dimensional lattice. And we can reduce the two-dimensional problem to the one-dimensional problem. By doing this, we only need to solve multiple sub-problems, each smaller than the original.

4 DEPTH-OPTIMAL SOLVER

Solver: We design and implement a solver for minimizing the depth of a SWAP-inserted circuit.

Definition 2. *Given a input logical circuit C, a hardware coupling architecture H, and an initial mapping M, our optimal solver returns a transformed circuit C' which has the minimal depth among all transformed circuits from C that are satisfy the coupling constraints of H, with the initial mapping M.*

In this section, we describe how we design and implement our optimal solver. Note that the solver only minimizes the depth. When considering gate count and/or estimated success probability (ESP), it is not optimal. We leave that as our future work. Interested readers for how to handle the practical cases (non-clique large problem graphs) by adapting the small case optimal solutions (clique problem graph) can skip this section and directly go to Section 5. We first describe the A* framework we use in Section 4.1. Then we provide the proof for the optimality of our A* approach in Section 4.2.

4.1 A* Framework

Search space. We use the A* framework to efficiently explore the search space. Each node in the A* search tree represents a state of the circuit during one cycle of scheduling. Each edge presents a state transition for advancing one cycle in the scheduling and SWAP insertion process. We define the following terms.

- **A state node** describes (1) the logical-to-physical qubit mapping at the beginning of a cycle, and (2) the set of gates being scheduled at the current cycle.
- **Root node** is the starting point of the search process. It has no gate executed at current cycle. But it already has an initial mapping from logical qubits to hardware qubits.
- **Terminal node** corresponds to the last cycle of a transformed circuit. At this node, the last gate(s) of the original circuit are being completed at the terminal node's cycle.

Our A* solver first initializes the priority queue with only the root node. It then repeats the following steps: Extracting a node from the priority queue, generating child node(s) based on valid state transitions, and pushing child node(s) into the queue. It terminates when a node retrieved from the priority queue is a terminal node.

A valid state transition from a parent node to a child node corresponds to the valid actions that can happen during one cycle. In this cycle, (a set of) original gate(s) can be scheduled, and/or (a set of) SWAP gate(s) can be scheduled. A node can obtain its entire set of child node(s) by exhaustively generating all potential combinations of remaining **executable** computation gates and SWAP gate(s) in the upcoming cycle. The executable gates and SWAPs are defined as gates with two qubits on connected hardware qubits.

At the beginning of every cycle, if one or multiple SWAP occurred in the last cycle, the logical-to-qubit mapping needs to be updated, and correspondingly, the mapping is incorporated into the circuit state of the current cycle.

The priority function of a node is the key to the A* algorithm paradigm. Each time the algorithm removes a node from the queue, it must be the node with highest priority. If we set the priority function to be *admissible*, the A* paradigm guarantees optimality [37]. We discuss our priority function in details next at Section 4.2.

4.2 Priority Function

Our priority function $f(v)$ is defined over a circuit-state node v. It estimates the lower-bound number of cycles needed for all paths that start from the root node and to a terminal node, via the node v. The priority $f(v)$ consists of two components $c(v), h(v)$ such that

$$f(v) = c(v) + h(v), \tag{1}$$

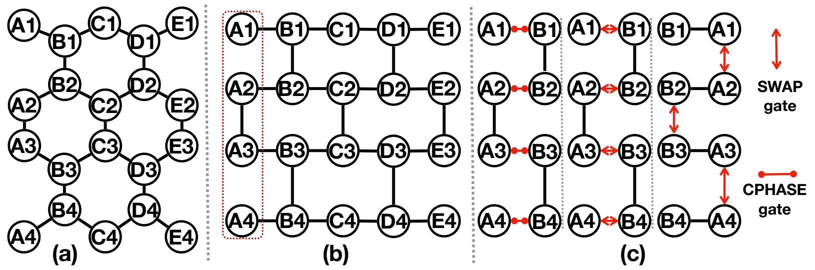


Figure 12: Hexagon coupling graph and partial compilation. (a) the hexagon graph in honey-comb structure; (b) the dragged hexagon graph in a square layout; (c) a layer of CPHASE gates followed by two layers of SWAP gates for 2xUnit.

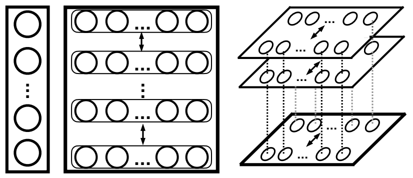


Figure 13: From 1D to 3D.

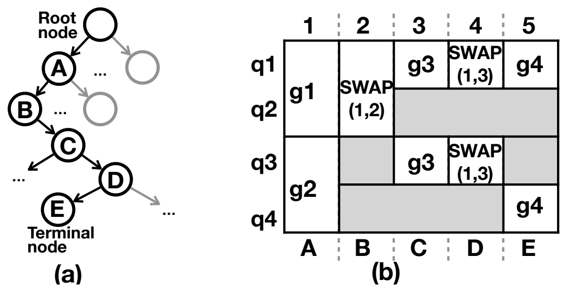


Figure 14: (a) shows a part of the A* search tree. (b) show the circuit execution cycle by cycle. A cycle's information include the set of gates (shown here), the qubit mapping (not directly shown here). Cycles A, B, C, D, and E, correspond to the tree nodes in (a). E is a terminal node as marked in (a).

where $c(v)$ represents the number of cycles it takes from the root node to the node v, which is the length of the path from the root node to v. This part is trivial. $h(v)$ represents a lower bound of the estimated cycles from the node v to some terminal node. The construction of $h(v)$ is not trivial.

Our priority function $h(v)$ considers two factors:

- The degree of a qubit q in the QAOA-graph $deg(q)$.
- The minimal number of SWAP gates that need to be inserted for a un-executable gate on two qubits q_i and q_j.

We first define the minimal number of cycles for scheduling all gates on a pair of qubits q_i and q_j.

Definition 3. *We define a lower bound of the minimal number of cycles for scheduling all gates on q_i and q_j, where there also exist a gate on q_i and q_j, and d is the distance between q_i and q_j.*

$$cost(q_i, q_j) = \min_{x=0..d-1} max(deg(q_i) + x, deg(q_j) + d - x). \quad (2)$$

Lemma 4.1. *For one pair of qubits q_i and q_j, where there is 2-qubit computation gate on them, our cost function $cost(q_i, q_j)$ indeed gives a lower bound on scheduling all the gates involving q_i and q_j.*

Proof. Since the distance between q_i and q_j is d, to move them next to each other, at least $d-1$ swaps are needed in total. We then split these $d-1$ swaps into two components: (1) the minimal SWAPs that q_i needs to take, and (2) the minimal SWAPs that q_j needs to take. In Equation 2, we consider d scenarios by splitting these $d-1$ moves among q_i and q_j. If q_i moves x steps, then q_j needs to move at least max(0, (d-1-x)) steps. There are a finite number of ways to set x, which is $x = 0..d-1$. Then we consider the original degree of q_i and q_j, q_i needs to execute x SWAPs, and $deg(q_i)$ computation gates, and q_j needs to execute $d-1-x$ SWAPs, and $deg(q_j)$ computation gates. Since the slower qubit dominates the overall execution time, we take the maximal of the two. Among all possible values of x, we take the minimal value since one can always choose the best way to split the $d-1$ moves. □

Having defined $cost(q_i, q_j))$ for a pair of qubits (q_i, q_j), we define the heuristic priority function $h(v)$.

Definition 4. *Heuristic cost function $h(v)$ gives the minimal number of cycles from v to a terminal node.*

$$h(v) = \max_{(q_i,q_j)\in E_{rem}(v)} cost(q_i, q_j) \quad (3)$$

$E_{rem}(v)$ is the set of un-executed gates implied by v and v's predecessors, from the QAOA-graph with edge set E.

Theorem 1. *Our cost function $h(v)$ is lower bound of all paths from v to a terminal node in our A* search tree.*

Proof. A compiled circuit C has larger or at least the same depth (the number of cycles assuming all gates take 1 cycle), as that of a compiled subcircuit of C. All gates on q_i and q_j are a subcircuit of C described by the edge set E_{rem}. Since E_{rem} describes the remaining circuit implied by v and v's predecessors, the path from v to a terminal node is bounded by the maximum cost of gates on all possible pairs of q_i and q_j with a gate on them. □

Theorem 2. *The priority function defined in Equation (1) is a lower bound of all paths from the root node to a terminal node via v. Hence it is admissible.*

Proof. Since the path from the root node to v is fixed, given by $c(v)$, and $h(v)$ is a lower bound for all paths from v to a terminal node, $c(v) + h(v)$ is a lower bound for all paths to a terminal node that must pass through v. Since it is proved to be a lower bound, the priority function is admissible. □

Example: We use an example to describe how we calculate the cost function $cost(q_i, q_j)$ in Fig. 15 for the input problem graph in Fig. 15(a). The initial mapping is shown on the left side of Fig. 15 (b). We let q_i and q_j be $q1$ and $q4$, where there is a gate g3 on them. Firstly, we do not know when should we schedule gate g3, so we set a gray box before and after it to represent the unknown scheduling. We set

$$cost(q_i, q_j) = min_{i=0..2} max(deg(q1) + i, deg(q4) + d - 1 - i).$$

Since $deg(q1) = 3$, $deg(q4) = 2$, and $d = 3$, we enumerate all possible values of i. Among those, we find the minimal cost(q1, q4) is when i = 1. Then, we set $cost(q1, q4) = 4$.

Discussion: Our optimal solver is for a given initial mapping. For our particular problem, since the input problem is a clique graph or a bi-clique graph, all initial mappings have the same behavior. It does not matter how we place the qubits.

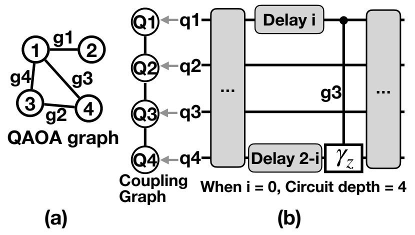


Figure 15: Cost function: (a) is the input QAOA problem graph. (b). The circuit of depth estimation using qubit pair q1 and q4.

5 HANDLING THE PRACTICAL CASES

5.1 Factor I: Flexibly Adapting the All-to-All Solution

IBM Heavy-hex. An architecture that stands out is the IBM heavy-hex, which looks like a honeycomb structure. It is possible to break it down into units. But based on our evaluation of different choices, it takes non-trivial amount of depth if we use the 1xUnit and 2xUnit approach. We decided to use a much simpler approach that have proven to be useful in practice.

Since the heavy-hex is quite sparse, we adapt the 1xUnit (line) solution of 2D grid to handle it.

Now we describe two main components in the heavy-hex architecture: (1) a longest path that can connect many nodes, and (2) the nodes that are off this path. It is shown in Fig. 16.

We categorize the operations into three types: (1) path-2-path interaction, (2) off-path-2-off-path interaction, and (3) path-2-off-path interaction. With these three, we can cover all-to-all interactions. We need two passes to cover the three.

In the first pass, all path-2-path interactions could be finished by applying the linear pattern for the longest path. In the meantime,

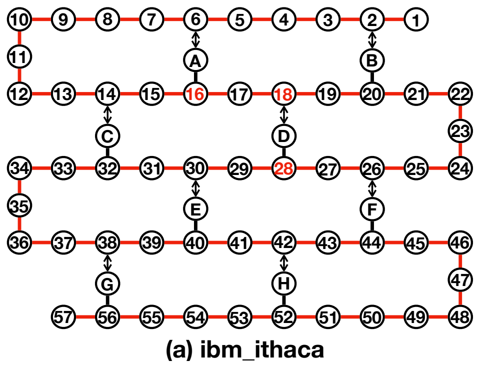


Figure 16: Heavy-hex: the numbered nodes are on the longest path, the lettered nodes are off-path.

we also handle path-2-off-path interactions, that is, we let each node interact with its off-path neighbor if there is any, before the node is moved to the next location. For each node on the path, it ensures that the node will interact with at least a part of the nodes in the off-path.

In the second pass, we swap off-path nodes with on-path nodes in one cycle, and perform another line solution together with on-path/off-path interaction like in the first pass. The course of nodes movement will be as if it is continuing that from the end of the first pass. The remaining path-2-off-path interactions would also be covered by the second pass. This guarantees that all-to-all interaction is achieved, and the detailed explanation/proof is in Appendix C.

5.2 Factor II: Non-clique problem graphs

To handle non-clique problem graphs, we can use a simple approach by exploiting the fact that every non-clique problem graph is a subgraph of the clique graph. The idea is to skip the 2-qubit gates that are not in the non-clique problem graph in the compiled circuit for the clique graph.

However, we are not satisfied with that. At certain points of the circuit, the remaining problem graph (by taking out the edges that correspond to scheduled gates) is very sparse, resulting in under utilization of the QPU. For very sparse problem graph, there is no need to rigidly follow the pattern of all-to-all interaction. We can just insert SWAPs greedily similar to prior studies [3, 4, 17, 19].

But the problem of purely using greedy solutions is that they cannot guarantee the compiled circuit has linear depth. Because it might over optimize for local optimums.

We want to take the best of both worlds. So we use a method to combine the benefits of both. We still perform the greedy compilation approach that handles the gates in the frontier and process them layer by layer. But at every layer where the qubit mapping is changed compared with last layer, we make a prediction by following the exact all-to-all solutions (with certain edges skipped) for the rest of circuit. By combining prediction for the rest of the circuit and the already processed partial circuit, we can get the gate count/fidelity/depth for the whole circuit. At the end, we look at all predictions, and check if any of them yields best whole compiled circuit than following the entire greedy compilation method. If so, we output the best compiled circuit.

5.3 Factor III: Qubit error variability

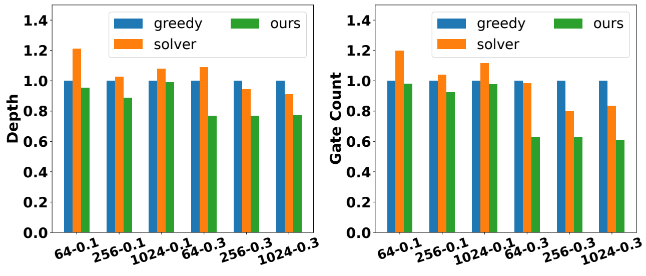
Besides the decoherence error determined by the circuit duration, the final fidelity is affected by error variability in hardware. For example, different coupling edges in the IBM machine have different CNOT error rates. The crosstalk triggered by two close and parallel CNOT gates also exists.

In the greedy component of our approach, we take those factors into the consideration. We use minimal weight perfect matching algorithms to place gates at low error coupling links. For the crosstalk issue, we integrate the graph coloring model.

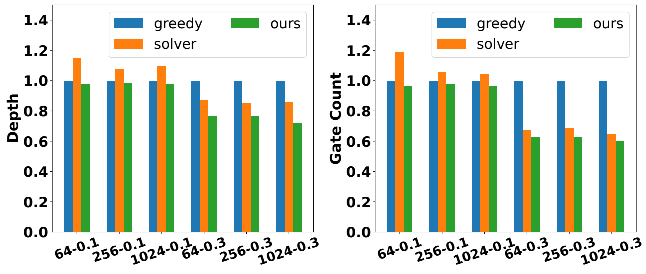
5.4 Solver Guided Solution Analysis

To quantitively analyze the strength and limitations of solver guided solution, we compare it with a pure greedy solution. As the result show in Fig. 17, we conducted comparisons on two types of architectures, Heavy-hex and Google Sycamore with a set of random graphs. We use random graphs with densities of 0.1, and 0.3. The number of vertices is 64, 256, and 1024. The "greedy" bar presents the data from the pure greedy solution. The "solver" bar represents data from the solver-guided solution. Bar "ours" presents our solution, combining both greedy and solver-guided solutions (detailed described in Section 6). All are normalized to the greedy version.

From Fig. 17, we can see that the greedy solution only has better performance than the solver-based solution, when the input problems are sparse, such as in graph 64-0.1. But the solver-based solution is better when the graphs are larger and denser. Interestingly, the mixed (our) solution beats both pure-greedy and solve-based ones for all benchmarks. That is, it is not just picking the best of the two, it is better than the best of the two.



(a) Depth in Heavy-hex (b) Gate Count in Heavyhex



(c) Depth in Sycamore (d) Gate Count in Sycamore

Figure 17: Pure-Greedy vs Solver vs Ours.

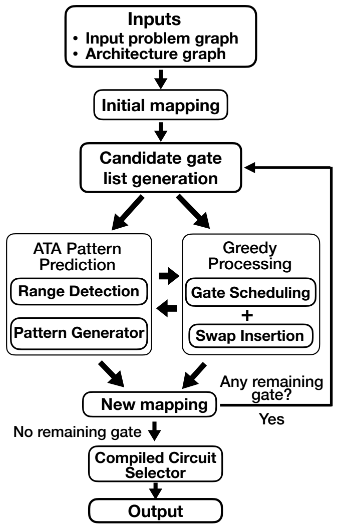


Figure 18: Our compiler framework

6 OUR COMPILER WORKFLOW

6.1 Our framework

Our framework iteratively processes candidate computation gates in the circuit. At the beginning, the candidate computation gate list includes all gates, since they all commute. Then at every cycle, it selects candidate gates to schedule and determines SWAP gates to insert. Whenever the qubit mapping is changed at the end of any cycle, it will predict for following the all-to-all solution (ATA solution) on the rest of the circuit, for instance, what the gate-count/depth/fidelity would be, and keep track of these predictions at different layers (cycles) along the compilation process. Once it reaches the end where there is no candidate gate left, it will choose a compiled circuit based on the prior recorded compilations. The overall compilation framework is shown in Fig. 18. We describe each component below.

6.2 The Greedy Processing Component

There are two sub-modules in the *greedy processing* component in Fig. 18. One is gate scheduling matching sub-module and the other is swap-insertion model.

In the gate-scheduling sub-module, we process hardware-compliant gates. We use graph coloring for this module. Each hardware-compliant gate is a node. Each edge represents if they share a qubit or if they have non-trivial crosstalk. Then we try to color the graph and choose the color that has maximal number of gates, and schedule these gates. By doing so, we take crosstalk into account.

The next-submodule is the *SWAP-insertion module*. At the same layer, we can insert SWAPs on idle qubits to help the two-qubit gates that are not hardware-compliant. We let each qubit be a node and each gate be an edge. For each potential SWAP gate (a SWAP that can reduce the distance of a separated pair of qubits that have to perform a computation gate), we assign weight $w = \frac{1}{e}$, e is the two-qubit gate error rate on the physical link. Then we find minimal weight perfect matching (MWPM) of these gates. The weights in MWPM characterize the error rate variability.

6.3 The ATA Pattern Prediction Component

The goal of ATA pattern prediction is to provide a bound for the compiled result of the rest of the circuit given current qubit mapping. After all, we only need a qubit mapping for following the all-to-all solution and skip the non-existent gates in the real problem graph. This component has two sub-modules: (1) Range detector, and (2) pattern generator.

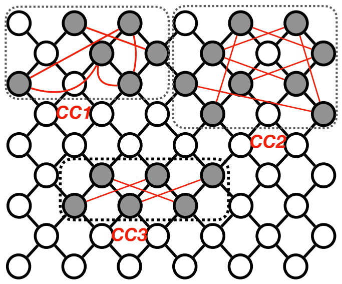


Figure 19: Range detection – we narrowed the whole architecture down to three sub-regions with respect to qubit interaction.

Range Detector. Note that as the candidate list is being updated, there are fewer and fewer gates in it. We process the sub-graph of the input problem graph that has active edges (remaining gates in the candidate list). Firstly, we find the disjoint connected components (CC) from the remaining QAOA problem graph, which refer to different set of qubits that have interaction (transitively), and we name these sets as interacting-qubit-set. Secondly, by checking with the current mapping π, we find regions π_{CC} in the architecture graph that correspond to different interacting-qubit-set. If the regions are disjoint, we make ATA prediction using a smaller-sized architecture with the same shape. If two regions overlap, we merge them into one region. For example, We have three non-overlapping regions (minimal sycamore shape that encloses an interacting-qubit-set) in Fig. 19, which means if we follow the ATA pattern, we can just limit the interaction within each bounded region, and hence given a better bound of depth/gate-count/fidelity, compared with following ATA for the entire region. We apply the same idea when we perform 2xUnit operation to further improve the bound.

Pattern generator. generates the pattern in the particular region with the given qubit mapping. This step is trivial as it simply follows the process cycles in the ATA solution until all gates in the given problem graph are processed.

6.4 Compiled Circuits Selector

When there is no gate in the candidate list, it means we have processed all gates. Now we would compare the final depth/gate-count/fidelity of the greedy compilation method, with that of the versions that have been previously recorded (partially greedy + partial ATA) at each layer when qubit mapping changes. If the estimated value of a partial greedy + partial ATA is better, we would revert compiled circuit back to that point and append it with ATA processed rest of circuit. This step guarantees that the final depth/gate-count/fidelity is bounded by the one that would otherwise be produced by the ATA solution. But it could be better than the ATA solution when the problem graph is sparse.

We choose the final version of compiled circuit by the value calculated through the cost function, $F = \alpha\% \ \frac{fD}{oD} + (1 - \alpha\%) \ fidelity^{\frac{1}{fG}}$. fD is the depth of compiled circuit, oD is the depth of original greedily compiled circuit, fG is the gate count of compiled circuit, and $fidelity$ is the product of error rate of gates in the circuit. The smaller value of F is better.

THEOREM 6.1. *Our method combines the benefits of both in our compilation framework such that it always produces a compiled circuit that is better or at least the same good as the one that would otherwise be produced by following the patterns in the clique-circuit.*

Brief Proof. Before we start the circuit compilation, given an initial mapping, the input problem graph and the underlying architecture, we can directly generate a compiled circuit by following the clique pattern exactly. We name it compiled circuit cc_0. The depth and gate count of cc_0 is linearly bounded because it is a purely solver-guided solution. Then we start doing compilation in greedy manner. At every point, we record the part of ciruit already compiled by the pure greey solution, and combine it with the rest of the circuit as if the rest of the circuit is generated by the solver-based solution. We obtain many hybrid solutions, and we name them as cc_i where $0 < i < k$ and k is the total number of solutions.

In Compiled Circuits Selector, we select the compiled circuit cc_{best} with the best F value defined above. That means cc_{best} is better than all cc_i where i denotes one of the hybrid solutions, including cc_0. Hence Compiled Circuits Selector will choose the circuit at least as good as cc_0, the one achieved by following the solver-based solution exactly. Therefore it is proved.

6.5 Limitations of Our Work

Our work is built upon the pattern discovered through our optimal solver, which is designed specifically for quantum circuits with commutable operators only. As such, there are two major limitations to our work.

Firstly, our work is not suitable for the circuit where the gate dependency is fixed. However, if the circuit contains mostly commutable gates, we may still be able to solve the compilation problem by adapting our work. We leave it as our future work. Secondly, the regularity of physical qubit connections is necessary to allow our method to be applied, as a high level idea. If the underlying architecture does not have regularity, our approach does not apply.

7 EVALUATION

In this section, we present a comprehensive analysis of our method by comparing it with state-of-art approaches QAIM [3], 2QAN [16], QAOA-OLSQ [30], SATMAP [20] and Paulihedral [19]. The comparison is conducted from various aspects, including depth, gate count, and fidelity.

7.1 Experiment Setup

Architecture. We use mainly IBM heavy-hex and Google Sycamore hardware. Both architectures have a regular pattern and can be expanded to any scale by following the pattern. IBM heavy-hex consists of a bunch of dodecagons. Google sycamore is a rotated 2D grid with certain vertices and edges missing on the boundary.

To show the scalability of our method, we scale both architectures to 1024 qubits and keep the shape close to a square. In the experiment, we use the minimum size of architecture that can handle the corresponding input problem graph.

We also use IBM machine Mumbai which has 27 qubits and has dodecagons in its structure as well. The IBM machine has error variability in its qubits and gate errors.

Metrics. We use the circuit depth, two-qubit gate count, and total variant distance (TVD) as metrics to evaluate the effectiveness of our method. Circuit depth is the length of the critical path in the compiled circuit which is correlated with the circuit duration. The circuit depth is expected as small as possible to reduce the decoherence error. We also use compilation time as a metric.

Two-qubit gate count is the number of CX gate in the compiled circuit including the original circuit gates and those decomposed from the added SWAP gates. Two-qubit gate error is dominant in the accumulated gate errors and fewer gate count means less accumulated gate errors. We decompose the compiled circuit into single-qubit basis gate and CX gates. TVD measures the distance between the output distribution of real machine experiment with the noisy free result from the simulation. So TVD can only be used for small-sized benchmarks as it needs ground truth. A smaller TVD is better.

Baselines. We compare our method with three baselines: Paulihedral [19], QAIM [3], and 2QAN [16]. Paulihedral is mainly designed for compiling arbitrary Pauli-string simulation, where QAOA and 2-local Hamiltonian simulation are a special case. QAIM is the first domain-specific compiler that tackles QAOA compilation. 2QAN is the one using gate unifying to augment the SWAP insertion strategy. We also did comparisons with QAOA-OLSQ proposed by Tan *et al.* [30] and SATMAP [20] by Molavi *et al.* They are both SAT solver based approaches.

Benchmarks. For benchmarks of input problem graphs, we use python library NetworkX [11] to generate two types of graphs, random graph and regular graph. We set the density of random graph as 0.3 and 0.5. We also try to set the density of regular graph close to 0.3 or 0.5 by varying the degree of each vertex. The number of vertices of two types of benchmarks are from 64 to 1024. For each size of benchmark, we randomly generate 10 cases and use the average values to do the comparison. For the input problem graphs to the 2-local Hamiltonian. We use next nearest neighboring (NNN) 1D-Ising model, NNN-2D-XY model, and NNN-3D-Heisenberg model. Each interaction graph has 64 vertices. These are of the same type as those used in 2QAN [16].

7.2 Comparison on IBM Heavy-hex

In this section, we show comparison on IBM heavy-hex architecture with the number of qubits from 64 to 1024. Since it is impossible to get TVD for such large number of qubits, we only show the circuit depth and gate count comparison. In most of our cases, our depth and gate count are both significantly smaller than that of the baselines, which indicates better fidelity for these large circuits.

In the comparisons of both types of graphs, the improvement of our method increases when the qubit number grows. It indicates our compiler scales well to large problems.

The comparison for depth on IBM heavy-hex is shown in Fig. 20. When the qubit number is larger than 256, the compilation overhead of QAIM and 2QAN is more than 24 hours, Fig. 20 only contains data points for qubit numbers up to 256 qubits. And for the results of 2QAN, even for 256-qubit, it takes more than a day to compile, since 2QAN use a quadratic solver to search all possible initial mappings and picks the one with the minimum total distance of two qubits of each gate in the coupling graph. So we do not include the results of 2QAN in Fig. 20. Instead we place the comparison with 2QAN results using a separate Table 1. For all comparisons on depth for IBM heavy-hex, our method has minimum depth. For the benchmarks with 64 qubits, our method could achieve the depth reduction of up to 47% over QAIM_IC, 68% over Paulihedral, and 43% over 2QAN. For the benchmarks with 256 qubits, the improvement is more significant. Our method could achieve the depth reduction of up to 67% over QAIM_IC, and 72% over Paulihedral.

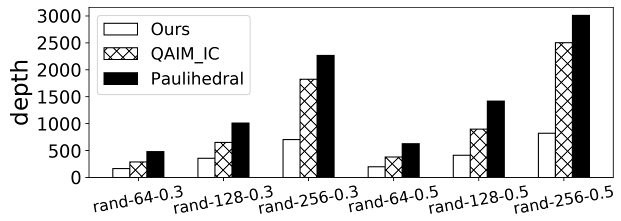
We also did the gate count comparison on two types of problem graphs with two different densities. The result is shown in Fig. 21. Similarly, we only show the comparison with qubit numbers up to 256. We also place the comparison with 2QAN in Table 1. The gate count comparison was shown on the log scale and our method outperforms all baselines. For the benchmarks with the number of 64 qubits, our method has the gate count reduction of up to 54% over QAIM_IC, 57% over Paulihedral, and 37% over 2QAN. The same as the circuit depth improvement, gate count reduction is more manifested in large benchmarks. For the benchmarks with the number of 256 qubits, our method has the gate count reduction of up to 63% over QAIM_IC, and 66% over Paulihedral.

For the QAOA circuits with 1024 qubits, we compare our method with Paulihedral with two types of graphs since this is the only one can handle such large circuits. The results shown in Table 2. Our method has 75% of depth reduction and 70% of gate count reduction over Paulihedral.

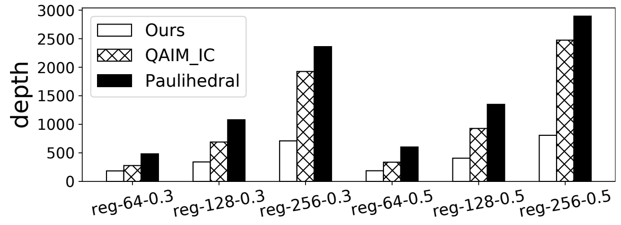
7.3 Comparison on Google Sycamore

Google sycamore architectures have better connectivity than IBM heavy-hex architectures which makes baselines perform better with their SWAP insertion strategy. It results in better circuit depth and gate count for baselines. However, our method still outperforms baselines in both gate count and depth comparison. The advantage is significant in large cases. Results are shown in Fig. 22 and Fig. 23.

For the same compilation overhead reason, some data of 2QAN for 128-qubit and 256-qubit is missing. Hence 2QAN is not presented

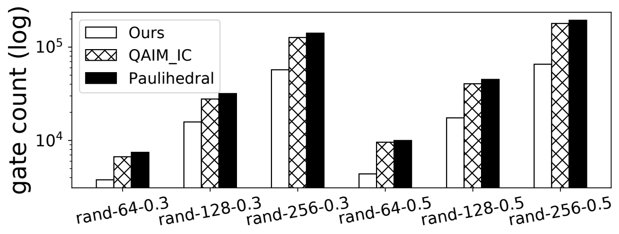


(a) Random graphs in IBM Heavy-hex

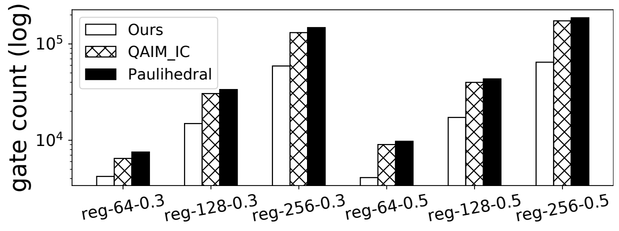


(b) Regular graphs in IBM Heavy-hex

Figure 20: Depth comparison for random graphs and regular graphs on heavy-hex architecture.



(a) Random graphs in IBM Heavy-hex



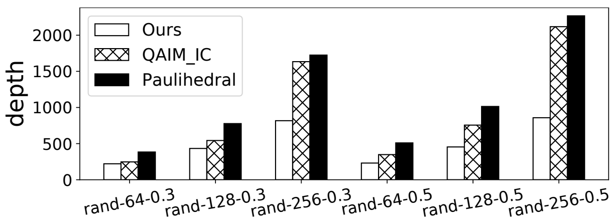
(b) Regular graphs in IBM Heavy-hex

Figure 21: Gate count comparison for random graphs and regular graphs on heavy-hex architecture.

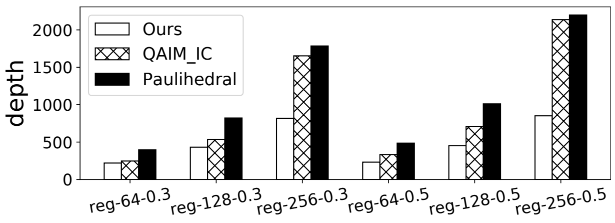
in Fig. 22 and Fig. 23. Instead 2QAN is presented in Table 1. In the depth comparison, we did experiments with two types of problem graphs again as the result is shown in Fig. 22. For the random graphs, our method could have the depth reduction of up to 60% over QAIM_IC, 62% over Paulihedral, and 12% over 2QAN.

The gate count comparison is shown in Fig. 23. The gate count comparisons are limited to benchmarks with 256 qubits as well. Our method is better than baselines for all types of graphs with different qubit counts and densities. Our method has gate count reduction of up to 44% over QAIM_IC, 48% over Paulihedral, and 5% over 2QAN.

We also did QAOA-1024 at Google Sycamore architecture as the results are shown in Table 2. We have 67% of depth reduction and 57% of gate count reduction over Paulihedral.

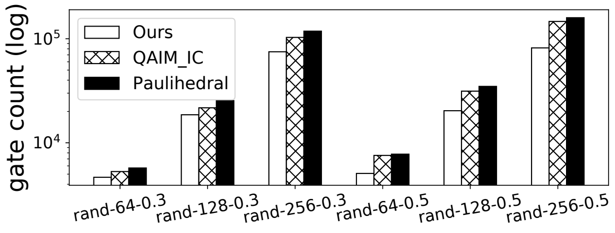


(a) Random graphs in Google Sycamore

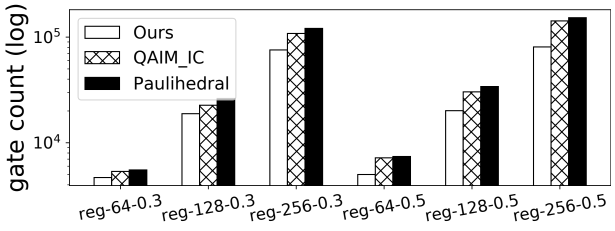


(b) Regular graphs in Google Sycamore

Figure 22: Depth comparisons for random graphs and regular graphs on Google Sycamore architecture.



(a) Random graphs in Google Sycamore



(b) Regular graphs in Google Sycamore

Figure 23: Gate Count comparisons for random graphs and regular graphs on Google Sycamore architecture.

7.4 Real Machine Experiments

In real machine experiments, we use TVD to evaluate our method. We compare our result with 2QAN since based on previous results,

Table 1: Comparison with 2QAN and QAIM

		Depth			Gate Count (CNOT)		
Arch.	Graphs	Ours	2QAN	QAIM	Ours	2QAN	QAIM
Heavy-hex	64-0.3	164	241	288	3789	4902	6695
	128-0.3	357	781	652	15785	19563	27799
	256-0.3	703	-	1825	57157	-	126653
	64-0.5	198	339	380	4385	6816	9580
	128-0.5	413	939	898	17458	29088	40487
	256-0.5	822	-	2503	65541	-	179085
Sycamore	64-0.3	220	235	247	4662	4785	5294
	128-0.3	433	-	543	18593	-	21691
	256-0.3	817	-	1634	74959	-	103010
	64-0.5	233	265	348	5041	5352	7564
	128-0.5	454	-	756	20266	-	31367
	256-0.5	858	-	2118	81567	-	146646

Table 2: Comparison for 1024 qubit graphs

		Depth		Gate Count(CNOT)	
Arch.	Graphs	Ours	Paulihedral	Ours	Paulihedral
Heavy-hex	1024-0.3	2910	10476	962514	2744864
	1024-0.5	3369	13719	1067113	3627376
	1024-320	2949	10728	967297	2796326
	1024-480	3299	13245	1051013	3509673
Sycamore	1024-0.3	3105	7830	1204734	2339009
	1024-0.5	3252	9975	1309333	3061684
	1024-320	3121	7932	1211841	2389790
	1024-480	3243	9615	1293761	2953371

2QAN has the best depth and gate count for small-sized circuits. For QAOA random 10-0.3, our method has TVD of 0.39 and 2QAN has TVD of 0.49. For QAOA random 20-0.3, our method has TVD of 0.62 and 2QAN has TVD of 0.66. This shows that our method is better in real machine experiments. QAOA random-20 has worse TVD because the larger circuit is more fragile to gate error and decoherence error.

We also run the full-fledged QAOA implementation to calculate max-cut using our compiled circuit. Running QAOA for multiple passes requires to use a classical optimizer to determine the rotation angles in the circuit (but the circuit structure, 2-qubit gates do not change). We use the default classical optimizer COBYLA in IBM Qiskit. We use QAOA circuits of for random graph 10-0.3 and random graph 20-0.3 and run them on IBM-Mumbai. The results are shown in Fig. 24 and 25. The x-axis is the number of optimization rounds. Each round runs the full circuit with given rotation angle numbers, for 8,000 shots. The y-axis is the negation of the expected max-cut value. The smaller the better. It shows that our compiled circuit converges faster in Fig. 24 and Fig. 25. The end-to-end results demonstrate the advantage of our method.

7.5 Comparison for 2-local Hamiltonian Problem Graphs

The results for 2-local Hamiltonian experiments are shown in Table 3. These were run on medium scale 64-qubit IBM heavy-hex architectures. Our compiler outperforms the 2QAN baseline in both circuit depth and gate count.

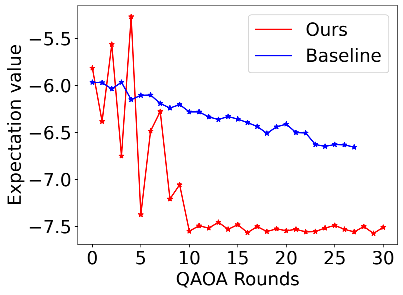


Figure 24: Full QAOA run at IBM Mumbai for 10-qubit random graph with density 30%.

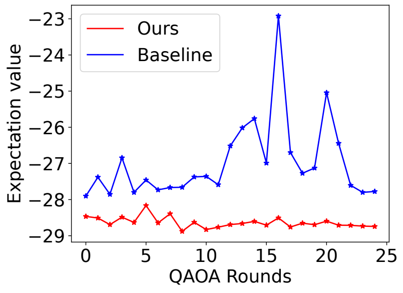


Figure 25: Full QAOA run at IBM Mumbai for 20-qubit random graph with density 30%.

Table 3: Comparison for 2-Local Hamtiltonian at IBM Heavy-hex

	Depth		Gate Count (CNOT)	
Benchmarks	Ours	2QAN	Ours	2QAN
1D-Ising	117	139	393	645
2D-XY	140	214	1050	1364
3D-Heisenberg	157	192	917	1000

7.6 Comparison with SAT Solver Based Solution

We did comparisons with QAOA-OLSQ by Tan *et al.* [30] and SATMAP by Molavi *et al.* [20] in 2D grid architecture. We choose the smallest coupling architecture that can fit the problem graph. The result shows in Table 4. Note that the column olsq represents the data of QAOA-OLSQ.

For most of the cases, our approach has better depth than QAOA-OLSQ and SATMAP, except one case 15-4, the 15-qubit density 40% graph, where our depth is 11 and QAOA-OLSQ's depth is 9. However, in this case, OLSQ takes more than 2 days to run. Our gate count is slightly worse than the QAOA-OLSQ method. Compared with SATMAP, our gate count is also worse than or similar to SATMAP. However, for the benchmarks with 15 vertices, our method is slightly better than SATMAP. All these benchmarks are compiled within 0.3 seconds by our compiler. But it takes hours for QAOA-OLSQ when qubit number is beyond 15 and graph density is beyond 30%. The compilation time of SATMAP is much shorter than QAOA-OLSQ, but still longer than ours.

Table 4: Comparison with SAT Solver-based Solution

	Depth			Gate Count			Compilation time(s)		
Graphs	Ours	olsq	satmap	Ours	olsq	satmap	Ours	olsq	satmap
10-2	4	5	5	12	10	12	0.001	0.24	5
10-3	6	7	10	22	19	22	0.003	33.2	6
10-4	11	12	12	32	25	30	0.04	283.4	76
12-2	3	4	3	16	12	12	0.006	0.32	5
12-3	8	7	11	26	22	25	0.09	4100.2	42
12-4	11	12	17	41	30	36	0.07	4.8 hrs	847
15-2	4	6	9	19	16	23	0.26	640.5	30
15-4	11	9	22	50	40	53	0.04	>2days	249

7.7 Compilation Time

We show the compilation overhead with respect to different QAOA graph sizes in Fig. 26. The X-axis is the number of qubits in the range of [64, 1024] and the Y-axis is the compilation time in seconds. Each QAOA problem graph is a random graph with a density of 0.3. The compilation overhead increases near-linear and the time for the circuit with 1024 qubits only takes about 30 seconds.

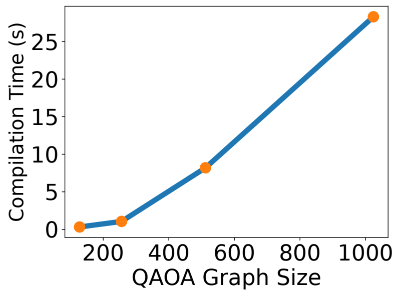


Figure 26: Compilation time

8 RELATED WORK

Circuit mapping plays an important role in the Quantum Supremacy realization due to the imperfect qubit and limited qubit connectivity. Previous studies in qubit mapping focus on the generic quantum circuits that the order of each gate is fixed to keep the circuit semantics [18, 21, 27, 28, 30, 31, 34–37].

Some recent studies of qubit mapping take permutable two-qubit gates into account. Tan *et al.* [30] uses constraint-based SAT-solver to achieve optimality by removing the gate dependency constraints from their generic approach. Alam *et al.* [3] model the QAOA qubit mapping as the binary bin-packing problem to heuristically insert SWAP gates and schedule unmapped CPHASE gates. Paulihedral [19] handles Pauli-string scheduling in Hamiltonian simulation and each CPHASE gate in the QAOA corresponds to a Pauli-string. 2QAN [15] proposes a quadratic solution for QAOA qubit mapping with respect to an objective function. It also takes the advantage of gate unifying to reduce circuit depth and gate count.

But none of those studies take architecture regularity into account. The previous method either suffers from large compilation overhead or tremendous gate count and depth increment. Our method is the first one that explores architectural regularity. In addition, our method is also integrated with a greedy component to avoid the disadvantage of rigidly following the all-to-all pattern.

In classical computing, the sorting networks and permutation networks are the closest to our study. Clique graphs are solved in permutation and sorting networks. But our problem is different.

For the permutation network, for instance, Abraham Waksman's work [32] can create arbitrary permutations for n input terminals to n output terminals using 2-node switch building blocks. However, if we directly apply the permutation network here, to enable all-to-all interaction, the depth is O(n*log(n)) since each permutation requires a depth of O(log(n)). Our work requires only O(n) depth for the linear architecture. Moreover, the permutation network can handle linear architecture, not multi-dimensional architecture.

For the sorting network works [5, 6, 14, 24]. They are relevant since each comparator operation involves two elements and a potential SWAP. However, it does not guarantee all-to-all interaction. For instance, when moving an element in insert-sort [14], it may stop at a location where it already satisfies the ordering constraint.

9 CONCLUSION

As modern quantum architectures start to exhibit regularity, it is time to rethink the design of quantum compilers. We proposed an optimal solver for finding all-to-all (ATA) interaction in regular quantum hardware and adapt the ATA pattern to compile for general programs with permutable 2-qubit operators. Experiments on Google Sycamore and IBM heavy-hex architectures show the advantage of our method.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive and helpful feedback. We thank Lingling Lao and Jiang Zhang for providing valuable comments during the discussion. This work is supported by grants from the Rutgers Research Council and NSF-FET-2129872. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Our open sourced optimal mapper. https://github.com/ata-pattern/ata-pattern.
[2] The IBM Quantum Heavy Hex Lattice, may 2022.
[3] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. Circuit compilation methodologies for quantum approximate optimization algorithm. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 215–228, 2020.
[4] Mahabubul Alam, Abdullah Ash Saki, and Swaroop Ghosh. An Efficient Circuit Compilation Flow for Quantum Approximate Optimization Algorithm. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20. IEEE Press, 2020.
[5] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.
[6] Michal Bidlo and Michal Dobeš. Evolutionary development of growing generic sorting networks by means of rewriting systems. *IEEE Transactions on Evolutionary Computation*, 24(2):232–244, 2020.
[7] Jerry Chow, Blake Johnson, Jay Gambetta, Rachel Zuckerman Sarango, and Saul. Ibm's roadmap for scaling quantum technology, Feb 2021.
[8] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Leo Zhou. The Quantum Approximate Optimization Algorithm and the Sherrington-Kirkpatrick Model at Infinite Size. *Quantum*, 6:759, jul 2022.
[9] Edward Farhi and Aram W Harrow. Quantum supremacy through the quantum approximate optimization algorithm. *arXiv preprint arXiv:1602.07674*, 2016.
[10] Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6/7):467–488, 1982.
[11] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[12] IBM. IBM Quantum System.
[13] Ian D Kivlichan, Jarrod McClean, Nathan Wiebe, Craig Gidney, Alá n Aspuru-Guzik, Garnet Kin-Lic Chan, and Ryan Babbush. Quantum Simulation of Electronic Structure with Linear Depth and Connectivity. *Physical Review Letters*, 120(11), mar 2018.
[14] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.
[15] Lingling Lao and Dan E Browne. 2QAN: A quantum compiler for 2-local qubit Hamiltonian simulation algorithms, 2021.
[16] Lingling Lao and Dan E Browne. 2QAN: A Quantum Compiler for 2-Local Qubit Hamiltonian Simulation Algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pages 351–365, New York, NY, USA, 2022. Association for Computing Machinery.
[17] Lingling Lao, Prakash Murali, Margaret Martonosi, and Dan Browne. Designing calibration and expressivity-efficient instruction sets for quantum computing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 846–859. IEEE, 2021.
[18] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1014. ACM, 2019.
[19] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, pages 554–569, New York, NY, USA, 2022. Association for Computing Machinery.
[20] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Qubit mapping and routing via maxsat, 2022.
[21] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1015–1029, New York, NY, USA, 2019. ACM.
[22] Bryan O'Gorman, William J Huggins, Eleanor G Rieffel, and K Birgitta Whaley. Generalized swap networks for near-term quantum computing, 2019.
[23] William D Oliver and Paul B Welander. Materials in superconducting quantum bits. *MRS Bulletin*, 38(10):816–825, 2013.
[24] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Differentiable sorting networks for scalable sorting and ranking supervision. 2021.
[25] QISKit: Open Source Quantum Information Science Kit. No Title. \url{https://https://qiskit.org/}.
[26] Rigetti. RigettiQPU, 2020.
[27] Marcos Yukio Siraichi, Vin\'\icius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. Qubit Allocation as a Combination of Subgraph Isomorphism and Token Swapping. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.
[28] Marcos Yukio Siraichi, Vin\'\icius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 113–125. ACM, 2018.
[29] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket>: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 6(1):14003, nov 2020.
[30] Bochen Tan and Jason Cong. Optimal Layout Synthesis for Quantum Computing. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.
[31] Swamit S Tannu and Moinuddin K Qureshi. Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 987–999, New York, NY, USA, 2019. ACM.
[32] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, jan 1968.
[33] Johannes Weidenfeller, Lucia C. Valor, Julien Gacon, Caroline Tornow, Luciano Bello, Stefan Woerner, and Daniel J. Egger. Scaling of the quantum approximate optimization algorithm on superconducting qubit based hardware, 2022.
[34] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 142. ACM, 2019.
[35] Chi Zhang, Ari B Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z Zhang. Time-Optimal Qubit Mapping. ASPLOS 2021, pages 360–374, New York, NY, USA, 2021. Association for Computing Machinery.
[36] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2018.
[37] Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1135–1138. IEEE, 2018.

APPENDIX

A MANUAL OPTIMIZATIONS FOR 2D GRID

A.1 Optimization I: Reducing 2xUnit solutions

Recall that we discussed in section 3.1 to divide the solution for clique input into 1xUnit solution and 2xUnit solution. Each 2xUnit solution are finished individually. However, the SWAP layers in one 2xUnit solution are beneficial to its nearby 2xUnit solution. For example, originally, we finish 2xUnit interactions A-B and C-D in parallel, and finish 2xUnit interactions B-C later in Fig. 27(a).

In two parallel 2xUnit solutions, all intra-unit SWAP gates in Unit C have relatively the same position as the intra-unit SWAP gates in Unit A as the example shown in Fig. 27(d) and (g). This makes the each qubit in Unit C relatively has the same position as Unit A. In addition, Unit B is adjacent to Unit C. So instead of having 2xUnit solution for Units B-C separately, we can merge it with A-B 2xUnit solution and C-D 2xUnit solution. As the example shown in Fig. 27(c) and (f), each B-C inter-unit layer has a preceding A-B and C-D inter-unit layer.

A.2 Optimization II: Merging Intra-unit operations

Since we apply 1xUnit solution to make each unit interact with each other, every unit will reach the bound once and only once. When we making inter-unit CPHASE gates, every two layers of inter-unit operations contains one top unit and one bottom unit that are not involving any inter-unit operations. Then we can perfectly add intra-unit CPHASE gates at idle units as the red gates in Fig. 27(c) and (f).

The intra-unit SWAP layers in each unit of 2xUnit solution is the same as the 1xUnit solution, so this guarantees that every intra-unit CPHASE gates in a unit are scheduled.

In the arXiv paper in February 2022 [33] which we consider as almost at the same time come up with solution to 2D grid, their solution consists of four repeated layers including two consecutive inter-unit CPHASE layers followed by a intra-unit CPHASE layer and one SWAP layer in the end. With the second optimization, our solution has three repeated layers. So our method could achieve approximately 25% of reduction in circuit depth.

The time complexity: Before we do any units exchange, it takes 3N cycles to complete bipartite all-to-all inter-unit interactions between each two adjacent units. It includes N cycles of even-odd units and odd-even units interaction respectively, and N-2 intra-unit swap cycles. If we consider one unit as a node, then we can apply linear pattern to make each unit adjacent to each other. Although N units exchange takes N-2 swap cycles, the transition from one units placement to another takes two consecutive swap cycles. So there are only N/2 units placement. In total, the time complexity of executing all-to-all interactions in NxN architecture is $1.5N^2+O(1)$.

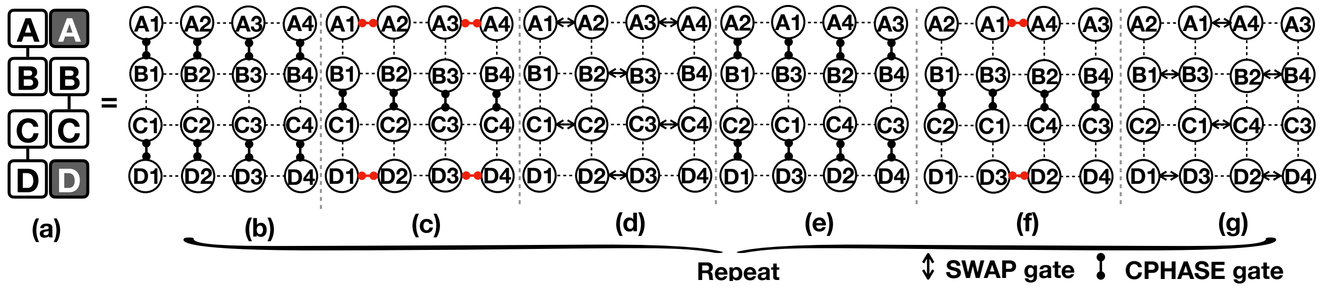


Figure 27: (a) The abstracted inter-unit interaction. Each square box is a unit standing for a row in 4x4 grid; (b) The detail of inter-unit interactions. The intra-unit interactions are merged into inter-unit interactions as the gates highlighted in red.

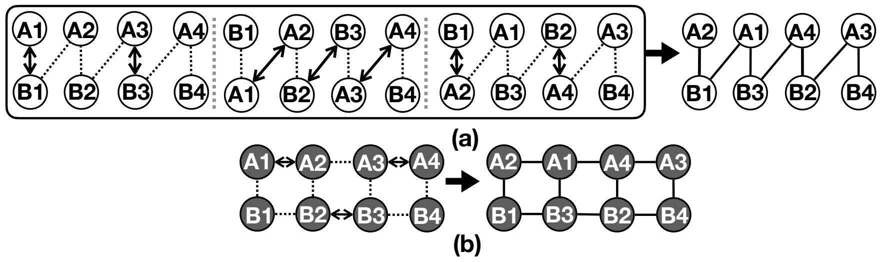


Figure 28: (a)Sycamore takes 3 swap cycles (one virtual SWAP layer to mimic the 2xN lattice inter-row interaction, (b) Corresponding 2xN lattice operation requires just 1 swap cycle

B OUR MANUAL SOLUTION FOR GOOGLE SYCAMORE

Recall that in 2xUnit solution for 2D grid, what we did is to let each row takes a different even-odd or odd-even SWAP layer simultaneously. This takes one SWAP layer. It turns out we can take 3 SWAP layers for 2xUnit of Sycamore to achieve the same effect. And since the top-unit and bottom-unit has direct links between elements at corresponding locations, we can let computation layer take place immeidately after such 3 SWAP layers, such that eventually the bipartite all-to-all interaction is achieved between the two neighboring units.

We illustrate the 3 SWAP layers of Sycamore that are analogue to the 1 SWAP layer of 2D grid in Fig. 28.

Inter-unit. We further propose a manual optimization, we can interleave the computation 2-qubit gates with these SWAP layers to save 25% of depth.

If we look at three consecutive SWAP layers, we can find out that the first layer and the third layer both contain two pair of idle qubits. For every two consecutive three SWAP-layers, the SWAP gates in the last SWAP layer of the first big step are in the alternating position of the SWAP gates in the first SWAP layer of the second big step. So we can let all of those SWAP gates scheduled in one layer and schedule CHPASE on the vacant layer. This is equivalent to a pure SWAP layer followed by a pure CPHASE layer. The CPHASE gates in the pure CPHASE layer apply on the new qubit permuration are the same as in 2D grid pattern. For example, we can rearrange the gates in Fig. 29 (a)(3) and (b)(1) to make Fig. 29 (a)(3) only contains SWAP gates and Fig. 29 (b)(1) only contains CPHASE gates. The CPHASE gates in Fig. 29 (a)(1) and Fig. 29 (d)(3) could also be considered as one pure SWAP layer plus one pure CPHASE layer. Though this example, we can see why the separated CPHASE layer could be merged into SWAP layers.

Intra-unit. So far we have discussed the inter-unit scheduling, but the intra-unit scheduling is still waiting to be explored.

One simple approach to schedule intra-unit gates is that we can form a linear connection for each two units and schedule those intra-unit gates by following the linear pattern.

However, if we look at the second cycle of each sub-figure in Fig. 29, we will see each diagonal edge connects a pair of qubits from the same unit and each pair is unique. So before or after each second cycle at the Fig. 29 (a) to (b), we can add one more cycle to schedule all intra-unit gates for unit A and unit B. The intra-unit scheduling for each two units could be done in parallel as well. So the depth cost for intra-unit scheduling is O(N), N is the size of unit.

Time complexity: Executing all-to-all interactions in NxN sycamore architectures has the same number of units placements as the executions in 2D grid. Both have N/2 different unit placements. The difference is that the bipartite all-to-all inter-unit interactions in here takes 4N cycles. It includes 3N swap cycles between odd-even units. One odd-even inter-unit interactions are merged into those swap cycles. The un-merged even-odd inter-unit interactions take N cycles. In total, sycamore solution takes $2N^2 + O(1)$ cycles.

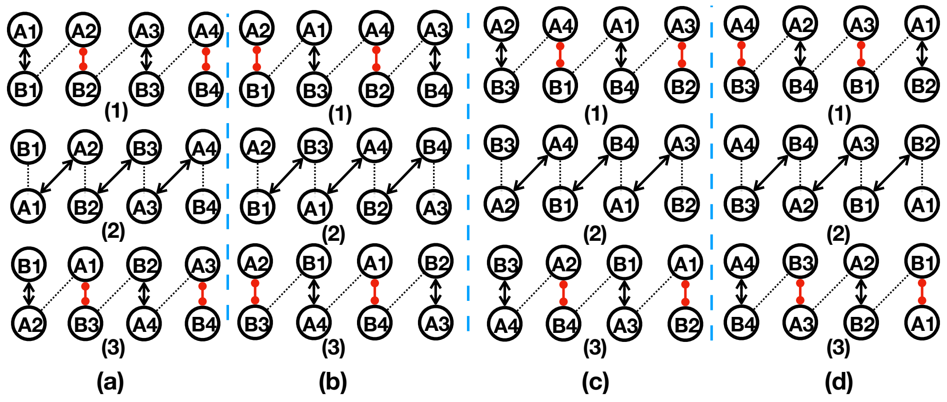


Figure 29: Basic Sycamore Inter-Unit Scheduling

C PROOF FOR FOR IBM HEAVY-HEX

By applying the linear pattern at the longest path two times in heavy-hex architecture, we are able to schedule all CPHASE gates corresponding to the clique input graph.

In the first pass of 1xUnit grid pattern, we can schedule all path-2-path operations. When a path qubit adjacent to a off path qubit, we would pause the pattern and schedule those path-2-off-path gate.

Then we swap the off-path qubit with one of its on-path neighbour and apply the second pass of 1xUnit solution. Since all off-path qubits are now on path, the second pass of pattern can schedule all off-path-2-off-path CPHASE gates.

The only question is that if the remaining path-2-off-path operations could be covered by the second pass or not. In the first pass of pattern, every qubit moves K steps, assuming the longest path has the length of K + 1. The qubit closer to the end point would cover more region and the qubit in the middle only traverses half of the longest path, so every qubit visited at least half of positions in the longest path. (A qubit at position i would firstly traverse to one end of path and turn the moving direction. In total it traverses K steps and stops at K - i position.) So any two qubits that one is from the top region and another one is from the bottom half of region can have the overlapping activity region and they can interact with each other after two passes of pattern.

Time complexity: The linear pattern takes O(N) cycles for the line size of N. In the heavy-hex architecture, executing all-to-all operations still have an overall time complexity of O(N), along with some additional constant cost for implementing path-2-off-path gates.