A Structured Method for Compiling and Optimizing QAOA Circuits in Quantum Computing

Yuwei Jin\(^1\), Jason Luo\(^1\), Lucent Fong\(^1\), Yanhao Chen\(^1\), Ari B. Hayes\(^1\), Chi Zhang\(^2\), Fei Hua\(^1\), Eddy Z. Zhang\(^1\)

\(^1\)Rutgers University
\(^2\)University of Pittsburgh

ABSTRACT

Quantum Approximation Optimization Algorithm (QAOA) is a highly advocated variational algorithm for solving the combinatorial optimization problem. One critical feature in the quantum circuit of QAOA algorithm is that it consists of two-qubit operators that commute. The flexibility in re-ordering the two-qubit gates allows compiler optimizations to generate circuits with better depths, gate count, and fidelity. However, it also imposes significant challenges due to additional freedom exposed in the compilation. Prior studies lack the following: (1) Performance guarantee, (2) Scalability, and (3) Awareness of regularity in scalable hardware.

We propose a structured method that ensures linear depth for any compiled QAOA circuit on multi-dimensional quantum architectures. We also demonstrate how our method runs on Google Sycamore and IBM Non-linear architectures in a scalable manner and in linear time. Overall, we can compile a circuit with up to 1024 qubits in 10 seconds with a 3.8X speedup in depth, 17% reduction in gate count, and 18X improvement for circuit ESP.

1. INTRODUCTION

Quantum computing has garnered considerable attention thanks to recent advances in quantum mechanics. The coherence time of a single qubit has increased exponentially in the past two decades\(^{23}\), analogue to Moore’s law for semiconductor computers. Industry is now able to build quantum computers up to between 53 and 127 qubits. Moreover, IBM is projected to release a quantum computer with over 1,000 qubits in the near future\(^7\).

Variational quantum algorithm (VQA)\(^{24}\) is an important class of quantum applications that holds promise for achieving near-term quantum supremacy. Today’s quantum devices are prone to errors. Quantum error correction (QEC) is necessary for applications that have relatively large depths such as Shor’s algorithm\(^{27}\). But QEC is prohibitive due to the scale of today’s quantum computers. For instance, to run a Shor’s algorithm with reasonable size (factoring a 1,000 bit number), it will need millions of physical qubits, which is not possible at this stage. VQA algorithms typically have modest depths and may not need QEC at this stage. Therefore it can exploit near-term quantum hardware.

Quantum approximate optimization algorithm (QAOA)\(^{9,8,10}\) is a highly advocated variational quantum algorithm. It is a general-purpose algorithm that has broad applications in combinatorial optimization, for instance, quadratic 0-1 programming and max-cut problem. We focus on the optimization of QAOA circuit in this paper.

To fully explore the power of QAOA algorithms, the QAOA circuits must be compiled to be resilient to noise. To compile and optimize a QAOA circuit, there are two approaches. The first one is the generic compiler\(^{37,18,21,28,29,39,38,32}\) including Qiskit\(^{25}\) and\(\text{T}\)\(\text{ket}\)\(^{30}\). The second is the application-specific compiler. The application-specific compiler takes into account the flexibility in permuting operators in QAOA circuits, while the generic compilers do not. An example is shown in Fig. 2 in which after permuting the gates, the logical circuit depth reduces by 25%.

\[\text{Figure 1: (a) Google Sycamore (b) IBM Heavy-hex}\]

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

The application-specific compilers have demonstrated better results than the generic compilers in terms of both performance and compilation time\(^{17,2}\). Tan et al.\(^{32}\) provide a constrained SAT solver to achieve the optimal compilation result for QAOA. Alam et al.\(^2\) use the heuristic of connectivity strength to improve the depth and fidelity of the compiled circuit. Paulihedral\(^{19}\) is a compiler framework that defines intermediate representation and optimizes the compilation of Pauli-strings in Hamiltionian simulation. 2QAN\(^{16}\) proposes a permutation-aware heuristic that has quadratic-time complexity and uses unitary unifying to enhance the performance.

However, existing application-specific approaches lack rigorous analytical investigation on the QAOA compilation problem. As far as we are concerned, none of the existing comp-
Optimal compilation is not a viable solution in the long run. However, it may not be able to handle large scale circuits. Our contributions are summarized as follows: exploitation in scalable quantum hardware. Our compiler provides scalability, performance guarantee, and regularity-exploitation in scalable quantum hardware. Our contributions are summarized as follows:

1. **Lack of scalability**: Optimal solvers such as that by Zhang et al. [37] can be adapted to compile QAOA circuits. However, it may not be able to handle large scale circuits. Optimal compilation is not a viable solution in the long run.

2. **No Provable Performance Guarantee** Heuristic compilers including QAIM [3, 2], Paulihedral [19], and 2QAN [16] can compile reasonably fast for circuits with hundreds of qubits. However, it is not clear how far the compiled circuit is from the optimal result, given any of the existing approaches. Based on our experiment results, when going beyond 256 qubits, QAIM runs for more than two hours. Paulihedral can run fast but provide less than desired compilation results for large-scale graphs. Using greedy heuristics can help solve the compilation problem in the near term, but we need to have an in-depth understanding of a given core heuristic as well as its application scenario.

3. **Obliviousness to Regularity in the Scalable Quantum Hardware** Neither optimal compilers [37] nor heuristic compilers [19, 16, 3, 2] take into account of the regular structure in today's quantum hardware that scale. We show Google Sycamore structure in Fig. 2(a) where an obvious pattern of diagonal links between qubits in a 2D lattice is manifested. Similarly recent IBM architectures show regularity in their structure: they use the heavy-hex architecture [1]. IBM architecture is shown in Fig. 2(b).

To this end, we propose a structured method for the compilation and optimization of QAOA circuits. Our method provides scalability, performance guarantee, and regularity-exploitation in scalable quantum hardware. Our contributions are summarized as follows:

- **Linear depth for multi-dimensional architecture**: Our compiled circuit is guaranteed to provide linear depth for Google Sycamore, IBM heavy-hex, and grid. Specifically, it guarantees $\frac{5}{4}$ depth for Google Sycamore architecture, $6n$ for IBM heavy-hex architecture, and $\frac{3}{2}$ depth for grid (if we ignore the sub-linear terms), in worst case scenario, where $n$ is the number of qubits.

- **Hardware regularity exploitation**: We exploit the hardware regularity through the multi-dimensional all-to-all permutation and swaps (Maaps) method we propose in this paper. The Maaps method provides a new way of thinking on achieving the all-to-all interaction in multi-dimensional architectures. We used Maaps to show how to come up with solutions for Google Sycamore, grid, and an additional imaginary hexagon architecture.

- **Enabling scalability**: Our compilation does not require an optimal solver to achieve a desired depth. It has linear time complexity. Hence it can compile large circuits very fast. It can run within 9 seconds for a circuit with 1,024 qubits. The compile time scales linearly with the number of qubits in quantum hardware, and yet the compiler provides linear depth guarantee.

- **Application beyond QAOA**: Our compiler framework can be applied to 2-local Hamiltonian simulation in addition to QAOA. The compilation and optimization of QAOA is essentially the same as that of 2-local Hamiltonian simulation. In both cases, a problem-graph determines the structure of the logical circuit, and the 2-qubit operators can permute. 2-local Hamiltonian is a common type of Hamiltonian simulation that originate in many physical systems, such as the traversing Ising model, XY model, and the Heiserburg model.

Overall we provide a simple yet effective compiler framework for compiling QAOA and 2-local Hamiltonian simulation circuits. Compared with QAIM [3], our experiments show that we can achieve up to 3.8X speedup for depth, 17% reduction in gate count, and 18X in fidelity improvement. In the meantime, our compiler has low overhead. We have evaluated circuits with up to 1,024 qubits in this paper and the compiler takes no more than 5 minutes, often running in less than 10 seconds, while prior study may take hours to compile a circuit with $> 256$ qubits.

We recognize that an independent study by Weidenfeller et al. [35] comes up with linear solutions for grid and IBM heavy-hex architectures at nearly the same time as we do (February 2022 vs. ours first 2xN grid solution in December 2021 and NxN grid solution in April 2022). Their solution for grid is similar to ours, except that our solution reduces 25% depth compared with theirs when SWAP is not implemented as three CNOTs, elaborated in Section 3.4.1.

## 2. BACKGROUND

### 2.1 Introduction to QAOA

The QAOA circuit is a multilevel parameterized quantum-classical hybrid circuit. It has $\rho$ optimization levels with two
occurs between two connected physical qubits. Examples they reside on two connected physical qubits. A SW AP gate in a 2-qubit gate need to be remapped via SW AP gates until to two connected physical qubits. Due to the instability of single-qubit and two-qubit gates. A two-qubit gate applies to the input problem graph in Fig. 2. (c) One possible compiled circuit of (b); (d) Another valid circuit corresponding to the same problem graph; (e) Optimal compiled circuit of (d); (f) Control-Z gate abstracted. For the logical circuit (b), since its depth is already 4, its compiled circuit must have length ≥ 4. For the logical circuit (d), its optimal compiled circuit (e) does not increase its original depth 3. Therefore, the logical circuit (d) is definitely better than the logical circuit (b).

2.2 Compilation of QAOA

2.2.1 Problem-graph induced circuit

Unlike other quantum applications, the structure of a QAOA circuit is not known until an input-dependent problem graph is specified. For instance the QAOA-Maxcut circuit is dependent on the input graph. In a QAOA-Maxcut circuit, a qubit corresponds to one vertex in the input graph. Each CPHASE gate corresponds to an edge in the problem graph, as shown in Fig. 2 the edges in (a) correspond to the two-qubit gates in (b) and (c). What distinguishes QAOA circuit from other circuit is that the two-qubit operators commute. That is, they can flexibly permute, and the circuit outcome is the same. For instance, in Fig. 2 both (b) and (c) are valid instances of the QAOA circuit for the input problem graph in (a). The commutativity feature exists in 2-local Hamiltonian simulation circuits too. Such flexibility can be exploited to optimize compilation, for instance, reduce the gate count and depth of the compiled circuit, as described in Section 2.2.2.

2.2.2 Compilation objective

Near-term quantum computer’s native gate set consists of single-qubit and two-qubit gates. A two-qubit gate applies to two connected physical qubits. Due to the instability of qubits and resonance between qubits and coupling edges, the connectivity between qubits on a quantum chip is limited. To tackle this problem, the locations of two logical qubits in a 2-qubit gate need to be remapped via SWAP gates until they reside on two connected physical qubits. A SWAP gate occurs between two connected physical qubits. Examples are shown in Fig. 3 where SWAP gates are inserted to make two quantum circuits executable. The goal of compilation is to yield better gate count, depth, and fidelity of the compiled circuit.

The flexibility in gate ordering in QAOA circuit introduces new opportunities for optimization at compile-time. Two resulting circuits from the same problem graph can lead to different optimal compiled circuits. We show an example in Fig. 3 where one induced circuit results in no depth change (3) even when a SWAP gate is inserted, while the other circuit itself already incurs a larger depth (4) than the first one, even without adding any SWAPs.

2.3 2-local Hamiltonian Simulation

Hamiltonian simulation is another important application of today’s universal quantum computers. Hamiltonian simulation if performed on classically computers is an intractable problem, as its complexity grows exponentially with the problem size. Richard Feynman is the first to propose to use quantum computers to efficiently solve the problem. 2-local Hamiltonian is a type of Hamiltonian that arise in many physical systems. It has the following form:

\[ H = \sum_{uv \in E} H_{uv} + \sum_{k \in V} H_k \]  

(1)

The interaction between the qubits in the 2-local Hamiltonian simulation can be represented by a graph \( G = \{V, E\} \) where \( H_{uv} \) is a two-qubit Hamiltonian and \( u - v \) represents an edge in \( E \), and \( H_k \) is a single-qubit Hamiltonian where \( k \) represents a node in \( V \). With a further Trotterization step for Hamiltonian simulation, all 2-qubit terms can commute flexibly. And Trotterization is a common approach for building a quantum circuit on universal quantum computer to asymptotically approximate the time evolution of a Hamiltonian simulation.

3. OUR METHOD

3.1 Solution for the Line Architecture

Consider \( n \) nodes on a rope, labeled from \( A_1 \) to \( A_N \). There exists a way to move each node around the rope, by swapping them with their neighbors, to form \( \lceil N/2 \rceil \) different permutations such that each node is adjacent to every other node
The idea is shown in Fig. 4. Let the physical positions on the rope be labeled $P_i$, $i \in [1, N]$. First, nodes located in position $P_i$, where $i$ is odd, are swapped with their neighbor $P_{i+1}$. Then, nodes that are now located in $P_j$ where $j$ is even, are swapped with their neighbor $P_{j+1}$. A circuit corresponding to the idea is shown in Fig. 5. We purposely add an additional pair of SWAPs at the end which will result in the qubits being reversed in its final state. This last pair of swaps is not necessary to have all neighbors interact, only for reversing qubits. Note that for odd-number qubits, in the last pair of SWAPs, we omit one for all-to-all connectivity. We do not discuss it due to space limit.

The line architecture solution has been proposed in previous studies [15, 22, 35]. Overall it takes $N-1$ SWAP layers (assuming each SWAP layer consists of parallel SWAPs) to achieve all-to-all connectivity and reversed qubit arrangement on the line. Certain representation lets one computation gate layer be followed by one SWAP layer [15, 35], and another representation lets two computation gate layers be followed by two SWAP layers [22] but will cut one SWAP layer in the last pair. Ours is the latter.

### 3.2 Solution for the 2xN Grid

The line architecture has been solved by previous studies as early as 2017 and 2019 [15, 22]. However, the solution for a multi-dimensional architecture has never been proposed until recently. We are the first to find a solution for the 2xN grid in 2021 [5] (in its Appendix). It is an important building block for the NxM grid solution. We believe Weidenfeller et al. [35] has a similar building block to this. It is also a building block for Google Sycamore and the hexagon architecture where we are the first to solve.

The idea is simple, as shown in Fig. 6. Note that each row performs a swap pattern that mirrors that for the line architecture, but the swaps alternate between adjacent units. The reason for the alternating swap pattern is as follows: imagine two parallel strings of nodes such that each node has a partner node on the other string (vertically). If the nodes on both strings move in the same direction, then the nodes will always have the same partner. This movement pattern enable both inter-row and intra-row all-to-all interaction. Interestingly, all intra-row interactions happen simultaneously (one parallel layer) at each step, and all inter-row interactions happen simultaneously, which means they can be separated if necessary.

### 3.3 Handling Multi-dimensional Architecture

To handle multi-dimensional architecture, our idea is to break down an $N$-dimensional architecture into multiple $(N-1)$-dimensional units. Assuming there are $M$ units, by looking at the connectivity between these units, it is at least a line. If it has more connectivity than a line, that’s okay too.

If a few condition can be met, we can handle the N-dimensional architecture by just considering every unit as if it is a “qubit” on a line, every unit-exchange as if it is a “SWAP” in a line, and every inter-unit interaction as if it is a “two-qubit gate” between two qubits. Note that the sub-problem of the N-1 dimensional architecture can be handled in the same way. The Maaps algorithm is shown in Algorithm 1.

**Algorithm 1 Maaps on multi-dimensional architecture**

1: **procedure** MAAPS($P$);
2: Divide $P$ into $M$ units: $P_0, P_1, \ldots, P_{M-1}$
3: 
4: if $| P_i | \neq 1$ then
5:   MAAPS($P_i$); $i=0, 1, \ldots, M$
6:   for $s=0; s=s+1; s<M/2$ do
7:     IE($P_i, P_{i+s}$); $i=0, 2, \ldots, M$
8:     IE($P_{j}, P_{j+1}$); $j=1, 3, \ldots, M$
9:   end for
10: else
11:  $P$ is a line now. Handle it as a line architecture.
12: end if
13: **end procedure**

We formalize the four conditions that must be satisfied to
handle any multi-dimensional at each recursive step:

- Define “units” such that the connectivity of the units forms a line.
- Define intra-unit (IA) operations. That is, all-to-all interaction within a unit, in linear time.
- Define inter-unit (IE) operations: Having all-to-all interaction between two neighboring units, in linear time.
- Define unit exchange (UE) operations: Exchanging the locations of two neighboring units in linear time.

With the four above conditions, Maaps can ensure linear time all-to-all interaction. It can be proved rigorously. We sketch the proof. Assuming the \( N - 1 \) dimensional sub-space size is \( X \), and the number of units is \( M \). The total size is \( M \times X \) for the \( N \)-dimensional space. Assuming the \( (N - 1) \)-dimensional sub-problem has linear solution \( O(X) \). Since the inter-unit interaction happens \( 2M + O(1) \) times and is linear, and the intra-unit interaction is linear \( O(X) \), the complexity is then \( O(M \times X) \) linear to the size of the entire \( N \)-dimensional space.

![Computation Setup](image)

Figure 7: Row-wise movement and interaction for 2D lattice. Each node represents a row. Assuming four rows \( A, B, C, \) and \( D \). \( A \) and \( D \) in computation stage 1 are idle and hence they can perform IA operations. Similarly for \( B \) and \( D \) in computation stage 2.

### 3.4 Google Sycamore, 2D Lattice, and Hexagon

We demonstrate how Maaps applies to 3 important architectures: 2D lattice, Sycamore, and hexagon. We start with the 2D lattice example as it illustrates the application of Maaps most clearly.

#### 3.4.1 2D Lattice

**Defining A Unit:** For 2D lattice, we define a unit as a row.

**Unit Exchange:** Unit exchange is simply applying vertical pairwise swaps between two rows. One parallel level of such swaps exchanges the locations of the entire two rows.

**Intra-row (IA) Operation:** Intra-row all-to-all interaction can follow that for the linear architecture, which is trivial.

**Inter-unit Interaction:** The goal of the IE interaction in 2D lattice is to ensure that every node in unit (row) \( A \) interacts with every node in its neighboring unit (row) \( B \). We can apply our solution for 2xN grid in Section 3.2. Recall that intra-row and inter-row operations can be decoupled into separate layers and either runs within minimal number of layers guaranteed by our 2xN solution, we only keep the inter-row operations. We present a concrete example in Fig. [9](#9), which is derived from our previous example [5]. We outline the algorithm for IE interactions on a 2D lattice in Algorithm 2.

Just naively applying the general Maaps method may not provide the best solution. We notice two optimization opportunities that happen to work for 2D grid case. The work by Weidenfeller et al. [35] also includes one of them, but not both. Hence, ours is slightly better, which reduces 25% depth. We describe them below.

#### Further Optimization I – Sharing SWAP Layer

In our representation of the line solution, after each pair of SWAP layers, we have two computation layers, one on odd-even units, and the other on even-odd units. An example is shown in Fig. [10](#10), where in steps (e) and (f) \{A-B, C-D\} interaction is odd-even unit IE, and B-C is even-odd unit IE.

In our general Maaps method, even-odd unit IE needs to be completed before odd-even IE operations, or the other way around, but we can mix them for the grid case.

We notice that the SWAP layers performed for the odd-even row pair can prepare for the layout change needed for the even-odd pair. In a 2xN operation, the top-row performs odd SWAPs (where the left index of a SWAP is an odd number, assuming the index starts from 1), and the bottom-row performs even SWAPs, or the other way around, as shown in Fig. [8](#8) (a) for rows A and B in grey boxes, and for rows C and D. It happens that it also creates a SWAP pattern that enables all-to-all connectivity for the even-odd row IE, as highlighted in red dashed box for rows B and C in Fig. [8](#8) (a). Hence we can use just one SWAP layer for both odd-even

---

**Algorithm 2 Inter-unit interaction for two rows in 2D lattice**

```plaintext
1: procedure IE_Lattice(A, B):
2:   r = 0
3:   s = sizeof(A)
4:   for i=0; i<s; i+=2 do
5:     G_interact(A_j, B_j); j=0, 1, ..., s
6:     SWAP(A_j, A_{j+1}); j=r, r+s, ..., s
7:     SWAP(B_j, B_{j+1}); j=1-r, 3-r, ..., s
8:   r = (r+1) mod 2
9: end for
10: end procedure
```

![Figure 8](image)

Figure 8: Grid Optimization I: (a) Odd-even row SWAP happens to help even-odd row interaction, (b) One additional SWAP layer added in addition to odd-even row SWAP.
We define a unit still as a "row" in the re-unit exchange then is trivial and only reduction layers (one IE and one IE/IA). We can reduce the depth while ours has one SW AP layer followed by two computation layers (two IE and one IA), the direction. Each node stays at the first or last location for ing towards a direction until it hits the border, then it changes the direction. Each node stays at the first or last location for exactly once. An example showing that it will cover all units in top row or bottom row is shown in Fig. 7.

Further Optimization II: Interleaving IE and IA.

The second optimization we apply is to overlap IA and IE operations. So far we haven’t talked about when to handle intra-unit (IA) case. It is possible that one can run IA for all rows at once since within each row it forms a linear line. In fact, we can even save this step by overlapping IA operations with IE without any additional cycles.

During the step of pair-wise IE: even-odd row IE, and odd-even row IE, there is always an unit or two that is idle. If the total number of rows is odd, then for the odd-even pair of IE, the last row is idle, and for the even-odd pair of IE, the first row is idle. For two consecutive even-odd and odd-even IE operations, there are always two rows that are idle. We can perform intra-unit (IA) operation for these idle rows at the same time as the IE operations. It happens that an IE cycle depth is the same as an IA cycle depth in our prior work [5] using the 2xN solution.

Note that this will cover all rows, since according to the unit movement pattern in the line proved by previous work [35] and our non-refereed publication [5], each node is moving towards a direction until it hits the border, then it changes the direction. Each node stays at the first or last location for exactly once. An example showing that it will cover all units in top row or bottom row is shown in Fig. 7.

This optimization is not present in the concurrent work by Weidenfeller et al. [35]. In total, they have one SWAP layer followed by three computation layers (two IE and one IA), while ours has one SWAP layer followed by two computation layers (one IE and one IE/IA). We can reduce the depth by 25%. Here we do not decompose SWAP into 3 CNOT or cancel CNOTs.

Putting it all together:

With the two optimizations, we can reduce the depth significantly, as shown in a full example in Fig. 10.

To calculate the depth, we assume it is a square lattice, both the row and column numbers are N, without loss of generality. It takes 3/2 cycles for each IE_{lattice}(A, B)’s iteration on average since the two SWAP steps (line 6 and 7). This will take a total of 3N-1 cycles (no need to swap for last iteration) since it takes N iterations for each node in a row to be neighbors with every other node in another row. These 3N-1 cycles repeat N/2 times since there are N/2 unit configurations that must be reached for all units to be neighbors with each other once.

There is a total of N-2 unit exchanges since there are N/2 configurations that need to be reached, but it takes 2 swap cycles to reach each configurations. No swaps are needed after the final configuration, hence the N-2 total swaps. Thus, the total run time is (3N-1)(N/2) + (N-2) cycles.

For method by Weidenfeller et al. [35], it does not overlap IE with IA. Rather it runs IE and IA separately after each set of horizontal SWAPs (two IEs and one IA, see Fig. 17 in their paper). Hence their depth is 2N^2 + O(N) in total. We reduce 25% depth compared with theirs.

3.4.2 Google Sycamore

Algorithm 3 A Mixed IE and IA Step for MxN Sycamore

For illustration purpose, we redraw the Google Sycamore case to look like a 2D lattice as illustrated in Fig. 12. Note that this does not reduce the complexity of the problem.

Defining A Unit: We define a unit still as a "row" in the redrawn Sycamore architecture in Fig. 12. For instance, A1, A2, A3, and A4 form a unit.

Unit Exchange: Unit exchange then is trivial and only requires the vertical links.

Inter-unit Interaction: Recall that in the 2xN architecture, we perform odd SWAPs and even SWAPs on two rows repeated. This ensures each node on the top row interacts with each node in the bottom row through vertical links, as shown in Fig. 6.

We now implement Sycamore’s IE by mimicking the this behavior of 2xN. We use the diagonal/vertical links between rows to achieve this goal but with more cycles. All together we use 3 SWAP layers to achieve this goal, which we refer to as a virtual horizontal SWAP layer. It takes 2 vertical SWAPs, and one diagonal SWAP. Our idea is shown as an example in Fig. 13 it is for mimicking the top-row odd
Putting everything together between two rows are split into two steps, shown as the red links on (f) and (h), the computation on (g) as well as the red links on (f) and (h) cover even-odd and odd-even row IEs respectively. This would still work for the even-number row case for Sycamore, but not for the odd-number row case. The last row in the odd-number row case does not have direct links between qubits within it, not like the last row in the grid. Moreover, since all other rows are busy now, it cannot use the diagonal links. Therefore we cannot satisfy the layout change needed for the even-odd row pair (the last and second-to-last rows).

In this case, we can fall back to the generic Maaps method, by not sharing the SWAP layer for two consecutive odd-even and even-odd row IE. Asymptotically, we increase the depth by a factor of 6/5. But we do not have to do so. Our idea is to simply neglect the interaction between the last physical row and the second to last physical row. Then we perform the missing inter-row IEs in a separate pass. The idea of omitting is shown as an example in Fig. 15.

It happens that the omitted IEs are between consecutive even-numbered rows, and between the second-to-last row and the last row. We sketch the proof here. Since even-numbered units move down the line until it hits the border and stops there for one SWAP cycle before it changes direction, each even numbered row will hit the bottom border once. Also since originally the distance between two even-numbered units is 2 and they move at the same speed before hitting the border, the last two physical rows will contain two even-numbered rows except one case. During the very first step, the IE between the second-to-last row (the largest even-numbered row) and the last row is also missing.

With this property, we can easily break down the missing IEs into two parallel layers. The first layer consists of pairs of rows 2-4, 6-8, 10-12, and so on. The second layer consists of the rest of the pairs, 4-6, 8-10, and so on. Since the distance between each pair is 2, we can apply concurrent SWAPs to move one next to the other. Then we revert the SWAPs to restore the natural number ordering, then we do the same for another parallel layer. In total it takes 6 layers including SWAP, computation and reverting, but it applies the factor in front of $N$ which is the square root of the total qubit number $N^2$, as each IE takes $N - 1$ depth.

**Runtime:** The IA loop is 5 cycles for an NxN Sycamore, the set of cycles will take $5N-1$ cycles. Thus, the total run time on any Google Sycamore architecture is at most $(5N-1)(N/2) + (N-2)$ cycles. Compared with our grid solution, it is $5/3$ factor slower. Compared with the line architecture, it is $5/4$ factor slower. Note that it is not possible to draw a line to connect all nodes in Sycamore. Hence we cannot directly

---

**Figure 10:** A detailed step for interleaving intra- and inter-unit operations, corresponding to computation stage 1 in Fig. 7.

**Figure 11:** Odd-number row case for Sycamore

**Figure 12:** Sycamore illustrated in a different way.

SWAP and bottom-row even SWAP case in grid. A similar set of 3 SWAPs can be achieved for mimicking the top-row even SWAP and bottom-row odd SWAP case in grid.

Now since all vertical links are maintained compared with the grid case, we will be able to perform inter-“row” computation after the “virtual horizontal swap” layers.

**Intra-unit Interaction:**

We notice that IA can be scheduled in between the actual SWAP layers of the “virtual horizontal SWAP” layer. Note that after the first SWAP layer, it connects the once separated qubits within a unit through the diagonal links. For instance, in Fig. 13 A1 would now be connected with A2 and A3 with A4. Thus we can insert an IA operation amid the steps of an IE operation, as shown in Fig. 14 (e).

**Optimization for Overlapping IE and Virtual SWAP:** Since during two SWAP layers within a virtual SWAP layer, half of the links remain idle, it allows us to schedule half the IE operations on these idle links. An example is shown in Fig. 14 (c), (f), (h), and (k). It is as if a vertical gate interaction between two rows are split into two steps, shown as the red links (gates) in Fig. 14 (f) and (h). We omit the discussion of the head and tail here due to space limit.

**Putting everything together:** We describe the interleaved IE and IA operations in Algorithm 3 and illustrate it with an example in Fig. 14. Note that both illustrate IE operations for all rows and columns in the Sycamore lattice, not just a pair of rows. In total it takes 5 cycles for such a big iteration (excluding the last iteration where certain swaps can be omitted) although we have drawn 8 of them.

**Discussion of the Odd-number Row Case.**

One can see that we performed a similar optimization to Optimization I in the grid case in Section 3.4.1. We let one virtual SWAP layer be shared by two consecutive odd-even row and even-odd row IEs. For instance, in Fig. 14, after a virtual SWAP layer (c), (e), and (f), the computation on (g) as well as the red links on (f) and (h) cover even-odd and odd-even row IEs respectively. This would still work for the even-number row case for Sycamore, but not for the odd-number row case. The last row in the odd-number row case does not have direct links between qubits within it, not like the last row in the grid. Moreover, since all other rows are busy now, it cannot use the diagonal links. Therefore we cannot satisfy the layout change needed for the even-odd row pair (the last and second-to-last rows).

In this case, we can fall back to the generic Maaps method, by not sharing the SWAP layer for two consecutive odd-even and even-odd row IE. Asymptotically, we increase the depth by a factor of 6/5. But we do not have to do so. Our idea is to simply neglect the interaction between the last physical row and the second to last physical row. Then we perform the missing inter-row IEs in a separate pass. The idea of omitting is shown as an example in Fig. 15.

It happens that the omitted IEs are between consecutive even-numbered rows, and between the second-to-last row and the last row. We sketch the proof here. Since even-numbered units move down the line until it hits the border and stops there for one SWAP cycle before it changes direction, each even numbered row will hit the bottom border once. Also since originally the distance between two even-numbered units is 2 and they move at the same speed before hitting the border, the last two physical rows will contain two even-numbered rows except one case. During the very first step, the IE between the second-to-last row (the largest even-numbered row) and the last row is also missing.

With this property, we can easily break down the missing IEs into two parallel layers. The first layer consists of pairs of rows 2-4, 6-8, 10-12, and so on. The second layer consists of the rest of the pairs, 4-6, 8-10, and so on. Since the distance between each pair is 2, we can apply concurrent SWAPs to move one next to the other. Then we revert the SWAPs to restore the natural number ordering, then we do the same for another parallel layer. In total it takes 6 layers including SWAP, computation and reverting, but it applies the factor in front of $N$ which is the square root of the total qubit number $N^2$, as each IE takes $N - 1$ depth.

**Runtime:** The IA loop is 5 cycles for an NxN Sycamore, the set of cycles will take $5N-1$ cycles. Thus, the total run time on any Google Sycamore architecture is at most $(5N-1)(N/2) + (N-2)$ cycles. Compared with our grid solution, it is $5/3$ factor slower. Compared with the line architecture, it is $5/4$ factor slower. Note that it is not possible to draw a line to connect all nodes in Sycamore. Hence we cannot directly
3.4.3 The Hexagon Architecture

Here we present an imaginary architecture – the connected hexagon architecture in Fig. 16(a). Each node except the boundary node has a degree of 3. If we flatten it out, it looks like the one in Fig. 16(b). Notice that there is often no line embedding that covers every node in the hexagon architecture, as shown in Fig. 16(b), so the line architecture solution cannot apply here. We let each unit be a vertical column of nodes, as highlighted in Fig. 16(b).

3.5 Application on Non-Linear IBM Machines

We can handle IBM heavy-hex architectures, by adapting the line solution.

Although there is no line embedding that covers all the nodes in these IBM architectures, we can still find a line embedding that covers the overwhelming majority of them. In Fig. 18, we show an example of one such line embedding, with the gray nodes being off the line embedding.

A very simple approach would be to run two full sets of iterations for the line architecture in a row. It will ensure inter-
action between all nodes in the line, as well as the interaction between nodes on and off the line. As mentioned previously and proved in prior and our studies [35][3], a node (whether it is even-numbered or odd numbered) moves towards a certain direction until it hits the border, once it hits the border, it stops for one step, and then changes the direction. In total it travels N steps with one step being stopped. It may not visit all physical locations, but if we let it continue and do another set of full iterations, it will cover each physical location, as shown in Fig. [19]. That ensures a node on the line is neighbor to each off-line node. Next we swap the nodes off the line onto the line and then run the line solution again. This guarantees to have all the nodes interact with each other in linear time. The total depth is \(6n+O(1)\) if the input graph is clique. The idea is illustrated in Fig. [18].

4. HANDLING REAL PROBLEM GRAPHS

Every problem graph of \(n\) vertices is a subgraph of \(n\)-clique graph. Since Section [3] presents a method of executing all edges of a clique problem graph in linear time, any non-clique problem graph can also be completed in linear time.

However, in practice, the problem graph is usually not a complete graph. Therefore, we further propose methods to adapt the Maaps algorithm to reduce the depth and gate count. We discuss two approaches: (1) Subgraph-isomorphism based, and (2) graph partition based.

4.1 The Subgraph-isomorphism Approach

When we have a problem graph that is not a clique, we can try to avoid executing gates that do not appear in the problem graph, and then eliminate the cycles that do not include any active gate. We find the minimal cycle from which no more gate execution occurs. That will give us the actual number of cycles to execute a problem graph.

We find that mapping the nodes in the problem graph to the physical qubits in Maaps structure is important since different mapping will result in different depths. We show an example in Fig. [20].

Finding the mapping from the problem graph to the Maaps structure is important, yet challenging. The goal is to minimize the number of continuous cycles that cover active gates. We use a subgraph-isomorphism approach.

Before we discuss this solution, we define a series of pattern graphs \(pg_k\), \(k = 1...n\) for Maaps pattern. If we gradually build a graph based on which edges in the problem graph have been executed with respect to the computation cycle number, we will have a series of graphs.

Since our problem is to find the minimal number cycles, we can try to fit the problem graph into the each of the \(pg_k\) graphs, from \(k = 1\) to \(l\), where \(l\) represents the first graph that can cover all edges in the problem graph. If the input problem graph is not a subgraph of \(pg_k\), we advance to \(pg_{k+1}\). This is guaranteed to terminate since any problem is a subgraph of the clique graph. At most we let \(k\) be \(n\) which is total number of vertices.

Fitting a graph into another graph is a well studied problem — the subgraph-isomorphism problem. We use a recent subgraph-isomorphism algorithm to implement it [13].

4.2 The graph partition approach

However, the subgraph-isomorphism approach has large overhead. Our experiments show that it takes un-necessarily long time to find a mapping when the qubit number goes above 27 qubits. Moreover, it has to follow Laaps or Maaps exactly, which may not necessarily be the best case.

When handling large problem graphs, we partition a large problem graph into small but dense components and minimize the cut edges [6][13][4]. The hardware coupling graph is also divided into multiple partitions such that each partition is mapped to one partition of the problem graph. We define the degree \(D_i\) of each partition \(G_i\) as the number of cut-edges with one node in \(G_i\). We map the partition \(G_i\) with highest \(D_i\) to the center of hardware coupling graph. Then for each mapped graph partition \(G_i\), we rank their unmapped neighbours and pick the one with the highest degree to be mapped close to its mapped neighbour. We repeat this until all partitions are mapped.

Within each hardware coupling partition, we use the Maaps method. Different partitions run concurrently. After all partitions are handled separately, we handle the remaining cut edges. At this stage, the sub-graph with remaining edges will be sparse. We can divide them into multiple disjoint connected components and handle each one of them separately, again with Maaps pattern. If it is very sparse, we can use a heuristic approach to move qubits that participate in two-qubit operators close to each other using the Floyd-Warshall algorithm, for one pair at one time.

Performance guarantee:. We acknowledge that introducing a heuristic method after graph partition might lose the performance guarantee. We tackle this problem for always keeping an upper-bound of the number of cycles during the compilation process. To start with, an initial mapping given by the graph partition method already implies how long it takes if we follow the exact Maaps pattern. We save this bound. Then at each step, given an updated mapping, we calculate how many more cycles based on following Maaps, and update the upper bound when necessary. In the end, if the compiled circuit has larger depth than the upper bound, we then revert to the point where this upper bound is achieved, and use it as the final compiled circuit. Hence it is still guaranteed to be better or the same as the linear depth we have defined in Section [3].

5. EVALUATION

We evaluate our proposed methodology by comparing our results with state-of-the-art QAIM-IC [2], QAOA-OLSO [32] and PauliHedral [19] with respect to depth, gate count, and fidelity. We performed experiments with three type of problem graphs: random graphs, regular graphs, and 2-local Hamiltonian simulation graphs (for 1-D NN Ising model, 2-D NNN XY model, and 3-D NNN Heisenberg model). We vary vertex numbers to up to 1024 and graph densities to up to 50%.

5.1 Experiment Setup

Target Architectures For large architecture, we use two
popular architectures. One is the Google Sycamore which can be expanded to any number of qubits by adding diagonal connections. The original Sycamore structure is shown in Fig.[1] The other is IBM Washington architecture in Fig. [1] which comprises of connected 12-node rings. Every two adjacent rings share 3 qubits. It can be expanded to any number of 12-node rings. IBM Brooklyn and Washington (IBM’s largest quantum machine) have such structure. IBM Brooklyn has 8 complete rings while IBM Washington has 18 (or 16 due to temporarily down links) complete rings.

For small-scale experiments, we use synthetic 2D lattice architecture similar to that in [2] and [16]. We use four grid sizes: $3 \times 4, 4 \times 4, 4 \times 5$ and $5 \times 5$.

**Metrics** We use depth and gate count of the compiled circuit. We also use the estimated success probability (ESP) to evaluate the quality of the compiled circuits. ESP is defined as the product of the success probability of individual gates. We also use the metric of compilation time to evaluate the efficiency of our compiler.

**Benchmarks** We use the library Networkx in python [12] to generate Erdős-Rényi random graphs and random $k$-regular graphs. For each type of graphs and each vertex number, we vary graph density to 0.3 and 0.5. In addition, we also use the interaction graph for 2-local Hamiltonian, including 1-D NNN Ising model, 2-D NNN XY model, and 3-D NNN Heisenberg model ('NNN' stands for the next nearest neigbouring) [16].

**Baseline** We compare our method with that by Alam et al. [2], which is named QAIM-IC, which is the fastest version among all proposed in their paper. QAOA-OLSQ [32] which is an optimal scheduler for QAOA. We also compare our approach with Paulihedral [19], reproduced by following the algorithm in the paper.

### 5.2 Impact on large-scale architectures

First, we show experiment results for the large scale problem graphs with different densities and qubit counts, on IBM heavy-hex and Google sycamore-like architectures.

**Depth** The depths for the large scale experiments are shown at log-scale in Fig. [23] and [24]. In all experiments and on both architectures, our compiler significantly outperforms both baselines with respect to depth.

On the IBM heavy-hex architectures, our compiler achieves up to 3X improvement in depth over QAIM for random graphs with 0.3 density, and up to 4X improvement for random graphs with 0.5 density; the improvement over Paulihedral is even greater. Similar speedups are seen for the regular graphs. In both cases, the improvement increases with the number of qubits, indicating that our compiler is able to scale better to larger problems compared to the baselines, thanks to the linear bound guaranteed by our approach.

On the Google sycamore-like architectures, we observe the same speedups over QAIM on both random and regular graphs. Our compiler also achieves lower depths for each benchmark on the sycamore-like architecture, compared to the heavy-hex architecture. This is because for the IBM architectures, we ran the standard Laaps pattern on a line-embedding, whereas for the sycamore-like architecture we used the specialized Maaps pattern. The improvement demonstrates that our 2D Maaps pattern is able to take advantage of the increased connectivity of the non-linear sycamore-like architecture to improve upon the basic linear pattern.

**Gate Count** The gate count results are shown at log-scale in Fig. [23]. We only show results for experiments up to 256 qubits, as the QAIM compiler could not handle benchmark sizes larger than this.

Our compiler performs better than the Paulihedral baseline, and performs on par with the QAIM baseline. While our compiler achieves better gate counts on the IBM heavy-hex architecture, the QAIM compiler achieves better gate counts on the Google sycamore-like architectures the 64 and 128 qubit benchmarks. This is because the Maaps pattern used for sycamore-like architectures uses many swaps to facilitate intra-row interactions, since there are no direct connections between qubits in a given row. Despite this, our compiler still achieves gate counts very close to the QAIM compiler, and even achieves a better gate count on the sycamore-like architecture for 256 qubits. This suggests our compiler is able to scale to larger architectures better than QAIM with respect to gate count.

When calculating ESP on large benchmarks, the large number of gates quickly drives the product down to 0, so we do not directly show the numbers for large benchmarks. However, gate count provides a good indicator of ESP; a smaller gate count indicates higher ESP. Our gate counts are on par with or better than that of QAIM and Paulihedral. Decoherence error is related to depth; our compiled circuits have significantly lower circuit depths, implying a much smaller
Figure 20: (a) Input problem graph. (b) A six-qubit linear physical coupling graph. (c) Following the line solution with one valid mapping, resulting in depth 8. (d) Following the line solution with another valid mapping, resulting in depth 5. The dashed-line gates in (c) and (d) are in original pattern but not in the input problem graph. The solid-line gates represents the gate in both.

Figure 21: Pattern graphs for Laaps-6, corresponding to the execution sequence in Fig. 5.

Table 2: Small scale Comparison

| Depth | Gate Count | ESP |
|-------|------------|-----|
| Graphs | Ours | QAIM | Ours | QAIM | Ours/QAIM |
| 10-0.3 | 6 | 11 | 19 | 23 | 1.27X |
| 15-0.3 | 15 | 25 | 64 | 65 | 1.06X |
| 20-0.3 | 22 | 33 | 119 | 129 | 1.83X |
| 25-0.3 | 29 | 52 | 194 | 215 | 3.57X |
| 10-0.5 | 15 | 22 | 44 | 49 | 1.35X |
| 15-0.5 | 17 | 32 | 87 | 95 | 1.62X |
| 20-0.5 | 24 | 53 | 159 | 190 | 6.54X |
| 25-0.5 | 34 | 66 | 259 | 307 | 18.34X |
| 10-3 | 6 | 14 | 24 | 28 | 1.27X |
| 10-4 | 10 | 13 | 33 | 34 | 1.06X |
| 15-4 | 11 | 19 | 57 | 60 | 1.20X |
| 15-6 | 16 | 34 | 79 | 83 | 1.27X |
| 20-6 | 17 | 28 | 107 | 113 | 1.44X |
| 20-8 | 22 | 36 | 142 | 155 | 2.20X |
| 25-8 | 28 | 47 | 209 | 219 | 1.83X |
| 25-10 | 30 | 71 | 235 | 279 | 14.39X |

de decoherence error compared to QAIM. Since fidelity combines these two factors of ESP and decoherence error, overall our compiler should achieve a better fidelity.

Compilation Time: Compilation times for large benchmarks are shown in Table 1. The time is given in seconds. Our compiler runs much faster than the baseline. This is because, aside from the initial mapping of qubits, which can be done efficiently using existing graph partition methods, our compiler mainly follows a pre-defined pattern, which requires no intensive computation and only simple data structures. Thus the running time of our compiler depends only on the runtime of the graph partition as well as the length of the pattern, and our compiler is able to handle much larger benchmarks with ease, compared to the baseline.

Sensitivity to Density: To explore the impact of density on performance, we fixed the number of qubits to 256, and ran experiments on the IBM heavy-hex architecture at various densities; the results are shown in Fig. 25. While the depth achieved by the QAIM compiler increases from 2000 to 3500 as the density increases from 0.3 to 0.9, the depth achieved by our compiler only increases slightly from 600 to 800. We did not compare Paulihedral here as it is worse than QAIM-IC.

5.3 Impact on small-scale architectures

We ran both random and regular graphs ranging from 10 to 25 qubits on small $N \times M$ 2D architectures. The results are shown in Table 2.

Our compiler achieves better depths than QAIM as well as better gate counts on all small benchmarks tested. Since ESP is correlated to gate count, our compiler also achieves better ESP values for almost all the small benchmarks. Unlike the larger benchmarks, where we used a heuristic based on graph partitioning to find an initial mapping, on smaller benchmarks we can afford to run a subgraph isomorphism algorithm to obtain the optimal mapping for following our Maaps pattern exactly.

In addition, we also compare our approach with the QAOA-OLSQ compiler [32]. The results are in Table 3. Due to the compilation overhead of QAOA-OLSQ, the comparison was only conducted for up to 15 qubits.

For the most of cases, our approach has better depth than QAOA-OLSQ, 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. But again, QAOA-OLSQ is a (near-)optimal solver and its compilation overhead is of magnitudes larger than ours. 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%.

5.4 2-local Hamiltonian problem graphs

The results for 2-local Hamiltonian experiments are shown
in Table 4. These were run on medium scale 64-qubit Google sycamore-like and IBM ring-life architectures. Our compiler beats the QAIM baseline for all 2-local benchmarks.

6. RELATED WORK

Existing compilation approaches include generic compilers that assume the gates in a circuit have a fixed set of partial order. There has been extensive study in generic compilers [37][18][21][28][29][39][38][32][33][35][26].

QAOA [9][8][10] however is a special type of application that allows commutativity among two-qubit operators. It exposes new optimization opportunity for generating better depth and fidelity circuits. Hence there has been a handful of studies for compiling and optimizing QAOA [17][2][32][19][16]. Tan et al. [32] provide a constrained SAT solver to achieve optimality. Alam et al. [2] use the heuristic of connectivity strength. Paulihedral [19] handles generic Pauli-strings in Hamiltonian simulation. 2QAN [16] proposes an algorithm that has quadratic-time complexity and uses unitary unifying to enhance the results. In particular, 2QAN not only handles QAOA circuit but also 2-local Hamiltonian simulation circuits.

But none of these studies provide performance guarantee or rigorous theoretical analysis as to how far the obtained circuit is from the optimal for multi-dimensional architectures. The consequence is that it may generate circuit with un-necessarily large depth and hence sabotage the depth advantage of QAOA or Hamiltonian simulation. Our study is the first that provides a structured method with performance guarantee for multi-dimensional architecture together with a concurrent work by Weidenfeller et al. [35]. It has low compilation overhead and hence scales to large architectures.

7. CONCLUSION

We present a novel pattern-based compiler framework for QAOA and 2-local Hamiltonian simulation circuit with linear bound. The pattern applies to multi-dimensional architecture. We demonstrate how to adapt it to Google Sycamore and IBM architectures. Our methodology excels in depth and gate count for both large and small scale benchmarks.
Figure 24: Depth comparison at Google

Figure 25: Depth of random graphs with 256 nodes and different densities for IBM architecture
REFERENCES

[1] “The ibm quantum hexahex lattice,” May 2022. [Online]. Available: https://research.ibm.com/blog/heavy-hex-lattice

[2] M. Alam, A. Ash-Saki, and S. Ghosh, “Circuit compilation methodologies for quantum approximate optimization algorithm,” in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 215–228.

[3] M. Alam, A. A. Saki, and S. Ghosh, “An efficient circuit compilation flow for quantum approximate optimization algorithm,” in Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference, ser. DAC ’20. IEEE Press, 2020.

[4] K. Andreev and H. Racke, “Balanced graph partitioning,” Theory of Computing Systems, vol. 39, no. 6, pp. 929–939, 2006.

[5] Anonymous, “Our non-refereed publication (pdf will be provided by pc chairs),” Dec 2021.

[6] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," Algorithm engineering, pp. 117–158, 2016.

[7] J. Chow, B. Johnson, J. Gambetta, R. Z. Sarango, and Saul, “Ibm’s roadmap for scaling quantum technology," Feb 2021. [Online]. Available: https://research.ibm.com/blogs/ibm-quantum-roadmap

[8] E. Farhi, J. Goldstone, S. Gutmann, and H. Neven, “Quantum algorithms for fixed qubit architectures,” 2017.

[9] E. Farhi, J. Goldstone, and S. Gutmann, “A quantum approximate optimization algorithm,” 2014.

[10] E. Farhi, J. Goldstone, S. Gutmann, and L. Zhou, “The quantum approximate optimization algorithm and the sherrington-kirkpatrick model at infinite size,” 2021.

[11] R. P. Feynman, “Simulating physics with computers,” International journal of theoretical physics, vol. 21, no. 6/7, pp. 467–488, 1982.

[12] A. Hagberg, P. Swart, and D. S. Chult, “Exploring network structure, dynamics, and function using networks,” Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[13] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, “Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together,” in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 1429–1446.

[14] G. Karypis and V. Kumar, “Multilevel algorithms for multi-constraint graph partitioning,” in SC ’98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing. IEEE, 1998, pp. 28–28.

[15] I. D. Kivlichan, J. McClean, N. Wiebe, C. Gidney, A. Aspuru-Guzik, G. K.-L. Chan, and R. Babbush, “Quantum simulation of electronic structure with linear depth and connectivity,” Physical Review Letters, vol. 120, no. 11, mar 2018. [Online]. Available: https://doi.org/10.1103/PhysRevLett.120.110501

[16] L. Lao and D. E. Browne, “2qan: A quantum compiler for 2-local qubit hamiltonian simulation algorithms,” 2021. [Online]. Available: https://arxiv.org/abs/2108.02990

[17] L. Lao, H. van Someren, I. Ashraf, and C. G. Almudever, “Timing and resource-aware mapping of quantum circuits to superconducting processors,” 2020.

[18] G. Li, Y. Ding, and Y. Xie, “Tackling the qubit mapping problem for nis-q era quantum devices,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2019, pp. 1001–1014.

[19] G. Li, A. Wu, Y. Shi, A. Javadi-Abhari, Y. Ding, and Y. Xie, “Paulihestra: 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, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 554–569. [Online]. Available: https://doi.org/10.1145/3530222.3507715

[20] S. Lloyd, “Universal quantum simulators,” Science, vol. 273, no. 5278, pp. 1073–1078, 1996. [Online]. Available: https://www.science.org/doi/10.1126/science.273.5278.1073

[21] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. 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, ser. ASPLOS ’19. New York, NY, USA: ACM, 2019, pp. 1015–1029. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304075

[22] B. O’Gorman, W. J. Huggins, E. G. Rieffel, and K. B. Whaley, “Generalized swap networks for near-term quantum computing,” 2019. [Online]. Available: https://arxiv.org/abs/1905.05118

[23] W. D. Oliver and P. B. Welander, “Materials in superconducting quantum bits,” MRS Bulletin, vol. 38, no. 10, p. 816–825, 2013.

[24] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O’Brien, “A variational eigenvalue solver on a photonic quantum processor,” Nature Communications, vol. 5, no. 1, Jul 2014. [Online]. Available: https://dx.doi.org/10.1038/ncomms5213

[25] QISKit: Open Source Quantum Information Science Kit, https://qiskit.org/

[26] A. Shafaei, M. Saeedi, and M. Pedram, “Qubit placement to minimize communication overhead in 2d quantum architectures,” in 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2014, pp. 495–500.

[27] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” SIAM J. Comput., vol. 26, no. 5, p. 1484–1509, Oct. 1997. [Online]. Available: https://doi.org/10.1137/S009753979529317Z

[28] M. Y. Siraichi, V. F. d. Santos, C. Collange, and F. M. q. a. Pereira, “Qubit allocation as a combination of subgraph isomorphism and token swapping,” Proc. ACM Program. Lang., vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360546

[29] M. Y. Siraichi, V. F. d. Santos, C. Collange, and F. M. Q. Pereira, “Qubit allocation,” in Proceedings of the 2018 International Symposium on Code Generation and Optimization. ACM, 2018, pp. 113–125.

[30] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, “|ket>: a retargetable compiler for NISQ devices,” Quantum Science and Technology, vol. 6, no. 1, p. 014003, nov 2020. [Online]. Available: https://doi.org/10.1088/2058-9565/ab6e92

[31] M. Suzuki, “General theory of fractal path integrals with applications to many-body theories and statistical physics,” Journal of Mathematical Physics, vol. 32, no. 2, pp. 400–407, 1991. [Online]. Available: https://doi.org/10.1063/1.529425

[32] B. Tan and J. Cong, “Optimal layout synthesis for quantum computing,” in Proceedings of the 39th International Conference on Computer-Aided Design, ser. ICCAD ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415620

[33] S. S. Tanu and M. K. Qureshi, “Not all qubits are created equal: A case for variability-aware policies for nis-q era quantum computers,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS ’19. New York, NY, USA: ACM, 2019, pp. 987–999. [Online]. Available: https://doi.org/10.1145/3297858.3304007

[34] H. F. Trotter, “On the product of semi-groups of operators,” Proceedings of the American Mathematical Society, vol. 10, no. 4, pp. 545–551, 1959. [Online]. Available: http://www.jstor.org/stable/2033649

[35] J. Weidenfeller, L. C. Valor, J. Gacon, C. Tornow, L. Bello, S. Woerner, and D. J. Egger, “Scaling the quantum approximate optimization algorithm on superconducting qubit based hardware,” 2020. [Online]. Available: https://arxiv.org/abs/2002.03459

[36] R. Wille, L. Burgholzer, and A. Zulehner, “Mapping quantum circuits to ibm qx architectures using the minimal number of swap tokens,” 2020. [Online]. Available: http://doi.acm.org/10.1145/3360546

[37] A. Zulehner, S. Gasser, and R. Wille, “Exact global reordering for nearest neighbor quantum circuits using A*, in International
Conference on Reversible Computation. Springer, 2017, pp. 185–201.

[39] A. Zulehner, A. Paler, and R. Wille, “Efficient mapping of quantum circuits to the ibm qx architectures,” in 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2018, pp. 1135–1138.