Artifacts Available V1.1
Artifacts Evaluated Functional V1.1
Results Replicated V1.1
Check for updates

Optimizing Quantum Fourier Transformation (QFT) Kernels for Modern NISQ and FT Architectures

Yuwei Jin*
Rutgers University
Piscataway, USA
jyw413482880@gmail.com

Xiangyu Gao*
New York University
New York City, USA
xg673@nyu.edu

Minghao Guo
Rutgers University
Piscataway, USA
mg1998@cs.rutgers.edu

Henry Chen
Rutgers University
Piscataway, USA
hc867@scarletmail.rutgers.edu

Fei Hua
Rutgers University
Piscataway, USA
huafei90@gmail.com

Chi Zhang
Independent Researcher
USA
raymond.chizhang@gmail.com

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

***Abstract*—Rapid development in quantum computing leads to the appearance of several quantum applications. Quantum Fourier Transformation (QFT) sits at the heart of many of these applications. Existing work leverages SAT solver or heuristics to generate a hardware-compliant circuit for QFT by inserting SWAP gates to remap logical qubits to physical qubits. However, they might face problems such as long compilation time due to the huge search space for SAT solver or suboptimal outcome in terms of the number of cycles to finish all gate operations. In this paper, we propose a domain-specific hardware mapping approach for QFT. We unify our insight of relaxed ordering and unit exploration in QFT to search for a qubit mapping solution with the help of program synthesis tools. Our method is the first one that guarantees linear-depth QFT circuits for Google Sycamore, IBM heavy-hex, and the lattice surgery, with respect to the number of qubits. Compared with state-of-the-art approaches, our method can save up to 53% in SWAP gate and 92% in depth.**

***Index Terms*—Qubit Mapping; Quantum Fourier Transform (QFT); Program Synthesis; IBM Heavy-hex; Google Sycamore; Lattice Surgery.**

I. Introduction

Quantum computing has exhibited its pronounced advantages in several fields, including cryptography [35], financial modeling [34], and chemical simulations [33].

Quantum Fourier Transform (QFT) is a computation kernel used in many important quantum applications [9] [7]. The applications that utilize the QFT kernel include but are not limited to the following: Shor's algorithm [35], amplitude estimation algorithm [34] [41], quantum phase estimation (QPE) [20], hidden subgroup problem (HSP) [10], HHL algorithm [14], and quantum walks [8]. These algorithms are fundamental for many domains, including cryptography, finance, risk analysis, quantum simulations, and chemistry and materials sciences. These algorithms include both *near-term* and *long-term* applications. They cover not only long-term applications such as Shor's algorithm, which requires a significant number of qubits and gates, but also near-term applications such as QPE, which requires a moderate number of qubits and gate resources.

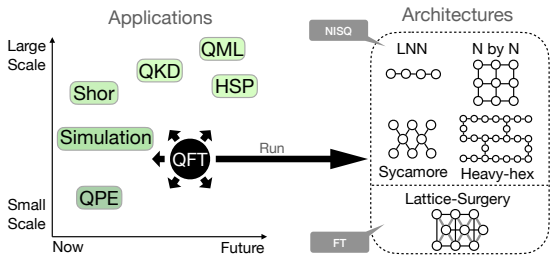


Figure 1: QFT serves as a computation kernel of diverse applications (e.g., Quantum Phase Estimation(QPE), Quantum Key Distribution(QKD), Quantum Machine Learning(QML), and Hidden Subgroup Problem(HSP)). There is a need to optimize QFT over diverse backends on the RHS.

Hence, optimizing the QFT kernel is of utmost importance. However, it is challenging for the following reasons.

(1) The QFT kernel is complex due to its requirement for all-to-all qubit interactions and strict dependency constraints, as the example shown in Fig. 2. Each qubit must interact with every other qubit using a controlled phase rotation (CPHASE) gate. Each qubit must act either as the control qubit or target qubit in the CPHASE gate, and follow a strict of ordering being control and target qubits. This complexity increases as the qubit number increases.

(2) The QFT kernel and other well-known quantum algorithms are often designed or optimized without specific hardware constraints in mind. Hence it is important to efficiently compile the logical QFT kernels into hardware circuits. For instance, the sparsity in today's hardware connectivity significantly hampers the execution of long-distance two-qubit gates. Two qubits must be physically adjacent to each other to execute a 2-qubit gate. This issue exists for both the noisy intermediate-scale quantum (NISQ) backends and certain fault-tolerant (FT) backends.

*These two authors contributed equally.

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

(3) The real-world quantum computing environment has the issues of noises, scalability, limited connectivity, and diversity in the backends. A QFT kernel may need to run in heterogeneous backends. These backends include the NISQ backends, such as the IBM heavy-hex device and the Google Sycamore device, and FT backends, such as the surface-code QEC devices equipped with the lattice surgery mode for 2-qubit gate implementations. It is challenging to provide efficient compilation solutions for a variety of backends and ensure consistent compilation quality. Moreover, the current compiler must recompile each time, given a different input size of the QFT kernel, and may fail to provide consistent quality across different input sizes.

Our Goal. Our paper focuses on optimizing the QFT kernel in the real quantum computing environment. As mentioned above, it is important to optimize QFT for heterogeneous backends, and have them ready before plugging them in different important quantum applications. We tackle this problem by leveraging the following key insights:

Key Insight 1: Break the Strict Ordering of Gates. The QFT kernel has abundant CPHASE gates. Each CPHASE gate commutes with the other. Hence, these gates do not have to follow the strict dependence order in the conventional logical circuits for the QFT kernel represented in the quantum textbooks [31]. Based on our experiment results, exploiting the commuting property does not affect the correctness of QFT, but makes a big difference when performing SWAP insertion during the compilation.

Key Insight 2: QFT Subkernel Partitioning. We develop a novel and flexible QFT sub-kernel partition methodology. This methodology allows reducing the compilation problem of the QFT kernel for a high-dimensional architecture into that for a low-dimensional architecture.

Key Insight 3: Unifying Different Methods. We unify the approaches from Insight 1 and Insight 2. Combining these two approaches can significantly improve the overall QFT kernel performance on hardware. For instance, while being able to reduce high-dimensional problem to low-dimensional problem, our sub-kernel splitting may cause extra delay in the circuit. However, breaking the ordering of gates may compensate for the delay of the circuits. We present a framework of methodology, and showcase multiple scenarios these two key insights can be flexibly combined.

Other Insights. We leverage program synthesis [38] methods to help enhance the compilation of the QFT kernels. It helps integrate human intelligence (educated guesses) for finding structured solutions to exploit the *regularity in both the QFT kernel and the scalable hardware we deal with*. Our experience in using program synthesis for compiling the QFT kernel is not only useful for QFT but may potentially be useful for compiling other quantum algorithms with a regular structure.

Our Contributions. With all the above, we have developed a unified framework of methodology for compiling the QFT kernel. Our framework has the following benefits: *(1) Our approach does not require recompilation when the number of qubits changes*; *(2) Our approach adapts to different backends, including both NISQ and FT backends; (3) Our approach provides consistent performance and fidelity guarantee for different input sizes and backends*.

Most importantly, for the first time, our work discovered *linear-depth hardware QFT kernel* for NISQ devices including IBM heavy-hex and for Google Sycamore, and FT devices equipped with the surface code QEC and the lattice surgery model. Our experiments have demonstrated a huge improvement in both depth and gate count over the state-of-the-art approaches. Our contributions are multifold:

- Guaranteed compilation quality: linear-depth solutions for the QFT kernel on different architectures: Google Sycamore, IBM Heavy-hex, and an FT backend with lattice surgery model.
- Commutativity exploitation for reordering CPHASE gates in the QFT kernel.
- A novel and flexible QFT sub-kernel partition that allows hierarchical decomposition of high-dimensional problems to low-dimensional problems.
- In-depth understanding of the optimization space for the QFT kernel on real hardware.
- Our hardware QFT kernel mapping solutions have up to 53% fewer SWAP gate count and 92% less depth than state-of-the-art approaches for up to 1024 qubits compared with SABRE [22].

II. Background

We introduce the hardware mapping problem first. Then we introduce a low-dimensional QFT kernel. We also describe both the NISQ backends and FT backends, in Section II-B and Section II-C. respectively.

A. Hardware Mapping

In superconducting quantum hardware, a two-qubit gate can only be executed when its two qubits are located in 2 adjacent physical qubits. In reality, due to the sparse connectivity of the current quantum machine, SWAP gates are required to change the hardware mapping and enable two-qubit gate interaction. For instance, for a line topology $q_0 \leftrightarrow q_1 \leftrightarrow q_2 \leftrightarrow q_3$, to execute a two-qubit gate between q_0 and q_3, one can insert 2 SWAP gates: SWAP(q_0, q_1) and SWAP(q_2, q_3) to move q0 and q3 closer.

B. A Low-dimensional QFT Kernel

Prior studies [29] [44] have discovered a hardware mapping solution for the QFT kernel on the linear nearest neighbor (LNN) architecture, which is a line of connected qubits. In our sub-kernel partition approach, we will recursively decompose the problem into the low-dimensional case, this LNN case serves as *our base case for low-dimensional problem*. This base case also serves as an inspiration for our use of the program synthesis tool when discovering QFT kernel solution on non-trival architectures. The logical circuit of the QFT kernel is shown in Fig. 2 (a).

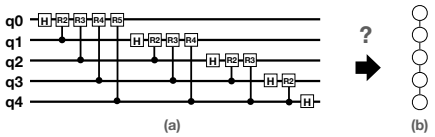


Figure 2: QFT Logical Circuit and a Line Topology

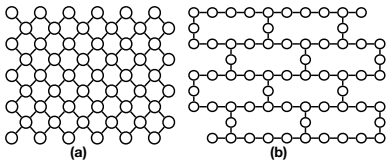


Figure 4: (a) Google Sycamore (b) Heavy-hex

We show the LNN solution via an example in Fig. 3. The circuit clearly exhibits a pattern. We will analyze this pattern. Before the analysis, we assume N is the number of qubits, q_i represents a logical qubit, and Q_i represents a physical qubit.

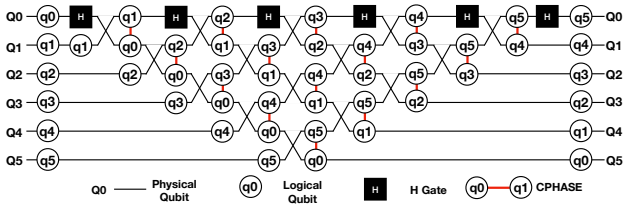


Figure 3: Hardware mapped QFT in LNN. q_i is a logical qubit. Q_i is a physical qubit. The single-qubit gate runs in parallel with two-qubit gates. Each qubit moves to the top first and then moves down, except q0, which directly moves down. When a qubit is at the top, it stops for one time step.

Initial mapping: Each logical qubit q_i is mapped to the physical qubit Q_i initially, $q_i \rightarrow Q_i$.

Repeating Steps: We denote a CPHASE gate between physical qubits Q_i and Q_{i+1} as $G(Q_i, Q_j)$, where Q_j is the control in the CPHASE gate, and Q_i is the target in the CPHASE gate. Each parallel layer is a sequence of consecutive pairs of SWAP or CPHASE starting from the physical qubit Q_0 or Q_1. The upper bound of the qubit index increases by one at one time for the physical qubits. Note that, interestingly, for the i-th parallel layer of CPHASE or SWAP gates, the logical qubits in each pair, have their indices sum to a constant number for each layer. At the very end, the qubits are as if reversed on the line, such that the mapping is $q_i \rightarrow Q_{N-1-i}$.

Implications: One may hope to find a Hamiltonian path that connects all the qubits and then apply the LNN solution for the QFT kernel. However, it is difficult to find such a path in modern architectures, whether NISQ or FT backends. Fig. 4 (a) and (b) show, respectively, Google's and IBM's superconducting architecture. It is possible to find a path, but such a path may not be able to connect all nodes. It can be proven that the Hamiltonian path does not exist. Furthermore, checking the existence of a Hamiltonian path in one graph is an NP-complete problem [1], meaning the cost spent in finding one Hamiltonian path might exceed the gain to leverage previously proposed efficient qubit mapping over LNN architecture.

Nonetheless, this LNN solution is still useful. Our proposed solutions, as described later, are either **non-trivial** extensions of the LNN solution or will use the LNN case as the base case when we perform recursive sub-kernel partitioning.

C. The Fault-tolerant Backend and the Lattice Surgery Model

The surface code is a promising method for implementing fault-tolerant quantum computing (FTQC) as it has a high tolerance error threshold [16], [27]. The lattice surgery [15] is one of the modes of the surface code FTQC. In the lattice surgery mode, a 2D grid of logical qubits is formed by tiling the plane with rectangular tiles, as shown in Fig. 5, where the grey tiles are logical computation qubits, and the white tiles are logical ancilla qubits. The ancilla qubits are necessary for implementing CNOT gates. Hence the ancilla qubits are interleaved with the data qubits. There are different ways to arrange ancilla qubits on the grid. The layout in Fig. 5 is a compact way for better resource usage.

For long-range CNOT gates, SWAP gates are necessary [5]. However, unlike the NISQ devices, where the SWAP has the same latency across all links, the SWAP has different latencies on different links. SWAPs between diagonal (grey) tiles are faster, they have depth of two by using two ancilla qubits at once. SWAPs on horizontal or vertical tiles have to be implemented using three CNOT gates, which have a depth of six, as each CNOT has a depth of two [5]. Note that a CNOT can happen also between two diagonal tiles, with the same latency as happening between horizontal or vertical tiles. Such links are illustrated in Fig. 5 (a). In Fig. 5 (b), we describe the links between qubits using a graph, where each node is a data qubit. Fig. 5 (c) is a stretched representation of Fig. 5, facilitating our further technical description.

Discussion: It is worth noting that no existing SWAP insertion approaches take the heterogeneous nature of the lattice surgery links for CNOT gates. The existing greedy SWAP insertion approaches such as Qiskit Sabre [22] assume each link has the same latency, but it is not true for lattice surgery. The existing analytical approach, the LNN approach we introduced, does not take the heterogeneous links into consideration either. While it is possible to find a Hamiltonian path, in the graph in Fig. 5, as the links have different latency, our approach significantly outperforms the LNN approach, as will be demonstrated in Section VII.

D. Program Synthesis

A program synthesis engine takes as an input a specification and an implementation. The specification describes what needs to be achieved, for instance, we require all CPHASE and H gates in QFT to be executed with respect to its dependencies in the transformed circuit. The implementation specifies the shape of the code that can potentially achieve the goal in the specification. However, such a code shape is roughly specified,

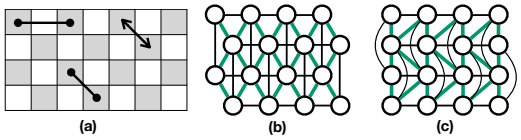


Figure 5: Lattice surgery mode: (a) Each tile represents a logical qubit. A grey tile is a data qubit. A white tile is an ancilla qubit. (b) The graph representing of qubits and their links. Green links are for faster SWAP. Black links are for slower SWAP. (c) A stretched grid of (b).

and certain parameters remain to be solved by the synthesizer. We leverage the regularity of the hardware, and represent the QFK kernel solutions as affine loops. Then we synthesize the loops using SKETCH [38] for certain scenarios in our sub-kernel partitioning method.

III. Our Key Insights

We describe how we break the dependences in Section III-A, perform sub-kernel partition in Section III-B, and one general scenario to combine the two in Section III-C.

A. Breaking the Dependence

The original QFT kernel (described in Fig. 2) has two types of dependence.

- **Type I dependence (has room to relax):** If two gates share the same control (or target), the one with a larger target (or control) index must run after the one with a smaller index. Assuming $G1 = G(q_i, q_j)$ and $G2 = G(q_i, q_k)$, if $j < k$ and $i \neq j$, G2 must run after G1. Same for the other way around for $G1 = G(q_j, q_i)$ and $G2 = G(q_k, q_i)$.
- **Type II dependence (cannot relax any more):** If one gate's control is another gate's target, the latter must run after the former. For instance, if $G1 = G(q_i, q_j)$, $G2 = G(q_j, q_k)$ where $i \neq j$ or $j \neq k$, G1 must run before G2.

There is a special case where a single-qubit Hadamard gate (H gate) also appears in the circuit. It still fits into the dependence constraint. We let H gate be presented as $G(q_i, q_i)$, allowing the control and the target to be identical in gate. Following the constraints defined above, it still works and defines the whole circuit's dependence.

However, we show that *Type I dependence* constraint is unnecessary. Two CPHASE gates sharing the same qubit can commute [4], [21]. This result holds since the CPHASE gate is a diagonal unitary. One may wonder why Type II dependence does not hold. The reason is that, we have H gate in between the CPHASE gates. The H gate does not commute with the CPHASE gate. Hence, from $G1 = G(q_i, q_j)$ to $G2 = G(q_j, q_k)$ where $i \neq j$, there is always one $G(q_j, q_j)$ between them. But for gates that share the control or target qubit, they can always appear in a row, in the original QFT.

Examples for Breaking the Dependence We show two scenarios for breaking such a dependence. Fig. 9 provides a simple example showing that breaking such dependency could help save running cycles, or provide a different ordering (that will be useful in the heavy-hex case later).

B. Sub-kernel Partitioning

We start with the 2-partition case, and then we extend it to the k-parition case for the QFT kernel.

2-partition QFT First, we divide qubits into 2 subsets, and then divide the QFT process into three steps, without violating the dependence constraints (both type I and II).

Let's assume we have qubits $U = \{q_0, ..., q_{N-1}\}$. U is divided into two sub-groups $U1$ (consecutive qubits $q_0, q_1, ..., q_k$) and $U2$ (consecutive qubits $q_{k+1}, ..., q_{N-1}$). We can prove that the following steps are correct.

- Step 1: Execute QFT on $U1$.
- Step 2: Execute the gates between $U1$ and $U2$, by preversing their original order among themselves.
- Step 3: Execute QFT on $U2$.

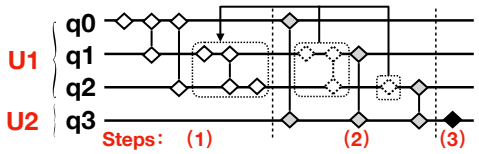


Figure 6: An example of dividing 4 QFT qubits into two subsets $U1$ and $U2$, and computation into thee steps. The dotted part shows the gates moved.

Due to the space limit, we sketch the proof for correctness. As long as a reordering satisfies these two types of dependence, it is valid. In the three steps above, we partition all gates involving qubits in $U1$ into two components: the interaction within $U1$ (Step 1), and the interaction between $U1$ and the rest of the qubits $U2$ (Step 2). Each component preserves its relative ordering before partition. We then first schedule all gates in Step 1 (within $U1$), all gates in Step 2 (between $U1$ and $U2$), and lastly Step 3 (gates within $U2$). An example is shown in Fig. 7.

Since it only has to satisfy Type II dependence as described in Section III-A, we only need to prove Type II dependence is preserved.

For type II dependence, assuming $G1 = G(q_i, q_j)$, $G2 = G(q_j, q_k)$, there are four cases for the location of i, j, and k. If all i, j, and k are in $U1$, since we preserve the original relative order among these gates themselves, type II dependence is preserved. If i, j are in $U1$, and k in $U2$, G2 runs after G1, since G1 is in step 1, and G2 is in step 2. If i in $U1$, and j, k in $U2$, G1 still happens before G2, since G1 belongs to Step 2, and G3 belongs to step 3. If i, j, and k are in $U2$, G1 happens before G2, as their relative ordering is preserved among these gates themselves in U2. Hence it is proved that such partition of QFT operations is correct.

In fact, it is trivial to see that Type I dependence is also preserved. We are simply regrouping gates that can run in parallel, as shown in the concrete example in Fig. 7.

k-partition QFT In the previous example, we demonstrate the partition of QFT qubits into two sets, and the partition of

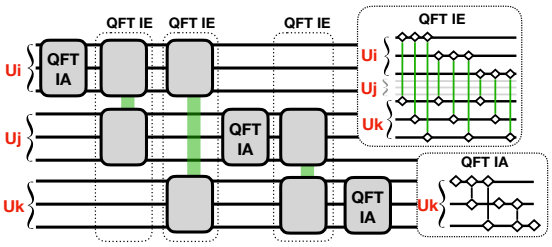


Figure 7: An example of the k-partition scheme for QFT.

QFT computation into three steps. It can in fact be extended to a k-partition QFT, where the qubits are divided into k subsets $U_0, U_1, U_2, ..., U_{k-1}$. Again each subset needs not to have the same size. We can first divide the qubits into two subsets of U_0 and $U_{P2} = \{U_1, U_2, U_{k-1}\}$. Then we partition U_{P2} into U_2 and $\{U_3 ... U_{k-1}\}$. The whole process applies, and the proof of correctness still holds.

We show the illustration of the k-partition QFT process in Fig. 7 and the pseudo-code in Fig. 8. The function QFT-IA denotes QFT for a range of consecutive qubits and a list of small ranges. The parameter $range\_list$ contains the list of small ranges. If the $range\_list$ is empty, QFT-IA degenerates to the traditional QFT operation, denoted as QFT-traditional, meaning we do not perform divide and conquer on this range of qubits. Otherwise, it performs a set of intra-QFT (QFT-IA) and inter-QFT (QFT-IE) on and between U_i, where $0 \leq i < range\_list.size()$. QFT-IE allows two small ranges of qubits to interact using CPHASE gates, and these two small ranges can have different sizes.

```
QFT(0, N) = QFT-IA([0, N], range_list)

QFT-IA(range, range_list) :
    iff range_list == NULL:
        QFT-traditional(range)
        return
    for range1 in range_list:
        QFT-IA(range1, NULL)
        for range2 in range_list
          & range2.idx > range1.idx:
            QFT-IE(range1, range2)

QFT-traditional(range):
    for i in range:
        H(i)
        for j in range & j > i:
            CPHASE(i, j)

QFT-IE(range1, range2):
    for i in range1:
        for j in range2:
            CPHASE(i, j)
```

Figure 8: The k-partition QFT process.

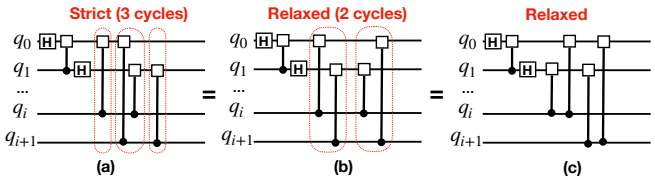


Figure 9: Benefits of relaxed ordering. (a) The original ordering of G(q_0, q_i), G(q_1, q_i), G(q_0, q_{i+1}), G(q_1, q_{i+1}). (b) (c) two other ways to reorder the gates

C. Unifying the Two Insights

We consider one possible scenario to unify these two insights. Since QFT-IE does not have any single-qubit gate (no gate $G(q_i, q_i)$), as it contains interaction between two sets, all gates in QFT-IE can commute. This corresponds to our discussion in Section III-A where we can break Type I dependence. Hence, we combine Insight 1 and Insight 2. The idea is that, after breaking down the computation into QFT-IA and QFT-IE, we can optimize the QFT-IE component using commutativity. At the logical circuit level, it does not improve the circuit depth. But in the hardware circuit level, since we have to insert SWAPs, this flexibility can offer a 2X speedup. We discuss that in Section V, IV and VI.

We then have two different versions of QFT-IE. We denote the first version as *QFT-IE-relaxed*, where gate reordering is exploited. We refer to the second version as *QFT-IE-strict*. Although QFT can directly use *QFT-IE-relaxed*, we include the discussion for *QFT-IE-strict* mainly because there are other circuits with similar structure to QFT but do not use CPHASE gates for two-qubit interaction.

The sub-kernel partition method QFT has two benefits. First, we can reduce a high-dimensional problem into low-dimensional problems recursively. Second, the unit size is broadly defined and each unit can have a different number of qubits.

Since we can solve the LNN mapping problem, if the units can be connected as if they are on "a line", one can run the unit-based linear QFT as shown in Fig. 14. The trick is that it needs to "SWAP" two adjacent units on the "line", and perform QFT-IE between two adjacent units on a "line", both in a hardware-compliant manner. For Google Sycamore and the lattice surgery grid, the units can form a "line", as described in Section V and VI.

IV. Linear-depth QFT on Heavy-hex

From Heavy-hex to Our Simplified Coupling Graph. We generate a coupling graph for the heavy-hex architecture by deleting some connection lines from the original heavy-hex architecture (details in Appendix of [17]). In the coupling graph (Fig. 10), those qubits with only one qubit as its neighbor are defined as **dangling points** (except the first qubit), and all other qubits are defined to be the **main line** points.

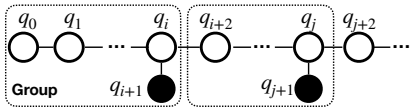


Figure 10: Initial mapping for Heavy-hex with dangling qubits (in black background): any node i with a node below has an index $i+1$, and the right node has an index $i+2$.

Initial Mapping in our Coupling Graph. We put logical qubits' initial mapping in Fig. 10. It follows a common rule: when we compare any two adjacent qubits, the qubit in the right/down position has a larger index.

Intuition behind our approach. Our method is a **non-trivial extension** of the LNN solution. We provide an in-depth understanding of the LNN solution first.

1) The displacement of all logical qubits follows an LNN pattern, characterized by a directional movement. Initially, all logical qubits shift towards the left. Upon reaching the leftmost position, they alter their trajectory.
2) Each logical qubit only starts moving after being involved in one CPHASE gate operation. We denote that as the *active* status of the qubit. A qubit will stop moving at one point, and we denote that as the *in-active* status of the qubit.

For the simplicity of description, we specifically let q_i denote the qubit connecting to the first dangling point, and q_j denote the qubit connecting to the second dangling point, and respectively q_{i+1} and q_{j+1} are first two dangling points.

Leveraging these findings, we describe our high-level idea below. We run the LNN operations on the first i qubits until we position the smallest-index active qubit on the main line adjacent to the dangling point, initially q_0. Then we move the smallest-index active qubit on the main line to the dangling position by a SWAP. The process iterates along the main line, moving the smallest-index active qubit once it reaches the next dangling point by a SWAP , and repeats. This sequence repositions first L smaller-index qubits to dangling positions (where L is the number of dangling points), disengaging them from LNN movement in the main line. The remaining qubits in the main line can continue to perform LNN QFT. We also stop every time after a SWAP layer and perform CPHASE cases, just like in the original QFT. Extra stops are needed for CPHASE between dangling points and their adjacent nodes.

Fig. 11 provides a concrete example with only one dangling point. Before q_0 arrives the position above q_4, all qubits follows the linear QFT solution; when q_0 is above q_4, we implement one SWAP gate between them; then, the main line is an intermediate step of an LNN QFT for all nodes from q_1 to q_{n-1} ((c) of Fig. 11). The main line can continue performing gate operations using LNN QFT solution.

How do we ensure it also performs CPHASE between q_0 and qubits from q_5 to q_{n-1}? In the straight line QFT, as shown in prior work [28], [44], each qubit q_i moves towards the left first, stops at the border for one step, and then to the right. All nodes after q_5 will move to the left end and then toward the right. When these nodes move to the left, they will be above q_0 at one point, where we let q_0 perform a CPHASE with these qubits. The interaction between q_0 and the nodes from q_5 to q_{n-1} are completed.

If there is more than one dangling point, we recursively handle this. We sketch the idea as follows. Specifically, it is as if we reduce one dangling point each time. Let's assume we have two nodes dangling from the main line (e.g., q_{i+1} and q_{j+1} in Fig. 10). We follow the same solution as the single-dangling-point case for a few steps first. The first time we swap q_0 with q_{i+1}, it is as if we can remove q_0 from the entire line, and then it reduces to the problem of having only one dangling point. We continue the solution for a straight line with one dangling point. Hence, we can extend our approach to two dangling points and, subsequently, to k dangling points.

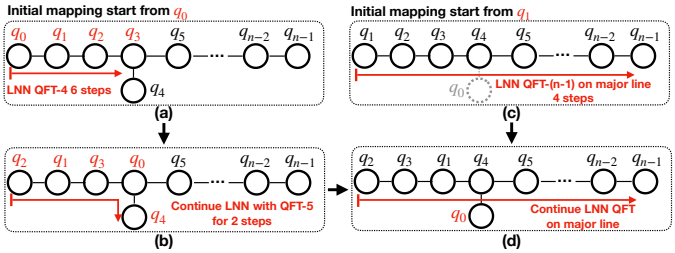


Figure 11: An example of running QFT in Heavy-hex with one dangling point. (a) Running LNN pattern in the main line with four active qubits. (b) Continue LNN pattern with the dangling qubit q4, which is identical to the LNN QFT-5 pattern. (c) Applying the LNN pattern on the main line with the first qubit q_1. (d) With dangling qubits, the qubit mapping would be the same as the case w/o dangling point. QFT-K means the QFT with k qubits.

We put the details in Alg. 1, and the final mapping of N qubits after the process is in Fig. 23 of Jin *et al.* [17].

Breaking the Dependence. In this method, we leveraged the insight of breaking the dependence in Section III-A. In the original QFT, q_0 interacts with all qubits q_2 to q_{last} before q_1 interacts with all of them. However, in our method, qubit q_0 is placed in the dangling point q_{i+1}. And qubit q_1 will be placed in the second dangling position of q_{j+1}. Hence, q_1 interacts with the qubits of indices larger than $j+2$ before q_0 interacts with them. This is okay, our Type II dependence only requires G(q_i, q_j) to happen before G(q_j, q_k). Since G(q_0, q_1) already has happened, for any qubit q_k, where $k \geq j+2$, G(q_1, q_k) can happen before G(q_0, q_k). An example is shown in Fig. 9(c).

Algorithm 1 Heavy-hex qubit mapping procedure

for $i = 0$; $i < N_1$; i++ **do**
  if $Q[0][i] < Q[0][i+1] \wedge C(Q[0][i], Q[0][i+1]) == 0$ **then**
    $C(Q[0][i], Q[0][i+1]) \leftarrow 1$; i++; ▷ CPHASE to the right qubit
  end if
  if $Q[0][i] < Q[0][i+1] \wedge C(Q[0][i], Q[0][i+1]) == 1$ **then**
    $SWAP(Q[0][i], Q[0][i+1])$; i++; ▷ SWAP with the right qubit
  end if
  if $Q[0][i] < Q[1][i] \wedge C(Q[0][i], Q[1][i]) == 0$ **then**
    $C(Q[0][i], Q[1][i]) \leftarrow 1$ ▷ CPHASE to the qubit below
  end if
  if $Q[0][i] > Q[1][i] \wedge C(Q[1][i], Q[0][i]) == 0$ **then**
    $C(Q[1][i], Q[0][i]) \leftarrow 1$ ▷ CPHASE to the qubit below
  end if
  if $Q[0][i] < Q[1][i] \wedge C(Q[0][i], Q[0][i+1]) == 1$ **then**
    $SWAP(Q[0][i], Q[0][i+1]) \leftarrow 1$ ▷ SWAP with the qubit below
  end if
end for

Time Complexity for Special Case. We calculate the time complexity for a special case with a dangling point for every four qubits on the main line. This is analogue to the heavy-hex case. In this case, each group consists of 5 qubits. 80% of qubits are in the main line while the remaining 20% are in the dangling position. The distance between 2 adjacent dangling points is 4. Our proof shows that an extra 25 steps are required for one added 5-qubit group. Therefore, the final complexity time is $5N + O(1)$, where N is the number of qubits in the

heavy-hex architecture.

Time Complexity Bound for General Case. Additionally, we also provide an upper bound for a general case where there are N1 qubits in the main line and N2 dangling qubits. The distance between 2 nearby dangling nodes might be different. The worst case is that every time there is a SWAP operation in the main line, another cycle is needed for the CPHASE operation with dangling points. The final time complexity upper bound is $6N + O(1)$. The detailed proof is in the appendix of Jin *et al*. [17].

V. Linear-depth QFT on Google Sycamore

Unit definition We combine every two rows together as a unit shown in Fig. 12. There are $m/2$ units in one $m * m$ Google Sycamore, each containing $2 * m$ qubits.

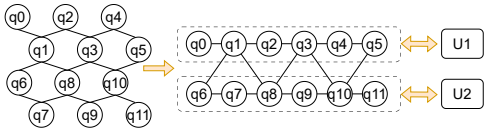


Figure 12: Unit definition in Sycamore architecture.

Unit SWAP. This design makes it possible to complete the unit swap between 2 adjacent units in 3 steps. We describe it for the particular simple example in Fig. 12.

```
parallelSWAP ({q1, q3, q5} {q6, q8, q10})
parallelSWAP ({q1, q3, q5} {q7, q9, q11})
             ({q0, q2, q4} {q6, q8, q10})
parallelSWAP ({q0, q2, q4} {q7, q9, q11})
```

We decompose the qubit mapping problem of QFT over the Sycamore architecture into two categories: intra-unit and inter-unit mapping.

Intra-unit QFT. Given all qubits within one unit are in a line, we leverage the mapping algorithm used for QFT over LNN architecture for intra-unit qubit mapping.

Inter-unit QFT (Relaxed Ordering). We apply the relaxed ordering in IE interactions between two adjacent units. We briefly show the qubit travel path in a LNN-inspired pattern (found by program synthesis) shown in Fig. 13 (a) for both the top and bottom units in one adjacent pair for bipartite all-to-all interactions. Each unit follows the LNN-inspired pattern, so every node is a neighbor to the other nodes in the same unit. Since both units are synced, and there are diagonal links between two units, this also ensures each node in one unit is a neighbor to all nodes in another unit once, except to the node in the same *the same column*.

Our approach. We discovered this inter-unit solution by parameterizing the relative LNN-inspired moving pattern in the top and bottom rows with the help of program synthesis (details in Appendix of [17]). Specifically, we find that if we sync the top and bottom to follow the same travel path in Fig. 13(a), an inter-unit interaction pattern emerges. In Fig. 13(b), there is a diagonal link between one qubit on the top row and the qubit on the bottom row *if the two qubits' column index differ by 1.* Note that the moving pattern in Fig. 13 (a) is similar to that in LNN, *but not the same*. For LNN, only on average half of the links are utilized, but in our pattern here, all links in a perfect matching are used.

Following the same travel path can guarantee that all top-row and bottom-row qubit pairs will be connected through the inter-unit link at least once, except for the pairs in the same column. Fortunately, figuring out a solution for these missing CPHASE gates is easy. We can SWAP one of them horizontally with its neighbor at the top (bottom) row, keep the other one on the bottom (top) row unchanged, run a CPHASE gate, and then revert the qubit to its original location with the same SWAP.

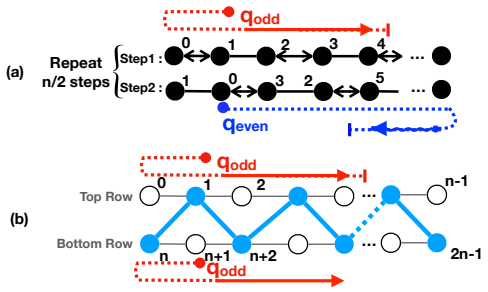


Figure 13: Travel paths for inter-unit QFT in Sycamore architecture. (a) Two consecutive SWAP layers for **one unit** over time. Each qubit has a different neighbor after one SWAP step. Both units (top and bottom) sync this way, although only one unit is drawn here. (b) The top and bottom rows sync with the same travel path. Qubits in a unit have different neighbors at each step from the other unit, using the top-down links highlighted in blue.

The QFT-IE-relaxed version is two times faster than the QFT-IE-strict version. A detailed solution is below:

```
for (i = 0; i <= m; i += 1)
   CPHASE on all inter-unit connections
  // Intra-unit swap
  beg = i mod 2
  intra_swap(beg_pos=beg, end_pos=m, UnitID=0)
  intra_swap(beg_pos=beg, end_pos=m, UnitID=1)
```

Another benefit of our solution is that both QFT-IE-strict and QFT-IE-relax will mirror the position of all qubits within a unit. It helps further processing that for QFT-IA.

Time Complexity Assuming $N = m * m$ Sycamore grid, where m is the original row size. In our formulation, each unit consists of 2m qubits, and there are $\frac{m}{2}$ units. Each unit is connected as if they are on a line, as described earlier. We will do the QFT-IA and QFT-IE using the divide-and-conquer method. The hardware-compliant unit QFT on LNN is presented in Fig. 14.

QFT-IE-relaxed For the QFT-IE-relaxed case, each QFT-IE takes $3 * (2m + 1)$ time steps, and each QFT-IA takes $4 * (2m) - 6$ time steps. Each unit SWAP takes three-time steps. There are in total $(m/2) + O(1)$ QFT-IE parallel steps, and in total $(m/2) + O(1)$ (QFT-IE, QFT-IA) mixed steps. There are a total of $2 * (m/2) - 3$ unit SWAP steps. Hence, the total time complexity is $7m^2 + O(m) = 7N + O(\sqrt{(N)})$, where N is the total number of qubits.

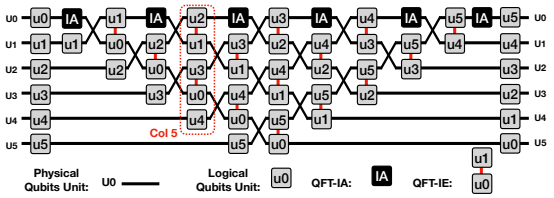


Figure 14: Unit-wise QFT using the recursive QFT scheme. This is analogous to the original LNN QFT solution.

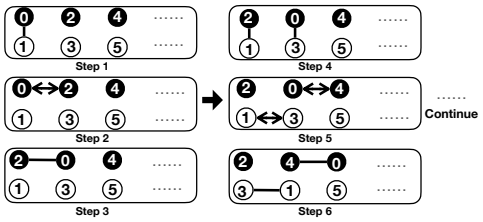


Figure 16: The first six steps of the 2×N QFT pattern.

VI. Linear-depth QFT on FT Backend

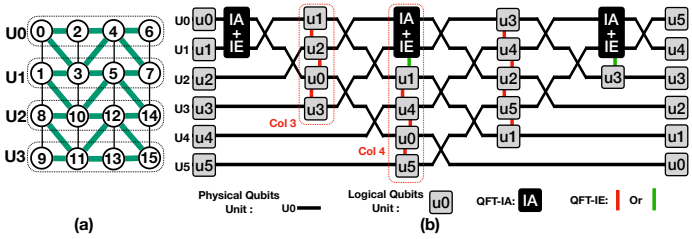


Figure 15: (a) Initial Mapping in Lattice Surgery. The layout of FT architecture is rotated to make all SWAP edges horizontal. Certain edges are unused and have been eliminated. (b) Optimized Unit Movement.

Unit Definition and Movement We consider each row in the (rotated) FT grid as a unit and place qubits in natural number ordering from left to right in a zigzag way for every two units (Fig.15(a)). Similarly, we still divide the problem into three sub-problems: intra-unit interaction QFT-IA, inter-unit interaction QFT-IE, and unit swapping.

We optimize the unit movement for the FT grid architecture as shown in the example in Fig. 15(b). We run every *two* parallel unit-SWAP layers in a row, before performing the inter-unit CPHASE operations, instead of having one unit-swap layer and one CPHASE layer interleaved. The unit swap is trivial by applying transversal SWAPs using the vertical links between two units in one step. Those vertical links are CNOT-only links, so one vertical SWAP costs three CNOT gates. This choice could save SWAP costs for inter-unit interactions, which will be discussed later.

Intra and inter-unit QFT: IA+IE. In our updated unit movement, every disjoint pair of units, (U_i, U_{i+1}), where i is an even index, would appear at the top two rows just once. Then, we can apply the 2×N QFT pattern, introduced in [44] to complete mixed intra- and inter- unit operations. For instance, the unit pairs (U_0, U_1), (U_2, U_3), and (U_4, U_5) depicted in Fig. 15 (b), which are enclosed in black boxes, execute the 2×N QFT pattern.

The intuition for 2×N QFT is to make the SWAP gates for intra-unit interactions benefit inter-unit interactions. Due to the gate dependency, it is as if we apply the LNN pattern on both the top and bottom units, but the bottom unit starts one step late. If the bottom unit does not start one step late, each qubit on the top unit always has the same (column) neighbor from the bottom unit, preventing full inter-unit operations. The initial mapping is designed to obey gate dependency. The 2×N QFT pattern contains three repeated steps: inter-unit interaction, intra-unit SWAP, and intra-unit interactions. We show a part of the 2×N pattern in Fig. 16.

The inter-unit interactions (pure IE – red links in figures) follow the QFT-IE-Relaxed strategy. We let each qubit in the top unit interact with every qubit in the bottom unit. The qubit movement within each unit is described in Fig. 13 (a). Again, the bottom unit starts one step late to prevent a qubit from always having the same neighbor qubit in the same column. We can achieve bipartite all-to-all inter-unit interactions with N steps of qubit movement and N steps of qubit interactions (this is developed via our program synthesis approach, details in Appendix of [17]).

Saving SWAP costs for inter-unit interactions. Since there is no gate dependency in the pure IE interactions like Col3 in Fig. 15(b). The compilation for this part is the same as compiling a QAOA circuit with bipartite all-to-all interactions in two units. We can adapt the SWAP saving strategy from [18].

Before our optimization, there is no inter-unit interactions between two consecutive pairs of units (the last unit of the first pair and the first unit of the second pair). For example, there is no red link between two units U_2 and U_0 in the fifth column of Fig. 14. But now, we have a link between unit U_2 and U_0 in the third column of Fig. 15(b). If we apply the same pattern to two pairs of units (U_1, U_2) and (U_0, U_3) in the third column of Fig. 15(b), then, the trajectory of qubits within U_1 is the same as the trajectory of qubits within U_0. Such that, if we can apply CPHASE gate between qubits in (U_1, U_2), we could also apply CPHASE gate between qubits in (U_2, U_0).

One special case of inter-unit interactions is highlighted in green in the fourth column of Fig. 15(b). Those inter-unit interactions involve one unit that is doing mixed intra- and inter-unit interactions. We pause the 2×N pattern on the top two units after or before each SWAP step in Fig. 16, letting the qubit on the physical $U1$ interact with qubits on the physical $U2$. The all-to-all interactions are still guaranteed since the qubit movement in the relaxed ordering is similar to the qubit movement in the 2×N QFT pattern with different start times.

Time Complexity. As for the time complexity, we use unit QFT on a line, as shown in Fig. 15(a). Assuming we have $N = m * m$ qubits. Each unit has m qubits.

QFT-IE-Relaxed In this case, we do not consider strict

ordering in QFT-IE. The time complexity of QFT-IE without QFT-IA is $3m + O(1)$. It happens $m/2 - 1$ times. The time complexity of QFT-IE is $1.5m^2 + O(m)$. Each mixed QFT-IE QFT-IA has complexity $6m + O(1)$. The detail of this time complexity refer to [44]. Due to the relax reordering, we pause mixed (QFT-IA, QFT-IE) for m steps for the inter-unit interactions between the mixed part and the unit below the mixed part. Mixed (QFT-IA, QFT-IE) steps has $7m + O(1)$ depth, and it happens $m/2$ times. For this part, the time complexity is $3.5m^2 + O(1)$. In total, the time complexity is $5m^2 + O(1) = 5N + O(1)$.

VII. Evaluation

We evaluate our QFT mapping approach against other compilation approaches over several hardware architectures, including NISQ architecture and FT architecture, in several aspects. Specifically, we want to ask 2 major questions. **Q1:** How long does it take to find the solution? **Q2:** What is the quality of these generated solutions?

We write a simulator [2] to verify the correctness of our outcome. We measure the quality of the compilation outcome, mainly based on circuit depth and gate count. Due to the noisy feature of quantum gate operations, smaller depth and fewer gate operations mean a lower possibility of being affected by external noise. We present the compilation time in seconds. If it cannot complete in 2 hours, we refer to it as "time-out" in Table I.

Finally, we explore the potential for applying our approach to larger-scale QFT circuits utilizing a fault-tolerant backend. It would be useful for us to exploit regularity in both a small-scale NISQ architecture and a large-scale FT architecture.

Baseline and benchmark selection. As for the baseline, we compare our approach against three approaches: LNN [28], SATMAP [30] and SABRE [22]. SATMAP searches over the whole space and output optimal solutions (with respect to gate count) at the cost of long-running time. SABRE does qubit mapping using a series of heuristics, making it possible to quickly output the potentially suboptimal results. LNN [44] is only tested on Lattice Surgery because we cannot find a hamiltonian path on Sycamore and Heavy-hex architectures. The benchmarks encompass various scales of QFT, each configured differently across a range of NISQ and FT architectures.

Diverse architecture backends. We use three types of architectures: Google Sycamore, heavy-hex, and lattice surgery, among which only lattice surgery is a FT architecture particularly suited for implementing larger QFT kernels, while others are NISQ architectures.

Google Sycamore has m by m configuration, where the total number of qubits is $N = m * m$. We test configurations where m is an even number. For heavy-hex, we unroll it to a line with dangling points [42]. Based on the description in Sec IV, there are $N/5$ groups and each group has 5 qubits. Among all these 5 qubits, 4 are in the main line and 1 sits as a dangling point. Therefore, we only test the heavy-hex architecture whose qubit number is a multiple of 5. For lattice surgery, the total number of qubits is $N = m * m$. Based on our approach in Sec VI, we only consider the case where m is larger than 10.

Architecture	# qubits	Our approach		SATMAP			SABRE		
		Depth	# SWAP	CT(s)	Depth	# SWAP	CT(s)	Depth	# SWAP
m*m Sycamore	2*2	10	6	1.75	10	3	0.28	11	3
m*m Sycamore	4*4	81	116	TLE	N/A	TLE	0.29	102	62
m*m Sycamore	6*6	208	540	TLE	N/A	N/A	0.46	363	484
Heavy-hex	2*5	39	40	439.79	44	37	0.30	43	36
Heavy-hex	4*5	89	160	TLE	N/A	N/A	0.31	134	196
Heavy-hex	6*5	139	360	TLE	N/A	N/A	0.56	229	523
Lattice surgery	10*10	476	2700	TLE	N/A	N/A	0.38	981	2365
Lattice surgery	20*20	1961	41880	TLE	N/A	N/A	8.67	11818	55474
Lattice surgery	30*30	4446	208800	TLE	N/A	N/A	55.26	47161	305807

Table I: Our approach vs. SATMAP and SABRE arcoss different architecture (CT: compilation time, TLE: timeout after 2h).

A. Quality of Compilation Outcome for Small-scale QFT kernel on NISQ Backends

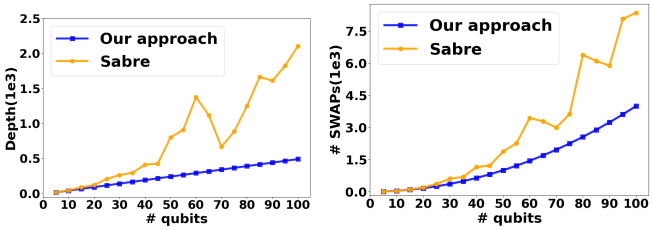
Due to the error rates, only small-scale QFT circuits are deemed suitable for execution on NISQ backends including Sycamore, and Heavy-hex. Thus, in our experiments conducted on these architectures, we assess the quality of compilation outcome across varying numbers of qubits, up to a maximum of 100, by adjusting the value of m.

In this evaluation, we focus on two principal aspects. Firstly, we examine the time each approach takes to output the compilation result, with a preference for faster speed. Secondly, we assess the quality of the outcomes, which includes considerations of circuit depth and the count of gates required to complete the QFT. Achieving the QFT with a smaller depth and a fewer number of SWAP gates is considered advantageous. What needs to be mentioned is that our method does not have compilation time as it is an analytical approach.

1) Compilation time: A portion of the experimental outcomes is presented in Table I. SATMAP, as an optimal solver, delivers favorable results in terms of circuit depth and gate count. However, its search space grows exponentially, leading to prolonged solution times. For example, when setting a time-out limit of 2 hours, SATMAP fails to produce results in most cases when there are more than 10 qubits.

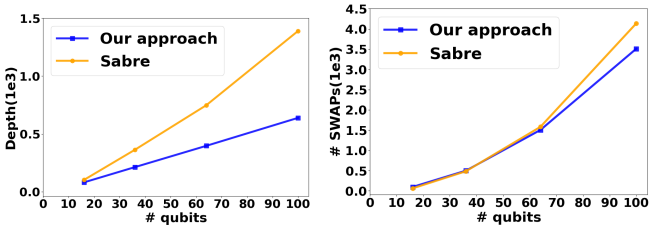
In comparison, SABRE achieves outcomes significantly faster than SATMAP. However, as the number of qubits increases in one configuration, the running time of SABRE increases as well (e.g., 55s for 30*30 lattice surgery).

2) Gate count and Depth: Regarding the number of SWAP gates, the general trend indicates that SABRE requires more SWAP gates or larger depth than ours, except for configurations with a very small number of qubits. Compared with SABRE, our approach have up to 53% fewer SWAP gate count and 92% fewer depth. In the Sycamore backend with up to 100 qubits, our methodology yields a depth cost approximately 50% lower than that of SABRE, alongside a 20% reduction in the number of SWAP gates. For the Heavy-hex backend, our approach significantly reduces the depth cost to just 24% of SABRE's and cuts the number of SWAP gates needed to 48% of those required by SABRE. It's noteworthy that SABRE's performance is not consistently stable; in certain instances, it



(a) Depth for Heavy-hex. (b) # SWAPs for Heavy-hex.

Figure 17: Our approach vs. SABRE for Heavy-hex.



(a) Depth for Sycamore. (b) # SWAPs for Sycamore.

Figure 18: Our approach vs. SABRE for Sycamore.

may result in a smaller gate count and depth for larger QFT sizes, particularly in the heavy-hex architectures.

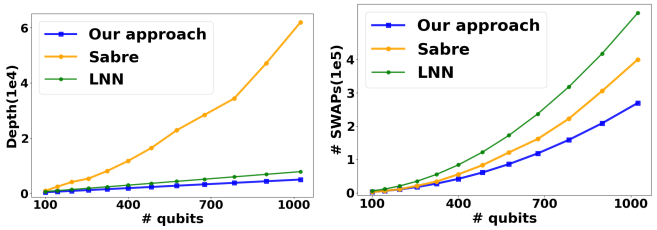
Generally, we discover that our approach produces significantly better outcomes for QFT qubit mapping in terms of circuit depth and gate count for small-scale Sycamore and heavy-hex backends.

B. Generalizing Our Framework to Larger-scale QFT Kernel on FT Backends

FT backends are specifically designed for circuits with a large number of qubits, as only fault-tolerant systems have the capability to effectively manage the high error rates inherent in large-scale circuits. Conversely, NISQ systems are limited to managing error rates in smaller circuits. Therefore, whether our framework can be expanded to large-scale QFT relies on its performance on FT backends. In this part of the evaluation, we evaluate the performance of our approach, SABRE, and SATMAP on lattice surgery architecture. The size of m is from 10 to 32, which leads to the number of qubits from 100 to 1024. We maintain the same aspects considered.

Due to the requirement of error correction, on the lattice surgery architecture, SWAP gate have different latencies on different links. We need to take into consideration these constraints. As SATMAP and SABRE lack the interface to configure such specific connections, we can't impose these constraints on them. Then we compare our approach against the version where all links are used for both baselines. Hence, if our approach can beat those baselines, we can conclude that our approach must be better than theirs.

Due to the large qubit scale, SATMAP's evaluation always encountered a timeout across all QFT sizes, as shown in Table I. SABRE generates the qubit mapping results faster than SATMAP at the cost of suboptimal results. The main reason is that SABRE incorporates several heuristics, some of which are greedy. The generated outcome may achieve even better performance (e.g., # SWAP gate operations) than ours over



(a) Depth for Lattice Surgery. (b) # SWAPs for Lattice Surgery.

Figure 19: Our approach vs. SABRE for Lattice Surgery.

small-scale architectures but as the size increases, its outcome becomes far worse than ours. Besides, the compilation time of SABRE increases proportional to the number of qubits in a particular architecture.

To be precise, as for the depth (Fig. 19), our approach is better than both SABRE and LNN among all sizes of qubits. The advantage becomes more significant as the size increases. Regarding the number of SWAP gates, our approach works better than SABRE when the number of qubits is larger than 144 and the advantage is also growing as the increasing of scale of QFT. Upon a detailed examination, for the scale of QFT comprising up to 1024 qubits, our approach demonstrates a significant advantage, achieving a depth cost that is roughly 92% lower than SABRE's outcome.

These results affirm that, despite the constraints favoring SABRE in the lattice surgery architecture (all links are used), our approach still delivers superior performance.

In conclusion, our framework exhibits enhanced performance in circuit depth and gate count for large-scale QFT circuits on FT backends.

C. Scalability of the outcome

Our approach relies on the systematic approach to complete the qubit mapping for QFT over diverse backends. In contrast, this simplicity and scalability are not observed with methods like SABRE and SATMAP.

SABRE, employs a look-ahead strategy to insert SWAPs, aiming to optimize not just for the immediate layer but future layers as well. It is difficult for us to find any common patterns that can be reused for a larger grid size. The randomness of SABRE's output is in Fig. 27 in Jin *et al*. [17]. Getting output from SATMAP has already been challenging due to its long compilation time. Hence it is not a viable solution for scalable architectures.

VIII. Related Work

Many studies focus on qubit mapping for a general class of applications [22], [25], [26], [30], [32], [36], [39], [40], [43], [44]. A general-purpose compiler takes an arbitrary program and an arbitrary architecture as input, and produces a compiled circuit for this architecture. The issue with this approach is that every time the program size changes, for instance, the qubit number changes, the program needs to be recompiled. We focus on domain-specific qubit mapping and do not require the compiler to recompile the program when the input size changes. Our approach produces a linear-depth QFT circuit for both NISQ and FT backends.

There are also other domain-specific compilers for various applications including quantum approximate optimization algorithms (QAOA) [3], [4], [18], [21], variational quantum eigensolvers (VQE) [19], [23], [24], and etc. For QFT, Maslov *et al*. [28] for the first time shows a linear time solution on the linear nearest neighbor (LNN) architecture. However, it is difficult to find a Hamiltonian path that connects all nodes in modern quantum architectures, limiting the applicability of this approach. Zhang *et al*. [44] improved upon Maslov's *et al*. [28] by discovering a linear-depth solution for a 2D grid with only two rows. However, 2xN grid architecture does not exist in modern architectures. Gao *et al*. [11] proposed a similar approach to do qubit mapping for QFT over the IBM Heavy-hex NISQ devices.

Leveraging program synthesis tools [38] to do the compiler design for domain-specific applications [6] [37] [13] [12] has already existed. However, to the best of our knowledge, our work is the first that demonstrates the usefulness of program synthesis for the compiler design domain of quantum computing, for the QFT kernel circuits.

IX. Conclusion

We propose a new QFT compilation framework for quantum application kernel over diverse quantum backends. Our approach outperforms the state-of-the-art approaches with less circuit depth and fewer gate count usage.

X. Acknowledgements

We extend our gratitude to the anonymous reviewers for their constructive and insightful feedback. This work was supported by grants from the Rutgers Research Council and NSF-FET-2129872. The opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Hamiltonian path problem. https://en.wikipedia.org/wiki/Hamiltonian_path_problem.

[2] Simulator to verify the QFT qubit mapping output. https://github.com/XiangyuG/qft_on_regular_architectures.

[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. *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, 2020.

[5] Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. Surface code compilation via edge-disjoint paths. *PRX Quantum*, 3:020342, May 2022.

[6] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[7] Francesco Bova, Avi Goldfarb, and Roger G Melko. Commercial applications of quantum computing. *EPJ quantum technology*, 8(1):2, 2021.

[8] Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM, June 2003.

[9] D. Coppersmith. An approximate fourier transform useful in quantum factoring, 2002.

[10] Mark Ettinger and Peter Hoyer. A quantum observable for the graph isomorphism problem, 1999.

[11] Xiangyu Gao, Yuwei Jin, Minghao Guo, Henry Chen, and Eddy Z. Zhang. Linear depth qft over ibm heavy-hex architecture, 2024.

[12] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. SIGCOMM '20, page 44–61, New York, NY, USA, 2020.

[13] Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Cat: A solver-aided compiler for packet-processing pipelines. In *ACM ASPLOS*, page 72–88, New York, NY, USA, 2023.

[14] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15), October 2009.

[15] Clare Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14:123011, 12 2012.

[16] Fei Hua, Yanhao Chen, Yuwei Jin, Chi Zhang, Ari Hayes, Youtao Zhang, and Eddy Z Zhang. Autobraid: A framework for enabling efficient surface code communication in quantum computing. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 925–936, 2021.

[17] Yuwei Jin, Xiangyu Gao, Minghao Guo, Henry Chen, Fei Hua, Chi Zhang, and Eddy Z. Zhang. Optimizing quantum fourier transformation (qft) kernels for modern nisq and ft architectures, 2024.

[18] Yuwei Jin, Fei Hua, Yanhao Chen, Ari Hayes, Chi Zhang, and Eddy Z. Zhang. Exploiting the regular structure of modern quantum architectures for compiling and optimizing programs with permutable operators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 108–124, New York, NY, USA, 2024. Association for Computing Machinery.

[19] Yuwei Jin, Zirui Li, Fei Hua, Yanhao Chen, Henry Chen, Yipeng Huang, and Eddy Z. Zhang. Tetris: A compilation framework for vqe applications, 2023.

[20] A. Yu. Kitaev. Quantum measurements and the abelian stabilizer problem, 1995.

[21] 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, page 351–365, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1014, 2019.

[23] Gushu Li, Yunong Shi, and Ali Javadi-Abhari. Software-hardware co-optimization for computational chemistry on superconducting quantum processors. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 832–845. IEEE Press, 2021.

[24] 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. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 554–569, 2022.

[25] Ji Liu, Peiyi Li, and Huiyang Zhou. Not all swaps have the same cost: A case for optimization-aware qubit routing, 2022.

[26] Ji Liu, Ed Younis, Mathias Weiden, Paul Hovland, John Kubiatowicz, and Costin Iancu. Tackling the qubit mapping problem with permutation-aware synthesis, 2023.

[27] Austin G Fowler Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *PHYSICAL REVIEW A covering atomic, molecular, and optical physics and quantum information*, 2012.

[28] Dmitri Maslov. Linear depth stabilizer and quantum fourier transformation circuits with no auxiliary qubits in finite-neighbor quantum architectures. *Physical Review A*, 76(5), Nov 2007.

[29] Dmitri Maslov. Advantages of using relative-phase toffoli gates with an application to multiple control toffoli optimization. *Phys. Rev. A*, 93:022311, Feb 2016.

[30] A. Molavi, A. Xu, M. Diges, L. Pick, S. Tannu, and A. Albarghouthi. Qubit mapping and routing via maxsat. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1078–1091, Los Alamitos, CA, USA, oct 2022. IEEE Computer Society.

[31] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[32] Siyuan Niu, Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. A hardware-aware heuristic for the qubit mapping problem in the nisq era. *IEEE Transactions on Quantum Engineering*, 1:1–14, 2020.

[33] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man Hong Yung, Xiao Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5, 2014.

[34] Patrick Rebentrost, Brajesh Gupt, and Thomas R. Bromley. Quantum computational finance: Monte carlo pricing of financial derivatives. *Phys. Rev. A*, 98:022321, Aug 2018.

[35] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, 1994.

[36] Marcos Yukio Siraichi, Vinicius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. Qubit allocation. *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 113–125, 2018.

[37] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, page 15–28, New York, NY, USA, 2016.

[38] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

[39] Bochen Tan, Dolev Bluvstein, Mikhail D. Lukin, and Jason Cong. Qubit mapping for reconfigurable atom arrays. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Bochen Tan and Jason Cong. Optimal layout synthesis for quantum computing. *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020.

[41] J.-C. Walter and G.T. Barkema. An introduction to monte carlo methods. *Physica A: Statistical Mechanics and its Applications*, 418:78–87, January 2015.

[42] 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.

[43] Chi Zhang, Yanhao Chen, Yuwei Jin, Wonsun Ahn, Youtao Zhang, and Eddy Z Zhang. A depth-aware swap insertion scheme for the qubit mapping problem. *arXiv preprint arXiv:2002.07289*, 2020.

[44] Chi Zhang, Ari B Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z Zhang. Time-optimal qubit mapping. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 360–374, 2021.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

This paper has the following 2 main contributions.

- C_1 A novel and flexible QFT sub-kernel partition that allows hierarchical decompostion of high-dimensional problems to low-dimensional problems.
- C_2 Guaranteed compilation quality: linear-depth solutions for the QFT kernel on different architectures: Google Sycamore, IBM Heavy-hex, and a fault-tolerant (FT) backend with lattice surgery model. Our hardware QFT kernel mapping solutions have up to 53% fewer SWAP gate count and 92% less depth than state-of-the-art approaches for up to 1024 qubits compared with SABRE.

B. Computational Artifacts

All computation artifacts are archived under one single open-source github repository (https://github.com/XiangyuG/qft_on_regular_architectures). We list the detailed mapping between artifcat ID and contributions in Table I-B.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Verify the gate execution (No figure or table)
A_2	C_2	Table 1, Figure 17-19

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This computational artifact is used to check the correctness of the contribution C_1.

The contribution C_1 is a qualitative study and analysis of the QFT circuit. It breaks a QFT into sub-kernels. The circuit compilation is divided into two stages: compilation between two kernels and within a kernel. Instead of compiling the circuit by following the original order, this breakdown changes the gate dependency without breaking the circuit semantics. To guarantee the correctness of this breakdown, we use artifact A_1.

Expected Results

The compiled QFT with kernel breakdown should pass the correctness check.

Expected Reproduction Time (in Minutes)

The expected computational time of this artifact is constant. There is no specific platform requirement.

Artifact Setup (incl. Inputs)

Hardware: There is no specific platform requirement.

Software: Python 3 and some standard libraries, such as networkX.

Datasets / Inputs: Compiled QFT circuit from our compilers.

Installation and Deployment: There is no Installation and Deployment.

Artifact Execution

Correctness check is integrated in the compilation process. The compiler does the checking automatically. Specifically, we use the function *check_qft_gates* in *util.py* to check whether all gates in QFT have been executed or not, and the function *OK_to_do_control* to check the correctness of gate dependency after circuit partition.

Artifact Analysis (incl. Outputs)

Our verifier passes all dependency and gate execution checks, meaning that our proposed approach is correct under all 3 state-of-the-art quantum architecture backends.

B. Computational Artifact A_2

Relation To Contributions

In this computational artifact, we provided a compiler for 3 different backends: Google Sycamore, IBM Heavy-hex, and a FT back-end with lattice surgery model. By default, the number of qubit of the logical QFT circuit is the same as the number of physical qubits in the given architecture. Once the number of physical qubits is specified, the compiler would automatically compiles the circuit with respect to the corresponding backend. Since the structure of backend and QFT circuit are known, their information are encoded in the compilation process.

Expected Results

The compiled QFT circuit has a linear-bounded circuit depth in terms of the number of qubits and gate count in terms of the number of CPHASE gates for all 3 state-of-the-art backends.

Expected Reproduction Time (in Minutes)

In our approach, the expected computational time is nearly constant (less than 1 second). Because the circuit is not compiled gate by gate. The whole circuit compilation follows a pre-defined pattern. However, the compilation time of 2 baselines (SATMAP and SABRE) increases as the number of qubit increases.

Artifact Setup (incl. Inputs)

Hardware: There is no specific platform requirement. We perform the experiment on a regular personal computer with the setting to be Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz, with 20 CPU(s) reaching up to 5200 MHz, and 30 GiB of memory. The operating system is Ubuntu 22.04.3 LTS.

Software: We use Qiskit (https://www.ibm.com/quantum/qiskit). Only basic Python 3 is sufficient (necessary python packages are list in README file to install beforehand) SKETCH program synthesis solver (https://people.csail.mit.edu/asolar/sketch-1.7.6.tar.gz).

Datasets / Inputs:

- Heavy-hex backend compiler takes the number of physical qubits as the input, automatically generating the hardware coupling graph and the compiled QFT circuit.
- Google sycamore backend compiler takes the number of rows and columns as the input, then automatically generates the hardware coupling graph and the compiled QFT circuit.
- FT backend has a grid structure. It also takes the number of rows and columns as the input, then automatically generates the hardware coupling graph and the compiled QFT circuit.

Installation and Deployment: There is no Installation and Deployment requirement.

Artifact Execution

We use the following script to verify all 3 backends.

```
python3 Googlesycamore_qft.py <the value of m for m*m grid>
```

```
python3 heavy_hex_qft.py <# qubits in the main row>
```

```
python3 lattice_sugery_qft_mix.py < an even value of #units >
```

Artifact Analysis (incl. Outputs)

We run the compiler multiple times, each time with different backends and different sizes. The output of the compiler includes the number of gate count and compiled circuit depth. Then we compare those data with the output of baselines in Table 1 and Figure 17-19.

Artifact Evaluation (AE)

A. Computational Artifact A_1

We only provide AE for computational artifact A2 mentioned in artifact description. Because A1 is only used to check the correctness of compiled results, and it is already included in our compilers. If the compiler finishes the compilation, the compiled circuits meet the requirements.

B. Computational Artifact A_2

Artifact Setup (incl. Inputs)

Please download the source code from the Zenodo:

```
https://doi.org/10.5281/zenodo.12594486
```

If the experiment is conducted in remote terminal, please use "ssh -Y" to receive the figures from the remote terminal.

Firstly, we recommend reviewer create a python virtual environment with **version 3.9**. Then install necessary libraries listed below:

```
pip install qiskit==0.43.1
pip install pandas
pip install matplotlib
```

Artifact Execution

The experiment workflow for our compilers is quite simple. To get the compiled program data in three different backends, just run following three commands:

```
python3 our_lattice_surgery.py
```

```
python3 our_heavy-hex.py
```

```
python3 our_sycamore.py
```

Results will be saved in folder csv_data.

To get the data from baseline, you need to **change the path to the folder sabre** and run code below for three different backends.

```
cd sabre
python3 sabre_qft.py 'N*N'
python3 sabre_qft.py 'sycamore'
python3 sabre_qft.py 'heavy-hex'
```

Please do not miss quotation marks and results will be saved in folder csv_data as well.

Artifact Analysis (incl. Outputs)

To show the figures in paper, you can implement the following code:

```
cd ..
python3 draw_figures.py
```

- The output data will be saved as a csv file under folder csv_data, including the compilation time, compiled circuit depth and swap gates count. Those are three metrics we only used in our paper. Both SWAP gate count and depth lower are better.
- There is no need to take the QFT circuit as the input since it is known once the number of qubits is specified and encoded during the compilation. The size of the QFT circuit is hard coded in a loop.
- As we stated we provided a linear-depth solution for the QFT kernel on different architectures. Our compilation result, depth grows linearly with the number of qubits.
- You will see six figures by implementing "draw_figures.py", generated in the order as they appear in the paper. For example, the first figure you will see is corresponding to Fig. 17(a) in our paper and the last figure is corresponding to Fig. 19(b).
- The results of baseline may vary due to the random seed in the program, draw in orange line. The last two figures are missing a green line, from a baseline that are not mainly compared in this paper. But they can be added simply using equation $8 * (m^2 - m - 1)$ for depth and $0.5 * (m^4 + m^3 - m^2 - m)$ for the swap count. m is the size of one column or row in the lattice surgery backend and $m * m$ is the #qubit shown in the x-axis.
- For the compilation time concern, the last two figures did not show the result of two largest cases (compared with the figure in our submitted version), where #qubit = 28 and 32. We can already see the improvement and trend with #qubit = [4,28).
- We did not include Table 1 in the AE because all the data in the table are already reflected in the six figures.