Making Qiskit Chiplet Aware

Peizhi Liu
Northwestern University
peizhiliu2023@u.northwestern.edu

Ruiqi Xu
Northwestern University
jerryxu2023@u.northwestern.edu

Kinsey Ho
Northwestern University
kinseyho2023@u.northwestern.edu

ABSTRACT

While quantum computing is a rapidly growing field, building quantum computers with a large number of high-quality qubits is still a challenging task. To improve qubit yield, recent work has demonstrated an interesting possibility, applying the chiplet design used in traditional computing to the quantum field [10]. However, the existing framework of quantum development tools has not supported this idea. In this paper, we explore ways to modify Qiskit, one of the most popular open-source tools, to enable chiplet transpilation with minimal code change.

CCS CONCEPTS

• **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → *Compilers*; • **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

Qiskit, chiplet, transpilation, hardware representation, quantum circuit optimizations

1 INTRODUCTION

While chiplet has been a popular topic in traditional architecture, with companies like Apple, Intel, and AMD releasing products that utilize this design approach, it is still relatively new in the concept of quantum computing. However, that does not mean quantum computers cannot benefit from it. While traditional architectures started encountering issues with chip yields in recent years, quantum computers have always suffered from poor scalability. The current state-of-the-art quantum computer is called Osprey and is built by IBM [1]. While it contains 433 qubits, it still does not come near to the requirement of fault-tolerant quantum computing, which can take millions of high-quality physical qubits due to the use of error correcting codes [7]. In order to improve the capability of quantum computers, engineers are actively looking for designs that support more qubits without sacrificing accuracy. A recent study suggests that chiplet-based designs might be a potential solution. They demonstrate that chiplet architectures can improve yield by anywhere between 9.6-92.6x [10], with smaller chiplets leading to better yields. The detailed improvements are shown in Figure 1. Therefore, in this work, we will look into the software support for future chiplet-based quantum computers, hoping to come up with a representation of such systems and build a transpilation tool on top of Qiskit [9]. While it is possible to run analyses on the entire circuit in Qiskit, we make the assumption that future circuits will contain thousands of qubits, meaning that the existing framework will struggle to produce the quantum assembly in a reasonable amount of time since the optimizations might not scale well with circuit size. We further claim that in cases where we are forced to turn off certain optimizations, considering those sub-circuits individually and having locally optimized results could be more desirable. Treating each chiplet as a separate target also offers an intuitive way to parallelize the computation, speeding up the analyses.

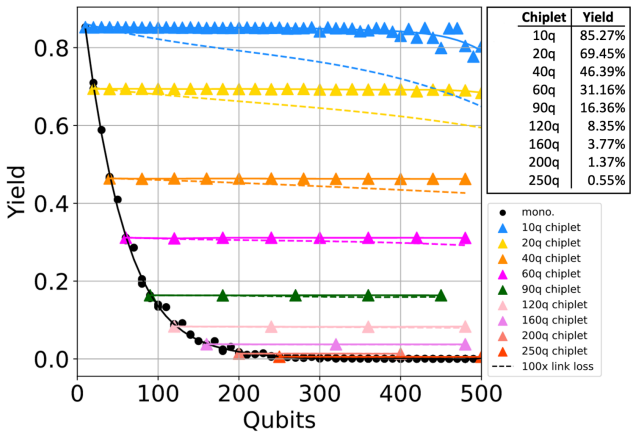

Figure 1: Chiplet Yield vs. Qubits from [10].

We pick Qiskit because it is already a commonly used and open-source platform with many features and support for a wide variety of backends. It provides us with a simple interface to build circuits, modify target machines, and test our approaches. Our target is to perform the task with minimal changes to the existing framework as the codebase is large, making it hard to fully understand the implications of our changes. Thus, we will treat the transpiler as a black box and reason about its operations based on what leads to better circuits.

2 PROBLEM SETUP

We set up the problem as follows. We are given a set of pre-cut quantum subcircuits $\{C_0, C_1, ..., C_n\}$. The sub-circuits have virtual qubits and quantum gates that we want to map onto a fixed topology of chiplets and physical qubits within each chiplet respectively. There will also be connectivity between chiplets that can perform gate operations.

We believe that the problem can be separated into two steps. In the first step, we need to map the overall circuit onto a set of chiplets based on the existing and required connectivity. The second step requires us to work on each sub-circuit, mapping the virtual qubits within the subcircuits to physical qubits on the target chiplets.

3 CIRCUIT TO CHIPLET MAPPING

The first transpilation step in mapping the overall circuit onto individual chiplets as seen in Figure 2, it can be done by constructing two graphs $G = (V, E)$ and $G' = (V', E')$. G can be thought of as

the computation graph for the subcircuit where each vertex is a subcircuit $V = \{C_0, C_1, ..., C_n\}$ and there is an edge $(C_i, C_j) \in E$ if there exists an operation between qubits of the two subcircuits (e.g. a swap gate). On the other hand, G' a graph representation of the chiplet topology where V' represents individual chiplets and E' represents connections between chiplets. The problem of circuit mapping hence reduces down to the task of finding a subgraph $G_0 = (V_0, E_0)$ such that $V_0 \subseteq V', E_0 \subseteq E' \cap (V_0 \times V_0)$ where $G \cong G_0$. This is the well-known sub-graph isomorphism problem seen in past literature, shown to be NP-hard [2].

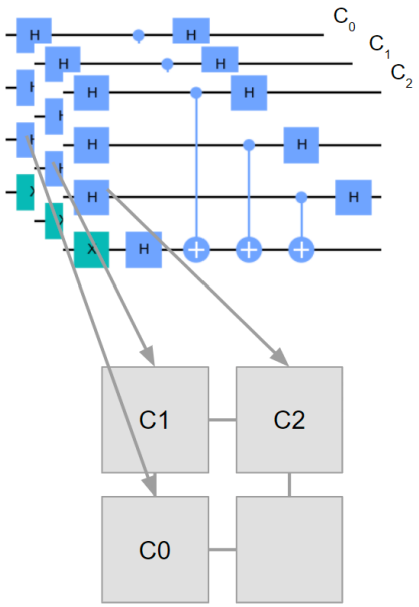

Figure 2: Chiplet Mapping

There are, however, several caveats to our particular circuit-mapping problem that make it, potentially, more tractable. First, the degree of vertices in both G and G' are likely bounded due to physical constraints on the connectivity of the chiplets [4]. Second, the number of chiplets in near-future chipset-based machines is small with only tens to hundreds of chiplets. Thus, even with a super-polynomial time algorithm, it may be feasible to perform the optimal mapping. Finally, there are various inexact approaches that are aimed at solving the chiplet mapping problem at larger scales.

Although we are not focusing on solving the subgraph isomorphism, we still find a few useful works that might help solve this problem in future works. *Deep Analysis on Subgraph Isomorphism* implements seven representative algorithms using C++ and compares their strengths and weaknesses [6]. Grand isomorphisms is a subgraph isomorphism solver implemented purely using Python. It has an intuitive interface and might be easier to integrate with Qiskit [11]. However, these works all assume that the nodes are uniform. In the chiplet design, we might have chiplets of various sizes, which imposes additional constraints on the solution because some sub-circuits cannot be mapped to all the chiplets. Another potentially interesting approach to efficient chiplet mapping is to utilize quantum algorithms for subgraph isomorphism [5], which requires only 50 qubits to encode graphs with the number of vertices on the order of 10^6.

4 SUBCIRCUIT TO QUBIT MAPPING

For the second step of the transpilation, which we mainly focused on for this project, the transpiler should map virtual qubits within the subcircuit to physical qubits on the target chiplet. To perform this transpilation, we introduce a technique of converting the actual chiplet topology into a physical qubit augmented with fixed ancilla qubits representation on Qiskit. Specifically, we propose a concept called the *Halo Region*. For every chiplet, it is formed by the connected qubits in other chiplets. To simplify the problem, we only treat the first-degree connections as candidates. We might need to include additional layers depending on if the circuit-cutting step requires swaps with chiplets beyond the first-degree ones. The *Halo Chiplets* of the red chiplet is shown in orange in Figure 3.

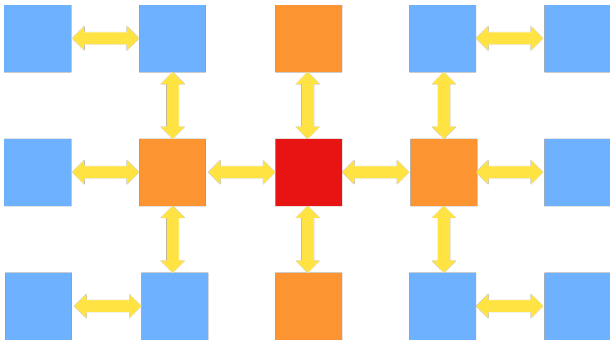

Figure 3: Halo Chiplets

For the connected qubits in those Halo Chiplets, we add them to the target circuit, connecting them based on the chiplet-level connections. In Figure 4, we assume that the qubit in the orange chiplet is connected to qubit 6 in the green chiplet, the qubit in the blue chiplet is connected to qubit 7, and the qubit in the red chiplet is connected to qubit 8. The Qiskit backend allows us to specify the operations that can be performed on these connections as well as the corresponding error and latency. In this case, we limit the set of allowed operations to swap only. We will discuss how to set those parameters in later sections.

Adding these qubits to the circuit allows us to treat the interactions between chiplets as swap gates. Whenever we need to send or receive quantum states, we simply perform a swap between the qubit with the interconnect and the corresponding halo qubit. This offers an intuitive representation of the circuit behaviors. However, this design also brings a challenge: because Qiskit tries to perform optimizations on the circuit, altering the mapping between physical and virtual qubits to reduce error and latency, we need a way to ensure that the halo qubits and the ones connected to them are mapped to the desired physical qubits. In some cases, we find that because interconnects tend to have higher error rates than other swap gates, the swap between the halo and regular qubits gets

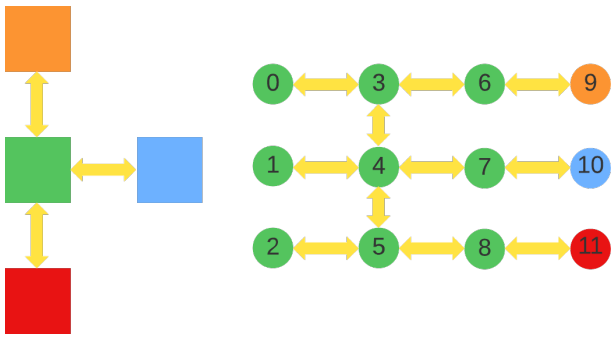

Figure 4: Halo Qubits

mapped to the regular qubits, meaning that neither qubits will not receive the desired data. We will discuss some potential solutions in the section following this.

5 METHODS

5.1 Using Initial_layout to Fix Location of Halo Qubits

We observe that the transpiler provided by Qiskit allows users to specify the initial mapping of qubits. If the circuit can be transpiled without breaking this assignment, Qiskit will respect the mapping [8]. For example, we can force virtual qubits to map to the physical qubit with the same index by passing it the array $[0, 1, ..., n-1]$, where n is the number of qubits. Alternatively, you can pass this mapping using a dictionary. However, one problem is that this approach does not support fixing a subset of all the qubits. The dictionaries and arrays have to have the same number of elements as the input circuit. Therefore, if we enforce an initial mapping, we potentially give up some optimization opportunities. Overall, although the solution provided by this approach is suboptimal, we believe that it is still useful as it allows us to retain the true parameters of the interconnect, meaning that it could provide some insights into the magnitude of circuit error.

5.2 Using Custom Gates

The motivation of this approach is that we observe not all the gates have to be available on all the qubits, meaning that the types of gates present can be used to manipulate and limit the types of circuits generated. It is possible to add three gates that are only available on 9, 10, and 11, and then apply these gates at the beginning of the circuit to force the bounding, solving the permutation problem described earlier.

First, we tried making the custom gate a combination of regular gates, known as the gate's *equivalence*. However, the problem with that is we cannot guarantee that the transpiler will perform the computation using the decomposed gates on regular qubits and pass the result to the halo qubits. Furthermore, it is tricky to pick the right level of error and delay. If the cost of that operation on the halo qubits is too low, some valid computations on the regular qubits might be assigned to the halo qubits since they are connected by swaps. On the other hand, if the cost is too high, the transpiler will break down the gate, run it on regular qubits, and transfer the results back to the halo qubits. In either case, we do not get the guarantee that the halo qubits will be mapped correctly. Even worse, it can break the correctness of the circuit. Although detecting this problem is easy: we can simply check if the custom gate is applied in places other than the beginning, recovering from the error is hard.

Then, we experiment with custom basis gates. By definition, basis gates cannot be decomposed. Intuitively, if we define the backend of the machine to only allow custom basis gate operations on certain qubits and not others, and we try to map custom operations on gates that do not support it, the transpiler not be able to perform the mapping. Furthermore, since the mapping of virtual to physical qubits remains static throughout the entire execution of the circuit, it is possible to statically map specific virtual qubits onto physical qubits by using these custom gates by issuing them at the beginning of the computation and removing them after transpilation. Figure 5 shows the custom gate we define called *water-gate* (courtesy of Mr. Nixon). It is only available on physical qubit 8. Thus, by applying it on virtual qubit 8, we ensure that the mapping of that qubit is fixed.

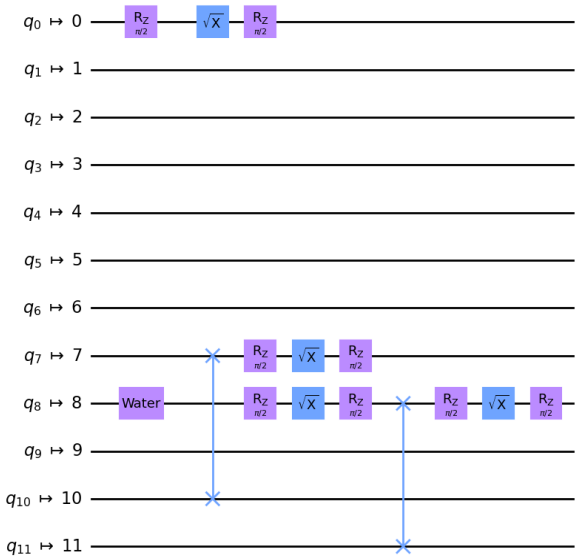

Figure 5: Custom Gate *Water*.

Alternatively, we might be able to perform one pass, find the basis gates that are not used by other qubits, assign one to each of the halo qubits, apply these gates at the beginning, and then run the circuits with halo qubits that support swap and the assigned basis gate. Although it would, in theory, save us the trouble of adding custom gates, in practice, we find that this approach has similar problems to using decomposable gates. It is hard to get the error rates correct so that the basis gates do not get mapped onto other qubits and get swapped over or the opposite. Furthermore, it requires two passes with modifications to the backend file after the first pass, significantly complicating the process. Additionally, we cannot guarantee that there will be free basis gates.

In short, we find that adding custom basis gates to the halo qubits indeed helps us fix the mapping issue. However, it does require more changes to the Qiskit framework.

5.3 Using Zero-error Swaps and Two Pass

Based on the insights gathered from the previous attempts, we believe that this problem can be broken down into two parts. First, we need to make sure that the halo qubits are not mapped onto the regular qubits, meaning that all the inter-chiplet communications have to happen on the swap gates between the halo qubits and regular qubits. Second, we need to make sure that the pairs do not get mapped in permuted orders. Based on our experiments, we find that Qiskit might permute pairs of qubits. For example, the virtual qubit pair 7 and 10 might get mapped to physical qubits 8 and 11, whereas the virtual qubit pair 8 and 11 is mapped to physical qubits 7 and 10. Although this seems to be harmless, the data that gets swapped over will be different as the halo qubits are connected to different chiplets.

For the first problem, the solution we propose is that we can set the error and latency of the swap gate to zero or a low value. That means the transpiler will see the halo qubits as ideal candidates for swaps and be encouraged to map the halo qubits there. Even if the chiplet has unused qubits, it will try to use the halo qubits to perform the swaps. At the same time, we do not need to worry about the regular swaps being assigned to the halo region because the regular qubits must perform operations other than swaps, making the halo qubits unsuitable.

The second problem is more tricky. We do not want to revert back to the initial_layout approach mentioned earlier. After looking into this, we found that we can bypass the limitation of Qiskit by performing two passes [3]. For the first pass, we allow Qiskit to run with the backend we made with no restrictions, meaning that it can perform all the optimizations. Then on the second pass, we apply an initial layout that swaps the permuted pairs, treating the physical qubits from the first pass as virtual qubits in the second pass. Although this might lead to worse error rates, the operation is necessary for preserving correctness. The proposed operations are demonstrated in Figure 6. The image on the left represents the case where the output is permuted. After applying initial_layout [0, 1, 2, 3, 4, 5, 6, 8, 7, 9, 11, 10], we get the circuit on the right.

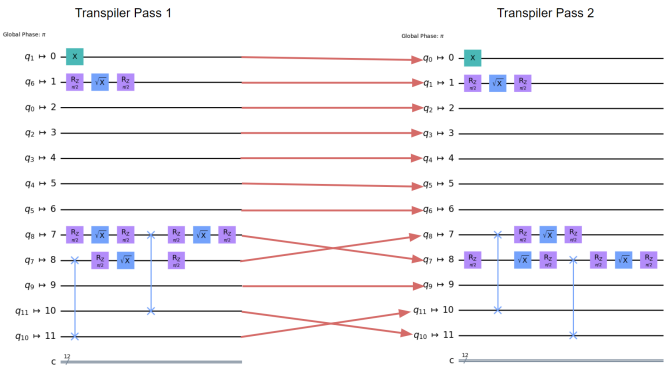

Figure 6: Fixing Permutation using initial_layout.

6 IMPLEMENTATION

6.1 Custom Backend Generator

Although the backend can be configured using a custom class, we can also define it using two JSON files and pass it to the provided parser. The first file we need to change is the backend props. For the "qubits" field in that file, we need to add one entry for each of the halo qubits, which contains fields like the T1 time, T2 time, frequency, anharmonicity, readout error, and readout length. The current implementation of the props writer simply copies the first entry and replicates it n times, where n is the number of halo qubits. For the "gates" field, we need to add all the necessary swap gates. Each gate entry includes the qubits involved, the type of gate, the gate error and length, and a name for that gate. The generator will set the error to zero and the gate length to the delay of the first gate. Note that the connection is directional. That means the qubits entry is ordered. The generator will automatically insert both orders. The gate name is the word "swap" followed by the indices of the first qubit and second qubit.

We also need to modify the backend conf file. The "n_qubits" field will be incremented by the number of halo qubits. The added connections need to be reflected in the coupling map of the swap gate. If swap gates do not exist, the Python writer will insert an entry with the name (swap in this case), parameters, quantum assembly, and coupling map. Additionally, we need to update the global coupling map with the new swap gates.

The inputs to both Python writers are the connections we want to add and the file name and the name of the original backend file. The connections are represented as an array of two-element arrays. The order does not matter in this case. We recommend basing the design on an existing backend since it provides some baseline for the error and latency values.

6.2 Generating and Applying the Chiplet Backend

In the "fake_chiplet.py" file on our repository, we demonstrate how to build the backend based on the JSON files. The backend takes the form of a subclass, with FakeBackendV2 from the fake_backend library being its parent class. The conf and props files can be passed in as member variables. The dirname variable allows you to point to the folder containing those files.

6.3 Building Custom Gates

In the example we provide, we present how to add a custom 1-qubit gate to the circuit. We base our design on the identity gate, which does not impact the result. The following steps detail how custom gates may be added with the current directory being `qiskit/`.

(1) Create a new custom gate file under `circuit/library/standard_gates/<gate_name>.py`
(2) Import the custom gate in `circuit/library/standard_gates/__init__.py` and add it to the list of standard gate name mappings
(3) Add custom gate class method to the QuantumCircuit class in `circuit/library/standard_gates/__init__.py`

(4) Modify the backend as a new basis gate with corresponding error models

Note that it is important to only use the custom basis gate operations on support qubits. Otherwise, the transpiler will fail.

7 ADDITIONAL COMPLEXITIES

There are additional complexities to the two circuit-to-qubit mapping processes to consider besides those already mentioned.

7.1 Dynamic Subcircuit to Chiplet Mapping

There may be, in fact, cases where the number of subcircuits to be mapped onto the chiplet is greater than the number of physical chiplets available. This is due to the varying durations of different subcircuits. Given N subcircuits, it might be possible to map the subcircuits onto M chiplets where $M < N$ if at any point in time t, the number of *active* subcircuits are n_t satisfies $n_t \leq M$. If we define the start time of a subcircuit i to be s_i and the end time of a subcircuit to be e_i, the subcircuit is *active* at time t if $t \in [s_i, e_i)$. The number of active subcircuits at any point in time t is therefore defined to be

$$n_t = \sum_{i=0}^{N-1} \mathbb{1}_{t \in [s_i, e_i)}. \tag{1}$$

First, to determine if there even is a possible mapping, we can reformulate this question as finding the chromatic number C (minimum coloring) of the computation graph. If the chromatic number is greater than the number of chiplets M, then it is guaranteed that these subcircuits cannot be mapped; otherwise, it is possible to map the circuit, albeit perhaps with delays (to allow subcircuit to finish before performing the swaps). For now, let us assume that if $C \leq M$, the subcircuits can be mapped without further adjustments to timing.

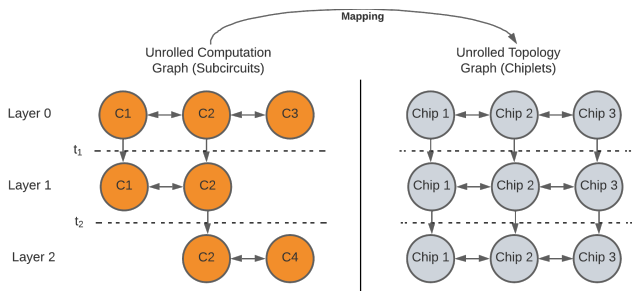

Figure 7: Temporal unrolling for dynamic subcircuit to chiplet mapping.

As for the actual subcircuit mapping, a first attempt is to address this problem by unrolling both the computation and topology graphs temporally (as a separate dimension) and performing a similar subgraph mapping as before. Figure 7 gives an example of how this can be done when $N = 4$ on a 3-chiplet linear topology. Unlike before, both the computation graph and the chiplet topology graph are unrolled temporally. Time slices are indicated by a change in the set of active subcircuit (either a new subcircuit is active or an existing subcircuit is no longer active). For example, in the figure, at t_1, subcircuit C_3 is no longer active, so a *layer* is created with directed edges connecting the still active subcircuits. Similarly, at t_2, subcircuit C_1 is no longer active but C_4 is now active. We see similar layers of nodes being replicated in the chiplet topology graph. Within each layer, for example, layer i, of the computation graph, there are connections (here we always assume bidirectional connections for correctness) between subcircuit nodes when there is an interaction between the subcircuits within time t_i and t_{i+1}. With this reformulation, we can then find a mapping between the computation graph G onto the topology graph G' to determine subcircuit to chiplet mapping.

Note that directed edges are needed between layers since temporally unrolling the graphs and using undirected edges will lead to the loss of temporal information, which can result in a temporal mapping of what was intended to be a spatial mapping onto the chiplet topology.

7.2 Aligning the Swaps

Since we are compiling each chiplet individually, we need to make sure that the corresponding swaps are aligned so that the chiplet that reaches the communication point early does not eagerly perform the swap. We propose two different solutions for that.

The first solution performs an additional software pass after the circuits are compiled. The approach is similar to the scanline algorithm. We push the swapping points onto a stack. Each swap operation has two values (one from each qubit). We then examine the elements one by one in the order of increasing time. If the pair of operations happen to be at the same time step, we remove them from the stack and move on. If we see one of the swaps happening first, we would have to delay the first swap so that it aligns with the second swap. Then, we need to update the time stamps of everything following the swap on the first qubit and put them back into the stack in sorted order. For delaying the operations, we can simply insert idle gates on all the qubits. We assume that forcing other qubits that do not depend on the swap qubit to stall does not lead to undesirable consequences. Otherwise, the complexity of the algorithm would increase significantly. This algorithm is illustrated in Figure 8. The operations on chiplet 0 has to be delayed for the swaps to align. Note that, in this case, the delay means that the second pair of swaps become unaligned, and that has to be fixed when the scanline arrives.

There is also a more straightforward hardware solution. We can perform a handshake at the swapping point. If one of the two qubits is not ready, we simply put it on hold until the other qubit reaches this execution point. In that way, the delay naturally propagates through the circuit without requiring other interventions. Furthermore, we can add a waiting register for each qubit. If the qubit is stalled due to another one, we store the index of that qubit. This allows all the independent qubits to run to completion more quickly. When the stalled qubit is able to continue making progress, we simply wake up all the qubits waiting for it and check if they can proceed.

7.3 Circuit Scoring

As we mentioned before, there are three possible methods to make the circuit follow all the necessary requirements. However, these three approaches might lead to different results. To figure out which

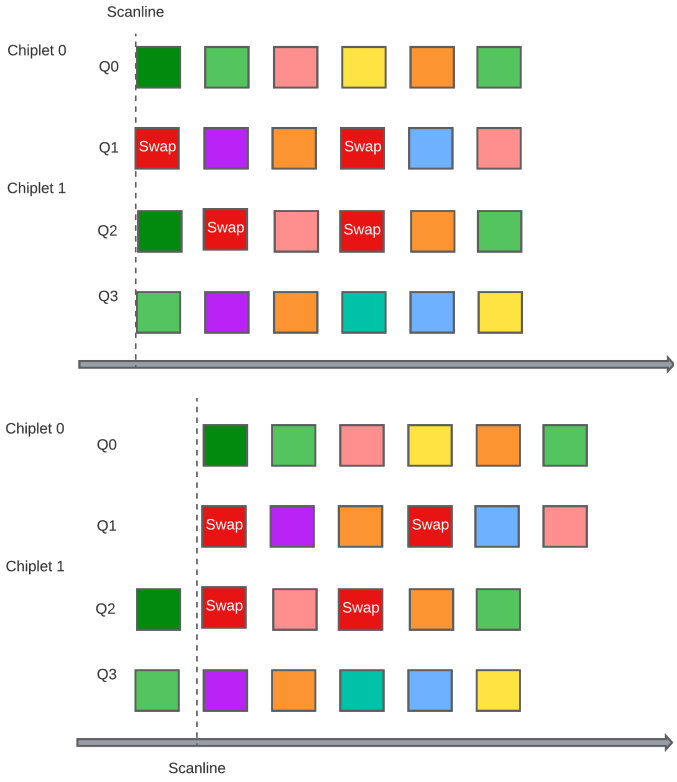

Figure 8: Scanline Algorithm for Swap Alignment

one is better, we can develop a scoring system. Two factors that are relevant in this case are the expected error and the total latency. For the expected error, we can use a simple heuristic. For every qubit, we consider the gates it goes through. Starting with an initial value of 1, we multiply the value by $(1-e_{gate})$. We call the value we get in the end the accuracy score of the qubit. Then we average the scores for all the qubits to get an overall accuracy score for the circuit. For the total latency, a straightforward way to get an estimate is through the circuit depth. However, if we want something more accurate, we can go through all the possible paths in the circuit and find the longest one based on gate delays.

With these two results, we can combine them in a way that rewards higher accuracy scores and lower latency. We can also weigh the importance of latency and accuracy based on the different characteristics of the machines. Based on the final scores, we can pick the optimal circuit.

8 CONCLUSION

In summary, we propose a novel representation of quantum circuits that can be used for chiplet-based designs by adding all the qubits that we can perform swaps on into the circuit. We discuss how to correctly limit those halo qubits to perform swaps only and introduce three approaches for fixing the virtual-to-physical mapping so that the halo qubits do not get moved around, causing incorrect data to be swapped over. However, while this representation correctly captures the instruction, it also creates problems like the alignment of swaps in time. Due to the limited amount of time, we propose a software and a hardware solution. We also explore optimizations that allow the number of subcircuits to exceed the number of chiplets.

Overall, there are still some unresolved challenges in making Qiskit chiplet-aware. However, we believe that chiplet design can bring immense benefits to the field of quantum computing, making quantum supremacy a reality.

9 DISTRIBUTION OF WORK

We worked on the code and paper together, so everyone has equal contributions to this.

ACKNOWLEDGMENTS

We are grateful for the time Professor Hardavellas spent discussing various ideas and implementations with us. The zero-error swap gate and the custom gate ideas are all inspired by our conversations with him.

REFERENCES

[1] Charles Q Choi. 2023. IBM's Quantum Leap: The Company Will Take Quantum Tech Past the 1,000-Qubit Mark in 2023. *IEEE Spectrum* 60, 1 (2023), 46–47.
[2] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.
[3] Ali Javadi-Abhari. 2022. Transpiler needs capability to enforce physical qubits to use and/or qubit layout. https://github.com/Qiskit/qiskit-terra/issues/8185
[4] Eugene M Luks. 1982. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences* 25, 1 (1982), 42–65.
[5] Nicola Mariella and Andrea Simonetto. 2023. A quantum algorithm for the sub-graph isomorphism problem. *ACM Transactions on Quantum Computing* 4, 2 (2023), 1–34.
[6] Jordan K. Matelsky, Elizabeth P. Reilly, Erik C. Johnson, Jennifer Stiso, Danielle S. Bassett, Brock A. Wester, and William Gray-Roncal. 2021. DotMotif: an open-source tool for connectome subgraph isomorphism search and graph queries. *Scientific Reports* 11, 1 (Jun 2021). https://doi.org/10.1038/s41598-021-91025-5
[7] Joe O'Gorman and Earl T. Campbell. 2017. Quantum computation with realistic magic-state factories. *Physical Review A* 95, 3 (mar 2017). https://doi.org/10.1103/physreva.95.032338
[8] Qiskit. 2023. qiskit.compiler.transpile - Qiskit 0.43.1 documentation. https://qiskit.org/documentation/stubs/qiskit.compiler.transpile.html
[9] Qiskit contributors. 2023. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2573505
[10] Kaitlin N. Smith, Gokul Subramanian Ravi, Jonathan M. Baker, and Frederic T. Chong. 2022. Scaling Superconducting Quantum Computers with Chiplet Architectures. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1092–1109. https://doi.org/10.1109/MICRO56248.2022.00078
[11] Li Zeng, Yan Jiang, Weixin Lu, and Lei Zou. 2021. Deep Analysis on Subgraph Isomorphism. arXiv:2012.06802 [cs.DB]

Received 10 June 2023