Circuit Transformations for Quantum Architectures

Andrew M. Childs

Joint Center for Quantum Information and Computer Science, University of Maryland, USA
Institute for Advanced Computer Studies, University of Maryland, USA
Department for Computer Science, University of Maryland, USA
amchilds@umd.edu

Eddie Schoute

Joint Center for Quantum Information and Computer Science, University of Maryland, USA
Institute for Advanced Computer Studies, University of Maryland, USA
Department for Computer Science, University of Maryland, USA
eschoute@cs.umd.edu

Cem M. Unsal

Department of Mathematics, University of Maryland, USA

Abstract

Quantum computer architectures impose restrictions on qubit interactions. We propose efficient circuit transformations that modify a given quantum circuit to fit an architecture, allowing for any initial and final mapping of circuit qubits to architecture qubits. To achieve this, we first consider the qubit movement subproblem and use the Routing via Matchings framework to prove tighter bounds on parallel routing. In practice, we only need to perform partial permutations, so we generalize Routing via Matchings to that setting. We give new routing procedures for common architecture graphs and for the generalized hierarchical product of graphs, which produces subgraphs of the Cartesian product. Secondly, for serial routing, we consider the Token Swapping framework and extend a 4-approximation algorithm for general graphs to support partial permutations. We apply these routing procedures to give several circuit transformations, using various heuristic qubit placement subroutines. We implement these transformations in software and compare their performance for large quantum circuits on grid and modular architectures, identifying strategies that work well in practice.

2012 ACM Subject Classification Computer systems organization → Quantum computing; Hardware → Quantum computation; Mathematics of computing → Graph theory; Applied computing → Physics; General and reference → General conference proceedings; Networks

Keywords and phrases quantum circuit, quantum architectures, circuit mapping

Digital Object Identifier 10.4230/LIPIcs.TQC.2019.3

Related Version Full version at `arXiv:1902.09102 [quant-ph]`.

Supplement Material Source code and result data available at `https://gitlab.umiacs.umd.edu/amchilds/arct`.

Funding This work was supported in part by the Army Research Office (MURI award number W911NF-16-1-0349), the Canadian Institute for Advanced Research, the National Science Foundation (grant number 1813814), and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Algorithms Teams and Quantum Testbed Pathfinder (award number DE-SC0019040) programs.

Acknowledgements The authors would like to thank Aniruddha Bapat for insights on the hierarchical product of graphs and suggestions for tightening the routing lower bound for these graphs. We would also like to thank Drew Risinger for helpful formative discussions.

© Andrew M. Childs, Eddie Schoute, and Cem M. Unsal;
licensed under Creative Commons License CC-BY

14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019).
Editors: Wim van Dam and Laura Mančinska; Article No. 3; pp. 3:1–3:24

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Quantum algorithms are typically formulated in a circuit model in which two-qubit gates can be performed between any pair of qubits. However, most realistic quantum architectures impose restrictions on qubit interactions. Thus a natural challenge is to find a way of implementing a given circuit on a given architecture with low overhead. We can do this by finding a time-efficient *architecture-respecting circuit transformation* – a mapping to a new circuit that preserves the function of the original quantum circuit up to an initial mapping of circuit qubits to architecture qubits and a final mapping of architecture qubits back to circuit qubits, where the new circuit is constrained to respect the architecture.

There have been many proposals for the design of quantum processors. Examples include trapped ion systems that enable interactions between any two ions in a trap [39] and superconducting qubit architectures with more limited interactions [19, 24, 44]. Many proposed architectures for scalable devices employ modularity, building a large device from interconnected subunits [39, 40, 12].

There is also a considerable amount of work on implementing circuits under architectural constraints. Some examples include implementations of Shor's algorithm [18], the quantum Fourier transform on 1D nearest-neighbor architectures [35], and quantum adders on nearest-neighbor architectures [15, 16]. However, the aforementioned works focus on analyzing specific circuits. Instead, we wish to find automated circuit transformations that can handle complex circuits and compare their performance when implemented in various architectures. Bounds on the efficacy of architecture-respecting circuit transformations and good automated tools for implementing them may be able to inform architecture design decisions [52]. Unfortunately, it is challenging to achieve good performance with an automated tool. Indeed, finding even one optimal placement for a set of gates is NP-hard [34].

Prior Work on Automated Architecture-Respecting Circuit Transformations

Several previous works use exhaustive approaches that take time exponential in the number of qubits (and hence can only be used for small instances). For example, Saeedi, Wille, and Drechsler [46] use SAT solvers to decompose circuits so they can be run on the path architecture; [33] finds an optimal circuit transformation on nearest-neighbor architectures by formulating the problem as a pseudo-boolean optimization; Venturelli et al. [50] use temporal planners to schedule gates; and [41] uses satisfiability modulo theory solvers to find mappings of the circuit with high success probability using calibration data. Other work has instead proposed minimizing the distance between all qubits in groups of gates on specific architectures [48, 54, 43], but this is also NP-hard in general. These and other papers add SWAP gates so that the logical state of a given physical qubit is transferred to a different physical qubit (henceforth, we simply refer to this as *qubit movement*, with the implicit understanding that only the logical state is moved).

As a heuristic solution, we can break the circuit into sets of disjoint gates and move qubits between each set. Metodi et al. [36] propose polynomial-time heuristic routines that prioritize gates with many dependents. Hirata et al. [22] propose exhaustive and heuristic searches for good placements of qubits on the path architecture to construct circuit transformations.

One can also use heuristic qubit placement and movement algorithms on fault-tolerant 2D grid architectures [30] or algorithms that are designed to handle the surface code [28]. We do not consider fault tolerance explicitly and instead work only at the logical level.

An exhaustive search of all permutations of n qubit locations takes time $O(n!)$ but can work well for small numbers of locations [53], or can be done selectively using A^* heuristic search [57, 58] or local search [34, 8]. By choosing a suitable initial placement of qubits, we

can further reduce the qubit movement cost. For example, [29] tries to find a good initial placement by repeatedly transforming the quantum circuit forwards and then backwards, taking the output qubit placement as input for the next iteration.

Others have considered a model in which one can perform fast measurements and adapt later parts of the computation based on the outcomes [20]. This model allows the movement of qubits with just a constant overhead at the cost of extra ancillas [45]. However, realizing such a model presents significant technical challenges and we do not consider it here.

Various bounds are also known for the cost of moving qubits. Sorting networks provide a way to upper bound the depth of the qubit movement circuit [27, 7, 13, 21]. We refer to [3] for a more complete overview of sorting networks.

Contribution

In this paper, we construct architecture-respecting circuit transformations that attempt to minimize the circuit depth or size overhead and have worst-case time complexity polynomial in the sizes of the circuit and architecture graph. We model the connectivity of the underlying hardware as a simple graph where vertices represent the qubits and edges represent places where a two-qubit gate can be performed.

As a simple and fast approach, we propose the *greedy swap circuit transformation* (Section 2.2.2). It inserts SWAPs on edges chosen to minimize the total distance between qubits involved in two-qubit gates until some gate(s) can be executed.

We then propose building architecture-respecting circuit transformations (Section 2.2.3) by combining algorithms for two basic subproblems: qubit movement (addressed by *permuters*, for which we provide theoretical performance guarantees) and qubit placement (addressed by *mappers*). For the latter, we specify a variety of heuristic strategies (Section 4) to find suitable placements of qubits from the input circuit, attempting to optimize for circuit size or depth. We implement these algorithms in software, which is publicly available under a free software license [47].

Consider now the problem of moving qubits on a given architecture graph. A sorting network sorts any fixed-length sequence of integers with a circuit of comparators, which compare two inputs and output them in some ordering. While sorting networks can be used to route qubits [7], they achieve a more general task, and the cost of routing can sometimes be lower with other methods. Specifically, we suggest ROUTING VIA MATCHINGS [2] (introduced in Section 3.1) as a more suitable framework for moving qubits in parallel. Deciding whether there exists a depth-k circuit for ROUTING VIA MATCHINGS is NP-complete in general for $k > 2$ [3], but optimal or near-optimal protocols are known for specific graph families [2, 56]. In some cases it is possible to implement any permutation asymptotically more efficiently than a general sorting network (see Table 1). On complete graphs, for example, any permutation can be implemented in a depth-2 circuit of transpositions [2], whereas an optimal sorting network has depth $\Theta(\log n)$ [1].

While it is common to consider only the worst-case routing performance, we also wish to route efficiently in practice. To improve practical performance, we generalize to *partial permutations* (permutations only defined on some subdomain) so that we can also move subsets of qubits efficiently. The destinations of the remaining qubits are unconstrained. In Section 3.1, we present routing algorithms for the path graph, the complete graph, and the *generalized hierarchical product* of graphs [6], which includes the Cartesian product of graphs and *modular* architectures as special cases [40]. Graphs obtained as hierarchical products have many good properties for quantum architectures [5]. We establish an upper bound on the routing number of a hierarchical product (Theorem 4) that matches prior work for total permutations on the Cartesian product of graphs [2] and depends on easily computable properties of the input partial permutation.

Table 1 Performance bounds for sorting networks versus routing via matchings (the routing number, $\mathrm{rt}(G)$; see (6)) where $|V| = n$. A (Δ, D)-tree has max degree Δ and diameter D. The generalized hierarchical product of the graphs G_1 and $G_2 = (V_2, E_2)$ is denoted by $\Pi_{\vec{v}}(G_1, G_2)$, for $\vec{v} \in \{0,1\}^{|V_2|}$ (see Definition 2). The special cases of the Cartesian product of graphs, the r-dimensional grid, and the modular graph are also listed. Let $\vec{1} \coloneqq [1 \dots 1]$ and $\vec{e}_1 \coloneqq [1\,0 \dots 0]$.

	Worst-case circuit depth	
Graph family	Sorting (comparators)	Routing nr. (transpositions)
path (P_n)	n [25]	n [2]
complete (K_n)	$\Theta(\log n)$ [1]	2 [2]
(Δ, D)-tree	$O(\min(\Delta, \log(n/D))\, n)$ [4]	$3n/2 + O(\log n)$ [56]
$\Pi_{\vec{v}}(G_1, G_2)$	not known	$\left\lceil \frac{\lvert V_2 \rvert}{\mathrm{ham}(\vec{v})} \right\rceil (\mathrm{rt}(G_1) + \mathrm{rt}(G_2)) + \mathrm{rt}(G_2)$
$G_1 \times G_2 = \Pi_{\vec{1}}(G_1, G_2)$	not known	$2\,\mathrm{rt}(G_1) + \mathrm{rt}(G_2)$ [2]
$\bigtimes_{i=1}^{r} P_{n_i}$	$n_1 + 2\sum_{i=2}^{r} n_i + o(\cdot)$ [26]	$n_1 + 2\sum_{i=2}^{r} n_i$ [2]
$\Pi_{\vec{e}_1}(K_{n_1}, K_{n_2})$	not known	$3n_2 + 2$

We also propose using Token Swapping [55] for minimizing the total number of SWAPs, which is relevant when optimizing for total circuit size (Section 3.2). We generalize this problem to partial permutations and obtain a 4-approximation algorithm (Theorem 7).

Finally, we evaluate our circuit transformations on large quantum circuits (Section 5) and compare their performance with the circuit transformation included in the Qiskit software (Section 2.2.1) [8]. We find that the relative performance varies significantly with the circuit type and architecture. When minimizing circuit size, the greedy swap circuit transformation is one of the best, though some improvement may be gained using some of our specialized circuit transformations. For depth, some of our specialized circuit transformations do best on random circuits on grid architectures, whereas Qiskit's circuit transformation does well on modular architectures. For quantum signal processing circuits [32] we find that the depth is best minimized by our greedy swap circuit transformation.

2 Constructing Circuit Transformations

Program transformations are algorithms that modify computer programs while retaining functionality [42]. In a similar vein, we define a *circuit transformation* as an algorithm that modifies an input quantum circuit to produce an output quantum circuit with the same functionality. We represent an architecture by a simple graph $G = (V, E)$, and let Q denote the set of qubits of the input circuit. A circuit transformation is *architecture-respecting* if it produces injective initial and final mappings of the form $\hat{p} \colon Q \to V$ and an architecture-respecting output circuit. The output circuit is architecture-respecting if for each two-qubit gate acting on (qubit) vertices v_1, v_2 we have $(v_1, v_2) \in E$ (where the ordering is irrelevant since G is undirected). Henceforth, we only consider circuit transformations that are architecture-respecting, and we refer to them simply as circuit transformations. We propose a construction for a general circuit transformation that may use the properties of the underlying architecture by relying on a specialized subroutine for moving qubits called a *permuter* (Section 3), and a subroutine determining where to place qubits, called a *mapper* (Section 4). We show in Appendix C that our circuit transformations are polynomial-time in the circuit size and architecture graph size.

To be able to transform a circuit, we must have $|Q| \leq |V|$, and the output circuit must contain a qubit for every vertex in the architecture. Throughout the circuit transformation, we keep track of the injective current placement of qubits $\hat{p} \colon Q \to V$. The initial and final

values of $\hat{p}$ are also the initial and final mappings, respectively, of qubits to the architecture. A gate is *executed* by appending it to the output circuit. Two-qubit gates with qubits $q_1, q_2 \in Q$ can only be executed when $(\hat{p}(q_1), \hat{p}(q_2)) \in E$. By adding SWAP gates to the output circuit, we can change $\hat{p}$ and thereby unitarily transform quantum circuits for execution on an architecture.

2.1 Definitions

Partial Functions and Partial Permutations. For sets X and Y, a *partial function* $f\colon X \rightharpoonup Y$ is a mapping from $\mathrm{dom}(f) \subseteq X$ to $\mathrm{image}(f) \coloneqq \{f(x) \mid x \in \mathrm{dom}(f)\} \subseteq Y$. However, $f(x)$ is undefined for $x \in X \setminus \mathrm{dom}(f)$. We consider such elements x *unmapped*. For $x \in \mathrm{dom}(f)$, we write $x \mapsto f(x)$ and say that x is *mapped to* $f(x)$. We can then define any partial function f as a set of mappings, $f \coloneqq \{x \mapsto y \mid x \in X, y \in Y\}$, where all preimages must be distinct (i.e., if $x \mapsto y \in f$ and $x' \mapsto y' \in f$ with $y \neq y'$, then $x \neq x'$). A *total* function $\hat{f}$ is a partial function where $\mathrm{dom}(\hat{f}) = X$ and is denoted $\hat{f}\colon X \to Y$. By the term “function” we will mean a total function.

A partial function f is *injective* iff $\forall x, x' \in \mathrm{dom}(f)$ with $x \neq x', f(x) \neq f(x')$. A function $\hat{f}\colon X \to Y$ is *surjective* iff $\forall y \in Y, \exists x \in X : f(x) = y$. A *bijective partial function* f is a partial function that is injective and is denoted $f\colon X \leftrightharpoons Y$ (note that such an f is necessarily surjective on its image). A *bijective function* $\hat{f}$ is both injective and surjective and is denoted by $\hat{f}\colon X \leftrightarrow Y$. For any bijective (partial) function f there exists an inverse function $f^{-1}\colon \mathrm{image}(f) \to \mathrm{dom}(f)$.

A partial permutation π is any bijective partial function with the same domain and codomain, i.e., $\pi\colon X \leftrightharpoons X$. Similarly, a total permutation is any $\sigma\colon X \leftrightarrow X$. By “permutation” we mean a total permutation.

We also define some notions specifically useful for this paper. An *unmapped vertex* is a vertex in $V \setminus \mathrm{dom}(\pi)$, for a graph $G = (V, E)$ and $\pi\colon V \leftrightharpoons V$. We define the union of partial functions $f\colon X \rightharpoonup Y$ and $g\colon X \rightharpoonup Y$ when $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$ as

$$(f \cup g)(x) \coloneqq \begin{cases} f(x) & \text{if } x \in \mathrm{dom}(f)\,, \\ g(x) & \text{if } x \in \mathrm{dom}(g)\,. \end{cases} \tag{1}$$

Furthermore, $(f \cup g)$ is a bijective partial function iff f and g are bijective partial functions and $\mathrm{image}(f) \cap \mathrm{image}(g) = \emptyset$. A *completion* of $\pi\colon X \leftrightharpoons X$ is a $\hat{\pi}\colon X \leftrightarrow X = (\pi \cup \sigma)$ for some $\sigma\colon X \leftrightharpoons X$, where $\mathrm{dom}(\sigma) = X \setminus \mathrm{dom}(\pi)$ and $\mathrm{image}(\sigma) = X \setminus \mathrm{image}(\pi)$.

Directed Acyclic Graph Representation of a Circuit. A quantum circuit can be viewed as a directed acyclic graph (DAG), where vertices represent gates and directed edges represent qubit dependencies. We define the first *layer* of the DAG, L, to be the set of all vertices without predecessors. By removing L and taking the first layer of the resulting DAG, we can define the second layer, and so on.

The size of a circuit is the number of gates it contains (i.e., the number of vertices in the DAG); the depth of a circuit is the number of layers. It is natural to minimize either the depth, corresponding to the execution time when gates can be applied in parallel, or the size, corresponding to the total number of operations that must be performed. We are mostly only interested in two-qubit gates and their qubits. Therefore, let us define $\mathrm{tg}\colon V_D \rightharpoonup Q \times Q$, where V_D is the set of DAG vertices, that outputs the pair of qubits acted on by the DAG vertex, for two-qubit gates. For simplicity, we denote $\mathrm{tg}(L) \coloneqq \{\mathrm{tg}(g) \mid g \in L, g \text{ is a two-qubit gate}\}$.

2.2 Architecture-Respecting Circuit Transformations

We now describe some specific architecture-respecting circuit transformations. We first describe two basic circuit transformations, one provided by the Qiskit software (Section 2.2.1) and another that uses a simple greedy approach (Section 2.2.2). Then, in Section 2.2.3 we specify a family of circuit transformations that builds on specialized procedures for qubit placement and routing.

2.2.1 Qiskit Circuit Transformation

The open-source quantum computing software framework Qiskit [8] contains a circuit transformation[1] that we build upon in one of our mappers (Section 4). We specify this transformation here and compare it with our other approaches to circuit transformations in Section 5.

We initialize $\hat{p}$ arbitrarily. Fix a number of trials, $k \in \mathbb{N}$, for each layer. We do the following in trial $i \in [k]$ where $[k] \coloneqq \{1, \dots, k\}$: For all $v, u \in V$, sample a symmetric weight $d_i(v,u) \coloneqq (1 + \mathcal{N}(0, 1/N))\, d(v,u)^2$ independently for $(v,u) \in V \times V$, where $\mathcal{N}(\mu, \sigma)$ represents a sample from the normal distribution with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma \geq 0$, and $d \colon V \times V \to \mathbb{N}$ is the shortest distance function on the architecture graph. We define an objective function as the sum of gate distances,

$$S \coloneqq \sum_{(q_1,q_2) \in \mathrm{tg}(L)} d_i(\hat{p}(q_1), \hat{p}(q_2))\,. \tag{2}$$

We now try to SWAP pairs of qubits to decrease S. Specifically, we construct a set of SWAPs by iterating over all edges $e \in E$ and greedily adding the corresponding SWAP if it decreases S and neither endpoint of e is already involved in some SWAP. We execute the set of SWAPs and update S. We then iterate this process until either $S = |\mathrm{tg}(L)|$; or there is no SWAP that decreases S; or we reach the upper bound of $2|V|$ iterations.

Now, if $S = |\mathrm{tg}(L)|$ then the algorithm has successfully found a sequence of SWAPs and all gates in L can be executed. The result of trial i is then set to this sequence of SWAPs. Otherwise, trial i is a failure. If there is at least one successful trial out of k trials, we execute the SWAPs of a successful trial with the fewest SWAPs and then execute all gates in L.

If no trial was successful, we apply the same routine for finding SWAPs that minimize S, but taking only a single gate $(q_1, q_2) \in \mathrm{tg}(L)$ at a time. Note that this results in a sequence of SWAPs along the shortest path between $\hat{p}(q_1)$ and $\hat{p}(q_2)$. After each such step we execute the selected gate. We repeat this until all gates in $\mathrm{tg}(L)$ have been executed and also execute all single-qubit gates in L. Finally, we remove the vertices in L from the input circuit DAG and iterate this process until all gates in the input circuit are executed.

2.2.2 Greedy Swap Circuit Transformation

We also describe a simple greedy approach to circuit transformations. Similar to the Qiskit circuit transformation described above, we prioritize SWAPs that maximally reduce the total distance between the qubits $\mathrm{tg}(L)$, but now using the simpler objective function

$$R \coloneqq \sum_{(q_1,q_2) \in \mathrm{tg}(L)} d(\hat{p}(q_1), \hat{p}(q_2))\,. \tag{3}$$

Note that this is different from (2), where a randomized distance d_i is used.

[1] We base our description on `qiskit.mapper.swap_mapper` from Qiskit version 0.6.1.

We construct an initial $\hat{p}$ as follows. Let us consider the first layer L' of the circuit consisting of only two-qubit gates (i.e., single-qubit gates are ignored), initialize $p' \colon Q \rightharpoonup V$ as undefined everywhere, and set $U := \emptyset \subseteq V$. We iteratively construct

$$p' \leftarrow p' + \{q_1 \mapsto v_1, q_2 \mapsto v_2 \mid (q_1, q_2) \in L', (v_1, v_2) \in M\}\,, \tag{4}$$

where $M \subseteq E$ is a maximum matching of G, remove (q_1, q_2) from L', set $U \leftarrow U \cup \{v_1, v_2\}$, and recompute M on the subgraph of G with the vertices $V \setminus U$.[2] The remaining qubits $Q \setminus \mathrm{dom}(p')$ are arbitrarily mapped to the available vertices $V \setminus \mathrm{image}(p')$ to obtain $\hat{p}$.

In every iteration, we construct a set of disjoint gates to execute. We first execute as many gates from L as possible given $\hat{p}$, and we remove these gates from the input circuit. Second, let E_i, for $i \in [2]$, be the set of edges where executing a SWAP would decrease R by i, excluding edges which already had a vertex involved in a gate this iteration. We then greedily execute gates from E_2 first and E_1 second, updating both E_is as we go. If we were not able to execute a gate from L and no SWAPs were executed, then, as a fallback, we deterministically pick a two-qubit gate $(q_1, q_2) \in \mathrm{tg}(L)$ and SWAP along the first edge on the shortest path between $\hat{p}(q_1)$ and $\hat{p}(q_2)$. We update $\hat{p}$ according to the inserted SWAPs, update L, and finally update R. This process is repeated until the input circuit is empty.

The fallback routine ensures that this circuit transformation always produces an output circuit. The value R strictly decreases in every iteration until a gate can be executed unless the fallback routine is performed, in which case R stays the same. On repeated calls to the fallback routine, the same two-qubit gate is picked deterministically until it is executed. This happens within $\mathrm{diam}(G) + 1$ iterations, where $\mathrm{diam}(G)$ denotes the diameter of G. By induction we see that the whole circuit will be executed.

2.2.3 Constructing Architecture-aware Circuit Transformations

We now present our construction for a general circuit transformation and make some definitions more precise. Let a *permuter* (Section 3) be a subroutine that, given $\pi \colon V \rightharpoonup V$, outputs a sequence of transpositions that implements π while respecting the architecture constraints. Let a *mapper* (Section 4) be a subroutine that, given $\hat{p}$, a permuter, and a quantum circuit, computes a new placement of qubits, $p \colon Q \rightharpoonup V$, such that some gates of the input circuit can be executed.

Initialize $\hat{p}$ in the same way as the greedy swap circuit transformation. We repeat the following steps until the entire circuit has been transformed:

1. Use the given mapper to find a placement, $p \colon Q \rightharpoonup V$, for the remaining input circuit;
2. Let "$\circ$" denote partial function composition, i.e., given $g \colon X \rightharpoonup Y$ and $f \colon Y \rightharpoonup Z$, $(f \circ g)(x) := f(g(x))$, for $x \in \mathrm{dom}(g)$ and $g(x) \in \mathrm{dom}(f)$. We use the permuter to find transpositions implementing $p \circ \hat{p}^{-1} \colon V \rightharpoonup V$ and replace the transpositions with SWAP gates to construct a permutation circuit to execute. We also update $\hat{p}$ to reflect the new placement of qubits after running the permutation circuit.
3. Execute all gates in L that can be executed in accordance with $\hat{p}$, remove these gates from the input circuit, and recompute L.

[2] This is equivalent to running the greedy depth mapper (Section 4) on the input circuit with only two-qubit gates, an arbitrary $\hat{p}$, and free permutations of qubits. In other words, the greedy depth mapper will pick a placement of qubits on the architecture unconstrained by movement of qubits, since this is the initial placement.

3 Partial Permutations via Transpositions

In this section we provide routing algorithms for implementing partial permutations via transpositions constrained to edges of a graph. We call such algorithms *permuters*. The Routing via Matchings and Token Swapping problems capture exactly our optimization goals of implementing a permutation of qubits on a quantum architecture while minimizing the circuit depth and size, respectively.

3.1 Partial Routing Via Matchings

The framework of Routing via Matchings captures how to permute qubits on a graph using a circuit of the smallest possible depth [2]. We first define a generalization of Routing via Matchings that allows for partial permutations and then provide permuters for implementing partial permutations for some architectures of interest.

▶ **Definition 1** (Partial Routing via Matchings). Partial Routing via Matchings *is the following optimization problem. Given a simple graph $G = (V, E)$ and a $\pi\colon V \rightharpoonup V$, the objective is to find the smallest $k \in \mathbb{N}$ such that there exist matchings $M_1, \dots, M_k \subseteq E$ on G, where each matching induces a permutation as a product of disjoint transpositions*

$$\pi_{M_i} = \prod_{(v,u)\in M_i} (v\ u), \quad \textit{such that} \quad \hat{\pi} = \prod_{i=1}^{k} \pi_{M_i} \tag{5}$$

is a completion of π.

Routing via Matchings is the special case of Partial Routing via Matchings where π is constrained to be a (total) permutation. The *partial routing number* of $\pi\colon V \rightharpoonup V$ on G is $\mathrm{rt}(G, \pi) := k$, where k obtains the minimum in Definition 1. The *routing number* [2] is the special case of the partial routing number where π is total. In this paper, we simply refer to the partial routing number as the routing number. The routing number of G is defined as

$$\mathrm{rt}(G) := \max_{\sigma\in \mathrm{Sym}(V)} \mathrm{rt}(G, \sigma)\,, \tag{6}$$

where we maximize over all permutations $\sigma\colon V \leftrightarrow V$ (here $\mathrm{Sym}(V)$ denotes the group of such permutations). Note that we only optimize over permutations, since for any $\pi\colon V \rightharpoonup V$,

$$\mathrm{rt}(G, \pi) = \min_{\hat{\pi}} \mathrm{rt}(G, \hat{\pi})\,, \tag{7}$$

where we minimize over all completions $\hat{\pi}$ of π.

An alternate way to interpret (Partial) Routing via Matchings is to assign *tokens* to all $v \in \mathrm{dom}(\pi)$ and destinations $\pi(v)$ for the tokens. A token can only by moved through an exchange of tokens between adjacent vertices. The goal is to move all tokens to their destination in as few matchings (specifying exchange locations) as possible. If a vertex does not hold a token at the time of an exchange with a neighbor, as can be the case in Partial Routing via Matchings, then after the exchange the neighbor will not hold a token.

We give simple constructions for permuters of the complete graph, K_n, and the path graph, P_n, for $n \in \mathbb{N}$. Let V be the vertex set of the respective graph and $\pi\colon V \rightharpoonup V$ given. For K_n, if $|\mathrm{dom}(\pi) \cup \mathrm{image}(\pi)| = 2|\mathrm{dom}(\pi)|$ all mappings are disjoint, so we return $\{(v, \pi(v)) \mid v \in \mathrm{dom}(\pi)\}$ as a single matching that implements π. Otherwise, we construct an arbitrary completion $\hat{\pi}$ of π and run the standard algorithm for Routing via Matchings for complete graphs on $\hat{\pi}$ [2], obtaining $\mathrm{rt}(K_n, \pi) \le 2$.

For P_n, let $V \cong [n]$, ordered from one end of the path to the other (picking ends arbitrarily). Iterate through $i \in V$ in ascending order, setting

$$\hat{\pi}(i) = \begin{cases} \pi(i) & \text{if } i \in \operatorname{dom}(\pi), \\ \min\left(V \setminus \operatorname{image}(\hat{\pi})\right) & \text{otherwise.} \end{cases} \tag{8}$$

We then run the standard path routing algorithm [2] on $\hat{\pi}$, obtaining $\operatorname{rt}(P_n, \pi) \leq n$. It remains an open question whether a tighter bound can be proven as a function of some property of π.

Hierarchical Product

The *generalized hierarchical product* (henceforth *hierarchical product*) of graphs [6] produces various subgraphs of the Cartesian product of graphs that include natural models of quantum computer architectures [5].

▶ **Definition 2** (Hierarchical Product [6]). *For $j \in \{1, 2\}$, let $G_j = (V_j, E_j)$ be a graph with $n_j := |V_j|$ vertices and adjacency matrix $A_j \in \mathcal{M}_{n_j}$, where $\mathcal{M}_k$ is the set of $k \times k$ boolean matrices for $k \in \mathbb{N}$. Then the* hierarchical product *$\Pi_{\vec{v}}(G_1, G_2)$, for $\vec{v} \in \{0, 1\}^{n_2}$, has vertex set $V_1 \times V_2$ and adjacency matrix $A_1 \otimes \operatorname{diag}(\vec{v}) + \mathbb{I}_{n_1} \otimes A_2$, where $\mathbb{I}_{n_1} \in M_{n_1}$ is the $n_1 \times n_1$ identity matrix, $M_1 \otimes M_2 \in \mathcal{M}_{n_1 n_2}$ is the Kronecker product of $M_1 \in \mathcal{M}_{n_1}$ and $M_2 \in \mathcal{M}_{n_2}$, and $\operatorname{diag}(\vec{v}) \in M_{n_2}$ is the diagonal matrix with the entries of $\vec{v}$ on the diagonal.*

Intuitively, this graph consists of n_1 copies of G_2, where the jth vertices in all copies of G_2 are connected by a copy of G_1 if $\vec{v}_j = 1$. We restrict ourselves to connected simple graphs, so A_1 and A_2 are symmetric 0–1 matrices and $\vec{v}$ is nonzero. An example of the hierarchical product of two path graphs is

$$\Pi_{[1\ 0\ 1]}(P_2, P_3) = \Pi_{[1\ 0\ 1]}\left(\begin{matrix} ② \\ | \\ ① \end{matrix}, ①—②—③ \right) = \begin{matrix} (2{,}1)—(2{,}2)—(2{,}3) \\ | \qquad\qquad\qquad | \\ (1{,}1)—(1{,}2)—(1{,}3) \end{matrix} \tag{9}$$

The Cartesian product is $\Pi_{\vec{1}}$, where $\vec{1} := [1 \ldots 1]$, and $\Pi_{\vec{e}_i}$ is the rooted product of graphs, rooted at the ith vertex of G_2.

We define the vertex-induced subgraph of any graph $G = (V, E)$ for vertex set $U \subseteq V$ as

$$G[U] := (U, E \cap (U \times U)) . \tag{10}$$

Now, let $G = (V, E) = \Pi_{\vec{v}}(G_1, G_2)$ and denote the vertices of G by $v = (v_1, v_2) \in V_1 \times V_2 = V$. We define $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i) := G\left[\{i\} \times V_2\right]$, for $i \in V_1$. Note that each $\mathcal{G}_i$ is isomorphic to G_2, so the permuter for G_2 can be used for $\mathcal{G}_i$. We also define the *communicator vertices* of $\mathcal{G}_i$ as the vertices $\{i\} \times \{j \in V_2 \mid \vec{v}_j = 1\} \subseteq \mathcal{V}_i$ and index them in ascending order (for some ordering of V). Note that the jth communicator vertex (of any $\mathcal{G}_i$) also belongs to $G[V_1 \times \{j\}]$, which is isomorphic to G_1.

A useful metric is the maximum number of vertices that need to leave or enter any $\mathcal{G}_i$ to implement π, defined as the degree of π,

$$\deg(\pi) := \max \bigcup_{i \in V_1} \left\{ |\{v \in \operatorname{dom}(\pi) \cap \mathcal{V}_i \mid \pi(v) \notin \mathcal{V}_i\}|, |\{v \in \operatorname{dom}(\pi) \setminus \mathcal{V}_i \mid \pi(v) \in \mathcal{V}_i\}| \right\} . \tag{11}$$

In every iteration of the routing algorithm, we route a set $R = \{v^{(i)} \in \mathcal{V}_i \mid i \in V_1\}$ such that all $\pi(v)_1$ are distinct, for $v \in R$ and $\pi(v) = (\pi(v)_1, \pi(v)_2) \in V$. Undefined values are always considered distinct. We call such R a set of *representative vertices*, and we view $v^{(i)}$ as the representative vertex of V_i.

Algorithm 3.1: Partial Routing via Matchings on the hierarchical product of graphs $\Pi_{\vec{v}}(G_1, G_2)$. In Line 1, *routing* means constructing a partial permutation σ on a subgraph (G_1 or G_2), using the applicable permuter to find transpositions implementing σ, and applying those transpositions to update π and each R_i.

```
  input  : π: V1 × V2 ⇀ V1 × V2; permuters on G1 and G2
1 Let R_i, for i ∈ [deg(π)], be given by Lemma 3
2 for i = 1, ..., ⌈deg(π)/ham(v⃗)⌉ :
3 |  foreach j ∈ V1 :
4 |  |  on 𝒢_j, for all k ∈ [ham(v⃗)], route the (unique) vertex v ∈ R_{(i−1)·ham(v⃗)+k} ∩ 𝒱_j
  |  |   to the k-th communicator vertex of 𝒢_j    // For R_ℓ with ℓ > deg(π), do
  |  |   nothing
5 |  foreach communicator vertex (v1, v2) of 𝒢1 :            // All copies of G1
6 |  |  on G[V1 × {v2}] = (V′, E′), route each v ∈ V′ ∩ dom(π) to (π(v)_1, v2) ∈ V′
7 foreach i ∈ V1 :
8 |  route all v ∈ dom(π) ∩ V_i to π(v) within 𝒢_i
9 return the transpositions that implement this routing
```

▶ **Lemma 3** (Proof in Appendix A.1). *For a graph $\Pi_{\vec{v}}(G_1, G_2)$, $\pi\colon V \rightharpoonup V$, let $d := \deg(\pi)$. We can find distinct sets of representative vertices R_i, for $i \in [d]$, such that*

$$\{v \in \mathrm{dom}(\pi) \mid v_1 \neq \pi(v)_1\} \subseteq \bigcup_{i \in [d]} R_i \,.$$

Algorithm 3.1 specifies a permuter for the hierarchical product. We prove the following performance bounds for this algorithm

▶ **Theorem 4.** *For a graph $\Pi_{\vec{v}}(G_1, G_2)$, Algorithm 3.1 finds a sequence of transpositions that implements $\pi\colon V \rightharpoonup V$ certifying that*

$$\mathrm{rt}(\Pi_{\vec{v}}(G_1, G_2), \pi) \leq \left\lceil \frac{\deg(\pi)}{\mathrm{ham}(\vec{v})} \right\rceil (\mathrm{rt}(G_1) + \mathrm{rt}(G_2)) + \mathrm{rt}(G_2)\,,$$

where $\mathrm{ham}(\vec{v})$ is the Hamming weight of $\vec{v}$, i.e., the number of ones in $\vec{v}$.

Proof. In every round of routing, we route $\mathrm{ham}(\vec{v})$ sets R_i to their destination $\mathcal{G}_j$s, for $j \in V_1$. In each round, we route on all copies of G_2 in parallel and then route on all copies of G_1 in parallel. After routing all R_i in at most $\lceil \deg(\pi)/\mathrm{ham}(\vec{v}) \rceil$ rounds, Lemma 3 ensures that only permutations local to each $\mathcal{G}_j$ remain. Finally, we route vertices to their destinations, as given by π, in each $\mathcal{G}_j$ independently using the permuter for G_2. ◀

As a possible optimization, we can remove some vertices from the partial permutations in the routing steps. For each removed vertex, we must ensure that the remaining steps of the routing algorithm remain valid. Specifically, let there be a $u \in \mathcal{G}_i \cap R_k$ for $i \in V_1$ and $k \in [\deg(\pi)]$. If $u \in \mathrm{dom}(\pi)$ and $\pi(u) \in \mathcal{V}_i$, then we remove it since it does not need to be routed outside of $\mathcal{G}_i$. Otherwise, if $u \notin \mathrm{dom}(\pi)$, we remove it unless $\{R_k \cap \mathrm{dom}(\pi) \mid \pi(v) \in \mathcal{G}_i\} \neq \emptyset$ since an unmapped vertex is expected at the communicator vertex in the second loop of the routing round. We apply this optimization in our implementation of the permuter for modular graphs (Appendix A.2).

We show a lower bound on the routing number of hierarchical products of graphs, which can be shown to be tight up to constant factors (see full arXiv version on title page).

▶ **Theorem 5.** *For a graph* $\Pi_{\vec{v}}(G_1, G_2)$ *and any* $\pi\colon V \rightharpoonup V$*,*

$$2\left\lceil \frac{\deg(\pi)}{\operatorname{ham}(\vec{v})} \right\rceil - 1 \leq \operatorname{rt}(\Pi_{\vec{v}}(G_1, G_2), \pi)\,.$$

Proof. Let us consider the token-based formulation of PARTIAL ROUTING VIA MATCHINGS. At most $\deg(\pi)$ tokens need to be moved out of any $\mathcal{G}_i$, for $i \in V_1$. Every matching can move at most $\operatorname{ham}(\vec{v})$ tokens out of their original $\mathcal{G}_i$. Once moved out, a new set of tokens must be moved onto the $\operatorname{ham}(\vec{v})$ communicator vertices. Therefore, it takes at least $2\lceil \deg(\pi)/\operatorname{ham}(\vec{v})\rceil - 1$ matchings to move $\deg(\pi)$ tokens out of any $\mathcal{G}_i$. ◀

For special cases of interest to quantum architectures, we analyze the modular graph in Appendix A.2 and provide a permuter specialized to Cartesian products of graphs with some heuristic optimizations in the full arXiv version.

3.2 Partial Token Swapping

The TOKEN SWAPPING problem is similar to ROUTING VIA MATCHINGS, but minimizes the total number of transpositions instead of the depth [55]. It follows that the induced permutation circuit is optimized for circuit size. The decision version of TOKEN SWAPPING was first shown to be NP-complete [38] and, later, hardness was shown in parametrized complexity of the number of swaps k [10]. For $\epsilon > 0$, a $(1+\epsilon)$-approximation algorithm is an algorithm that produces a solution within a factor $(1+\epsilon)$ of optimal for all valid inputs. Here, we define a generalized version of TOKEN SWAPPING that allows for partial permutations, and then give a 4-approximation algorithm for this problem on connected simple graphs that generalizes a previous 4-approximation algorithm for total permutations [38].

▶ **Definition 6** (PARTIAL TOKEN SWAPPING)**.** *We define* PARTIAL TOKEN SWAPPING *as an optimization problem. Given are a graph* $G = (V, E)$ *and partial permutation* $\pi\colon V \rightharpoonup V$*. The objective is to find the smallest* $k \in \mathbb{N}$ *such that* $\hat{\pi} = (u_1\ v_1)(u_2\ v_2)\ldots(u_k\ v_k)$*, for* $\hat{\pi}$ *some completion of* π *and* $(u_i, v_i) \in E$ *for* $i \in [k]$*.*

Analogous to the routing number, we define the *routing size* of $\pi\colon V \rightharpoonup V$ on G, $\operatorname{rs}(G, \pi)$, to be the minimum k in Definition 6, and the routing size of G as

$$\operatorname{rs}(G) \coloneqq \max_{\sigma \in \operatorname{Sym}(V)} \operatorname{rs}(G, \sigma)\,. \tag{12}$$

TOKEN SWAPPING is the special case of PARTIAL TOKEN SWAPPING where π is constrained to be a total permutation. PARTIAL TOKEN SWAPPING also has an equivalent token-based formulation, similar to PARTIAL ROUTING VIA MATCHINGS.

We now describe a permuter that aims to minimize the circuit size. Miltzow et al. [38] gave a 4-approximation algorithm for TOKEN SWAPPING. Here, we generalize their results to PARTIAL TOKEN SWAPPING and prove that our generalized algorithm is also a 4-approximation algorithm. For this section, we consider the token-based formulation of PARTIAL TOKEN SWAPPING (recall the notion of tokens introduced in Section 3.1).

The main idea of Miltzow et al. is to perform SWAPs that reduce the sum of all distances of tokens to their destinations. We use the following definitions from [38]: An *unhappy swap* is "an edge swap where one of the tokens swapped is already on its target and the other token reduces its distance to its target vertex (by one)", and a *happy swap chain* is a path of $\ell + 1$ distinct vertices $v_1 v_2 \ldots v_\ell$, such that swapping all $(v_i, v_{i+1}) \in E$, for $i \in [\ell - 1]$, in increasing order strictly reduces the distances of all tokens in the chain to their destinations.

Algorithm 3.2: Routing tokens to their destinations while minimizing the number of transpositions. We add an extra step that performs no-token swaps to the algorithm of [38]. For $v \in V$, $N(v) \subseteq V$ denotes the set of neighbors of v. The partial permutation $\mathrm{id}|_{\mathrm{dom}(\pi)} \colon V \rightharpoonup V$ is the restriction of the identity function $\mathrm{id} \colon V \leftrightarrow V$ to $\mathrm{dom}(\pi)$ (so it is undefined outside of $\mathrm{dom}(\pi)$).

```
  input  : π: V ⇀ V
1 while π ≠ id|dom(π) :
2   if there exists a happy swap chain v1 v2 ... vℓ then
3     Perform transpositions (v1 v2)(v2 v3)...(vℓ−1 vℓ)
4   else if ∃v ∈ dom(π), ∃u ∈ N(v) \ dom(π) : d(u, π(v)) < d(v, π(v)) then
5     Perform no-token swap (v u)                        // u has no token
6   else
7     There exists an unhappy swap; perform it
8   Update π according to the transpositions that were performed
9 return The sequence of transpositions that was performed
```

When considering a partial permutation, not all vertices have a token assigned to them. We add an extra step to the approximation algorithm for Token Swapping to make use of this: Before considering an unhappy swap, we first try to swap a token to a tokenless neighbor if it brings the token closer to its destination. We call this a *no-token swap*. The approximation algorithm for Partial Token Swapping is specified in full in Algorithm 3.2.

▶ **Theorem 7** (Proof in Appendix A.3). *Given a simple connected graph $G = (V, E)$ and $\pi \colon V \rightharpoonup V$, Algorithm 3.2 uses at most $4 \cdot \mathrm{rs}(G, \pi)$ transpositions.*

In the full arXiv version we also show that, when restricted to tree graphs, Algorithm 3.2 is a 3-approximation algorithm or worse, even though it is a 2-approximation algorithm on trees for total permutations [38].

4 Placing Qubits on the Architecture

A *mapping* algorithm (or *mapper*) finds an assignment of circuit qubits to architecture vertices such that gates can be executed efficiently. We specify mappers in terms of the routing number and the routing size. In practice, we replace these quantities with the upper bounds that result from applying our permuters.

Mappers construct *placements* of circuit qubits onto qubits of the architecture. A placement is a bijective partial function $p \colon Q \rightharpoonup V$, where $G = (V, E)$ is the architecture graph. A mapper has access to the *current placement* $\hat{p} \colon Q \to V$ provided by the circuit transformation. Given a placement p and the current placement $\hat{p}$, we can compute a partial permutation $p \circ \hat{p}^{-1} \colon V \rightharpoonup V$ that implements p. All our mappers construct a placement p that is initially undefined everywhere and modify it until finished.

We briefly describe the mappers (and their abbreviations) that we implement and evaluate (see Appendix B for details). We propose mappers optimizing for circuit depth (*depth mappers*) and for circuit size (*size mappers*). For size mappers, specifically, if there is any gate that can be performed without moving qubits, then there is no disadvantage to doing that immediately since it will have to be performed eventually. If there is any such gate, we simply return the empty placement. Thus we assume, for all size mappers, that there are no gates to be performed in-place. Let L be the first layer of gates of the input circuit.

The *greedy depth mapper* (greedy depth) repeatedly places the highest-cost gate in L at its lowest-cost location, where the cost is the routing number to achieve the placement.

The *incremental depth mapper* (incremental) guarantees placement of only the lowest-cost gate, instead of trying to place (almost) all gates in L, and incrementally improves the situation for the other gates.

The *greedy size mapper* (greedy size) is the same as the greedy depth mapper, except that we replace $\mathrm{rt}(\cdot)$ with $\mathrm{rs}(\cdot)$ in the objective function.

The *simple size mapper* (simple) places only the lowest-cost gate at its lowest-cost location.

The *extension size mapper* (extend) first places one gate using the same approach as the simple size mapper. We then try to only place another gate if it is cheaper to place now rather than in a later call to the mapper.

The *Qiskit-based size mapper* (qiskit) is based on Qiskit's circuit transformation. We slightly modify the circuit transformation routine in that it picks edges to SWAP one at a time instead of finding a maximal matching. Since this is a mapper, we only execute one iteration of the circuit transformation: for the first layer L. We return the final placement $\hat{p}$ that would be induced by executing all SWAPs found during the mapping process.

5 Results

We implement the circuit transformation introduced in Section 2.2.3 with a variety of mappers and appropriate permuters. We also implement the greedy swap transformation described in Section 2.2.2. We check the validity of our implementations by testing closeness in fidelity of the original output state and that of the transformed circuit for random input states of 11 qubits on random circuits [47] (described in the next section).

Evaluation Criteria. When testing the performance of these circuit transformations, each is allocated at most 8GB of RAM and 2 days to transform all circuits of a data point. For each data point we transform 10 random circuits and 1 quantum signal processing (QSP) circuit. We consider a 2-day runtime acceptable, given that classical computational resources are plentiful compared to quantum ones. We generate the data on a heterogeneous cluster with Intel Opteron 2354 and Intel Xeon X5560 processors.

The Cartesian permuter (see full arXiv version), the general size permuter (Section 3.2), and Qiskit's circuit transformation (Section 2.2.1) are randomized. We run multiple trials of these permuters and take the best result. Most of the time, trials produce equally good permutation circuits, although occasionally they deviate by a few SWAP gates. Our mappers run permuters $O(|L||E|)$ times, so we do only 4 trials to quickly remove any bad outliers. In contrast, our circuit transformation only directly runs a permuter once per layer of gates, so in this case we perform a slower 100 trials in an attempt to save a few SWAPs. We leave the number of trials for Qiskit's circuit transformation at its default of 40.

We test the performance of circuit transformations for the grid, $P_{n_1} \times P_{n_2}$, using the permuter for Cartesian graphs and for the modular architecture (Appendix A.2), $M_G(n_1, n_2)$, for $n_1, n_2 \in \mathbb{N}$. For an N-qubit circuit, we set $n_1 = n_2 = \lceil \sqrt{N} \rceil$ so that there are enough qubits in the architecture to contain the circuit. By Theorem 4, we know that taking $n_1 = n_2$ minimizes the routing time for our routing strategy among all grids with the same number of qubits. It is less clear how to balance parameters for the modular architecture since Corollary 8 does not depend on n_1 and n_2. For $n_1 \ll n_2$ or $n_2 \ll n_1$, less movement of qubits is needed, since many qubits are adjacent to one another. Thus, we take $n_1 = n_2$ in an attempt to consider a hard case. For some values of N, it may also be possible to find

parameters $n_1' \neq n_2'$ such that $N \leq n_1' n_2' < \lceil\sqrt{N}\rceil^2 = n_1 n_2$, requiring fewer qubits. However, this introduces unwanted size-dependent behavior in our results when $|n_1' - n_2'| \gg 0$ for one circuit size and $n_1' \approx n_2'$ for the next, so we find it preferable to fix $n_1 = n_2$.

We compare the transformed circuits in terms of their *weighted depth* and *weighted size*. For both trapped-ion and superconducting qubits, two-qubit gates typically have longer execution times and lower fidelities than single-qubit gates [31]. Even among two-qubit gates there is a difference between execution times. Assuming fast local unitaries, the SWAP gate has 1–3 times the interaction cost of a CNOT depending on the physical interactions used to realize the gates [51]. For simplicity, we assign unit cost for one-qubit gates, cost 10 for CNOT, and cost 30 for SWAP. We define the weighted size of a circuit as the sum of all gate weights and the weighted depth of a circuit as the maximum-weight path in the DAG of the circuit, where the weight of a path is the sum of the weights of the gates along it.

We consider two circuit families: random circuits and QSP circuits [32]. Random circuits have been proposed for quantum computational supremacy experiments on near-term quantum devices [9, 11]. Such proposals typically construct random circuits so that architecture constraints are automatically obeyed. For our purposes, random circuits provide a class of examples with little structure for circuit transformations to exploit, so we expect them to represent a hard case with large overhead. We generate a fixed set of 10 random circuits of depth 20 for various qubit counts. For each layer, we bin the qubits into pairs uniformly at random and assign each pair of qubits a Haar-random unitary from SU(4). Finally, we decompose each unitary into the smallest possible number of CNOT + SU(2) gates [49]. This random circuit generator is provided by Qiskit [8].

We consider QSP circuits for Hamiltonian simulation as an example of a realistic quantum algorithm. We use the unoptimized circuits provided in [14], decomposed into Z rotations, CNOT gates, and single-qubit Clifford gates. The QSP algorithm requires precise angles that turn out to be expensive to compute. Therefore, [14] uses randomized angles instead, giving a circuit that does not correctly implement the Hamiltonian simulation. Nevertheless, the circuit corresponds to an accurate implementation of QSP, up to rotation angles, and can be used for benchmarking resources. Furthermore, the circuit transformations we construct are unaffected by those angles. We only consider one pair of *phased iterates* of the QSP algorithm ($V^\dagger_{\phi_i+\pi} V_{\phi_{i-1}}$ as in [14, Eq. 31]). A full QSP circuit for the architecture can be constructed by iterating the mapped circuit of such phased iterates, a permutation circuit between iterations, state preparation, and state unpreparation. The cost of the transformed phased iterates dominates all other costs of the construction, so the total cost can be estimated by taking our result times the number of iterations.

The circuit transformations from Section 2.2.3 are constructed from a permuter and a mapper. We denote such circuit transformations by $\mathrm{tf}\colon \{\mathrm{d,s}\} \times M_p$, where M_p is the set of all mappers (referred to by their abbreviations, see Section 4), "d" denotes an appropriate depth permuter (Section 3.1), and "s" denotes the general size permuter (Section 3.2). For example, by tf(d,greedy depth) we denote a circuit transformation with a depth permuter for the architecture and the greedy depth mapper (Section 4).

Numerical Results. Figure 1 plots our results. We first consider the random circuit results. For the grid, we find that tf(d,incremental) shows much slower growth of weighted depth than circuit transformations that do not use depth-optimized permuters (Section 3.1). We also note that tf(d,qiskit) performs much better than Qiskit's circuit transformation (Section 2.2.1), suggesting that depth-optimized permuters can offer a significant advantage. On the modular graph, Qiskit's circuit transformation is much better at minimizing the weighted depth, but

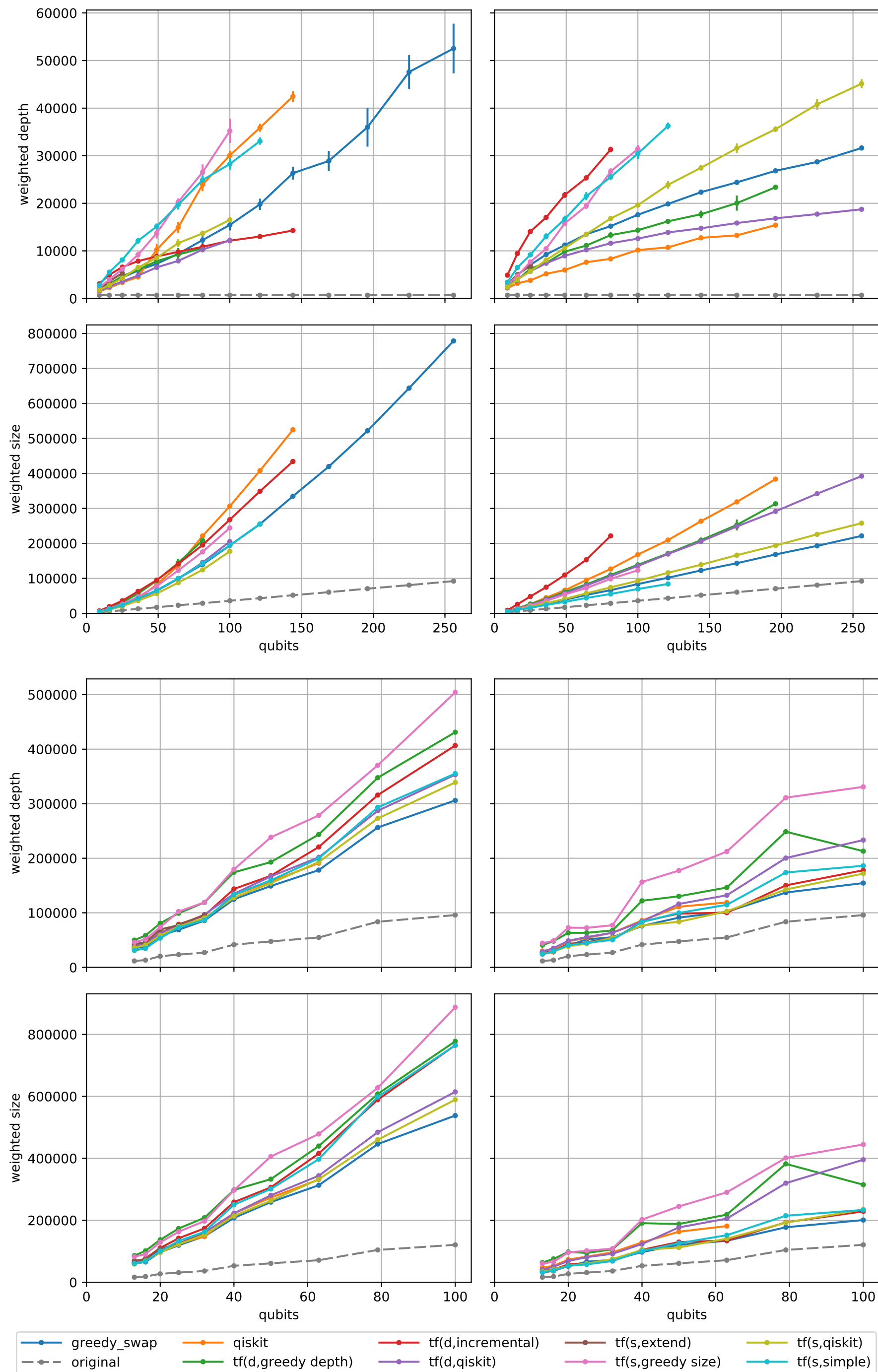

Figure 1 The weighted depth and weighted size for transformed random circuits (top two rows) and QSP circuits (bottom two rows) on the grid architecture (left column) and the modular architecture (right column). We generate fixed sets of 10 random circuits for increasing qubit counts and plot the mean and standard deviation for each data point. One QSP circuit is considered for each data point. The metrics for the original circuit are also given to make the overhead introduced in circuit transformations explicit; note that the original circuit does not respect the architecture constraints. The notation tf: $\{d,s\} \times M_p$ indicates a circuit transformation constructed from either an appropriate depth ("d") permuter or the size ("s") permuter and one of our mappers (Section 4).

tf(d,qiskit) starts closing the gap for larger sizes. Unfortunately, we do not know if tf(d,qiskit) performs better at larger sizes because Qiskit's circuit transformation is not fast enough to generate the relevant data. Up to 100 qubits tf(s,qiskit) achieves the best weighted size on grid architectures, and tf(s,simple) does best on modular architectures up to 121 qubits. For all sizes the greedy swap circuit transformation (Section 2.2.2) performs as one of the best at optimizing for weighted circuit size. The greedy swap circuit transformation is also able transform larger circuits within the time limit as expected from its lower time complexity.

For larger QSP circuits, the greedy circuit transformation (Section 2.2.2) is the clear winner in both weighted depth and weighted size, suggesting that it may be a good approach for practical quantum circuits. Surprisingly, tf(s,qiskit) also performs fairly well at minimizing the depth despite targeting the circuit size.

6 Future Work

We would like to better understand what circuit transformations work best for which architectures, quantum algorithms, and objective functions. We also would like to use the tools of Partial Routing via Matchings and Partial Token Swapping to establish bounds on the overhead of specific architectures. Ideally, we could use these tools and circuit transformations to design architectures that offer good performance subject to realistic hardware constraints and to compute realistic resource estimates for implementations of quantum algorithms.

There are many ways our methods could be improved. It would be interesting to know whether one can do better than just using swap gates to route qubits. Our mapper algorithms may also be improved by including some form of lookahead to consider later layers of the given circuit, or by specializing mappers to particular architectures.

Modeling the architecture as a simple graph loses information about the underlying hardware. For example, in the IBM system the architecture edges have directionality indicating the control and target of cnots. In implementations of the modular architecture, the interconnecting links are probably much noisier and slower than local operations. In general, gate costs and times can vary significantly across a hardware implementation and sometimes even vary over time [41]. Adapting to variable costs and keeping track of operations performed asynchronously is challenging but could be worthwhile for architectures that support a mixture of fast and slow operations.

Finally, we hope that future progress on the challenges addressed in this paper will be facilitated by a suitable set of benchmarks of large quantum circuits. We publicly make available and license our source code, benchmark circuits, and results (in `TSV` format) [47] and encourage others to do the same.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing - STOC '83*. ACM Press, 1983. doi:10.1145/800061.808726.
2. Noga Alon, F. R. K. Chung, and R. L. Graham. Routing Permutations on Graphs via Matchings. *SIAM Journal on Discrete Mathematics*, 7(3):513–530, May 1994. doi:10.1137/s0895480192236628.
3. Indranil Banerjee and Dana Richards. New Results on Routing via Matchings on Graphs. In *Fundamentals of Computation Theory*, pages 69–81. Springer Berlin Heidelberg, 2017. doi:10.1007/978-3-662-55751-8_7.

4 Indranil Banerjee, Dana Richards, and Igor Shinkar. Sorting Networks on Restricted Topologies. In *SOFSEM 2019: Theory and Practice of Computer Science*, pages 54–66. Springer International Publishing, 2019. doi:10.1007/978-3-030-10801-4_6.

5 Aniruddha Bapat, Zachary Eldredge, James R. Garrison, Abhinav Deshpande, Frederic T. Chong, and Alexey V. Gorshkov. Unitary entanglement construction in hierarchical networks. *Physical Review A*, 98(6), 2018. doi:10.1103/PhysRevA.98.062328.

6 L. Barrière, C. Dalfó, M. A. Fiol, and M. Mitjana. The generalized hierarchical product of graphs. *Discrete Mathematics*, 309(12):3871–3881, June 2009. doi:10.1016/j.disc.2008.10.028.

7 R. Beals, S. Brierley, O. Gray, A. W. Harrow, S. Kutin, N. Linden, D. Shepherd, and M. Stather. Efficient distributed quantum computing. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 469(2153), 2013. doi:10.1098/rspa.2012.0686.

8 Luciano Bello, Jim Challenger, Andrew Cross, Ismael Faro, Jay Gambetta, Juan Gomez, Ali Javadi-Abhari, Paco Martin, Diego Moreda, Jesus Perez, Erick Winston, and Chris Wood. Qiskit, 2017. URL: https://www.qiskit.org/.

9 Sergio Boixo, Sergei V. Isakov, Vadim N. Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J. Bremner, John M. Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595–600, 2018. doi:10.1038/s41567-018-0124-x.

10 Édouard Bonnet, Tillmann Miltzow, and Paweł Rzążewski. Complexity of Token Swapping and its Variants. *Algorithmica*, 80(9):2656–2682, October 2017. doi:10.1007/s00453-017-0387-0.

11 Adam Bouland, Bill Fefferman, Chinmay Nirkhe, and Umesh Vazirani. On the complexity and verification of quantum random circuit sampling. *Nature Physics*, October 2018. doi:10.1038/s41567-018-0318-2.

12 Teresa Brecht, Wolfgang Pfaff, Chen Wang, Yiwen Chu, Luigi Frunzio, Michel H. Devoret, and Robert J. Schoelkopf. Multilayer microwave integrated quantum circuits for scalable quantum computing. *npj Quantum Information*, 2(16002), 2016. doi:10.1038/npjqi.2016.2.

13 Stephen Brierley. Efficient Implementation of Quantum Circuits with Limited Qubit Interactions. *Quantum Info. Comput.*, 17(13-14):1096–1104, November 2017.

14 Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences*, 115(38):9456–9461, 2018. doi:10.1073/pnas.1801723115.

15 Byung-Soo Choi and Rodney van Meter. On the Effect of Quantum Interaction Distance on Quantum Addition Circuits. *ACM Journal on Emerging Technologies in Computing Systems*, 7(3):1–17, August 2011. doi:10.1145/2000502.2000504.

16 Byung-Soo Choi and Rodney van Meter. A $\Theta(\sqrt{N})$-depth Quantum Adder on the 2D NTC Quantum Computer Architecture. *J. Emerg. Technol. Comput. Syst.*, 8(3):24:1–24:22, August 2012. doi:10.1145/2287696.2287707.

17 Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962. doi:10.1145/367766.368168.

18 Austin G. Fowler, Simon J. Devitt, and Lloyd C. L. Hollenberg. Implementation of Shor's Algorithm on a Linear Nearest Neighbour Qubit Array. *Quant. Info. Comput. 4, 237-251 (2004)*, 2004.

19 Google Quantum AI Lab. A Preview of Bristlecone, Google's New Quantum Processor. URL: https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html.

20 Jeff Heckey, Shruti Patil, Ali JavadiAbhari, Adam Holmes, Daniel Kudrow, Kenneth R. Brown, Diana Franklin, Frederic T. Chong, and Margaret Martonosi. Compiler Management of Communication and Parallelism for Quantum Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*. ACM Press, 2015. doi:10.1145/2694344.2694357.

21 Steven Herbert. On the depth overhead incurred when running quantum algorithms on near-term quantum computers with limited qubit connectivity.

22 Yuichi Hirata, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima. An Efficient Method to Convert Arbitrary Quantum Circuits to Ones on a Linear Nearest Neighbor Architecture. In *2009 Third International Conference on Quantum, Nano and Micro Technologies*. IEEE, February 2009. doi:10.1109/icqnm.2009.25.

23 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, December 1973. doi:10.1137/0202019.

24 IBM Q Team. IBM Q Experience Devices, 2018. URL: https://quantumexperience.ng.bluemix.net/qx/devices.

25 Donald E. Knuth. *Networks for Sorting*, volume 3 of *The Art of Computer Programming*, pages 219–247. Addison-Wesley Professional, second edition, 1998.

26 Manfred Kunde. Optimal sorting on multi-dimensionally mesh-connected computers. In *STACS 87*, pages 408–419. Springer-Verlag, 1987. doi:10.1007/bfb0039623.

27 Samuel A. Kutin, David Petrie Moulton, and Lawren M. Smithline. Computation at a distance. *Chicago Journal of Theoretical Computer Science*, 13(1):1–17, 2007. doi:10.4086/cjtcs.2007.001.

28 L. Lao, B. van Wee, I. Ashraf, J. van Someren, N. Khammassi, K. Bertels, and C. G. Almudever. Mapping of lattice surgery-based quantum circuits on surface code architectures. *Quantum Science and Technology*, 4(1):015005, September 2018. doi:10.1088/2058-9565/aadd1a.

29 Gushu Li, Yufei Ding, and Yuan Xie. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices.

30 Chia-Chun Lin, Susmita Sur-Kolay, and Niraj K. Jha. PAQCS: Physical design-aware fault-tolerant quantum circuit synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(7):1221–1234, July 2015. doi:10.1109/tvlsi.2014.2337302.

31 Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences*, 114(13):3305–3310, March 2017. doi:10.1073/pnas.1618020114.

32 Guang Hao Low and Isaac L. Chuang. Optimal Hamiltonian Simulation by Quantum Signal Processing. *Physical Review Letters*, 118(1), January 2017. doi:10.1103/physrevlett.118.010501.

33 Aaron Lye, Robert Wille, and Rolf Drechsler. Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits. In *The 20th Asia and South Pacific Design Automation Conference*. IEEE, January 2015. doi:10.1109/aspdac.2015.7059001.

34 D. Maslov, S. M. Falconer, and M. Mosca. Quantum Circuit Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):752–763, 2008. doi:10.1109/tcad.2008.917562.

35 Dmitri Maslov. Linear depth stabilizer and quantum Fourier transformation circuits with no auxiliary qubits in finite-neighbor quantum architectures. *Physical Review A*, 76(5), November 2007. doi:10.1103/physreva.76.052310.

36 Tzvetan S. Metodi, Darshan D. Thaker, Andrew W. Cross, Frederic T. Chong, and Isaac L. Chuang. Scheduling physical operations in a quantum information processor. In Eric J. Donkor, Andrew R. Pirich, and Howard E. Brandt, editors, *Quantum Information and Computation IV*, page 6244. SPIE, 2006. doi:10.1117/12.666419.

37 Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}|E|)$ algoithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*. IEEE, October 1980. doi:10.1109/sfcs.1980.12.

38 Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. Approximation and Hardness for Token Swapping. In Piotr Sankowski and Christos Zaroliagis, editors, *Annual European Symposium on Algorithms*, volume 24, pages 185:1–185:15. Leibniz International Proceedings in Informatics (LIPIcs), 2016.

39 C. Monroe and J. Kim. Scaling the Ion Trap Quantum Processor. *Science*, 339(6124):1164–1169, 2013. doi:10.1126/science.1231298.

40 C. Monroe, R. Raussendorf, A. Ruthven, K. R. Brown, P. Maunz, L.-M. Duan, and J. Kim. Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects. *Physical Review A*, 89(2), February 2014. doi:10.1103/physreva.89.022317.

41 Prakash Murali, Jonathan M. Baker, Ali Javadi Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers, 2019.

42 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 1999. doi:10.1007/978-3-662-03811-6.

43 M. Pedram and A. Shafaei. Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures. *IEEE Circuits and Systems Magazine*, 16(2):62–74, 2016. doi:10.1109/MCAS.2016.2549950.

44 Rigetti. QPU Specifications, 2018. URL: https://www.rigetti.com/qpu.

45 David J. Rosenbaum. Optimal Quantum Circuits for Nearest-Neighbor Architectures. In Simone Severini and Fernando Brandao, editors, *8th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2013)*, volume 22 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 294–307, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TQC.2013.294.

46 Mehdi Saeedi, Robert Wille, and Rolf Drechsler. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing*, 10(3):355–377, June 2011. doi:10.1007/s11128-010-0201-2.

47 Eddie Schoute, Cem Unsal, and Andrew Childs. arct, 2019. URL: https://gitlab.umiacs.umd.edu/amchilds/arct.

48 Alireza Shafaei, Mehdi Saeedi, and Massoud Pedram. Qubit placement to minimize communication overhead in 2D quantum architectures. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, January 2014. doi:10.1109/aspdac.2014.6742940.

49 Farrokh Vatan and Colin Williams. Optimal quantum circuits for general two-qubit gates. *Physical Review A*, 69(3), 2004. doi:10.1103/physreva.69.032315.

50 Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Compiling quantum circuits to realistic hardware architectures using temporal planners. *Quantum Science and Technology*, 3(2):025004, February 2018. doi:10.1088/2058-9565/aaa331.

51 G. Vidal, K. Hammerer, and J. I. Cirac. Interaction Cost of Nonlocal Gates. *Physical Review Letters*, 88(23), 2002. doi:10.1103/physrevlett.88.237902.

52 Mark Whitney, Nemanja Isailovic, Yatish Patel, and John Kubiatowicz. Automated generation of layout and control for quantum circuits. In *Proceedings of the 4th international conference on Computing frontiers - CF '07*, pages 83–94. ACM Press, 2007. doi:10.1145/1242531.1242546.

53 Robert Wille, Oliver Keszocze, Marcel Walter, Patrick Rohrs, Anupam Chattopadhyay, and Rolf Drechsler. Look-ahead schemes for nearest neighbor optimization of 1D and 2D quantum circuits. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, January 2016. doi:10.1109/aspdac.2016.7428026.

54 Robert Wille, Aaron Lye, and Rolf Drechsler. Exact Reordering of Circuit Lines for Nearest Neighbor Quantum Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(12):1818–1831, December 2014. doi:10.1109/tcad.2014.2356463.

55 Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping Labeled Tokens on Graphs. In *Lecture Notes in Computer Science*, pages 364–375. Springer International Publishing, 2014. doi:10.1007/978-3-319-07890-8_31.

56 Louxin Zhang. Optimal Bounds for Matching Routing on Trees. *SIAM Journal on Discrete Mathematics*, 12(1):64–77, January 1999. doi:10.1137/s0895480197323159.

57 Alwin Zulehner, Alexandru Paler, and Robert Wille. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018. doi:10.1109/tcad.2018.2846658.

58 Alwin Zulehner and Robert Wille. Compiling SU(4) quantum circuits to IBM QX architectures. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference on - ASPDAC '19*, pages 185–190. ACM Press, 2019. doi:10.1145/3287624.3287704.

A Partial Permutations via Transpositions

Here we present proofs and statements that were omitted in the main text of Section 3.

A.1 Hierarchical Product

Proof of Lemma 3. Let $G = (U, V, E)$ be a bipartite multi-graph, with $U = V := [n_1]$ the left and right vertex sets, and the edge multi-set

$$E = \{(v_1, \pi(v)_1) \mid v \in \mathrm{dom}(\pi)\}\,. \tag{13}$$

Each vertex $k \in U$ belongs to at most d edges (k, l), for $l \in V$ and $k \neq l$, and each vertex $l' \in V$ belongs to at most d edges (k', l'), for $k' \in U$ and $k' \neq l'$. However, for any $k \in U$ there could be as many as n_2 edges (k, k). For all $k \in U$ we remove as many $(k, k) \in E$ as necessary to ensure that the maximum degree of any vertex in G is d.

We make G d-regular by repeating the following: If $\nexists k \in U$ with $\deg(k) < d$ we are done. Otherwise, such a k exists and $\exists k' \in V$ with $\deg(k') < d$, since

$$\sum_{k \in U} \deg(k) = \sum_{k' \in V} \deg(k')\,. \tag{14}$$

It follows that there exist vertices $u \in \mathcal{V}_k \setminus \mathrm{dom}(\pi)$ and $v \in \mathcal{V}_{k'} \setminus \mathrm{image}(\pi)$. For the purposes of this proof, we set $\pi(u) = v$, effectively adding an edge (k, k') to E.

Now we have modified π so that G is d-regular. By Hall's marriage theorem, there exists a perfect matching in G, and removing it results in a $(d-1)$-regular graph. We iterate this to find d distinct perfect matchings in G. Each edge $(k, k') \in E$ corresponds to some $v \in \mathcal{V}_k$ and $u \in \mathcal{V}_{k'}$, with $\pi(v) = u$. Therefore, each perfect matching corresponds to a set of representative vertices, R_i. Since all perfect matchings are distinct, and all $e \in E$ are covered by some matching, the Lemma follows. ◀

A.2 Modular Graph

Large-scale quantum computation may benefit from a modular design, with many interconnected subunits [39, 40, 12]. As a simple model of a modular quantum processor consisting of n_1 modules with n_2 qubits each, we consider the *modular graph* $M_G(n_1, n_2) := \Pi_{\vec{e}_1}(K_{n_1}, K_{n_2}) = (V, E)$, where $\vec{e}_i \in \{0,1\}^{n_2}$, for $i \in [n_2]$, is the ith standard basis vector. In this architecture, two qubits in the same module can be directly coupled, and any two modules can be coupled through their unique communicator qubits. With one minor modification to Theorem 4, we get the following bounds on the routing number of the modular graph.

▶ **Corollary 8.** *For $n_1, n_2 \in \mathbb{N}$ and $\pi\colon V \rightharpoonup V$, we have* $\deg(\pi) \leq \mathrm{rt}(M_G(n_1, n_2), \pi) \leq 3\deg(\pi) + 2$.

Proof. Directly applying Theorem 4 gives $\mathrm{rt}(M_G(n_1, n_2), \pi) \leq 4\deg(\pi) + 2$. However, only one token needs to be routed to the communicator vertex in every round of Algorithm 3.1. We note that this can be routed with one set of parallel transpositions (Section 3.1), saving us one matching every round. To show the lower bound, we apply Theorem 5 with $\mathrm{ham}(\vec{e}_1) = 1$. ◀

A.3 Partial Token Swapping

Proof of Theorem 7. The proof is very similar to [38, Theorem 7] with some minor modifications to account for no-token swaps. Let

$$S \coloneqq \sum_{v \in \mathrm{dom}(\pi)} d(v, \pi(v)), \tag{15}$$

and let s_u, s_h, s_{nt} be the number of unhappy, happy, and no-token swaps, respectively. We know that $\mathrm{rs}(G, \pi) \geq S/2$ since each swap can only reduce S by two. A no-token swap reduces S by one. A happy swap chain of length ℓ reduces S by $\ell + 1$. As such, over the course of the algorithm, $s_h + s_{nt} \leq S$. For an unhappy swap, the token that is swapped away from its destination must next be involved in a happy swap or a no-token swap, so $s_u \leq s_h + s_{nt}$. Thus, we get $s_u + s_h + s_{nt} \leq 4 \cdot \mathrm{rs}(G, \pi)$ as desired. ◀

B Specifics of Mappers

Here, we describe our mappers (Section 4) in full. Let M be a maximum matching in the architecture graph.

B.1 Greedy Depth Mapper

The *greedy depth mapper* iteratively places the highest-cost gate at its lowest-cost location, where cost is measured in terms of the routing number to achieve the placement. More precisely, we initialize the set of used vertices $U \coloneqq \emptyset$ and find a placement $p' \coloneqq \{q_1 \mapsto v_1, q_2 \mapsto v_2\}$ that attains the optimum

$$\max_{(q_1,q_2) \in \mathrm{tg}(L)} \min_{(v_1,v_2) \in M} \mathrm{rt}\big(G, (p \cup \{q_1 \mapsto v_1, q_2 \mapsto v_2\}) \circ \hat{p}^{-1}\big), \tag{16}$$

where we consider both orderings of edges from M, $(v, u), (u, v) \in M$, since edges are undirected. Then, we update $U \leftarrow U \cup \mathrm{dom}(p')$ and recompute M for the graph $G[V \setminus U]$ (recall (10)); we remove the gate associated to (q_1, q_2) from L; we set $p \leftarrow p \cup p'$; and we iterate until $\mathrm{tg}(L) = \emptyset$ or $M = \emptyset$. Finally, we return the placement p.

B.2 Incremental Depth Mapper

Instead of trying to place (almost) all gates in L, the *incremental depth mapper* guarantees placement of only the lowest-cost gate, as given by the routing number, and incrementally improves the situation for the other gates. Specifically, we first find a placement $p_{\min} \coloneqq \{q_1 \mapsto v_1, q_2 \mapsto v_2\}$ that attains the optimum

$$c'_{\min} \coloneqq \min_{(q_1,q_2) \in \mathrm{tg}(L)} \min_{(v_1,v_2) \in E} \mathrm{rt}(G, (p \cup \{q_1 \mapsto v_1, q_2 \mapsto v_2\}) \circ \hat{p}^{-1}), \tag{17}$$

where we consider both orderings of E, $(u, v), (v, u) \in E$. We set $p \leftarrow p_{\min}$ and define $U \coloneqq \{u, v\}$. Let $c_{\min} \coloneqq \max\{c'_{\min}, 1\}$.

We find a placement for the remaining two-qubit gates that (individually) does not exceed $c_{\min}$. We iterate in arbitrary order over $(q_1, q_2) \in \mathrm{tg}(L)$ and do the following: For $i \in [2]$, we construct a set of eligible vertices

$$U_i \coloneqq \big\{v \in V \setminus U \mid \mathrm{rt}\big(G, (p \cup \{q_i \mapsto v\}) \circ \hat{p}^{-1}\big) \leq c_{\min}\big\}. \tag{18}$$

Now we try to find $v_1^* \neq v_2^*$ as $(v_1^*, v_2^*) \coloneqq \arg\min_{(v_1,v_2) \in U_1 \times U_2} d(v_1, v_2)$. If such (v_1^*, v_2^*) does not exist, we do not include q_1 and q_2 in p; otherwise, we set $p \leftarrow p \cup \{q_1 \mapsto v_1^*, q_2 \mapsto v_2^*\}$ and update $U \leftarrow U \cup \{v_1^*, v_2^*\}$. After iterating over all gates in $\mathrm{tg}(L)$, we return p.

B.3 Greedy Size Mapper

The *greedy size mapper* the same as the greedy depth mapper, except that we replace $\mathrm{rt}(\cdot)$ with $\mathrm{rs}(\cdot)$ in (16).

B.4 Simple Size Mapper

The *simple size mapper* places only the lowest-cost gate at its lowest-cost location. More precisely, we find a placement $p := \{q_1 \mapsto v_1, q_2 \mapsto v_2\}$ that attains the optimum

$$\min_{(q_1,q_2)\in \mathrm{tg}(L)} \min_{(v_1,v_2)\in E} \mathrm{rs}\big(G, (p \cup \{q_1 \mapsto v_1, q_2 \mapsto v_2\}) \circ \hat{p}^{-1}\big) \tag{19}$$

where we consider all orderings of the edges of E, and return p. Note that we have replaced $\mathrm{rt}(\cdot)$ with $\mathrm{rs}(\cdot)$ in (17).

B.5 Extension Size Mapper

The *extension size mapper* first finds an initial placement p using (19). Let $c'_{\min}$ be the value attained at the optimum for (19). After finding the initial placement, we try to only place another gate if it is cheaper to place now rather than in a later call to the mapper.

Specifically, for the current p and $\hat{p}$, we define $\hat{p}' : Q \to V$ as the placement after performing the permutation circuit constructed from transpositions achieving $\mathrm{rs}(G, p \circ \hat{p}^{-1})$. Let $U := \emptyset$. Now we define a heuristic for the number of saved transpositions, $s\colon Q \times Q \to \mathbb{N}$, as

$$\begin{aligned} s(q_1, q_2) := {} & \mathrm{rs}\big(G, p \circ \hat{p}^{-1}\big) + \min_{(v_1,v_2)\in E} \mathrm{rs}\Big(G, \{q_1 \mapsto v_1, q_2 \mapsto v_2\} \circ (\hat{p}')^{-1}\Big) \\ & - \min_{(u_1,u_2)\in E'} \mathrm{rs}\big(G, (p \cup \{q_1 \mapsto u_1, q_2 \mapsto u_2\}) \circ \hat{p}^{-1}\big)\,, \end{aligned} \tag{20}$$

where E' is the edge set of $G[V \setminus U]$ and we consider all orderings of the edges of E and E'.

The extension size mapper iterates the following. We find the gate $(q_1^*, q_2^*) \in \mathrm{tg}(L)$ attaining $s_{\max} := \max_{(q_1,q_2)\in \mathrm{tg}(L)} s(q_1, q_2)$, and let $(u_1^*, u_2^*) \in E'$ be the edge attaining $s_{\max}$ as given by (20). If $s_{\max} \geq 0$, we set $p \leftarrow p \cup \{q_1^* \mapsto u_1^*, q_2^* \mapsto u_2^*\}$, remove the gate (q_1^*, q_2^*) from L, update $U \leftarrow U \cup \{v_1^*, v_2^*\}$, and iterate; otherwise, we stop and return p.

B.6 Qiskit-based Mapper

Finally, we implement a mapper that is based on Qiskit's circuit transformation (described in Section 2.2.1). Since this is a mapper, we only execute one iteration of the circuit transformation: for the first layer L. We also do not modify the output circuit, but instead return the final $\hat{p}$ that would be induced by executing all SWAPs found during the mapping process.

We make three changes to Qiskit's circuit transformation. The first is that when minimizing S, instead of choosing a maximal set of SWAPs in every iteration, we choose only one SWAP along an edge $e \in E$ that minimizes S. The second is that the upper bound on the number of iterations is raised to $|V|^2$, since we only apply one SWAP per iteration. Thirdly, if no trial is successful, we fall back to the simple size mapper and return the placement it finds, which places only one gate in this iteration.

C Time Complexity Analysis

To show that our proposed algorithms have polynomial worst case time complexity, we compute time complexities of our circuit transformations, permuters, and mappers explicitly.

C.1 Circuit Transformations

Greedy Swap Circuit Transformation. We ignore the initial placement since it is insignificant for large circuits. A gate from L is executed in at most $\operatorname{diam}(G)$ iterations, where $\operatorname{diam}(G)$ is the diameter of G. In every iteration, $O(|E|)$ edges are checked to determine gates that can be executed and SWAPs that will decrease R. Therefore, the total time complexity is $O(|C||E| \operatorname{diam}(G))$, where $|C|$ denotes the size of circuit C. There is a tighter bound in terms of output circuit C' since every iteration creates a layer in the transformed circuit, the complexity is $O(\operatorname{depth}(C')|E|)$, where $\operatorname{depth}(C')$ denotes the circuit depth of C'.

Specialized Circuit Transformations (Section 2.2.3). We again ignore the time complexity of computing the initial placement. Let t_m be an upper bound on the time complexity of the mapper, and let t_p be an upper bound on the time complexity of the permuter. Computing $p \circ \hat{p}^{-1}$ takes time $O(|V|)$. The number of transpositions produced by the permuter is at most t_p, so executing the associated SWAPs takes time $O(t_p)$. Only one gate from L may be executed every iteration so we upper bound the number of iterations by $|C|$. We find a time complexity of $O(|C|\,(t_m + |V| + t_p))$. Clearly, if $t_p, t_m \in \operatorname{poly}(|C|, |V|)$ then our circuit transformation is also poly-time as desired.

C.2 Permuters

We show that the permuters are polynomial-time in the input size.

Complete Graph. The time complexity of the ROUTING VIA MATCHINGS algorithm for K_n is $O(n)$ [2]. The other operations described above also take time $O(n)$, so we get a time complexity of $O(n)$ for the complete graph permuter.

Path Graph. Constructing the completion $\hat{\pi}$ takes time $O(|V|)$. The total complexity for running the path permuter is $O(|V|^2)$, where the time complexity of the ROUTING VIA MATCHINGS algorithm [2] dominates the construction of $\hat{\pi}$.

Hierarchical Product. Let t_1 and t_2 upper bound the time complexity of algorithms for PARTIAL ROUTING VIA MATCHINGS on G_1 and G_2, respectively. We first find $\deg(\pi)$ distinct sets of representative vertices by Lemma 3. The time to find one set of representative vertices is dominated by the time to find the maximum bipartite matching, $O(n_1^{2.5})$ [23]. Then, for $\lceil \deg(\pi)/\operatorname{ham}(\vec{v}) \rceil$ iterations, we route on all copies of G_2 and then G_1 in parallel. Overall, we get a time complexity of

$$O\left(\deg(\pi) \cdot n_1^{2.5} + \left\lceil \frac{\deg(\pi)}{\operatorname{ham}(\vec{v})} \right\rceil (\operatorname{ham}(\vec{v})t_1 + n_1 t_2) + n_1 t_2\right). \tag{21}$$

Modular Architecture. We evaluate the time complexity of this permuter using Equation (21). We have $t_1 = O(n_1)$ and $t_2 = O(n_2)$, giving an overall time complexity of $O(dn_1^{2.5} + n_1 n_2)$, where we noted that $t_2 = O(1)$ while doing the $\deg(\pi)$ rounds of routing.

Approximation Algorithm for Partial Token Swapping. Computing an all-to-all distance matrix takes time $\Theta(|V|^3)$ using the Floyd-Warshall algorithm [17], but this cost needs only to be incurred once for a graph so we do not include it. A happy or unhappy swap can be found in time $O(|E|)$ by finding cycles in an auxiliary directed graph [38]. Similarly, finding no-token swaps has time complexity $O(|E|)$. Therefore, we get a total time complexity of $O(S|E|) \leq O(|V|^2|E|)$.

C.3 Mappers

We give polynomially-sized upper bounds on the time complexity of the mappers as a function of the time complexity of the permuter, t_p.

Greedy Depth Mapper. We perform at most $\min\{|L|, |M|\}$ iterations to place gates. In each iteration, we find a p' according to (16) in time $O(|L||M|t_p)$. Thus, the time complexity for one call of the mapper is

$$O\Big(\min\{|L|, |M|\} \left(|L||M|t_p + \sqrt{|V|}|E|\right)\Big), \tag{22}$$

where $O(\sqrt{|V|}|E|)$ is the complexity of computing a maximum matching [37].

Incremental Depth Mapper. We get $O(|L| \left(|E|t_p + |V|t_p + |V|^2\right))$ for the time complexity of the incremental depth mapper. This assumes we have access to the all-pairs distance matrix of the architecture graph, which can be precomputed in time $\Theta(|V|^3)$ [17] (independent of the input circuit).

Simple Size Mapper. The time complexity of the simple size mapper is $O(|L||E|t_p)$.

Extension Size Mapper. Calculating $s(q_1, q_2)$ for any $q_1, q_2 \in Q$ takes time $O(|E|t_p)$. Therefore, the total time complexity of the extension size mapper is $O(|L|^2|E|t_p)$.

Qiskit-based Mapper. First, we compute an all-to-all distance matrix in time $\Theta(|V|^3)$ [17], which we ignore since it is a one-time cost dependent only on the architecture. Each of the $O(|V|^2)$ iterations has a time complexity of $O(|E||L|)$. Thus, the Qiskit mapper has time complexity $O(|V|^2|E||L|)$.