Optimized Quantum Circuit Generation with SPIRAL

Scott Mionis
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania
smionis@andrew.cmu.edu

Franz Franchetti
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania
franzf@andrew.cmu.edu

Jason Larkin
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania
jmlarkin@sei.cmu.edu

***Abstract*—Quantum computers have been at the bleeding edge of computing technology for nearly 40 years and research continues to progress due to their immense promise. However, despite hardware and algorithm breakthroughs, the software infrastructure that compiles programs for these devices requires further development and has traditionally been under-emphasized. In this work, we present a novel approach to compiling more efficient quantum programs. We capture the input algorithm as a high-level mathematical transform and generate a multitude of architecture-compliant programs directly from that specification; this is achieved by casting the problem as a sparse matrix factorization task and recursively searching over a host of applicable divide-and-conquer decomposition rules. This approach allows us to leverage high-level symmetries of the target transform to explore global rewrites and select the best factorization from the top-down, a task that is nearly impossible given only a program stream. We implement the proposed framework with SPIRAL [4], a code generation platform founded on the GAP [6] computer algebra system; we ultimately demonstrate that SPIRAL is a viable supplemental tool for future quantum frameworks.**

***Index Terms*—compilers, Fourier transform, SPIRAL, quantum computing, circuit optimization, code generation**

Introduction. Unearthing the full taxonomy of applications benefiting from quantum technology has only just begun. However, while both this nascent quantum theory and the hardware supporting it are developing rapidly, it fast outpaces the true capabilities of current software infrastructure. Compiling quantum circuits is non-trivial for three reasons. First, current-era devices maintain only sparse connectivity between qubits. Since many operations require the operand qubits to be physically connected in the target device, this means that quantum programs assuming dense connectivity must often be routed onto the hardware. If done naïvely, this process often requires the compiler to insert an overwhelming number of data movement operations. Additionally, these data movement operations alone can violate the practicability of a program since quantum states are fragile and degrade rapidly if the critical path is too long. Finally, the problem of minimizing such data movement operations can be reduced to the maximum common edge subgraph problem [5], and thus is NP-complete. Current compilation technologies, such as IBM's Qiskit platform [1], take in programs expressed at the *circuit* level, a low-level and imperative representation of quantum programs. These technologies then map an input circuit assuming full connectivity to the true geometry of the target device by inserting SWAP operations through a localized, peephole heuristic algorithm. These algorithms are often insufficient, and the number of SWAP operations can grow exponentially with circuit size. The inherent problem with this approach lies in the input representation; mapping a *specific* circuit onto an architecture is a fundamentally more restricted version of the problem we actually wish to solve. Rather, the compiler should find the most efficient circuit, that when run on the target architecture, produces the same *result* as the input circuit.

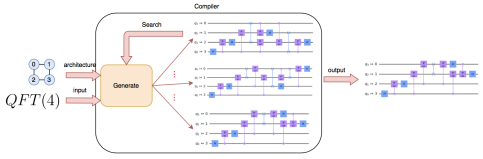

Fig. 1. Proposed generative approach. Example shown is for a 4-qubit quantum Fourier transform (QFT).

To address the aforementioned motivations, we will take, as input, a declarative representation of the target algorithm as expressed as a high-level symbolic transform object. We will use SPIRAL to decompose this transform in a variety of ways, leveraging algorithm-specific decomposition identities such as the Cooley-Tukey [3] rule for Fourier transforms. The sparse factorizations of our symbolic transform uniquely map to circuits that implement the desired algorithm. We can then search over this global space of possible factorizations with algorithm-informed heuristics and choose the best with respect to some measure, in our case, SWAP count.

Circuit Generation as Matrix Factorization. The state of an N-qubit system for arbitrary N can be represented by a 2^N-length complex column vector. Every quantum program (excluding those with intermediate measurements) can perform computation solely by rotating this state vector in a length-preserving manner; the effect of an N-qubit quantum program is thus succinctly described by a $2^N \times 2^N$ unitary matrix. Quantum devices typically support different sets of basic linear algebra operators, called quantum *gates*, as atomic operations. These gates, when combined into a circuit, represent a sparse matrix factorization of the overall transformation; the global transform can be derived by composing the matrix definitions

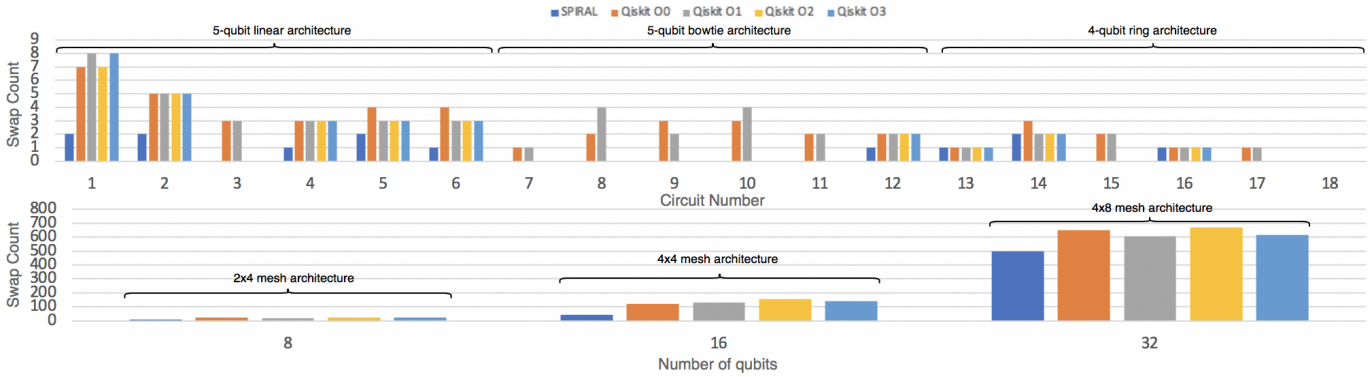

Fig. 2. Swap minimization comparison between SPIRAL and Qiskit on low-level benchmark circuits [top], QFT [bottom]. Target architectures are annotated.

of parallel and sequential gates with the tensor and matrix product respectively. Unfortunately, since the overall transform matrix is exponential in size with respect to N, it is typically impossible to reconstruct from circuit representation, and hence no high-level scheduling decisions can be informed by it. We address this in SPIRAL by starting with a symbolic representation of the desired transformation and searching over various ways of breaking it down into a gate-level expression, a process that allows us to leverage the definition of our algorithm to generate intelligent schedules.

SPIRAL Quantum Compiler. Formalizing the above, we use SPIRAL to solve the following minimization problem.

$$\text{circuit}_{\text{opt}}(Mat) = \underset{m \in \text{ Factorizations}(Mat)}{\arg\min} \text{Cost}(m)$$

Mat is the desired $2^N \times 2^N$ transform matrix and Factorizations returns a set of valid decompositions of Mat into gate-level expressions. Cost is a minimization metric, in our case, the number of SWAP operations. We capture Mat as a composition of symbolic transform kernels such as $\text{qHT}(n)$ and $\text{qFT}(n)$, an n-qubit Walsh-Hadamard transform or QFT respectively. SPIRAL generates factorizations by applying a rule-based decomposition procedure to recursively reduce the input to a gate-level expression, after which this sparse factorization can be unparsed as quantum assembly (QASM) [2]. For example, the Hadamard transform can be easily decomposed into the tensor product of 2×2 Hadamard gates with the following rules.

$$\text{qHT}(nm) \quad \mapsto \quad \text{qHT}(n) \otimes \text{qHT}(m) \tag{1}$$

$$\text{qHT}(2) \quad \mapsto \quad \text{H} \tag{2}$$

Decomposing the top-level object in a manner that minimally satisfies connectivity constraints involves not only searching over algorithmic decompositions, however, but also over the possible placements of these kernels onto specific hardware qubits in connectivity map of the hardware, a process we call *embedding*. Exploring various embeddings equates to introducing permutation matrices into our factorization, and can similarly be enumerated as breakdown rules. The resulting expression from any particular partial order of rule applications, or rule tree, can be simplified via a set of semantics-preserving rewrite rules and then evaluated based on cost. SPIRAL finds the best of these expressions, and hence the best output circuit, by implementing a dynamic programming search procedure to find the best rule tree in a manner similar to constructive proof search. SPIRAL has access to both a high-level definition of the transform and the architecture geometry; it uses these to heuristically prune the search space.

Results. We implemented the aforementioned system in SPIRAL and evaluated it on a series of test programs with stringent connectivity requirements. These programs were specified to SPIRAL at a low level of abstraction, our goal being to show that SPIRAL generalizes to arbitrary programs and still produces circuits that require fewer SWAP operations than those produced by Qiskit for the same input. As seen in Figure 2, SPIRAL performs competitively on these programs as it has access to a strictly larger search space than does Qiskit, and we would expect these results to generalize to a larger set of benchmarks. However, since these inputs are already sparse formulas, much of the SPIRAL compilation framework is bypassed. Therefore, a more exciting application of SPIRAL beyond these small test programs lies in our ability to apply algorithm-specific placement heuristics for *high-level* operators. We specifically looked at the QFT and developed a series of initial heuristics that inform SPIRAL breakdown by adapting various parallel Fourier transform algorithms. We show distinct savings when compared to Qiskit on common mesh architectures; these promising initial results indicate that similar approaches could be valuable.

References

[1] H. Abraham et al., "Qiskit: An Open-source Framework for Quantum Computing," 2019. doi:10.5281/zenodo.2562110

[2] A. W. Cross et al., "OpenQASM 3: A broader and deeper quantum assembly language" [arxiv:2104.14722].

[3] F. Franchetti and M. Püschel, "Fast Fourier Transform," in *Encyclopedia of Parallel Computing*, D. A. Padua (Editor).

[4] F. Franchetti and M. Püschel et al. "Spiral," in *Encyclopedia of Parallel Computing*, 2011, D. A. Padua (Editor).

[5] G. G. Guerreschi, "Scheduler of quantum circuits based on dynamical pattern improvement and its application to hardware design," 2019, arXiv:1912.00035v1 [quant-ph].

[6] Martin Schönert et al. GAP – Groups, Algorithms, and Programming – version 3 release 4 patchlevel 4. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.