Quantum Circuit Optimization with SPIRAL: A First Look

Scott Mionis
Electrical and Computer Engineering
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*—Compilation and optimization of quantum circuits is an integral part of the quantum computing toolchain. In many Noisy Intermediate-Scale Quantum (NISQ) devices, only loose connectivity between qubits is maintained, meaning a valid quantum circuit often requires swapping physical qubits in order to satisfy adjacency requirements. Optimizing circuits to minimize such swaps, as well as additional metrics like gate count and circuit depth, is imperative towards utilizing the quantum hardware of both today and the near future. In this work, we leverage SPIRAL, a code generation system for linear transforms built on GAP's computer algebra system, and present an application of such a system towards optimizing quantum circuits. SPIRAL natively understands tensor products, complex matrices and symbolic matrices, and the proven decomposition and rewriting capabilities are uniquely predisposed to optimize quantum circuits. Specifically, by defining the optimization problem in terms of SPIRAL's breakdown and rewriting system, we construct a search problem that can be solved with techniquies like dynamic programming. The optimal circuit can then translated to QASM code, where it is executable on a real quantum device. We demonstrate that the power of SPIRAL could provide a valuable tool for future software frameworks.**

***Index Terms*—quantum, SPIRAL, optimization, quantum circuit, future quantum frameworks**

Introduction. SPIRAL, built on the GAP computer algebra system, starts with a formal framework that expresses algorithms and hardware architecture specifications in an internal representation [1]. Using architectural directives, SPIRAL then finds efficient computational kernels for a specified algorithm by running a search task. To achieve this goal, SPIRAL captures mathematical identities for each object in it's high-level algorithm syntax, expressing them in the form of breakdown rules. During the process of decomposing algorithms into more basic components, these breakdown rules are applied recursively; a specific sequence of rule applications, or rule tree, is only one of many paths from high-level representation to lower-level implementation. To simplify these lower-level expressions, an extensive rewriting system also exists to perform substitutions. For the application of SPIRAL to the quantum domain that is explored in this work, we leverage a subset of this existing framework in order to similarly convert high-level algorithm descriptions into compositions of gates. These expressions can then in turn be unparsed as QASM [6] and executed on a real quantum device.

Defining quantum notation. Quantum circuits are traditionally expressed as a matrix and tensor product of unitary matrices which represent the basic gates in the quantum system. Additionally, The matrix definitions of most gates can be viewed as complex rotation or permutation matrices. SPIRAL internally supports this notation, and many rules for these mathematical objects can be immediately transferred from SPIRAL's other domains, such as in signal processing [2]. To capture quantum formulas in this framework, we define both high-level *transform* objects and low-level *gate* objects, with breakdown rules to translate downwards. Transform objects represent logical operations that may comprise of multiple basic gates, such as the n-qubit hadamard transform $\mathrm{qHT}(n)$. Gate objects are the set of basic implementable gates for the target hardware, such as the hadamard gate qH, CNOT gate $\mathrm{CNOT}(i, j)$, or identity I. Defining the decomposition of a transform such as the $\mathrm{qHT}(n)$ into basic qH blocks can be done via the following rules expressed in SPIRAL, Where $\otimes$ is the tensor or kronecker product:

$$\mathrm{qHT}(nm) = \mathrm{qHT}(n) \otimes \mathrm{qHT}(m) \quad (1)$$

$$\mathrm{qHT}(1) = \mathrm{qH} \quad (2)$$

Capturing Quantum Circuits. Given the quantum notation implemented in SPIRAL, a circuit can be expressed as a series of transform objects applied to specific qubits. Mathematically, this translates to a matrix product of $2^n \times 2^n$ matrices, n being the number of qubits, and each matrix representing a single layer of the circuit. To express this in a format that can be expanded by SPIRAL's breakdown rules, we proceed to define a $\mathrm{qEmbed}(\ell, \mathrm{arch}, t)$ transform object to represent the $2^n \times 2^n$ matrix formed by applying transform object t to the qubits in list ℓ, given qubit connectivity graph arch. Decomposing this object can be done many different ways. Given the connectivity requirements that t imposes on ℓ, we can position the qubits in any physical orientation satisfying those constraints. Therefore, using arch, we can find any viably-connected subcomponent of the physical toplogy. We then position our logical qubits into the chosen orientation before applying t. Finding the globally optimal orientation is nontrivial, and can be found by performing an intelligent search. If t is recursively defined as the matrix product of multiple $\mathrm{qEmbed}(\ell, \mathrm{arch}, t)$ objects, the same process is followed over the sub-topology; arch is now the

original topology graph pruned of any physical qubits outside of its scope.

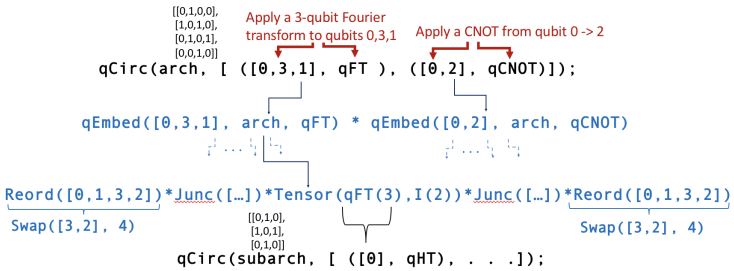

Fig. 1. Decomposition of Transform Objects

To string together the matrix product of several $\mathrm{qEmbed}(\ell, \mathrm{arch}, t)$ objects we define a wrapper object, qCirc, which is the top-level input to SPIRAL.

Embedding a Transform. Recursively decomposing a qEmbed is done in three nested stages. Given a desired logical to physical qubit layout, A *Reorder* step implicitly represents the swaps needed to create that mapping. A *Junction* step then freely reorders the logical argument qubits into canonically-ordered positions 0 through $n-1$, where n is the size of the embedded transform. We can then apply the given transform to the upper n qubits and the identity to all others, before reversing the Junction and Reorder steps. Reorder objects are kept adjacent so that they can be cancelled in the rewrite stage.

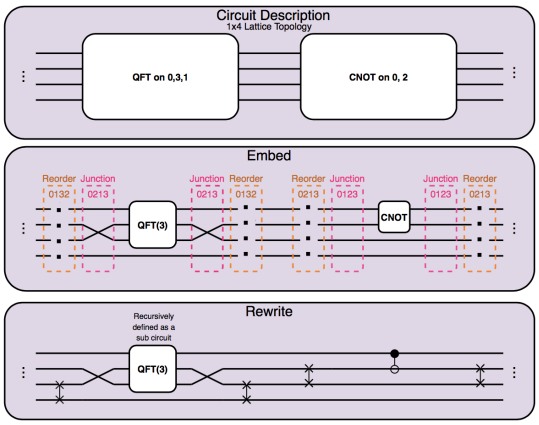

Fig. 2. Embedding a circuit, circuit view

Quantum Circuit Reduction. After an expression is completely simplified to gate objects, we can further reduce the circuit using SPIRAL's rewriting system [5]. These rewrite rules can perform direct or conditional substitutions over associative operators such as matrix products or tensor products. We specifically combine any adjacent Reorder stages and simplify them before converting them to CNOT objects. For example, $\mathrm{Reorder}([0,2,3,1]) \circ \mathrm{Reorder}([0,3,2,1])$ can be reduced to a cheaper $\mathrm{Reorder}([0,1,3,2])$, and a $\mathrm{Reorder}([0,2,3,1]) \circ \mathrm{Reorder}([0,3,1,2])$ yields an Identity transformation. Architecture-induced SWAPs can be minimized by finding valid Reorder steps that cancel the most effectively.

Circuit Optimization. Finding the optimal circuit is now a search problem over the space of possible circuits using a defined cost function. It can be summarized symbolically as:

$$rt_{opt}(arch) = \arg\min_{rt,arch} \mathrm{Cost}(\mathrm{Rewrite}(\mathrm{Breakdown}(rt, circ, arch)))$$

Where $\mathrm{Breakdown}(rt, circ, arch)$ applies rule tree rt to circuit transform object $circ$ using the qubit adjacency matrix $arch$, $\mathrm{Rewrite}(c)$ applies rewrite rules to simplify the expression c, and $\mathrm{Cost}(t)$ is the cost function. With this formulation, SPIRAL's DP [3] function can perform a Dynamic Programming search over the circuit space, and return the optimal circuit defined as $\mathrm{Rewrite}(\mathrm{Breakdown}(rt_{opt}, circ, arch))$. This optimal circuit maps directly to executable QASM code, with only a slight difference in syntax. We can either fix the $arch$ parameter, or search over row and column permutations to explore global qubit reorderings.

Results. We tested SPIRAL against IBM's Quiskit optimizer [4] for several small test circuits, with cost defined as the number of CNOT gates in the final circuit. To verify our results, we transferred SPIRAL-generated circuits to QASM code and executed on IBM's Tenerife and Bogota devices. As seen below, SPIRAL can compete on CNOT count with level 1 and 2 optimization for these test circuits despite running an unstructured search, and without incorporation of advanced gate-cancellation rules [7]. We believe this is a strong indication that taking the system beyond proof-of-concept could empower a valuable tool, given the historically strong scalability of the SPIRAL system in other transform applications.

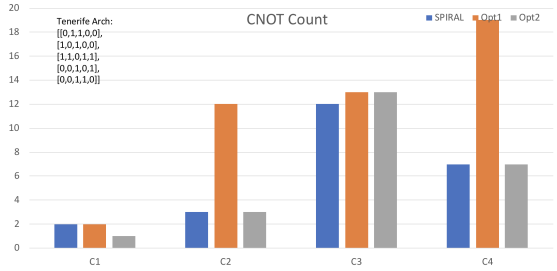

Fig. 3. Comparison of SPIRAL with Quiskit Optimization on several trial circuits and IBM Tenerife Topology

Future Work. For larger numbers of qubits, the space is far too large for naïve search. However, having an implicit representation for all possible circuits is powerful when multiple permutations are non-trivial. For large qubit sizes we plan to further structure the search problem with breakdown-rule heuristics, pruning unpromising branches early in the rule tree. By also capturing additional gate simplification techniques as rewrite rules, we are confident that SPIRAL's capabilites could prove valuable as a future supplement to the existing taxonomy of quantum optimization software.

REFERENCES

[1] F. Franchetti, et al. SPIRAL: Extreme Performance Portability, Proceedings of the IEEE, Vol. 106, No. 11, 2018. Special Issue on From High Level Specification to High Performance Code

[2] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, N. Rizzolo SPIRAL: Code Generation for DSP Transforms Proceedings of the IEEE Special Issue on "Program Generation, Optimization, and Adaptation," Vol. 93, No. 2, 2005, pages 232-275.

[3] F. Franchetti and M. Püschel Generating SIMD Vectorized Permutations Proceedings of International Conference on Compiler Construction (CC) 2008

[4] Gadi Aleksandrowicz, Thomas Alexander, et al. Qiskit: An Open-source Framework for Quantum Computing, 2019.

[5] F. Franchetti, Y. Voronenko, M. Püschel A Rewriting System for the Vectorization of Signal Transforms Proceedings High Performance Computing for Computational Science (VECPAR) 2006, LNCS 4395, pages 363-377

[6] Andrew W. Cross, Lev S. Bishop et al, "Open Quantum Assembly Language," arXiv preprint arXiv:1707.03429, 2017.

[7] E. Reiffel and W. Polak, Quantum Computing A Gentle Introduction. Cambridge, MA: MIT Press, 2011