Optimized Quantum Circuit Generation with SPIRAL

Scott Mionis

CMU-CS-21-114
May 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Franz Franchetti, Chair
Seth Goldstein

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Copyright © 2021 Scott Mionis

Keywords: compilers, Fourier transform, SPIRAL, quantum computing, circuit optimization, code generation

Dedicated to whomever finds something of significance in this work and manages to employ it successfully

Abstract

Quantum computers [55] have been at the bleeding edge of computing technology for nearly 40 years, and while there are several barriers that prevent their immediate utility, research in this area continues to progress due to their immense promise. Specifically, these systems have been theorized to decrease the complexity of several important problems, most tangibly violating the security of RSA encryption [56] and possibly the extended Church-Turing thesis [8]. In an effort to harness the revolutionary potential of these systems, research has largely focused around the physical construction of at-scale quantum devices. However, the software infrastructure that compiles programs for these devices also requires further development before the benefits of the quantum era can be realized; this area has been traditionally under-emphasized.

In the near-term, Noisy Intermediate-Scale Quantum (NISQ) devices maintain only sparse connectivity between qubits, meaning that quantum programs assuming dense connectivity must be efficiently routed onto the hardware. If done naïvely, this process often requires the compiler to insert an overwhelming number of data movement operations; these alone can violate the practicability of the program since quantum states are fragile and degrade rapidly if the critical path is too long. Existing approaches have made great strides in making this process more efficient, but have failed to capitalize on relevant advancements in the classical domain and are plagued by a representation mismatch that limits the scope of program transformations that can be applied.

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 after generating a multitude of architecture-compliant programs directly from that specification, we apply traditional search techniques to select the best output. This approach allows us to produce shorter quantum programs as we can leverage high-level symmetries of the target transform to perform global rewrites; this task is nearly impossible given only a program stream. To implement the proposed framework we leverage SPIRAL [26][24][54], a code generation platform built on the GAP [57] computer algebra system, and restate quantum computing in terms of pure linear algebra such that we can treat this seemingly domain-specific problem as a generic matrix factorization task. We ultimately demonstrate that SPIRAL is a viable supplemental tool for future quantum software frameworks, and provides tangible benefits when used to compile symmetric algorithms like the Fourier transform.

Acknowledgments

Thank you to my family and extended family, who may not be the principal beneficiaries of this work but are enthusiastic supporters of it nonetheless. Additional thanks are due to my research advisors, specifically Dr. Franz Franchetti and Dr. Seth Goldstein, as well as Dr. Jason Larkin from the Software Engineering Institute who has advised me on this project. I am additionally indebted to all the faculty and students of Carnegie Mellon University whom I have had the privilege of interacting with during my studies.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 The Quantum Optimization Problem . . . 1
  - 1.1.2 Existing Results . . . 3
  - 1.1.3 Challenges to Overcoming Deficiencies . . . 4
- 1.2 Overview . . . 7
- 1.3 Contributions . . . 9
- 1.4 Organization . . . 10

2 A Primer on Quantum Information Science **11**
- 2.1 What are Quantum Computers? . . . 11
- 2.2 Qubit Fundamentals . . . 13
  - 2.2.1 One Qubit . . . 13
  - 2.2.2 Multiple Qubits . . . 16
- 2.3 Quantum Circuits . . . 18
  - 2.3.1 Circuits as Matrix Factorizations . . . 19
  - 2.3.2 Connectivity . . . 21
- 2.4 Problem Formulation . . . 24
  - 2.4.1 Formalization . . . 24
  - 2.4.2 General Approach . . . 24
- 2.5 Conclusions . . . 26

3 SPIRAL Quantum Compiler **28**
- 3.1 Approach . . . 28
- 3.2 SPIRAL . . . 29
  - 3.2.1 GAP . . . 29
  - 3.2.2 Intermediate Representations . . . 29
  - 3.2.3 Rewriting System . . . 31
- 3.3 Quantum Circuit Generation . . . 31
  - 3.3.1 System Inputs . . . 32
  - 3.3.2 Formula Breakdown . . . 33
  - 3.3.3 Formula Rewriting . . . 40
  - 3.3.4 Global Reordering . . . 43
  - 3.3.5 Search . . . 46

3.3.6 Heuristics . . . 47
3.3.7 System Outputs . . . 47
3.4 Conclusions . . . 49

4 Case Study: Quantum Fourier Transform **50**
4.1 Why is the QFT important? . . . 51
4.2 The FFT Butterfly . . . 51
4.3 Classical-to-Quantum Translation . . . 54
4.4 QFT Optimization in SPIRAL . . . 57
4.4.1 General Approach . . . 58
4.4.2 Static Hypercube Algorithm . . . 59
4.4.3 Mesh Embedding . . . 63
4.4.4 Dynamic Hypercube Algorithm . . . 65
4.4.5 Diagonal Hypercube Algorithm . . . 68
4.4.6 Non-powers of two . . . 70
4.5 Conclusions . . . 71

5 Evaluation **72**
5.1 Generalized Connectivity Satisfaction . . . 72
5.1.1 Results . . . 73
5.1.2 Analysis . . . 75
5.2 Quantum Fourier Transform . . . 77
5.2.1 QFT on mesh architectures . . . 77
5.2.2 QFT on non-mesh architectures . . . 79
5.3 Conclusions . . . 83

6 Conclusions **85**
6.1 Overview . . . 85
6.2 Directions for Future Work . . . 86
6.2.1 Proof of correctness . . . 88
6.2.2 Error-correcting Codes . . . 88
6.3 Closing Remarks . . . 89

Bibliography **91**

List of Figures

1.1 Increasing sizes of QFT on a fully-connected architecture [left] and nearest-neighbor grid [right]. Y-axis scaling differs. Data is from Table 1.1. 3
1.2 Traditional quantum circuit compilation approach (transpilation): input circuit [left], output circuit [right]. Swap operations are expressed graphically as x-terminated lines. 5
1.3 Functionally-equivalent QFT circuits transpiled onto a ring architecture. 6
1.4 Proposed generative approach to compiling optimized quantum circuits. 8

2.1 Google's sycamore quantum computer. Sourced from [10]. 12
2.2 Measurement of ϕ in $|0\rangle / |1\rangle$ basis [left], measurement of ϕ in $|+\rangle / |-\rangle$ basis [right]. The flattened circle representation that is displayed assumes real coordinates. 15
2.3 Bloch Sphere in traditional notation [left], Represented as a 3D vector space [right]. 15
2.4 A quantum circuit [left] and its QASM representation [right]. 18
2.5 Various IBM quantum computer connectivity maps. Sourced from [44]. 22
2.6 Expression tree factorizations of transform Mat. 25
2.7 Expression tree contraction and simplification. 26

3.1 Breakdown rules in classical SPIRAL. 30
3.2 Control flow graph of QSPIRAL compiler stages. 32
3.3 Breakdown rules for quantum Hadamard transform. 34
3.4 Possible placements for non-terminal QFT on a 4×4 lattice; scattered placement (e.g. (c)) will be rejected due to connectivity constraints. In this example, $k = 4$ and $N = 16$. 35
3.5 Scheduling a QFT transform on adjacent [top] and non-adjacent [bottom] qubits. Directly stating the bottom operation as a tensor product is problematic. 36
3.6 Decomposition of qCirc and adjacency matrix pruning. 37
3.7 Base rule for qCNOT fails to fire if connectivity for qubits i and j is not met in adjacency matrix $arch$. 38
3.8 Full breakdown of a two-transform algorithm; colored objects indicate non-terminals that are subject to further decomposition. 39
3.9 Subset of QSPIRAL rewrite rules. Equation (3.18) is implemented in Figure 3.11. 40
3.10 Rewrite stage control flow graph. 40
3.11 Reorder object consolidation. 41
3.12 $\mathrm{Reorder}$ and $\mathrm{Junction}$ resolution in QSPIRAL backend. 42

3.13 Formula rewriting in QSPIRAL: a) expression after $\mathrm{Junction}$ and $\mathrm{Reorder}$ are removed, b) expression after tensor contraction, c) expression after rewrite rule cancellation. 42
3.14 Changing the starting configuration of qubits to minimize reordering cost: old configuration [top], optimal configuration incurring zero staging cost [bottom]. . 43
3.15 Permuting the starting configuration of an SPL formula. Transform placements are untouched but staging cost is reduced to 0. 45
3.16 Unparsing QSPIRAL's symbolic matrix factorization; 4-qubit QFT factorization [left] 4-qubit QFT QASM program [right]. 48
3.17 Equivalent matrix expansion of 4-qubit QFT circuit (excluding vector normalization terms) [left] and a 16-point DFT [right]. 48

4.1 8-point radix-2 DIT FFT algorithm. Sourced from [52]. 52
4.2 8-point radix-2 DIF FFT algorithm. Sourced from [52]. 53
4.3 Cooley-Tukey decomposition in sparse matrix form [left], state vector dataflow representation [right]. Sourced from [25]. 54
4.4 Implementing FFT butterflies with Hadamard gates. Different H placements yield different butterfly stages. 55
4.5 Implementing a 4-point FFT with Hadamard and controlled rotation gates. 56
4.6 Implementing an 8-point FFT with Hadamard and controlled rotation gates. . . . 56
4.7 Transposing the DIF QFT [left] into the DIT QFT [right]. 56
4.8 Comparison between radix-2 QFT [left] and radix-4 QFT [right]. 57
4.9 4-qubit QFT circuit. 59
4.10 A 16-qubit hypercube. 60
4.11 The 4-qubit planar swap sequence. 61
4.12 A 16-qubit QFT via Cooley-Tukey decomposition. 62
4.13 Modified 16-qubit Cooley-Tukey QFT, with hypercube-informed twiddle scheduling. 62
4.14 Expanded static hypercube QFT decomposition (final bit reversal omitted). . . . 63
4.15 Embedding a 16-qubit hypercube into a 4×4 mesh topology [top], 32-qubit into 4×8 mesh [bottom]. 64
4.16 A 16-qubit hypercube embedding after performing a 3-dimensional cube rotation. 65
4.17 A 16-qubit hypercube embedding after performing a 4-dimensional cube rotation. 65
4.18 The 16-qubit dynamic hypercube QFT decomposition (static hypercube with additional embedding rotations). 67
4.19 A 64-qubit hypercube. Qubit 0 has 6 neighbors with edges marked in red. 68
4.20 Partitioning of N-qubit hypercubes into rings of $N/4$-qubit hypercubes. 69
4.21 Division of N-qubit hypercube embedding into 4 $N/4$ regions. Partial flattening onto mesh topology [left], high-level view [right]. 69
4.22 Twisted torus handshaking pattern. 70

5.1 Benchmarks for evaluating generalized connectivity satisfaction. 73
5.2 SWAP count on test circuits for three quantum architectures. X-axis is circuit number, in the order shown by Figure 5.1. 74

5.3 Search space histogram evaluated with respect to SWAP count: athens test circuit 6 [left], athens test circuit 4 [right]. Circuits resulting from all valid rule trees are plotted with swap count on the x-axis and circuit count on the y-axis. . . 76
5.4 Naïve scheduling problem for low-level circuit input, with a reordering step between every gate. 76
5.5 Diagonal hypercube QFT SWAP count evaluation on lattice: 4-32 [left], 4-64 [right]. Data is from Table 5.1. 77
5.6 Dynamic hypercube QFT SWAP count evaluation on lattice: 4-32 [left], 4-64 [right]. Data is from Table 5.1. 79
5.7 Chosen non-mesh architectures. Images sourced from Qiskit backend and adapted from [44]. 80
5.8 Hypercube QFT SWAP count evaluation on non-mesh. Data is from Tables 5.2, 5.3 and 5.4. 81

6.1 Full overview of quantum circuit generation procedure. 87

List of Tables

1.1 Qiskit QFT gate counts across optimization levels, on mesh architectures. 4
1.2 Qiskit 128-qubit QFT gate counts on a 12×12 mesh architecture. 4

2.1 Expressing a valid unitary matrix decomposition in Backus-Naur form (BNF) [4]. 24

3.1 Subset of QSPIRAL non-terminals. 32
3.2 QSPIRAL algorithm syntax. 33
3.3 Modified QSPIRAL algorithm syntax, with explicit architecture parameter. . . . 37
3.4 Subset of QSPIRAL's SPL syntax, excluding size annotations. 38

5.1 Hypercube QFT SWAP count on lattice. 78
5.2 4-qubit QFT SWAP count on non-mesh. 82
5.3 16-qubit QFT SWAP count on non-mesh. 83
5.4 32-qubit QFT SWAP count on non-mesh. 84

Chapter 1

Introduction

1.1 Motivation

Quantum computing has been the subject of intense research in the past decades due to the unparalleled speedup it promises for certain forms of computation. By leveraging quantum mechanics to store large amounts of information in a few particles, called quantum bits or *qubits*, quantum computers have the potential to make several classically-difficult problems tractable; there is even evidence that these devices could violate the extended Church-Turing thesis [8]. Unearthing the full taxonomy of applications benefiting from quantum technology has only just begun, and immense promise has been shown with the development of algorithms for factorization and search [59][30]. However, while this nascent quantum theory is developing rapidly, it fast outpaces the true capabilities of modern hardware and software infrastructure. The effort to construct larger and/or noiseless physical devices is the object of much research in the field, and currently poses a significant physical barrier. However, even if our hardware challenges were overcome overnight, the existing software toolchains for these devices betray the fact that our technological deficiency lies not only in the chip itself but also in the languages and compilers needed to express and run quantum programs.

We analyze the deficiencies of existing infrastructure by first focusing on the tasks which make quantum compilation difficult. We then present results generated from the leading edge of existing technology and explain why those results are inadequate. Subsequently, we describe the challenges researchers face when trying to improve these results, and why a radically different approach is needed.

1.1.1 The Quantum Optimization Problem

Before discussing the driving factors behind the deficiencies present in existing compilation technologies, we focus first on what makes the optimization problem itself hard. In general, compiling quantum circuits is non-trivial for the following reasons.

- **Connectivity.** Unlike classical computers, quantum computers execute a series of operations that directly modify qubits in-place in the hardware. The performance of quantum computers is thus directly linked to where these qubits are located in the physical device.

This is primarily due to the set of multi-qubit operations, called *controlled* operations, that typically operate on two qubits at a time and require the qubits to be physically adjacent in the device. Adjacency can be achieved by some physical circuit or bus in the chip itself, but for manufacturing and feasibility reasons, qubits are typically only sparsely connected. For controlled operations between qubits that are not linked in this manner, data movement instructions must be inserted by the compiler to dynamically shift values around and meet this constraint.

- **Coherence.** Qubits are able to retain massive amounts of information only because they are kept in specific quantum states; information could be stored, for example, in the spin of an electron or the polarization of a photon. These states are fragile and extremely susceptible to environmental interference; unfortunately they degrade rapidly if too many operations are performed on them. As a result, longer programs typically yield more error-prone results, and so the critical path of a program must be kept as short as possible. Quantum operations can also be imperfectly implemented, and these otherwise small errors can cascade if allowed to accumulate over long operation sequences.
- **Complexity.** Routing a quantum algorithm efficiently onto hardware, in a manner that minimizes the number of data movement operations needed while meeting the connectivity constraints of the hardware, is an NP-complete problem. Specifically, elements of this problem can be reduced to the maximum common edge subgraph problem [31]. Therefore, finding circuits with a minimal number of data movement instructions requires developing effective heuristics for this problem.

These three factors constructively interfere with one another to make the compilation problem (at least for an *optimizing* compiler) a distinctly difficult task. The connectivity constraint introduces structural hazards that require compiler intervention, typically resulting in bloated code sizes if done inefficiently. The coherence constraint means that if too many data movement operations are inserted, the quality of the results, or even the practicability of executing the program, will suffer. Finally, the complexity constraint implies that inserting these data movement operations in a minimalistic way is NP-hard.

To motivate the importance of limiting the number of data movement operations, we can look at architectural statistics for the SWAP operation (the quantum data movement primitive) on various quantum computers. Google's Sycamore computer has a SWAP error rate that is 5-6 times higher than the error rate for single-qubit operations [3]. Similar results can be seen from devices by IBM [48] and IONQ [17]. In addition, SWAP operations are simply more computationally intensive to implement than single-qubit gates, due both to the nature of the task and the fact that single-qubit operations can often be performed in parallel. Finally, ignoring the specifics of the SWAP operation itself, the other operations in the program that actually serve to *forward* the computation are often fixed properties of the algorithm; a quantum Fourier transform (QFT) requires a fixed number of rotation gates and Hadamard gates. Conversely, the number of SWAP operations typically grows unnecessarily with increasing program size, usually due to the inadequacies of our connectivity solvers coupled with the sparsity of the hardware. If critical path is a concern, data movement must be the primary target for minimization.

1.1.2 Existing Results

We have elaborated on why the quantum optimization problem is difficult, and why solving it effectively is important. We now elaborate on the deficiencies of existing approaches.

To show this, the compilation statistics in Figure 1.1 were generated by compiling a series of Fourier transform circuits onto a nearest-neighbor grid topology with IBM's Qiskit platform [1]. The data-movement primitive used in these systems is the SWAP operation, which swaps the values of two adjacent qubits. Given the aforementioned constraints, we would want to have as few swaps as possible. On a device with full connectivity, no swaps should be necessary at all.

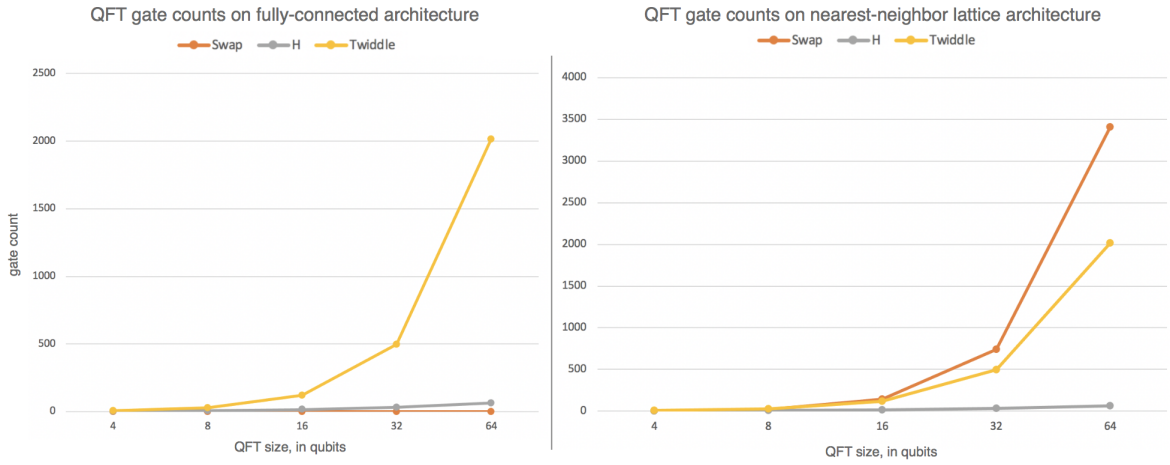


Figure 1.1: Increasing sizes of QFT on a fully-connected architecture [left] and nearest-neighbor grid [right]. Y-axis scaling differs. Data is from Table 1.1.

As the number of qubits in the circuit increases, and hardware layouts remain sparse with respect to edges, we can see that the data movement overhead becomes unacceptably large; the proportion of SWAP operations to actual computational operations becomes untenable. Not only is the largest portion of runtime dedicated to moving values around, but coherence may become problematic since the compiler-inserted data movement operations rapidly increase circuit size. In addition to coherency concerns, SWAP operations also have a high error rate, and so this also means our results may get progressively worse as the number of qubits scales. As seen further with the 128-qubit QFT in Table 1.2, this unsavory trend is exponential; this is unsustainable if the goal is to run even larger algorithms in the future. This problem is, in our view, *as limiting* to the growth of quantum technology as the physical constraints barring us from manufacturing larger devices.

Qiskit QFT gate counts on a nearest-neighbor lattice						
Qubits	Optimization	Connectivity	SWAP	Twiddle	H	Circuit Depth
8	0	4x2	26	28	8	30
16	0	4x4	123	120	16	66
32	0	6x6	781	496	32	214
64	0	8x8	3893	2016	64	715
8	1	4x2	20	28	8	26
16	1	4x4	132	120	16	67
32	1	6x6	818	496	32	215
64	1	8x8	4156	2016	64	765
8	2	4x2	24	28	8	28
16	2	4x4	156	120	16	81
32	2	6x6	844	496	32	223
64	2	8x8	4149	2016	64	776
8	3	4x2	22	28	8	26
16	3	4x4	143	120	16	68
32	3	6x6	740	496	32	190
64	3	8x8	3406	2016	64	501

Table 1.1: Qiskit QFT gate counts across optimization levels, on mesh architectures.

Qiskit 128-qubit QFT gate counts on a nearest-neighbor lattice						
Qubits	Optimization	Connectivity	SWAP	Twiddle	H	Circuit Depth
128	0	12x12	21793	8128	128	2572
128	1	12x12	22059	8128	128	2401
128	2	12x12	20303	8128	128	2241
128	3	12x12	18660	8128	128	1865

Table 1.2: Qiskit 128-qubit QFT gate counts on a 12×12 mesh architecture.

1.1.3 Challenges to Overcoming Deficiencies

We have analyzed the problem itself and why existing approaches are inadequate. However, there are several outstanding challenges that prevent immediate progress from being made.

The first challenge in developing better systems is due to the nature of quantum programs. A significant consideration is that quantum programs are mathematical objects as opposed to "RAM mutators." Quantum computers derive their power from the parallel nature of qubits; each individual qubit state can be written as a 2-dimensional complex vector, the values of which can be modified and measured. The state of a quantum system at any point in the computation is determined by the values of these vectors. This is a departure from the classical world, where the state of a system is a function of register, cache and memory values (or for a turing machine model, the location of the tape head, the tape contents and the DFA state [63]). This vector representation seems drastically simpler, but it also poses significant computational problems.

The state of an N-qubit system for arbitrary N is represented by a 2^N-length complex column vector. Additionally, since a quantum program is essentially limited to rotating this state vector in a length-preserving manner (a fact derived in Chapter 2), the effect of an N-qubit quantum program is succinctly described by a $2^N \times 2^N$ transform matrix that acts on the input state vector. This clearly aligns with our view of quantum programs as pure mathematical operators, and this transform matrix can be derived by composing the individual mathematical operators that make up a quantum program. However, it is unfortunate that the transform matrix is exponentially large with respect to N.

For even small values of N, this matrix is often too large to compute or store conveniently. This poses a problem for verification which typically requires expanding and comparing the transform matrices for the two circuits in question in order to establish equality. This also means that when compiling a quantum program stream, even though we *could in principle* derive the overall $2^N \times 2^N$ transform matrix, it is almost never practical to do so. The primary side-effect of this is that it is nearly impossible for the compiler to know exactly what the input program is computing; this is also a problem faced by many classical compilers for imperative programs. Compilers in this space are therefore traditionally limited to making local changes on a small scale, since these are all that can be done safely while making sure that the program remains functionally equivalent to the input. This makes it difficult to detect more complex semantics-preserving algorithm rewrites; if the compiler *knew* what the input algorithm was, it could apply more ambitious manipulations. The fact that quantum programs operate in the $O(2^N)$ vector space rather than the $O(N)$ qubit space, then, is both the biggest benefit of quantum systems and a primary contributor to the bottleneck described in the above sections.

This unfortunate reality is exacerbated by the structure of traditional compiler architectures in this space, particularly by the representations used to define the inputs and outputs of the system. The most common low-level representation of quantum programs is the *quantum circuit*, where individual operations are graphically represented as gates applied to input qubits in a layout similar to that of a boolean circuit or hardware datapath diagram. This representation fits because it matches our understanding that a quantum program is simply a feed-forward datapath that applies linear algebraic operators to the input vector. A common approach, then, is to take such a circuit as input and have the compiler map it onto the target hardware by inserting SWAP operations (among other manipulations that are of less consequence in this thesis), as is shown in Figure 1.2.

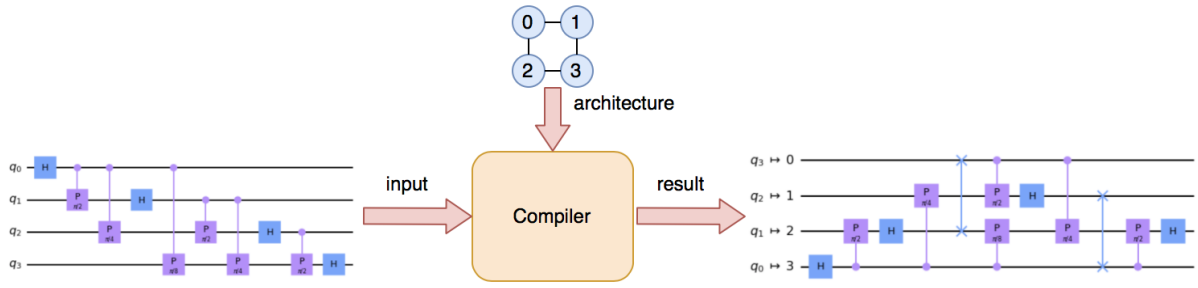


Figure 1.2: Traditional quantum circuit compilation approach (transpilation): input circuit [left], output circuit [right]. Swap operations are expressed graphically as x-terminated lines.

The compiler takes in both a circuit definition and a target architecture; the connectivity information of various hardware locations is typically expressed as a graph. The compiler performs local, peephole optimizations to try to insert the fewest swap operations possible that still make the input circuit executable on the target architecture, an approach commonly referred to as "transpilation". This approach is problematic due to the low-level representation used to express the input. As stated before, each quantum circuit effectively implements a specific $2^N \times 2^N$ linear transformation, but this is a surjective mapping and this transformation is usually implementable in circuit form in various ways. Figure 1.3 displays Qiskit transpilation results for 4 different circuits that all implement the same transformation, specifically a 2^4-point Fourier transform.

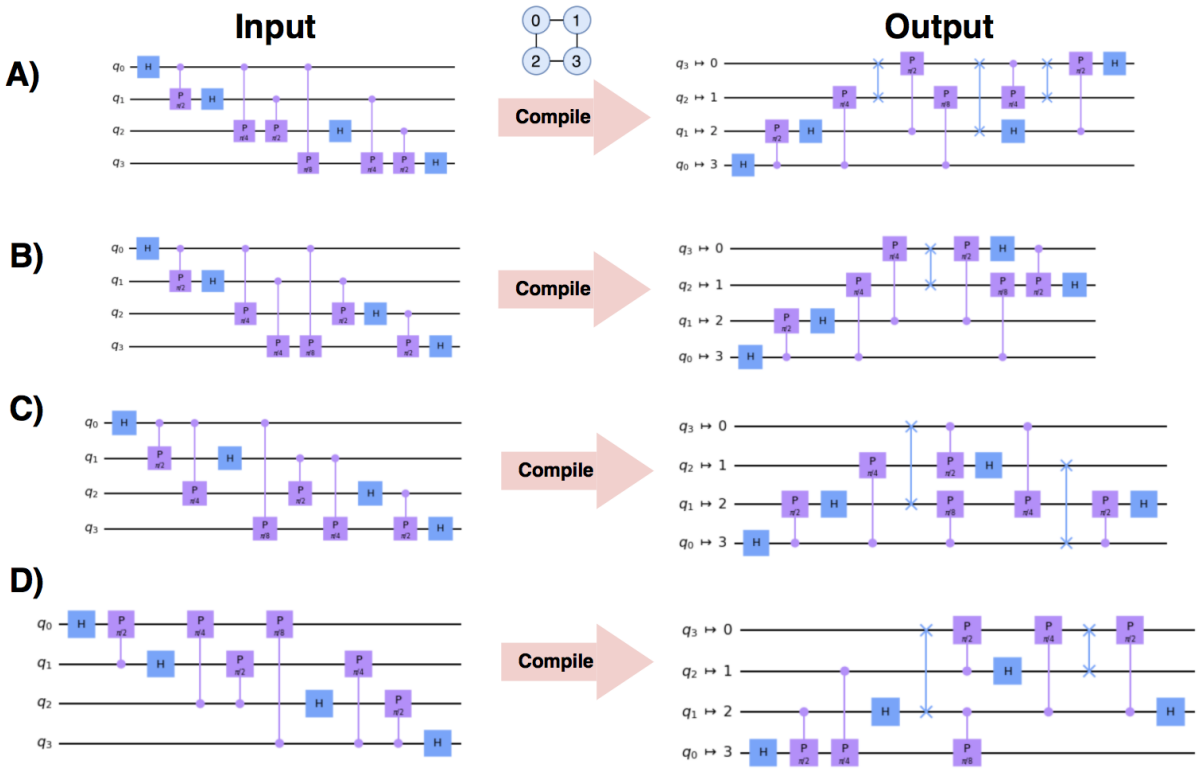


Figure 1.3: Functionally-equivalent QFT circuits transpiled onto a ring architecture.

For each of the input circuits in Figure 1.3, the compiler has successfully mapped it onto the target architecture. However, mapping a *specific* circuit onto an architecture is fundamentally not the problem that we want the compiler to solve. For example, if we were to transpile circuits A, C or D from Figure 1.3, we would much rather receive circuit B's result than our own. What the user actually wants the compiler to find is the most efficient circuit, that when run on the target architecture, produces the same *result* as the input circuit. Notice that this search problem is completely agnostic to the input circuit besides using it to determine the $2^N \times 2^N$ transform that the output has to implement. Unfortunately, as we have stated above that reconstructing this $2^N \times 2^N$ transform is impossible, in order to achieve this goal with the traditional transpiler paradigm we would have to find an efficient way of searching over the space of circuits that implement the desired transform using only the single gate-level implementation we are afforded.

We can show that this is nearly impossible.

Some circuits (like circuits A, B and C from Figure 1.3) are related to one another by a semantics-preserving reordering of operations. Conceivably, this fraction of the overall circuit space could be reconstructed even if only given a single circuit that implements the desired transform. More generally, however, we realize that global manipulations of signal processing algorithms, such as the transformation of the DFT algorithm into the more efficient FFT algorithm, rely on group-theoretic arguments [18] and are extremely difficult to derive when simply manipulating a compiled program stream. A concrete example of this is the transposition of a decimation-in-time FFT into a decimation-in-frequency FFT [52], as shown by circuits A and D in Figure 1.3. We cannot recognize and apply these symmetries without first knowing what we are trying to compute, and hence using circuit representation artificially constrains the space of output circuits that we can reasonably generate. Conveying this in familiar classical terminology, we quite simply seek to express the input program in a declarative language as opposed to an imperative one, and hence leverage higher-level scheduling techniques without being constrained by any specific implementation.

In the next section we propose a novel approach that starts with a high-level algorithm description (a symbolic representation of the elusive $2^N \times 2^N$ transform matrix) and constructively forms a program to efficiently execute it on the target hardware. This allows us to apply high-level global manipulations, and enables the efficient evaluation of a much larger space of circuits that achieve the desired transform rather than just ones related to the input by local permutations or simplifications.

1.2 Overview

This thesis addresses the aforementioned motivations by proposing a new approach for compiling and optimizing quantum circuits. Specifically, we will apply a generative approach. We previously recognized the need to shift our focus away from compiling individual circuits and towards treating the transform itself as a first-class design constraint; the task at hand is to find the best circuit that implements a specified input transform on the given architecture. We will intuitively cast this as a generic search problem, illuminating the fact that this problem lives completely within the realm of traditional computer science. Before our approach will become apparent, however, the questions remain of how to internally capture and represent the inconveniently-large $2^N \times 2^N$ transform matrix, and how to efficiently generate the entire space of possible circuits from such an abstract functional description.

With respect to the first problem, we notice that there are very few useful quantum algorithms currently, many of which reuse kernels like the Fourier transform. We can thus capture these algorithms symbolically, and represent nearly any useful quantum algorithm as a composition of these symbolic transforms. An example of these might be the $\mathrm{QFT}(n)$ for an n-qubit Fourier transform, or the $\mathrm{QHT}(n)$ for an n-qubit Hadamard transform. By capturing generalized input in terms of higher-level objects, we can leverage an additional layer of abstraction to allow us more latitude in applying complex rewrites.

The second problem (i.e. how best to generate a space of quantum circuits from an abstract definition) leads directly into our proposed compiler design as shown in Figure 1.4. Our

system will enable an efficient global search over a heuristically-pruned subset of all possible circuit implementations of the desired transformation. We will take, as input, both the symbolic representation of the target algorithm as expressed as a high-level transform and hardware specifications including the connectivity graph of the target device. We will then generate and search over the space of circuits that implement this transformation on the target hardware and choose the best with respect to some cost measure, probably including the number of swaps.

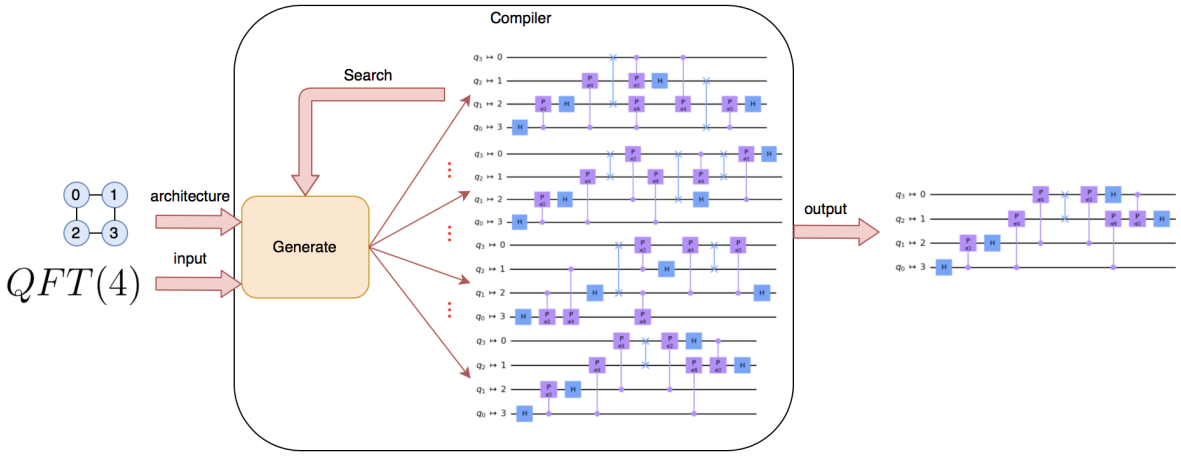


Figure 1.4: Proposed generative approach to compiling optimized quantum circuits.

To inform how it is possible to decompose a high-level specification into a quantum circuit, we first cover the foundational mathematics of quantum computing with a particular interest in framing them with traditional linear algebra. Specifically, we are able to establish, as stated before, that the state of an N-qubit system is a 2^N-length complex column vector and that a quantum program is a linear transformation that acts on this vector. We also show that quantum gates are small unitary matrices that can be combined by the matrix product when applied in series and by the tensor product when applied in parallel. Applied to our problem, this means that any given quantum circuit is just a sparse factorization of some overall transform matrix, written purely in terms of the tensor and matrix products of quantum gates. Generating a quantum circuit from a high-level symbolic transform is then just a matrix factorization problem, and we will leverage a computer algebra system to solve exactly this.

We structure this factorization problem further by introducing the concept of breakdown rules. We will decompose our input algorithm by applying a series of divide-and-conquer decomposition rules to break it into smaller transforms until the entire expression is ultimately written in terms of quantum gates. By searching over all such sequences of decomposition rules we effectively search over all circuit representations of the input. Since optimizing the hardware locations of various qubits is critical to both minimizing swaps and ensuring we meet the connectivity requirements of the architecture, we will also need to search over various partitions of the connectivity graph at each breakdown stage, expressing an overall transform executing on connectivity graph G as a composition of smaller transforms executing on subsets of G. Since we know, as we decompose the algorithm, both a high-level specification of the transform and the

connectivity information, we can use these to inform heuristics regarding both the partitioning of these qubits and which breakdown rules we should try. Given this circuit generation framework, we can then apply any of the myriad search techniques such as dynamic programming [6] to find an optimal or heuristically-optimal circuit and output our result.

We implement the aforementioned system with the SPIRAL [24] framework, a code generation system that applies roughly the same methodology as described to successfully optimize linear transforms for classical architectures. Based on the GAP [57] computer algebra system, SPIRAL is built to manipulate key mathematical operators such as the matrix product and tensor product, and its focus on computational group theory provides advanced algorithm rewriting capabilities. We successfully implement a backtracking search procedure to perform the proposed rule-based decomposition.

An added benefit of the SPIRAL system is its past success in optimizing the fast Fourier transform (FFT) for classical architectures. Since the quantum Fourier transform (QFT) is simply the FFT implemented in quantum circuitry, and is itself an extremely important kernel in the quantum domain, we can leverage both SPIRAL's advanced library of decomposition rules and existing parallel FFT literature in order to construct several effective QFT decomposition heuristics. By taking a structured approach to the problem, recognizing and taking advantage of patterns as opposed to treating the QFT as an arbitrary series of gates, we show tangible savings on SWAP operations when compared with other compilation approaches. The methodology we employ to integrate these heuristics can be extended to include any other relevant algorithms and heuristics. Throughout this work, we stress that there are direct parallels between quantum and classical algorithms, meaning that research in the two domains should arguably be interchangeable to some degree.

We end with an analysis of our system, and notes regarding directions for future work. Beyond simply porting classical SPIRAL's decomposition and heuristics libraries over to the quantum domain, there are several areas such as circuit verification and error correction in which this system could be uniquely valuable. We believe this work strongly motivates developing approaches similar to ours in order to effectively compile high-level algorithms for the quantum hardware of the future.

1.3 Contributions

The contributions of this thesis include

- A formalization of optimizing quantum circuits with respect to critical path, written in terms of traditional linear algebra constructs and classical computer science terminology
- The description and implementation of a novel compilation framework for generating optimized quantum circuits
- A case study of optimizing Fourier transforms for quantum architectures
- The description and implementation of novel heuristics for implementing Fourier transforms on quantum architectures
- A qualitative and quantitative evaluation of the proposed system

The primary claim argued by this thesis is that *a quantum compilation approach that intends*

to solve the data movement problem in a scalable manner should benefit from taking high-level algorithm symmetries into account, and that this approach can be heavily informed by existing literature in the classical domain. While we focus on the Fourier transform in this work, and thus only substantiate this claim for a specific algorithm, we expect that our approach can extend to a broader set of quantum kernels. The eventual goal of any such system must be to eliminate the exponential domination of data movement operations shown in Figure 1.1.

1.4 Organization

Chapter 2 provides relevant background material regarding quantum computation and explains the mathematics behind quantum circuits. We frame the quantum optimization problem in terms of traditional computer science constructs and linear algebra, discarding much of the quantum-specific terminology and notation typically used in the field; this is done in order to illuminate the similarities between this research area and the others we draw from in order to inform our approach. Chapter 3 provides an overview of our SPIRAL implementation, in which we describe relevant specifics regarding both the general SPIRAL framework and the formalization of quantum circuits as linear transforms subject to sparse decomposition. Chapter 4 discusses a particular algorithm in detail, the quantum Fourier transform, and proposes a technique for efficiently mapping this algorithm onto quantum architectures. Chapter 5 discusses our results and how they further motivate our approach. We conclude with Chapter 6, in which we present concluding remarks and provide a road map for future work.

Chapter 2

A Primer on Quantum Information Science

This chapter presents background material on quantum information science with a specific focus on quantum computing. Much of the traditional terminology and notation is discarded in favor of generic linear algebra, which allows us to draw clearer parallels to classical techniques in later chapters. First, Section 2.1 gives a brief introduction to quantum computing and why it is currently a promising field of study. Section 2.2 defines the primitives comprising a quantum system and details the basic principles regarding their behavior. Then, Section 2.3 describes quantum circuits and the gates that construct them, briefly exploring the hardware architecture of quantum computers, and how these constraints relate to the general approach presented in the introductory chapter. The optimization problem is formalized in terms of traditional computer science constructs in Section 2.4, where a general approach is outlined. Conclusions are drawn in Section 2.5, after which Chapter 3 discusses the SPIRAL implementation of a solver for the problem we outline.

2.1 What are Quantum Computers?

Quantum computers were conceived in the early 1980's and are widely credited to Richard Feynman [22], Paul Benioff [7] and Yuri Manin [41]. Behind the invention was the desire to leverage the intricate quantum-mechanical states of particles in order to store and compute on massive amounts of information. Specific requirements regarding the practicability and utility of quantum computers as real devices were not outlined until later [16].

A quantum computer is powered by quantum bits, or qubits. These qubits store information in quantum states, similarly to how classical bits can be represented by groups of electrons or numerous other physical processes [36]. At the quantum level, however, these states can represent more information than can binary counters. Due to the principle of superposition [58], the quantum nature of these states leads to an exponential relationship between the number of qubits and the amount of information that can be stored and computed on. By constructing complex physical devices to manipulate and read these states, we are able to perform potentially amazing computational feats, such as factor large integers in polynomial time with Shor's algorithm [59].

In a major departure from the traditional von Neumann model [9], quantum computers perform all operations directly on these qubits, something that is referred to as *in-place* computa-

tion [34]. This means that instead of shuffling data around in memory and reading the results of execution units, quantum computers are directly manipulating the qubit particles themselves by subjecting them to various processes. Hence, if these qubit particles must interact with other qubits in the system, their physical location becomes very important. To capture this, we traditionally express quantum programs as circuits, representing the fact that quantum programs are feed-forward datapaths that apply successive mathematical operations, or *gates*, to the input qubits. In this sense, these computers share design principles with systolic [39][40] and spatial dataflow architectures [53]. These operations can alternatively be viewed as ordered instructions in the quantum assembly (QASM) language [14], which is simply a text serialization of the circuit representation.

There is ongoing research into the complexity classes that define quantum computers and the relationship they have with the classical P/NP distinctions. There are several algorithms in existence that show a polynomial speedup when run on quantum computers versus their classical brethren [50]; these speedups are possible due to the exponential increase in the amount of data we can represent with qubits when compared against that possible with classical bits. Potentially more exciting, however, is the possibility of classically-intractable problems becoming feasible on a quantum computer, a possibility that has attracted large-scale research in the field [3] and has driven the development of devices such as the one shown in Figure 2.1. If found to be true, this would violate the extended Church-Turing thesis.

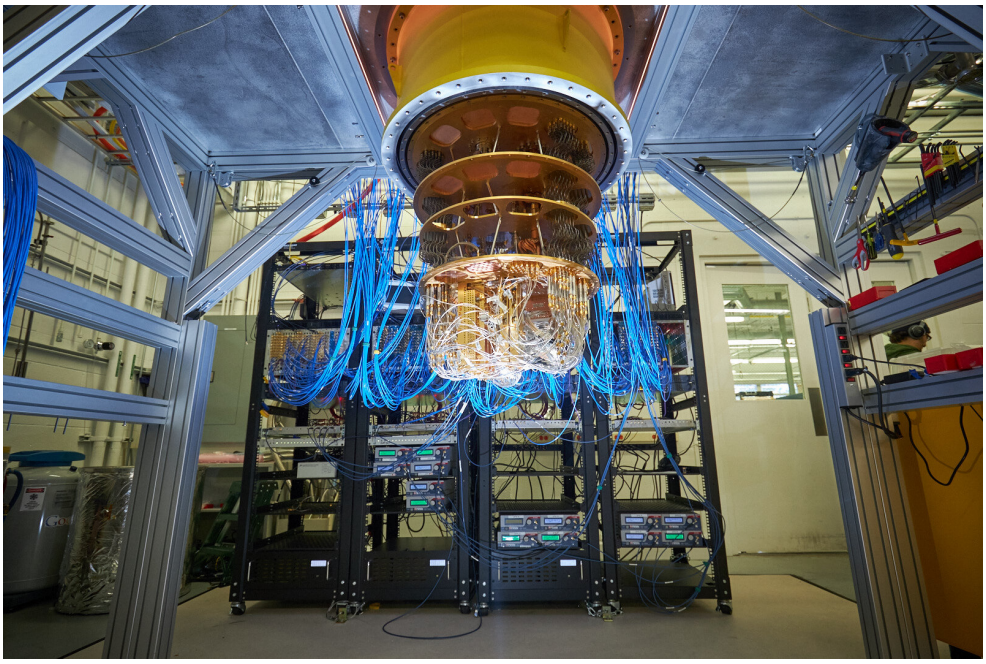


Figure 2.1: Google's sycamore quantum computer. Sourced from [10].

The construction of these devices is immensely difficult. The computers themselves must be isolated from their environment; this is needed in order to protect qubits from being affected by anything other than the specific laser pulses or other physical processes that are used to implement quantum operations. As a result, building larger systems with more qubits, and investigat-

ing which particles make the most stable and resilient qubits, has required immense engineering effort. Luckily, great progress has been made on both the hardware and software sides of quantum computer development, and major corporations have consistently revealed larger quantum computers with successively greater utility. However, as stated in the first chapter, something must change in order to achieve the level of scaling necessary to make these devices truly useful. Specifically, we may require radically different approaches to software compilation in order to support the necessary scaling of our algorithms to the hopefully immense quantum computers of the future.

In order to construct an approach that could satisfy this need we will draw inspiration from the classical domain. In order to do so we must first translate the mathematics into a format that lends itself well to drawing these parallels.

2.2 Qubit Fundamentals

Quantum computation is a relatively new field of study within the broader area of computer science because it concerns the architecture and algorithms for a radically different computational model. While truly understanding the physics warrants further study, the principles dictating how to compute with these quantum bits lie more purely in the field of traditional computer science.

2.2.1 One Qubit

Qubits can be described by a dual-state system. A qubit can be in one of two orthogonal states, whether that be the up/down spin of an electron or the vertical/horizontal polarization of light. The expressiveness of a qubit comes from the principle of superposition, which implies that besides being in one state or the other, that a qubit can additionally be in-between these two states. Specifically, it can be in each of the two states with some *amplitude*. Let us call two such states $|0\rangle$ and $|1\rangle$ and the overall qubit state $|\phi\rangle$. $|\phi\rangle$ is traditionally defined by this set of equations.

$$|\phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \text{ where } \alpha^2 + \beta^2 = 1 \text{ and } \alpha, \beta \in \mathbb{C} \tag{2.1}$$

But we can also define any number of orthogonal states

$$|-\rangle = \frac{1}{\sqrt{2}}\,|0\rangle - \frac{1}{\sqrt{2}}\,|1\rangle \tag{2.2}$$

$$|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle \tag{2.3}$$

and then express the same $|\phi\rangle$ in terms of those states.

$$|\phi\rangle = \gamma\,|+\rangle + \omega\,|-\rangle \text{ where } \gamma^2 + \omega^2 = 1 \text{ and } \gamma, \omega \in \mathbb{C} \tag{2.4}$$

We can change the states that $|\phi\rangle$ is written in terms of by translating the amplitudes.

$$\alpha = \frac{\gamma}{\sqrt{2}} + \frac{\omega}{\sqrt{2}} \tag{2.5}$$

$$\beta = \frac{\gamma}{\sqrt{2}} - \frac{\omega}{\sqrt{2}} \tag{2.6}$$

These equations merely define the mutable state of a qubit, but to actually read out these values we must *measure*. Measurement is defined with respect to a set of orthogonal states, i.e. a $|0\rangle$ / $|1\rangle$ measurement operator or a $|+\rangle$ / $|-\rangle$ measurement operator, each of which return a $|0\rangle$ / $|1\rangle$ or $|+\rangle$ / $|-\rangle$ judgement respectively. Measuring a quantum state is non-deterministic; the classical probability with which the measurement operator will output a judgement is equivalent to the squared amplitude of that state. For example, measuring the $|\phi\rangle$ qubit state defined by Equation (2.1) with a $|0\rangle$ / $|1\rangle$ measurement gate will return $|0\rangle$ with probability α^2 and $|1\rangle$ with probability β^2. Measuring it with a $|+\rangle$ / $|-\rangle$ measurement gate will return $|+\rangle$ with probability γ^2 and $|-\rangle$ with probability ω^2. Measurement is also *destructive*, and the qubit state itself is modified by the measurement operator. Specifically, the qubit collapses into a state described by an amplitude of 1 on the state which was read and 0 on the state which was not; in other words, the qubit becomes exactly what was measured.

The notation used above, commonly referred to as Dirac notation [15], constitutes primarily what is used in quantum computing literature. This notation, in our rudimentary opinion, unnecessarily obfuscates the simplicity of the mathematics for the immediate purposes of this thesis. Therefore, we will now present these topics in terms of linear algebra; this is more representative of SPIRAL's internal treatment. Partially rephrasing the above, we can represent a qubit as a 2-dimensional complex vector.

$$\phi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \text{ where } ||\phi|| = 1 \text{ and } \alpha, \beta \in \mathbb{C} \tag{2.7}$$

It is plain to see that this vector representation can be written in terms of any two orthonormal bases in this space. In fact, $|0\rangle$ and $|1\rangle$ are commonly written as $[1, 0]^T$ and $[0, 1]^T$ respectively, from which other bases like $|+\rangle$ and $|-\rangle$ can be derived.

$$\phi = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.8}$$

$$\phi = \gamma\,|+\rangle + \omega\,|-\rangle = \gamma \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} + \omega \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} \tag{2.9}$$

The magnitudes assigned to these bases are directly the amplitudes associated with the corresponding states. Measurement probabilities, therefore, are simply derived by projecting the ϕ vector onto the measurement bases and taking the magnitude squared of the resulting projections, as shown in Figure 2.2. The destructive nature of measurement is also captured by this, since measurement itself is a projection operation onto the measured basis vector.

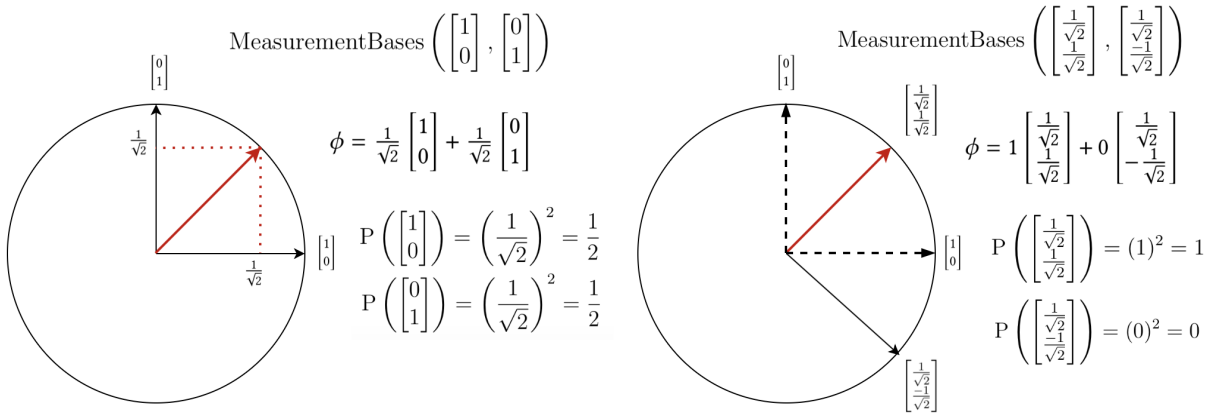


Figure 2.2: Measurement of ϕ in $|0\rangle$ / $|1\rangle$ basis [left], measurement of ϕ in $|+\rangle$ / $|-\rangle$ basis [right]. The flattened circle representation that is displayed assumes real coordinates.

There remains an apparent inconsistency with our formulation, namely that the qubit state vector ϕ is 4-dimensional, with 2 real dimensions and 2 complex dimensions. However, one may note that qubit states are traditionally charted on a 3-dimensional sphere called the *Bloch Sphere*, as shown in Figure 2.3.

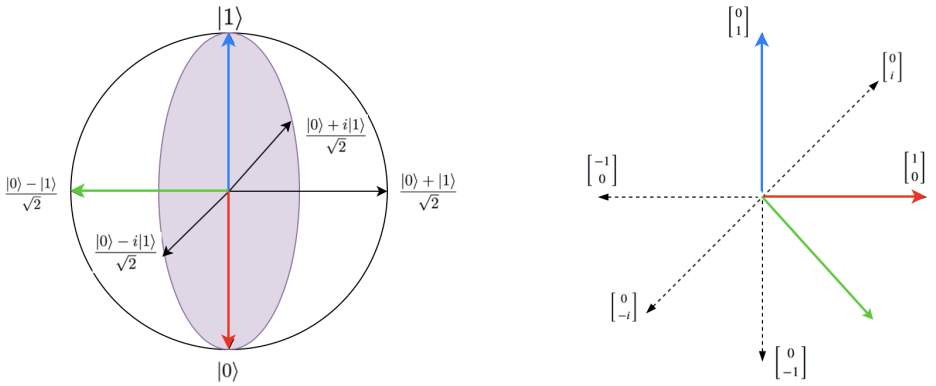


Figure 2.3: Bloch Sphere in traditional notation [left], Represented as a 3D vector space [right].

We must then reconcile our 4-dimensional vector space with this apparent 3-dimensional representation. The key lies in the fact that we need only represent *distinguishable* qubit states, where the only method we have of distinguishing two qubit states is the measurement operator. Recall that the measurement operator is a projection operator, and that the classical probability of reading a value is the magnitude squared of the projection onto that basis. Given a qubit state ϕ and a set of basis vectors X, Y

$$\text{Prob}(X) = |X \cdot \phi|^2 \tag{2.10}$$
$$\text{Prob}(Y) = |Y \cdot \phi|^2 \tag{2.11}$$

Multiplying ϕ by a global phase term ξ where $|\xi| = 1$ we get

$$\begin{aligned} \text{Prob}(X) &= |X \cdot \xi\phi|^2 = |X \cdot \phi|^2 && (2.12) \\ \text{Prob}(Y) &= |Y \cdot \xi\phi|^2 = |Y \cdot \phi|^2 && (2.13) \end{aligned}$$

Therefore the probability of measurement with respect to arbitrary basis vectors X, Y is exactly the same for qubit state vectors ϕ and $\xi\phi$. Therefore we can represent our 4-dimensional qubit state ϕ in three dimensions by performing the following transformation, leveraging this property to convert Equation (2.15) to Equation (2.16).

$$\begin{aligned} \phi &= \begin{bmatrix} a \\ b \end{bmatrix} \text{ where } a, b \in \mathbb{C} && (2.14) \\ \phi &= \begin{bmatrix} ce^{j\theta} \\ de^{j\omega} \end{bmatrix} \text{ where } c, d \in \mathbb{R} && (2.15) \\ \phi &\equiv e^{-j\theta} \begin{bmatrix} ce^{j\theta} \\ de^{j\omega} \end{bmatrix} \text{ where } c, d \in \mathbb{R} && (2.16) \\ \phi &\equiv \begin{bmatrix} c \\ de^{j(\omega-\theta)} \end{bmatrix} \text{ where } c, d \in \mathbb{R} && (2.17) \\ \phi &\equiv \begin{bmatrix} a \\ b \end{bmatrix} \text{ where } a \in \mathbb{R}, b \in \mathbb{C} && (2.18) \end{aligned}$$

It is worth noting that ϕ is still a 4-dimensional vector. However, one of the complex dimensions is aliased out due to the limitations of measurement. Vectors with no observable difference are therefore treated as equivalent, and no downstream operations in any quantum program can make this difference become apparent at a later stage.

2.2.2 Multiple Qubits

For a generalized number of qubits N, we can express the state of the system with a 2^N-length complex column vector with unit norm. This *joint state vector*, denoted in the following formulas as Φ_N, can be constructed via the Kronecker product of individual qubit states. We can construct higher-dimensional bases from taking the Kronecker product of single-qubit bases.

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.19}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.20}$$

And hence we can simply express a multi-qubit system as a complex vector in a 2^N-dimensional space

$$\Phi_2 = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \text{ where } a, b, c, d \in \mathbb{C} \quad \Phi_3 = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} \text{ where } a, b, c, d, e, f, g, h \in \mathbb{C} \tag{2.21}$$

albeit remembering from before that global phase is not distinguishable by a measurement device.

It is worth noting in this context the concept of *quantum entanglement*, not because it is particularly important for our particular work, but rather because viewing it in a linear-algebraic context does much to dispel confusion regarding the important and infamous topic. Central to this topic is the Bell [5] state, or EPR (Einstein Podolsky Rosen) [20] pair, which resembles the following.

$$\mathrm{Bell} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.22}$$

Unlike other multi-qubit states, this cannot be factored into a tensor product of individual qubit states. The proof of this is fairly trivial [51]. Assume, for instance, there were two arbitrary single-qubit states α and β such that $\alpha \otimes \beta = \mathrm{Bell}$.

$$\alpha = \begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} \quad \beta = \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} \quad \mathrm{Bell} = \alpha \otimes \beta = \begin{bmatrix} \alpha_0\beta_0 \\ \alpha_0\beta_1 \\ \alpha_1\beta_0 \\ \alpha_1\beta_1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.23}$$

The system of equations used to solve for the elements of α and β have no solution. This is the formal definition of entanglement. For our purposes, "spooky action at a distance" is an algebraic result.

The primary takeaway from the above mathematics is that at any point in the quantum program, the complete state of the system can be expressed by a 2^N-length column vector. Since the only operations that can be performed on these state vectors are rotations, we can view quantum computers as devices that efficiently affect linear (unitary) transforms on the joint state vector representing the quantum properties of qubits. By taking a linear-algebraic approach, we need not worry about the specific laser pulses used to implement these transforms, and can treat quantum computers as mechanically complex (but theoretically simple) accelerators for implementing linear transform algorithms.

2.3 Quantum Circuits

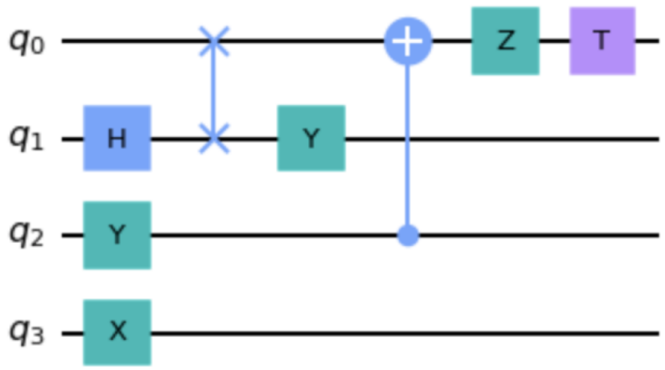
While the theoretical underpinnings of classical algorithms can be traced to the Turing machine or lambda calculus [11] models, quantum algorithms are typically expressed as circuits, reflecting the fact that a quantum program is essentially a feed-forward datapath. Following from this representation, the fundamental building blocks of a quantum circuit are referred to as quantum *gates*. Gates comprise the set of atomic operations that a quantum architecture can perform, similarly to how an instruction set architecture (ISA) bridges the gap between classical hardware and software. While the set of supported or *basis* gates varies from chip to chip, there is still a canonical set of operations in the literature. Gates that operate on a single qubit can be represented by a 2×2 unitary matrix, and specialized 4×4 multi-qubit gates are discussed in section 2.3.2. For example, the Hadamard (H) gate performs a Walsh-Hadamard transform on a single-qubit state vector, an X gate reverses the elements and a Z gate negates the second vector component.

$$\mathrm{H} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \mathrm{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathrm{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.24}$$

Single-qubit gates implement a variety of rotations around a qubit's 3-dimensional vector space, and hence can be rewritten as special cases of a generic rotation matrix. An arbitrary qubit state vector can be altered by any rotation transformation that preserves unit length (i.e. a unitary matrix); this constraint is owed to the fact that vector magnitudes equate to probabilities.

$$\mathrm{GRot}(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\theta)}\cos(\theta/2) \end{bmatrix} \tag{2.25}$$

Circuits are typically expressed graphically, in a manner similar to a hardware datapath. Circuits can also be expressed in QASM format and executed on a variety of platforms, but QASM is simply a text serialization of the circuit diagram as seen in Figure 2.4.




```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
h       q[1];
swap    q[0], q[1];
y       q[2];
x       q[3];
cx      q[2], q[0];
y       q[1];
z       q[0];
t       q[0];
```

Figure 2.4: A quantum circuit [left] and its QASM representation [right].

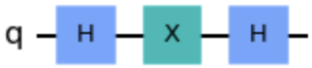
For this section, and for the rest of the thesis, we assume that all measurement operations are done at the end of the circuit. Besides usually being the case, the principle of deferred measurement [50] additionally allows us to convert an arbitrary program with intermediate measurements

into one with measurements at the end. Otherwise, in the case where intermediate measurements were taken, the state of a quantum system would cease to be a simple vector, or *pure* state. Rather, we would have to classify a quantum state as a *mixed* state, where the system takes on one of several state vectors, each with classical probability. This is not immediately relevant to the problem we are trying to solve, and thus the situation is not considered by this thesis.

2.3.1 Circuits as Matrix Factorizations

While graphical (circuit) and program stream (QASM) representations are valuable, quantum circuits are best expressed as pure mathematical objects. We show that functionally, a quantum circuit on N qubits implements a $2^N \times 2^N$ unitary transformation on the input joint state vector. We also show that an implementation of this transform in circuit form is just one of many possible matrix factorizations, where the overall transform is expressed in terms of basis gates.

Starting with the application of gates in series, we can see that this corresponds to the matrix product of the gates. Consider the 3-gate circuit



which, progressing through the stages, can be expressed by the following series of unitary transformations.

$$\phi = \begin{bmatrix} a \\ b \end{bmatrix} \tag{2.26}$$

$$\mathrm{H}\phi = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \tag{2.27}$$

$$\mathrm{XH}\phi = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \tag{2.28}$$

$$\mathrm{HXH}\phi = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \tag{2.29}$$

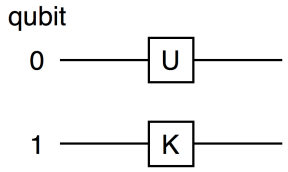
$$\mathrm{HXH}\phi = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \mathrm{Z}\phi \tag{2.30}$$

In this example, if the Z gate were supported on the target device, we could rewrite the overall transform as a single-qubit circuit containing a single Z gate. This is a trivial example, but it shows that the same global transform can have numerous circuit implementations which directly correspond to matrix factorizations of that transform.

Additionally, applying two gates in parallel corresponds to applying the tensor product of these gates to the joint state vector. To see this, assume a 2-qubit circuit with arbitrary gates applied in parallel. We will denote the upper gate U and the lower gate K.

We show below that applying U to qubit 0 and K to qubit 1 is equivalent to applying $U \otimes K$ to the joint state vector of qubits 0 and 1.

$$U = \begin{bmatrix} u_0 & u_1 \\ u_2 & u_3 \end{bmatrix} \quad K = \begin{bmatrix} k_0 & k_1 \\ k_2 & k_3 \end{bmatrix} \tag{2.31}$$



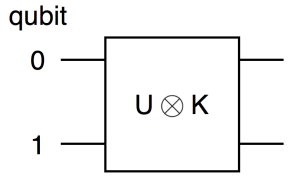
$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \implies \left(U \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) \otimes \left(K \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} u_0 \\ u_2 \end{bmatrix} \otimes \begin{bmatrix} k_0 \\ k_2 \end{bmatrix} = \begin{bmatrix} u_0 k_0 \\ u_0 k_2 \\ u_2 k_0 \\ u_2 k_2 \end{bmatrix} \tag{2.32}$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \implies \left(U \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) \otimes \left(K \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} u_0 \\ u_2 \end{bmatrix} \otimes \begin{bmatrix} k_1 \\ k_3 \end{bmatrix} = \begin{bmatrix} u_0 k_1 \\ u_0 k_3 \\ u_2 k_1 \\ u_2 k_3 \end{bmatrix} \tag{2.33}$$

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} \implies \left(U \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) \otimes \left(K \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} u_1 \\ u_3 \end{bmatrix} \otimes \begin{bmatrix} k_0 \\ k_2 \end{bmatrix} = \begin{bmatrix} u_1 k_0 \\ u_1 k_2 \\ u_3 k_0 \\ u_3 k_2 \end{bmatrix} \tag{2.34}$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \implies \left(U \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) \otimes \left(K \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} u_1 \\ u_3 \end{bmatrix} \otimes \begin{bmatrix} k_1 \\ k_3 \end{bmatrix} = \begin{bmatrix} u_1 k_1 \\ u_1 k_1 \\ u_3 k_3 \\ u_3 k_3 \end{bmatrix} \tag{2.35}$$

$$\implies \begin{bmatrix} u_0 k_0 & u_0 k_1 & u_1 k_0 & u_1 k_1 \\ u_0 k_2 & u_0 k_3 & u_1 k_2 & u_1 k_3 \\ u_2 k_0 & u_2 k_1 & u_3 k_0 & u_3 k_3 \\ u_2 k_2 & u_2 k_3 & u_3 k_2 & u_3 k_3 \end{bmatrix} = U \otimes K \tag{2.36}$$



Using the matrix and tensor product identities, this means that a quantum circuit can be fully reduced to a single, unitary transformation in a 2^N-dimensional complex vector space. A specific circuit implementing this transform is just a sparse factorization written in terms of quantum gates and the identity operator.

Phrased in this way, quantum circuit generation lies completely in the realm of traditional linear algebra. For example, a 4×4 Walsh-Hadamard transform is commonly decomposed as such.

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & 1\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & -1\end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 1 & -1\end{bmatrix} \tag{2.37}$$

$$\mathrm{H}_4 = (\mathrm{H}_2 \otimes \mathrm{I}_2)(\mathrm{I}_2 \otimes \mathrm{H}_2) = (\mathrm{H}_2 \otimes \mathrm{H}_2) \tag{2.38}$$

Where H_2 is the 2×2 Hadamard matrix, H_4 is the 4×4 Hadamard matrix, and I_2 is the 2×2 identity matrix. Viewing this mathematical deconstruction in terms of quantum computing, Equation (2.38) directly gives us a circuit implementation of the desired transform. Specifically, it implies that a 4-qubit Hadamard transform can be implemented by applying a 2×2 Hadamard gate to each input qubit. Additionally, the Cooley-Tukey decomposition rule for the discrete Fourier transform (DFT) [25] also directly yields a recipe for generating a quantum circuit.

$$\mathrm{DFT}_n = (\mathrm{DFT}_k \otimes \mathrm{I}_m)\mathrm{T}^n_m(\mathrm{I}_k \otimes \mathrm{DFT}_m)\mathrm{L}^n_k; n = km$$

Where T^n_m is a diagonal matrix of twiddle factors and L^n_k is a stride permutation. As will be shown later, twiddle factors directly correspond to controlled rotation gates, and permutation matrices directly correspond to a sequence of data movement gates, or SWAP gates. Because a quantum circuit is simply a graphical representation of a mathematical decomposition, SPIRAL, or any other computer algebra system, is uniquely predisposed to search over these decompositions, as it robustly does so already for a variety of functions. This mathematical notation captures a quantum program much more effectively than the program stream model of QASM, and is in our view the most general representation available.

We now constrain this equality between circuits and matrix factorizations by introducing the concept of connectivity. If a multi-qubit operation cannot be factored into individual 2×2 operations, this transform inherently requires communication between the operand qubits. Due to the in-place nature of quantum computing these values must be physically adjacent in the computer itself, meaning that any circuit that does not meet this constraint is invalid.

2.3.2 Connectivity

Besides single-qubit gates, there additionally exist multi-qubit gates that act on the joint state vector directly rather than being decomposable into independent and parallel operations. This may have been inferred already, as the Bell state would be impossible to construct without them. These gates take the form of *controlled* gates, where a controlling qubit toggles the application of a single-qubit gate to another. The CNOT gate (or $\mathrm{Controlled_X}$ gate) is one such example, where the rotation being applied is that characterized by the X gate. Generically we can write these 2-qubit gates as such.

$$U = \begin{bmatrix} u_0 & u_1 \\ u_2 & u_3 \end{bmatrix}, \quad \text{Controlled}_U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_0 & u_1 \\ 0 & 0 & u_2 & u_3 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & u_0 & 0 & u_1 \\ 0 & 0 & 1 & 0 \\ 0 & u_2 & 0 & u_3 \end{bmatrix} \tag{2.39}$$

The leftmost definition of Controlled_U has qubit 0 controlling and the other with qubit 1 controlling. It is worth noting that measurement and the destructive effects accompanying it are not applied here. Rather, these gates can simply be seen as unitary transformations that are applied to a two-qubit joint state vector, and which happen to not be factorable into the tensor product of individual 2×2 matrices. These gates are extremely important for nearly all relevant quantum algorithms, as without them it is impossible to do any multi-qubit computation that isn't simply parallel computation on individual qubits.

However, inherent physical limitations necessitate the qubits being operated on to be *physically connected* in the hardware architecture, meaning the hardware locations these qubit values are stored in must be linked by circuits or buses in the chip itself. Due to hardware constraints, cost and manufacturing limitations not being the least of which, these hardware locations are typically only sparsely connected, meaning that qubit values often have to be dynamically moved around in order to satisfy the connectivity constraints of each gate in the circuit. Connectivity maps for various IBM devices are shown in Figure 2.5.

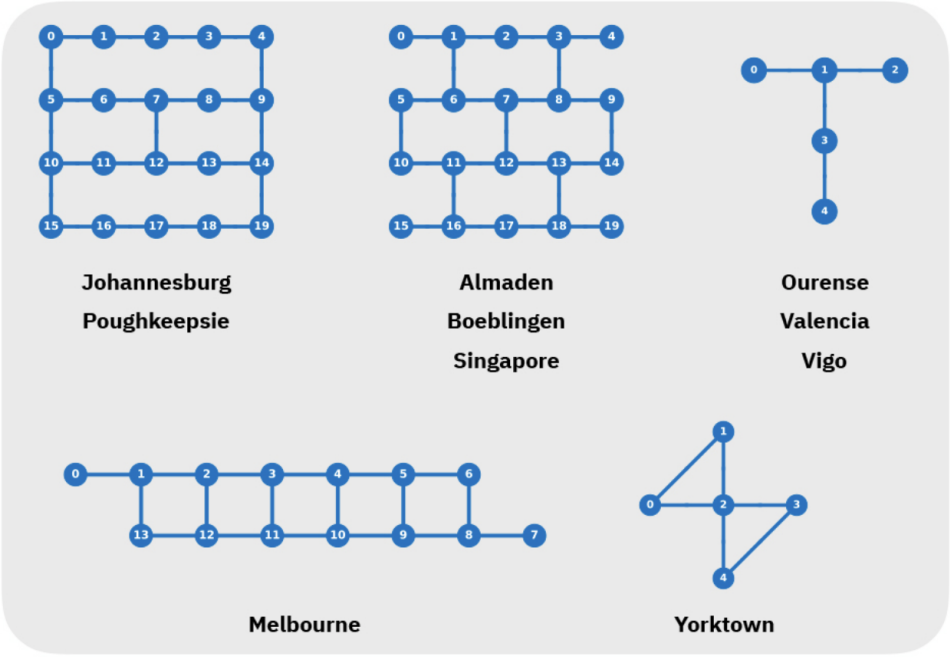


Figure 2.5: Various IBM quantum computer connectivity maps. Sourced from [44].

For readers familiar with the backend of a traditional compiler and with the x86 architecture, these connectivity constraints can be viewed similarly to how the *div* instruction requires certain

operands to be in particular registers. Similarly to how classical compilers will insert register-to-register moves to meet these requirements, quantum compilers will also insert swaps to ensure connectivity is met.

Since quantum programs are better viewed as mathematical objects than a sequence of imperative instructions, however, we internally consider data movement to be a specific class of permutation matrix that permutes the *logical-to-physical* mapping of values to hardware locations. This is augmented by the fact that there are additional limitations in the quantum world; the no-cloning theorem [66] means that qubit values cannot be duplicated or saved, implying that somehow capturing a qubit value and storing it in memory like a stack location is quite impossible. This is a big reason why the data movement primitive in quantum systems is the SWAP operation and not a copy or move operation. Despite these differences, however, the concepts are ultimately the same.

SWAP implements a permutation matrix that swaps the elements of a joint state vector.

$$\mathrm{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.40}$$

Noticeably, this operation can be decomposed 2 different ways

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.41}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.42}$$

Which, in circuit form, correspond to two different series of CNOT gates. In fact, any higher-level mathematical object can be expressed purely in terms of controlled gates and single-qubit gates.

We can now reconcile this nomenclature with the overall goal stated in the introductory chapter. We initially expressed our desire to treat quantum circuits as sparse matrix factorizations, and thus be able to generate circuits that implement a higher-level transform simply by performing matrix decompositions. We have seen in this section that it is possible; every quantum circuit (excluding those with intermediate measurements) can be written as a sparse factorization of an overall $2^N \times 2^N$ transform. We have also seen that traditional divide-and-conquer rules from the classical domain, such as those used to decompose the Fourier or Hadamard transforms, are exactly recipes for creating quantum circuits. We also know that not all matrix factorizations are valid circuits for a given architecture, and thus in the next section we will have to remember the relevant connectivity information as we recursively deconstruct the input. We will next use these concepts to formalize a generic search problem and propose a method for solving it.

2.4 Problem Formulation

We have derived the basics regarding quantum circuits, their formulation as mathematical objects (namely high-dimensional unitary transformations), and the constraints imposed by the hardware. Now we formalize the optimization problem before applying SPIRAL as a solver in Chapter 3.

2.4.1 Formalization

We now formalize the quantum optimization problem as

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

Where Mat is the desired $2^N \times 2^N$ transform matrix, and Factorizations is an operator on this matrix that returns a set of valid decomposition into the tensor and matrix product of quantum gates. It is valid to express such a decomposition as an expression tree, the language we use to denote this being expressed in Table 2.1.

$\langle basic_gate \rangle_1 ::=$	H	
	X	
	Z	
	Y	
	I	
	...	
$\langle cnot \rangle_2 ::=$	CNOT_1	# qubit 1 controls qubit 0
	CNOT_0	# qubit 0 controls qubit 1
$\langle circuit \rangle_n ::=$	$\langle circuit \rangle_n * \langle circuit \rangle_n$	
	$\langle circuit \rangle_k \otimes \langle circuit \rangle_m$	where n = mk
	$\langle cnot \rangle_2$	where n = 2
	$\langle basic_gate \rangle_1$	where n = 1

Table 2.1: Expressing a valid unitary matrix decomposition in Backus-Naur form (BNF) [4].

Finally, given a complete set of factorizations (and hence a complete set of circuit implementations), we want to find the best one with respect to some cost measure. To minimize data-movement instructions, Cost would simply be the number of SWAP gates in the circuit.

2.4.2 General Approach

While SPIRAL is an extremely useful framework for enacting the proposed circuit generation procedure, the procedure itself is not inherently tied to the SPIRAL system. Therefore, we present a general approach before describing implementation details in the next chapter.

The first major phase will be a decomposition phase, or breakdown phase, that directly implements the Factorization procedure outlined above. We will start with a high-level, symbolic

representation of the desired transform (or a composition of transforms), and by recursively applying a series of divide-and-conquer decomposition rules, we will construct a sparse factorization that uniquely maps to a circuit implementation of the desired transform. This factorization can be written as an expression tree, as shown in Figure 2.6.

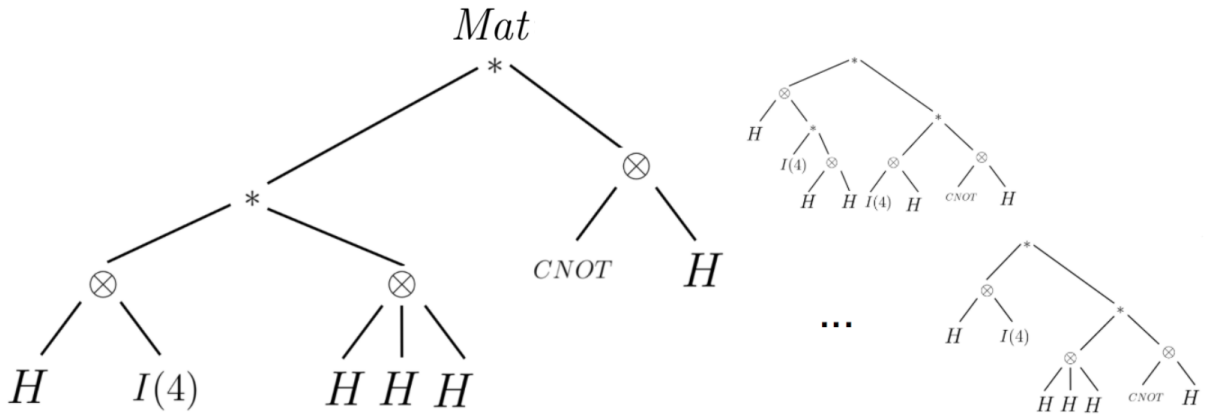


Figure 2.6: Expression tree factorizations of transform Mat.

An example of such a decomposition rule is the Cooley-Tukey rule for FFTs. This rule breaks a large DFT object into two smaller DFT objects, which can then be recursively decomposed. The rule terminates when all DFTs are of size 2, which is when we hit a base case that equates the $\mathrm{DFT}(2)$ to the Hadamard gate H. Notice that the span of possible factorizations (i.e. the space of possible expression trees) is immense. Even just Cooley-Tukey can be applied several different ways.

The recipe for forming a particular expression tree is a specific sequence of rule applications, which we will call a *rule tree*. Our task is to search over all rule trees that decompose the input; this equates to searching over all decompositions of the input, and thus, all circuits that implement the input.

Some rule trees are invalid; rules may fail if connectivity is not met for the controlled operations we attempt to place. In this case we can backtrack to the last applied rule. In order to detect this case we must remember the subset of qubits we are acting on in each recursive subproblem. This approach is directly inspired by backtracking proof search in an area like constructive logic [29], in which a context is generally kept. This also allows us to employ heuristics at a rule-by-rule granularity, and since we have both algorithmic and architectural information, we should be able to make intelligent decisions to prune unpromising rule trees early on in the search procedure. By enumerating all valid rule trees in our search space, we also enumerate all possible factorizations, and hence all valid circuits for the target architecture.

Once we have a procedure capable of forming all valid expression trees (or at least a relevant subset of expression trees), we should be able to simplify these expression trees by condensing branches and cancelling leaves. This constitutes the local, peephole simplifications that make up the larger part of existing compilation approaches, and is displayed in Figure 2.7.

The group of all such simplified expressions forms a space over which we wish to minimize the number of SWAP gates. We will search over a heuristically-pruned subset of all possible rule

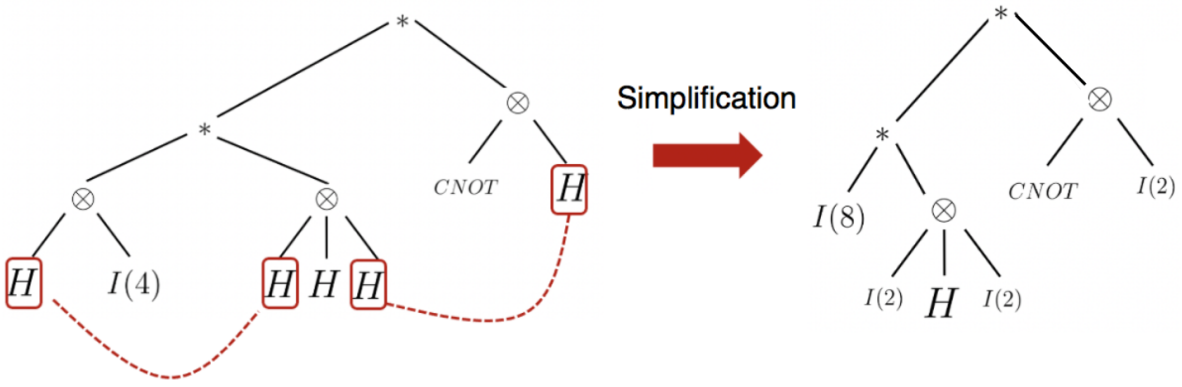


Figure 2.7: Expression tree contraction and simplification.

trees to find the one that, when applied to the input, yields the cheapest circuit representation after applying our simplification rules. Treating the rule tree as the x variable and the other compilation stages as a projection operator F that takes this x and maps it to a specific point in the circuit space, we can treat this search problem quite generically.

$$x_{opt} = \arg\min_{x} \text{Cost}(\text{F}(x))$$

We can apply any of the myriad minimization methods that have been studied by computer scientists to solve this problem.

These circuit decompositions exhibit the subformula property in the sense that branches of these expression trees that are linked by matrix multiplications are themselves valid subcircuits; this means we can evaluate the cost of various subtrees and use this feedback to inform which rules we try. SPIRAL will leverage this property by applying a dynamic programming search to solve this problem. We also have access to both the connectivity graph and the symbolic algorithm representation at each decision point in the breakdown process, and can use these to inform possible heuristics. These methods are enumerated and described in the next chapter.

2.5 Conclusions

In this chapter we summarized the fundamental principles governing the construction and execution of quantum programs. We showed that the building block of quantum computers, the qubit, can be written as a 2-dimensional complex vector, and that the state of an N-qubit system can be written as a 2^N-dimensional complex vector. Therefore, we derived that every quantum circuit (excluding those with intermediary measurement gates) directly implements a $2^N \times 2^N$ linear transform on that state vector, and that this transform matrix is simply the tensor product of parallel quantum gates and the matrix product of sequential ones. We also established that there is typically a very large space of quantum circuits that implement the same $2^N \times 2^N$ transform.

Given this, we concretely formalized our proposed approach. Instead of starting with a circuit and having to derive the transform from it, a process which is computationally infeasible for large N, we propose starting with a symbolic representation of the transform, from which we

can derive a host of circuits. We can do this by factoring the symbolic input into a sparse matrix representation that can be written purely in terms of quantum gates; this decomposition can be written as an expression tree. Given a set of rules that govern how to decompose symbolic objects into gates, we can try breaking down the input in many different ways and enforce connectivity by constraining the applicability of certain rules. We can then search over these rule trees to find the one that yields the best circuit. For this, we leverage SPIRAL.

Chapter 3

SPIRAL Quantum Compiler

This chapter outlines the primary contribution of this thesis: a novel framework for generating optimized quantum circuits. Section 3.1 revisits the generalized approach presented in the previous chapter in order to refine the requirements that the SPIRAL implementation must meet. Section 3.2 then gives a brief overview of the SPIRAL system, its capabilities, and the core features which are leveraged in the quantum compiler. Section 3.3 gives an overview of the quantum compiler and code generator, showing in detail how we construct and search over quantum circuits with recursive breakdown rules and an extensive rewriting system. Conclusions are drawn in Section 3.4. We further discuss algorithm-specific heuristics in Chapter 4.

3.1 Approach

We now revisit the general approach outlined in the previous chapter and refine it further to reflect SPIRAL's implementation.

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

To solve the variant of the problem that is posed above, there are a few outstanding requirements. First, we must construct a system that efficiently implements Factorizations; in principle, this system would be able to construct *all* non-trivially unique factorizations of the input matrix. Additionally, we must be able to do so without expanding Mat itself since this matrix is exponentially large with respect to the number of qubits; this implies that Mat must be expressed symbolically. These considerations suggest a top-down rules-based approach; algorithmic decompositions, especially those based on group-theoretic symmetries like FFT decompositions, are nearly impossible to discover and employ when simply inspecting a pre-compiled program stream that cannot be expanded. Additionally, since the space of $\text{Factorizations}(Mat)$ expressions is far too large to expect decent search performance, Factorizations should be intelligent enough to prune obviously suboptimal factorizations by using the symbolic Mat definition, and the transforms it breaks down into, to choose decent heuristics. Critically, the architecture of the target device is also provided as input, and so it can also be used to inform these heuristics, specifically to ensure we are finding a reasonably optimized placement of the desired algorithm

on the target hardware.

Next, we must ensure that our search procedure is efficient. The Cost operator should be lightweight, and ideally it should grow at most asymptotically with respect to the circuit depth; this would approximately be the cost of linearly scanning the circuit for swap operations, not taking into account the complexity of scanning the nested contents of a tensor operator itself.

Finally, we should leverage well-studied search techniques like dynamic programming to efficiently evaluate and select $\mathrm{circuit}_{\mathrm{opt}}$. We will, in the rest of this chapter, enumerate the features of SPIRAL that allow us to implement the aforementioned system.

3.2 SPIRAL

SPIRAL [26] is a program generation system for linear transforms and other functions, and has been successful in producing extremely high performance code for a variety of hardware architectures. SPIRAL takes in a high-level algorithm specification, and along with other architectural and microarchitectural parameters, outputs optimized code in a variety of languages and for a variety of accelerated architectures [46]. SPIRAL has been developed at Carnegie Mellon University over the past 20 years alongside industry partners; it has been both the subject of numerous successful publications and has found commercial use for generating vendor math libraries. It is built on the GAP [57] computer algebra system, and thus has the advantage of leveraging both a large library of algebraic objects and native support for computational group theory.

3.2.1 GAP

GAP is a system for computational discrete algebra, particularly emphasizing group theory. GAP consists of a programming language, a large library of algebraic objects that can be accessed in the language and a massive array of functions that implement important mathematical operations. The GAP language is interpreted, can be compiled, and despite having control structures very similar to those of Pascal [65], it supports a variety of operators inspired by more functional languages. Through a variety of modular packages, functions and library objects which are mostly written in the GAP language, advanced mathematical capabilities are provided for a wide variety of tasks including computing the properties of groups, constructing graphs, solving polynomials and factoring prime numbers. SPIRAL is built on the GAP system and leverages the basic capabilities it provides to construct a code generation framework.

3.2.2 Intermediate Representations

SPIRAL compiles an input by transitioning through several internal representations that progressively lower to hardware assembly, similarly to a traditional compiler. The top-level input to SPIRAL is a linear transform, as expressed by a symbolic object called a *non-terminal*. Mathematically, any linear transform can be expressed succinctly as the following

$$x \longmapsto Mx \tag{3.1}$$

Where x is the input vector and M is a fixed matrix implementing the desired transform. In the quantum case we will discuss in the next section, every M is square.

The question remains of how best to map this computation onto the target hardware. Directly computing this matrix, without applying any optimizations that may result from the sparsity of the matrix, will require $O(n^2)$ many operations. However, most algorithms of this nature are not randomized, but rather exhibit symmetry that can be leveraged to reformulate this computation into a cheaper alternative. The most famous example of this transformation is the $O(n \log n)$ FFT algorithm that implements the otherwise $O(n^2)$ DFT. Additionally, these reformulations can be expressed as a factorization of M into a product of sparse matrices. This sparse matrix formula is represented in a lower-level syntax called *Signal Processing Language*, or SPL [67]. The decomposition of non-terminal M into the sparse formula expressed in SPL is achieved through a recursive rule-based procedure, similar in motivation to proof search in a logical language like Prolog [12]. Each rule in this system, called a *breakdown rule*, is a divide-and-conquer algorithm that breaks a non-terminal into smaller and smaller pieces until the formula is finally expressed purely in terms of SPL objects. The most famous example of such a rule is the Cooley-Tukey FFT rule discussed previously, and others are shown in Figure 3.1. The motivation behind this transformation is that each sparse operation in the SPL language can be treated as a separate computational kernel which can be mapped to a variety of vector architectures or hardware accelerators. In this manner, SPIRAL has reduced a high-level operation into a sequence of small kernels that can be more efficiently implemented. In the quantum world, each SPL object is a quantum gate or operation, but this transformation serves the same underlying purpose.

There is much latitude in deciding which SPL formula is the optimal representation of M for the target hardware. It may be the case that certain kernels are easier to compute than others. Therefore, in order to find the cheapest SPL representation for our target, we must perform a search over all ways of translating the matrix M into an SPL formula. If we call a specific way of decomposing M a *rule tree*, or rather a partial ordering of rule applications, this problem equates to searching over all rule trees. SPIRAL natively implements a dynamic programming search procedure that efficiently performs this search with respect to some cost measure, and hence is able to find the best low-level representation for a high-level transform. Heuristics play a large role in limiting the number of rule trees that SPIRAL has to search over.

$$\mathrm{DFT}_{mn} \longmapsto (\mathrm{DFT}_m \otimes \mathrm{I}_n)\mathrm{D}_{m,n}(\mathrm{I}_m \otimes \mathrm{DFT}_n)\mathrm{L}^{nm}_m \tag{3.2}$$

$$\mathrm{DFT}_n \longmapsto \mathrm{X}_n\mathrm{RDFT}_n \tag{3.3}$$

$$\mathrm{WHT}_n \longmapsto \mathrm{WHT}_m \otimes \mathrm{WHT}_n \tag{3.4}$$

$$\mathrm{MDDFT}_{n_1\times\ldots\times n_k} \longmapsto \mathrm{MDDFT}_{n_1\times\ldots\times n_r} \otimes \mathrm{MDDFT}_{n_{r+1}\times\ldots\times n_k} \tag{3.5}$$

$$\mathrm{MDDFT}_n \longmapsto \mathrm{DFT}_n \tag{3.6}$$

$$\mathrm{Haar}_n \longmapsto \mathrm{L}^n_2(\mathrm{I}_{n/2} \otimes \mathrm{DFT}_2) \tag{3.7}$$

$$\tag{3.8}$$

Figure 3.1: Breakdown rules in classical SPIRAL.

SPIRAL then implements a backend compiler to reduce the SPL language into code for the target hardware, often progressing through a series of further intermediate representations. For the quantum application that we will discuss in the next section, the target language will be QASM, and we will similarly perform a series of simplification passes to ultimately reduce our SPL expression into a directly executable assembly file.

3.2.3 Rewriting System

SPIRAL contains an extensive multi-stage rewriting system [27] for simplifying SPL formulas; this is one such pass that helps convert the SPL formula into real code. This phase applies a series of semantics-preserving identities such as distributing multiplication across addition, cancelling inverse matrices and other local modifications. Nearly any rule involving local variable substitution can be simplistically framed as a rewrite rule, which the SPIRAL rewrite engine will then attempt to apply, along with any other chosen rules, wherever applicable.

In our work, we leverage this system to implement a circuit simplification pass that runs after the breakdown stage, both removing a few compiler internals and applying quantum gate identities such as $\mathrm{H} \cdot \mathrm{H} = \mathrm{I}$ (which captures the fact that the Hadamard gate is its own inverse). Translating between different sets of basis gates can also be done through this system.

3.3 Quantum Circuit Generation

Now that we have covered the general architecture of the SPIRAL system, we detail how to leverage this program generation infrastructure to solve the problem formalized in the previous chapter. We elaborate on the architecture of the SPIRAL quantum compiler, which we will often refer to as "QSPIRAL" for brevity. We show that a generative and rule-based approach, coupled with the full power of the GAP computer algebra system, lends itself extremely well to generating optimized quantum circuits. In fact, the quantum system itself is implemented as a modular package that can be attached to the open-sourced SPIRAL framework, and the existing classical framework almost exactly provides the tools we need to compile quantum circuits.

QSPIRAL takes, as input, a high-level symbolic representation of the algorithm and an adjacency matrix describing the hardware connectivity map (Section 3.3.1). Then, QSPIRAL searches over all rule trees in the search space, trying out the various ways of decomposing the input algorithm into low-level circuits that will run on the target architecture. To evaluate the cost of a specific rule tree (i.e. the cost of the circuit it produces), we first apply it to the input to obtain an SPL expression (Section 3.2.2), and then convert this SPL expression to QASM with the backend rewrite system (Section 3.2.3). The resulting QASM code can then be evaluated based on the number of SWAP operations it contains. This forms a generic minimization problem; the search procedure will attempt to find the best rule tree, and the circuit this rule tree produces becomes the compiler output. This solution is not necessarily unique, and the output selected will depend heavily on the heuristics used to traverse the space of rule trees. Examples of these heuristics for a specific algorithm are provided in Chapter 4, but these are useless without a generalized system within which they can be applied. QSPIRAL is such a system, and is described in this section.

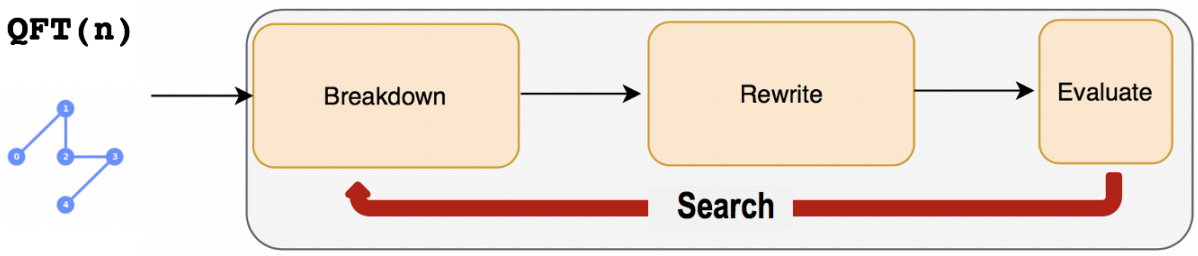


Figure 3.2: Control flow graph of QSPIRAL compiler stages.

We will discuss the various phases of the compiler (Figure 3.2) in sequence, starting with the system inputs, detailing the breakdown phase, and then describing the internal workings of the rewrite phase. Finally, we describe the search procedure, and how the tight integration of these steps can enable advanced search techniques.

3.3.1 System Inputs

The primary input to QSPIRAL is a high-level representation of the desired algorithm. An algorithm in QSPIRAL, like in classical SPIRAL, is a transform non-terminal that symbolically represents a linear transform. We can capture important quantum kernels as symbols in our domain-specific input syntax, eliminating the need to accept arbitrary $2^N \times 2^N$ matrices as input (which would be impractical). A subset of these symbols is shown in Table 3.1.

$\mathrm{qHT}(n)$	Walsh-Hadamard transform on n qubits
$\mathrm{qXT}(n)$	Pauli X matrix on n qubits
$\mathrm{qYT}(n)$	Pauli Y matrix on n qubits
$\mathrm{qFT}(n)$	quantum Fourier transform on n qubits
$\mathrm{qReord}(\ell, n)$	Permutation ℓ applied on n qubits

Table 3.1: Subset of QSPIRAL non-terminals.

With this notation we have succeeded in allowing the user to compile certain operators, but practical quantum circuits rarely perform a *single* transform on all qubits in the system; Shor's algorithm, for example, includes the QFT alongside other transformations. Therefore, we need a way to form larger composite algorithms from these non-terminals. Our solution is an additional non-terminal called qCirc, which represents several chained non-terminals applied to different portions of the input vector. The qCirc object is itself a non-terminal, and hence can be recursively instantiated as shown in Table 3.2.

The $index_list$ construct indicates which logical qubits are affected by the $non_terminal$ in question. These indices are entirely arbitrary with respect to the initial logical-to-physical mapping; the programmer does not care about the physical location of qubit index k as long that qubit undergoes the intended operations. However, due to connectivity requirements and the cost of swapping values, the hardware very much cares about the physical location of qubit index k.

$non_terminal ::=$	$\mathrm{qCirc}(n, \langle op_list \rangle)$
	$\mid \mathrm{qHT}(n)$
	$\mid \mathrm{qFT}(n)$
	...
$op_list ::=$	$[\langle index_list \rangle, \langle non_terminal \rangle] :: \langle op_list \rangle$
	$[\,]$
$index_list ::=$	$[\langle nat \rangle] :: \langle index_list \rangle$
	$[\langle nat \rangle]$
$nat ::=$	n where $n \in \mathbb{N}$

Table 3.2: QSPIRAL algorithm syntax.

Ideally, QSPIRAL would automatically put qubit index k in the best hardware location, given the operations it has to undergo and the other qubits it has to communicate with. Determining a good starting configuration is done in the global reordering phase of the rewrite system (Section 3.3.4).

Also provided as input is the connectivity map of the target hardware, as expressed in an $N \times N$ adjacency matrix. While other representations are available, such as Compressed Sparse Row (CSR) representation [21], the qubit adjacency matrix scales with N rather than 2^N and therefore the uncompressed representation does not cause a memory bottleneck in our system. Additionally, since matrix multiplication is efficiently implemented in GAP, the adjacency matrix representation allows simple path-finding techniques (e.g. analyzing the powers of the adjacency matrix) to be used internally, which proves useful in several situations. The top-level QSPIRAL input, usually a qCirc object, is tagged with this adjacency information, and hence it is available throughout the entire decomposition process. As we subdivide our formula in the breakdown phase, this adjacency matrix gets smaller, effectively yielding a monotonically non-increasing problem size.

3.3.2 Formula Breakdown

After a quantum algorithm is captured in the high-level syntax, the breakdown phase decomposes our target object through a series of divide-and-conquer decomposition rules. In this section we describe rule trees and how they can be used to convert the symbolic input into an SPL expression. The resulting SPL expression is *almost* a quantum circuit, but in addition to quantum gates it also contains a few compiler internals, and thus further compilation is needed. This is achieved by the rewrite phase described in the next section, which finally converts SPL into QASM.

Given a QSPIRAL non-terminal, we can search through our internal library of decomposition rules to find those that are applicable. Rules of this nature, as mentioned before, are similar to the Cooley-Tukey rule for Fourier transforms; they break a larger non-terminal into the matrix and tensor product of smaller ones. When a non-terminal is sufficiently small it can be converted into an SPL object. Once all non-terminals have been reduced to SPL objects, the breakdown is complete. At each step, there are usually many rules that can be applied, thus leading to several

different ways of reducing the input; each such sequence of rules is called a rule tree. An example of the breakdown rules for the Hadamard transform is shown in Figure 3.3.

$$\mathrm{qHT}_n \longmapsto \mathrm{qHT}_k \otimes \mathrm{qHT}_m; m = nk \tag{3.9}$$
$$\mathrm{qHT}_1 \longmapsto \mathrm{H} \tag{3.10}$$

Figure 3.3: Breakdown rules for quantum Hadamard transform.

In this trivial decomposition scheme, a Hadamard transform can be decomposed into the tensor product of smaller Hadamard transforms, and eventually all Hadamard transforms of size 1 can be terminated as a single H quantum gate. This approach transfers directly from classical SPIRAL; the space of circuits that implement a transform matrix is exactly the set of decompositions of the matrix into compositions of basis gates. We can draw parallels between the quantum and classical domains; Equation (3.9) is almost directly taken from SPIRAL's pre-existing library of sparse factorization rules.

While a large subset of the decomposition rules used in classical SPIRAL are immediately applicable to the quantum domain by virtue of being expressed in the same mathematical language, additional decomposition rules must be inserted to handle our custom qCirc transform object. Recall that this object represents the application of various non-terminals to different logical qubits, and since quantum computers execute in-place, this object can be decomposed in many different ways. Specifically, each of the non-terminals in the argument list should be applied to its specified logical qubits, but these logical qubits can be located in any set of hardware locations. We can effectively compute each of these non-terminals on any subset of hardware locations in the device by inserting the proper data movement operations. These different placements reflect the wealth of scheduling options available, and our breakdown system must allow us to explore these options such that we can search over them. This means that the breakdown rules for the qCirc object concern not simply algorithmic decompositions but the partitioning of hardware qubits, a process we call *embedding*. In order to understand this process for complex algorithms, let us first focus on embedding a single non-terminal.

Starting with a single transform, let us analyze the task of embedding this transform onto a quantum device. If this transform computes on k qubits, and the device has a total of N hardware qubits, this transform can be executed in-place on any subgroup of k qubits in the device. There are loosely $\frac{N!}{(N-k)!}$ choices for this, not considering that in most cases, the size-k subgroup has to be connected. Various placements of these logical qubits for a given transformation are shown in Figure 3.4.

The breakdown system should allow us to apply various rules to rewrite the transform in terms of any of these arrangements, thereby allowing us to search over these arrangements by enumerating the various rule trees. For readers familiar with classical compiler internals, this logical-to-physical mapping could be viewed as a register allocation, with logical qubits being temporary values and hardware locations being registers. We are, in essence, choosing a register allocation on which to compute our non-terminal. Since quantum computers execute in-place, the performance of this non-terminal will be better or worse depending on which registers its

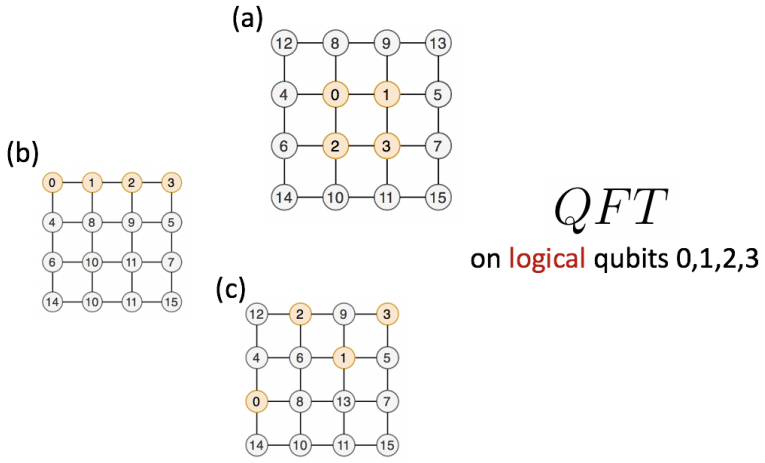


Figure 3.4: Possible placements for non-terminal QFT on a 4×4 lattice; scattered placement (e.g. (c)) will be rejected due to connectivity constraints. In this example, $k = 4$ and $N = 16$.

operands are placed in.

As stated in the introductory chapter, the primary measure of performance used in this work is the amount of data movement needed, so the performance is purely determined by the physical positioning of the input values with respect to the other qubits they might need to communicate with. The cost then (in our case, the number of SWAP instructions) of any arrangement is twofold; the *execution* cost is the cost of executing the desired transform in-place on the chosen hardware locations, and the *staging* cost is the cost of moving the desired operand qubits into and out of those hardware locations. In the classical register allocation analogy, the performance of each assembly instruction would be better or worse depending on which registers its operands are placed in (execution cost), and microarchitectural limits placed on which register-to-register moves are allowed in the ISA mean that there are varying costs of actually getting the desired temporaries into those registers to start with (staging cost). This dichotomy between execution and staging costs is a primary example of how circuit optimization is a balancing act between local and global optimization; minimizing execution cost may require moving logical qubits into physical locations that are impractical given their starting positions. For now, we assume some arbitrary initial mapping and optimize this further in the global reordering phase.

Given a breakdown system that allows us to write an embedded transform in terms of any valid arrangement of qubits, it is now necessary to delve into how one such arrangement can be expressed in the target SPL language; this language, you may recall, represents a sparse matrix factorization of the input. Luckily, sequences of swap operations can simply be written as permutation matrices (a fact we showed in Chapter 2), meaning that we can write this breakdown rule similarly to the following.

$$\text{Embed}(N, L, T) \quad \longmapsto \quad R_N^L \, \text{Apply}(T, L) R_N^{-L}; \text{ for any valid } R_N^L \tag{3.11}$$

Where N is the total number of hardware qubits, L is the ordered list of logical operand qubits (where $|L| = k$) and T is the transform we wish to apply to the qubits in L. R_N^L is any

$N \times N$ permutation matrix that rearranges the qubits in L and R_N^{-L} simply restores the original ordering by undoing this permutation. Since permutation matrices are ultimately decomposed into SWAP gates, the R_N^L matrix has the effect of putting the logical qubits in L anywhere we want in the N-qubit architecture. We can write this formula in terms of any logical-to-physical mapping by simply writing it in terms of a different R_N^L matrix. If $R_N^L = \mathrm{I}(N)$, or rather the N-qubit identity matrix, we would incur a staging cost of 0.

Now that we have effectively chosen which hardware qubits we want to execute T on by virtue of selecting an R_N^L matrix, we would like to continue the breakdown by applying the T decomposition rules as shown in Figure 3.3 for the case of $T = \mathrm{qHT}$. To do this, we must unwrap the $\mathrm{Apply}(T, L)$ operator, ideally representing it as a tensor product of the symbolic T matrix with the identity operator. This is slightly problematic, however, if the R_N^L matrix places the input qubits in hardware locations that are not adjacent in the canonical state vector ordering, as shown in Figure 3.5. In this case, it is difficult to express the operation as a single tensor product, and hence it is an issue if we wish to maintain the invariant that, at every breakdown stage, our formula remains a valid matrix factorization of the input.

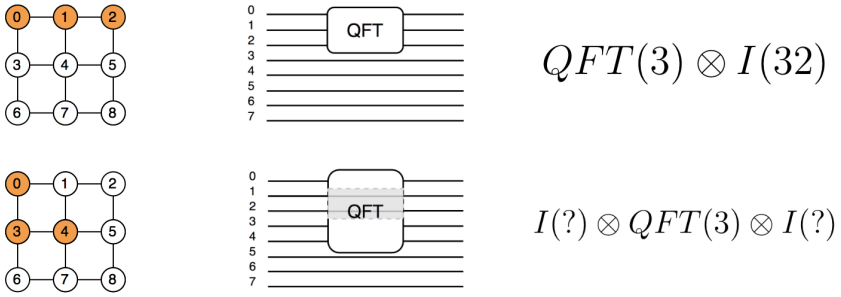


Figure 3.5: Scheduling a QFT transform on adjacent [top] and non-adjacent [bottom] qubits. Directly stating the bottom operation as a tensor product is problematic.

To handle this we introduce an additional matrix, J_N^L, that freely reorders the state vector and places the qubits in L in adjacent vector indices such that we can express the application of T succinctly as a tensor product; we arbitrarily choose the top k indices (i.e. $0, \ldots, (k-1)$). This permutation matrix is a compiler internal and will be fully removed once decomposition is complete, but it allows us to flatten the representation as shown.

$$\mathrm{Embed}(N, L, T) \quad \longmapsto \quad R_N^L J_N^L (T \otimes I(N - |L|)) J_N^{-L} R_N^{-L}; \text{ for any valid } R_N^L \qquad (3.12)$$

We will not decompose R_N^L or J_N^L any further because these objects will be compiled out in the rewrite system; J_N^L will get removed completely and R_N^L will be simplified and eventually converted to sequences of SWAP gates. For now, however, we can continue to break down T into an SPL formula consisting of quantum gates, having made the high-level scheduling decision of what hardware qubits T is computed on.

For multiple transforms that need to be embedded in sequence, we can simply take the matrix product of the embedding of each one, as follows.

$$\text{Circuit}(N, [[L_1, T_1], [L_2, T_2]]) \longmapsto \text{Embed}(N, L_1, T_1)\,\text{Embed}(N, L_2, T_2) \quad (3.13)$$

Now that we have discussed how to partition qubits with breakdown rules, we can reconcile the above approach with our qCirc syntax. We modify our high-level syntax with an additional qEmbed object, and explicitly tag our non-terminals with the $arch$ adjacency matrix (Table 3.3).

$non_terminal ::=$	$\text{qCirc}(n, arch, \langle op_list \rangle)$
	$\text{qEmbed}(n, arch, [\langle index_list \rangle, \langle non_terminal \rangle])$
	$\mid \text{qHT}(n, arch)$
	$\mid \text{qFT}(n, arch)$
	$\mid \text{qCNOT}(n, i, j, arch)$
	...
$op_list ::=$	$[\langle index_list \rangle, \langle non_terminal \rangle] :: \langle op_list \rangle$
	$[\,]$
$index_list ::=$	$[\langle nat \rangle] :: \langle index_list \rangle$
	$[\langle nat \rangle]$
$nat ::=$	n where $n \in \mathbb{N}$

Table 3.3: Modified QSPIRAL algorithm syntax, with explicit architecture parameter.

Like before, we decompose our high-level qCirc input into the matrix product of qEmbed objects for each of the non-terminals in the argument list. Note that the embed step will prune the $arch$ matrix such that the embedded non-terminal is operating over only a subset of the total qubits; the rows and columns of this smaller adjacency matrix will be renamed appropriately to match the input ordering of the qubits, as shown in Figure 3.6.

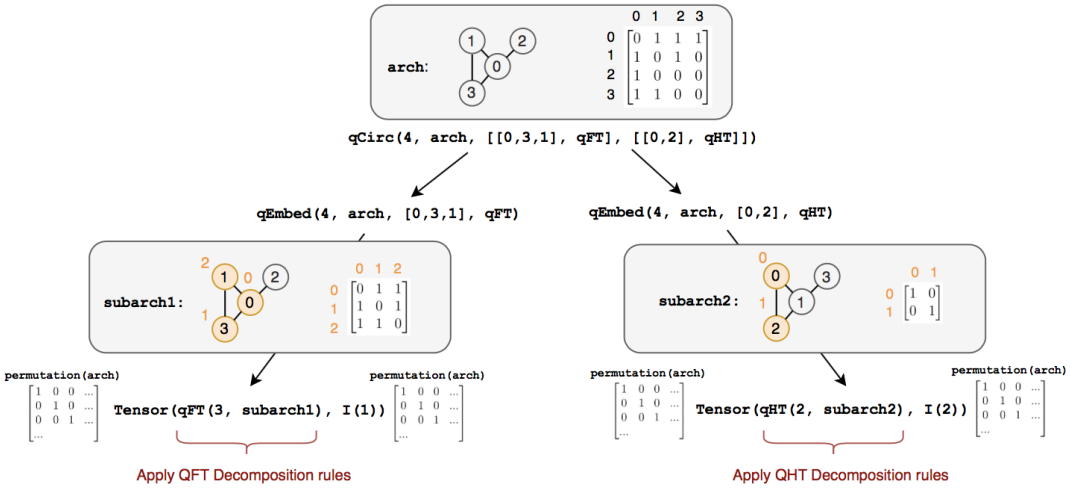


Figure 3.6: Decomposition of qCirc and adjacency matrix pruning.

By propagating relevant connectivity information throughout the process, we are be able to enforce that all rule trees are valid with respect to architecture connectivity. Specifically, we

$$\mathrm{qCNOT}(n, i, j, arch) \mapsto Tensor(\mathrm{I}(i), \mathrm{CNOT}(i, j), \mathrm{I}((n-1)-j)); \text{ if } \mathrm{hasedge}(i, j, arch)$$

Figure 3.7: Base rule for qCNOT fails to fire if connectivity for qubits i and j is not met in adjacency matrix $arch$.

enforce that each rule application does not violate connectivity; if there are no valid options at a particular breakdown step, then the current rule tree is invalid and we must backtrack to the last applied rule. An example of a rule that fails to simplify if connectivity is violated is the qCNOT rule shown in Figure 3.7. The qCNOT is a simple transform that can be implemented with a single controlled rotation (e.g. a CNOT gate).

We additionally capture the R_N^L and J_N^L matrices as SPL objects, denoted $\mathrm{Reorder}$ and $\mathrm{Junction}$ respectively. An SPL formula is therefore a fully-decomposed algorithm written completely in terms of quantum gates and these two compiler internals, as shown in Table 3.4. A complete decomposition of a formula using these primitives is shown in Figure 3.8.

$spl_formula ::=$	$\mathrm{Tensor}(\langle spl_object \rangle, \langle spl_object \rangle)$
	$\langle spl_formula \rangle * \langle spl_formula \rangle$
$spl_object ::=$	$\mathrm{Reorder}(\langle idx_list \rangle, arch, \langle dir \rangle)$
	$\mathrm{Junction}(\langle idx_list \rangle, arch, \langle dir \rangle)$
	$\mid \mathrm{H}$
	$\mid \mathrm{X}$
	$\mid \mathrm{SWAP}(i, j)$
	...
$idx_list ::=$	$[\langle nat \rangle] :: \langle index_list \rangle$
	$[\langle nat \rangle]$
$dir ::=$	1 # forward
	-1 # backward

Table 3.4: Subset of QSPIRAL's SPL syntax, excluding size annotations.

One question that may remain is why we include the R_N^{-L} permutation at all, since it will often be wasteful to return to our arbitrary starting positions if there is no reason to do so. This is done for several reasons. First, disallowing side-effects means we can search over the decomposition of each non-terminal in the top-level qCirc argument list independently, and only combine them in the rewrite phase. This simplifies both the breakdown system and the dynamic programming search procedure we will discuss later. Next, the R_N^{-L} is likely to cancel with the next forward R_N^L permutation, in which case we pay no price for this simplicity. Finally, as seen in the global rewrite phase described later, maintaining a single, global context at the top level allows us to heuristically determine the optimal global context and change the starting configuration; this will cancel these reorderings in most cases.

To summarize, we have formalized the instruction scheduling problem as a matrix factorization problem, and provided a mechanism by which we can explore various hardware permuta-

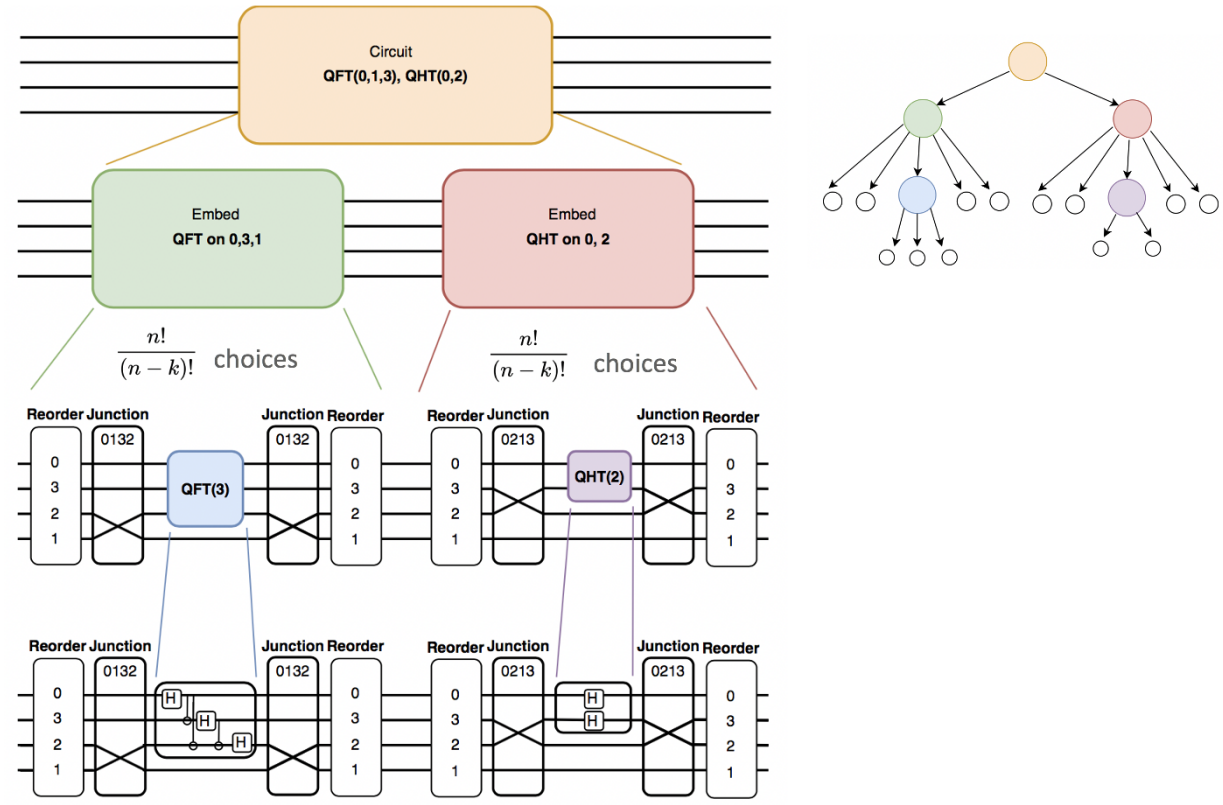


Figure 3.8: Full breakdown of a two-transform algorithm; colored objects indicate non-terminals that are subject to further decomposition.

tions leveraging the SPIRAL breakdown system. The resulting SPL expression is then passed into the rewrite phase to convert it into QASM code.

3.3.3 Formula Rewriting

Once the decomposition phase is complete, an SPL formula undergoes a series of simplification and rewriting steps in order to reduce the representation to executable QASM. These steps are intended to eliminate the remaining compiler objects and perform any peephole simplifications that are applicable. To implement this phase, we make heavy use of SPIRAL's existing rewrite rule framework. We can apply a series of semantics-preserving identities to the SPL expression in order to eliminate extra gates. Among these rules are single-qubit gate cancellation rules, tensor flattening rules and object contraction rules, as shown in Figure 3.9.

$$\mathrm{CNOT}_{ij}\mathrm{CNOT}_{ij} \mapsto \mathrm{I}_{2^{|i-j|}} \tag{3.14}$$
$$\mathrm{H}_2\mathrm{H}_2 \mapsto \mathrm{I}_2 \tag{3.15}$$
$$\mathrm{H}_2\mathrm{X}_2\mathrm{H}_2 \mapsto \mathrm{Z}_2 \tag{3.16}$$
$$\mathrm{Tensor}(l_1, ..., l_k, \mathrm{Tensor}(r_1, ..., r_n)) \mapsto \mathrm{Tensor}(l_1, ..., l_k, r_1, ..., r_n) \tag{3.17}$$
$$\mathrm{Reorder}(l_1, n, arch)\,\mathrm{Reorder}(l_2, n, arch) \mapsto \mathrm{Reorder}(l_3, n, arch) \tag{3.18}$$

Figure 3.9: Subset of QSPIRAL rewrite rules. Equation (3.18) is implemented in Figure 3.11.

The general layout of the rewriting phase is shown in Figure 3.10. The global reordering phase is described separately in the next section, as it is a completely self-contained module.

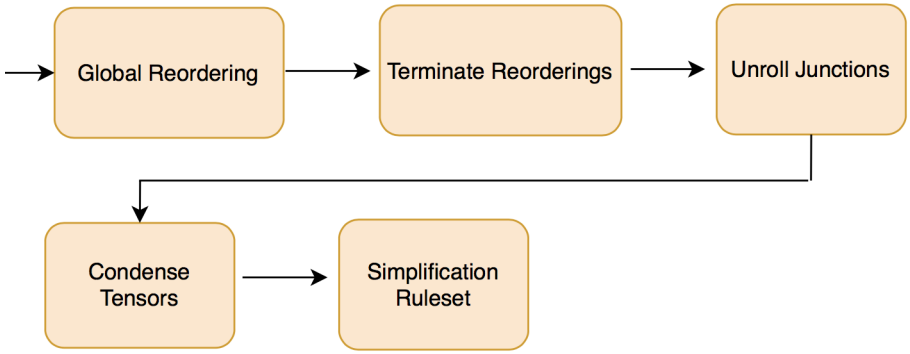


Figure 3.10: Rewrite stage control flow graph.

First, Reorder objects can be condensed. During the breakdown process shown in Figure 3.8, Reorder objects are intentionally left on the outside of module boundaries, meaning they can easily be consolidated (they are directly composed with the matrix multiply operator, and are adjacent in the SPL formula) to provide more effective transitions between the logical-to-physical mappings preferred by the adjacent algorithm stages. This process is shown in Figure 3.11. A

seemingly inefficient forward permutation may be globally optimal if it cancels efficiently with the backwards permutation of the preceding transform.

Once all Reorder objects are consolidated, they can be converted to sequences of SWAP gates. The task of generating the minimum number of swaps needed to achieve a given permutation is a well-studied group-theoretic problem [19], and since each Reorder object is tagged with the architecture adjacency matrix (and additionally, since we restore the global ordering at module boundaries, all cancellable Reorder objects are operating on the *same* adjacency matrix), we can directly generate a connectivity-compliant set of swaps to replace each Reorder.

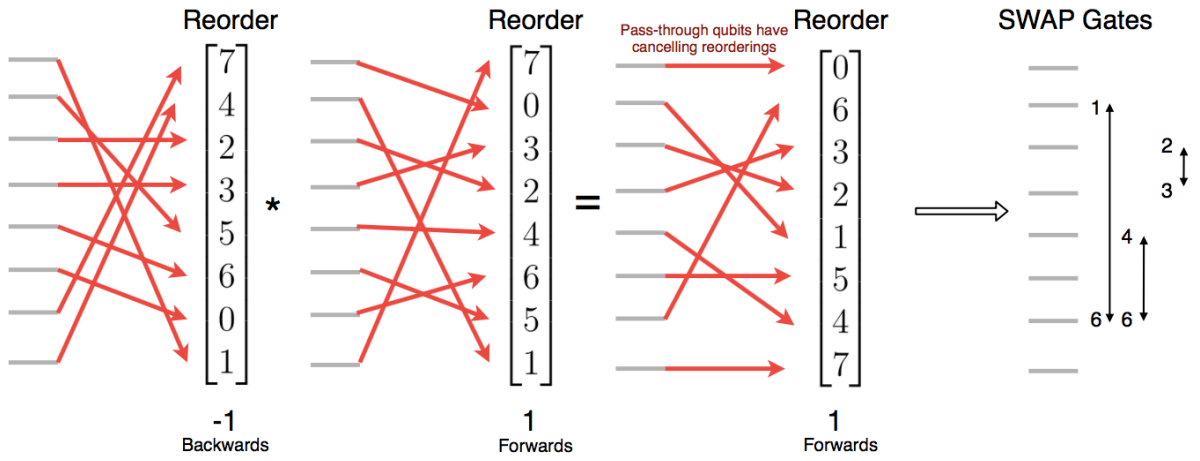


Figure 3.11: Reorder object consolidation.

After terminating Reorder objects, the only remaining non-gate compiler internal is the Junction construct. It was impossible to eliminate these objects while black-box non-terminals still existed in the formula, but at this point the rest of the circuit has been fully decomposed; individual gates can easily be rescheduled to run on different qubits in the vector. Therefore, we can eliminate these extra permutations by linearly scanning the circuit and untwisting qubit indices whenever a Junction is found. We do this recursively, in case there are multiple layers of embedded objects. This process is shown in Figure 3.12.

At this point we are left with a valid quantum circuit expressed purely in terms of symbolic matrices that equate to basis gates. However, it is still very possible that gate cancellations can be made. Our rewrite rules will handle these cancellations by simplifying the matrix product of individual gates. However, the task still remains of rewriting the circuit formula such that all potentially cancellable gates are adjacent. To do this, we sweep backwards through our SPL formula and sift symbolic matrices leftwards through tensor contraction. Afterwards we can pattern-match and apply semantics-perserving rewrite rules to simplify our formula. This implements the expression tree simplification discussed in Chapter 2.

The final tensor expression directly represents a simplified circuit instantiation of the desired transform. This mathematical expression uniquely maps to a quantum circuit as seen in Figure 3.13, and can be evaluated based on the number of SWAP operations.

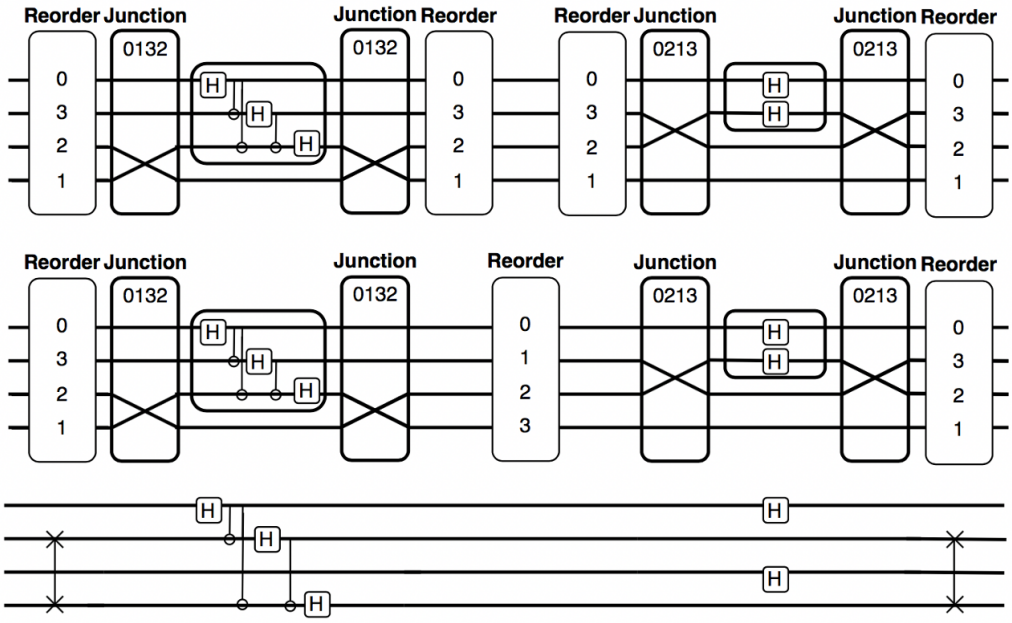


Figure 3.12: Reorder and Junction resolution in QSPIRAL backend.

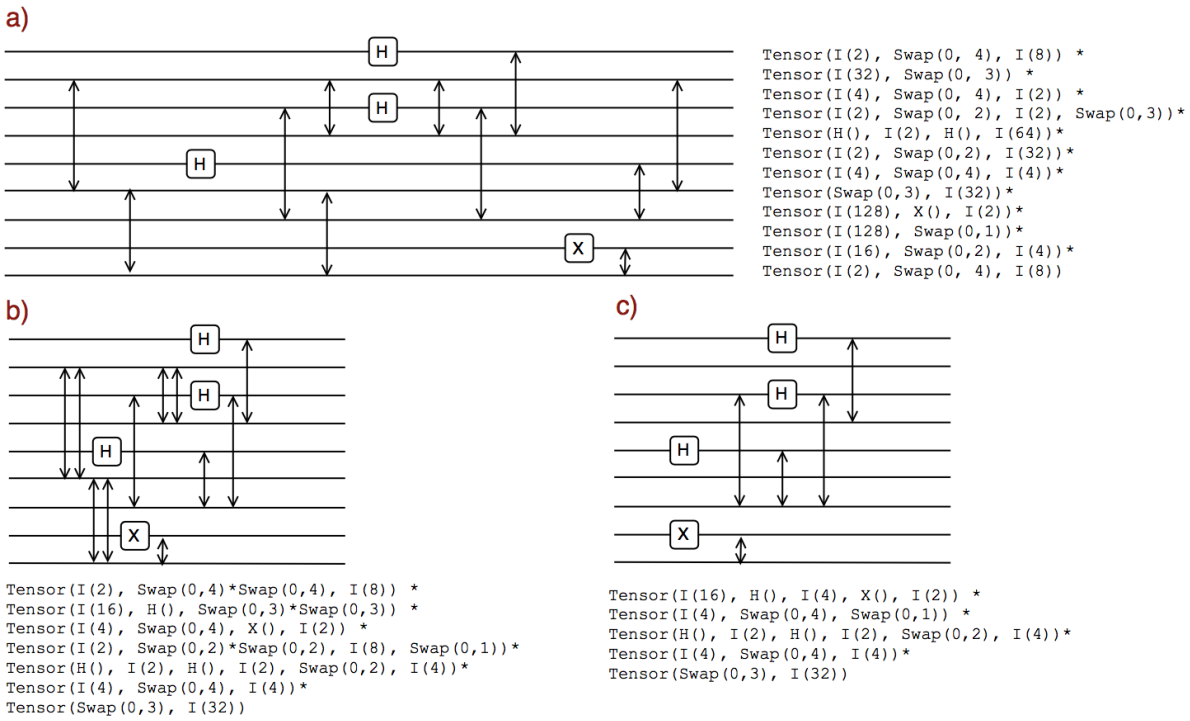


Figure 3.13: Formula rewriting in QSPIRAL: a) expression after Junction and Reorder are removed, b) expression after tensor contraction, c) expression after rewrite rule cancellation.

3.3.4 Global Reordering

Revisiting the two types of cost for data-movement explored in the previous sections (*execution* and *staging* cost), we can see that this dichotomy directly reflects the need to balance local with global optimization. Execution cost concerns only the cost of executing a transform on the chosen hardware locations, and staging cost concerns the data movement necessary to prepare the desired qubit placement; staging costs, by definition, rely on the starting configuration of the qubits. While we assumed in the previous section that there is a fixed starting assignment of logical qubits to hardware locations, we should actually be able to permute this ordering in order to give us the lowest staging cost. Ideally, if all the logical qubits were already in the desired hardware locations, staging cost would be 0. Rewriting our circuit in terms of a different starting configuration is trivial; all that is required is to change the top-level $\mathrm{Reorder}$ objects by writing them with respect to this new ordering. Finding the optimal configuration is not necessarily so trivial.

Given a single transform this task is simple; whatever logical-to-physical mapping we chose to execute the transform on during the breakdown phase can directly become the starting configuration. This means the staging cost becomes 0 and execution cost will dominate the overall cost measure, as shown in Figure 3.14. For an algorithm composed of multiple transforms this is more complex, however, since the same logical qubit could be wanted in several different hardware locations throughout the algorithm.

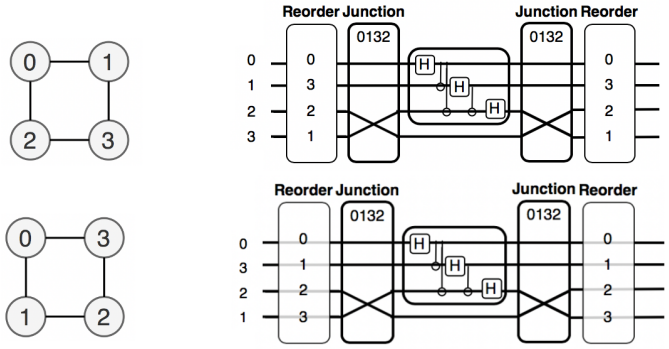


Figure 3.14: Changing the starting configuration of qubits to minimize reordering cost: old configuration [top], optimal configuration incurring zero staging cost [bottom].

Implementing this in a naïve manner, we could simply try recompiling the algorithm for all starting configurations of qubits. This corresponds to recompiling the algorithm on all $N!$ row/column permutations of the input adjacency matrix. Intuitively, this brute-force approach rapidly becomes infeasible. Therefore, a heuristic approach is needed; to implement such an approach, we leverage the as-of-yet unreduced $\mathrm{Reorder}$ objects to tell us which transform kernels want certain logical qubits in which orientations.

Scanning the SPL formula and inspecting the $\mathrm{Reorder}$ objects in the expression, we can compute for each logical qubit index the *affinity* it has for a given hardware location. This can simply take the form of a sum, and then we could greedily assign a logical qubit index to the hardware location it is most often swapped to (i.e. the location it has the greatest affinity for).

In the case where multiple logical indices have a high affinity for the same hardware location, however, something more complex is needed. Therefore we also tag each $\mathrm{Reorder}$ with two additional fields of metadata that help us make a more educated decision. First, we include the list of indices that are affected by the embedded transform. The reason for this is the following: despite $\mathrm{Reorder}$ being a size N permutation, N being the total number of qubits, the embedded transform only cares about the k qubits that are actually computed on, and k is frequently less than N. This permutation is agnostic to the hardware locations of the other $N - k$ qubits as long as they do not conflict with the locations of the k affected ones. Therefore, we should not take the positions of the other $N - k$ qubits into account when calculating affinities. As a second field of metadata, we also include a weight parameter that roughly correlates to the expected execution cost of the embedded transform. This allows more important transforms to dominate the affinity calculation. Using this, we can calculate affinities by taking a weighted sum across all $\mathrm{Reorders}$ and greedily constructing a starting configuration. This starting configuration is not guaranteed to be optimal, but it is more representative of the desired data placement, and so it is likely to greatly reduce the staging costs.

As stated above, once this new starting configuration is chosen, the only change to the SPL formula would be restating the top-level $\mathrm{Reorder}$ objects in terms of the new starting configuration. This can be done by applying the dataflow simplification seen in Figure 3.11. We then make sure to output a vector representing the permuted starting ordering such that the qubit indices can be reordered classically.

This process, shown in Figure 3.15, can introduce seemingly spurious swaps at the end of the circuit; such an unlucky case can happen if the final transform in the circuit requires a reordering that does not align exactly with the chosen global reordering. In this case, it is fairly simple to remove trailing swaps and output a vector representing the output ordering of the circuit, which now may be slightly different than the input ordering. This is also trivial to account for classically.

The global reordering phase is the final module in the QSPIRAL compiler that is needed; we now have described the complete compilation path from high-level specification to quantum circuit. In the next section, we will delve into the search procedure and how QSPIRAL selects the optimal output.

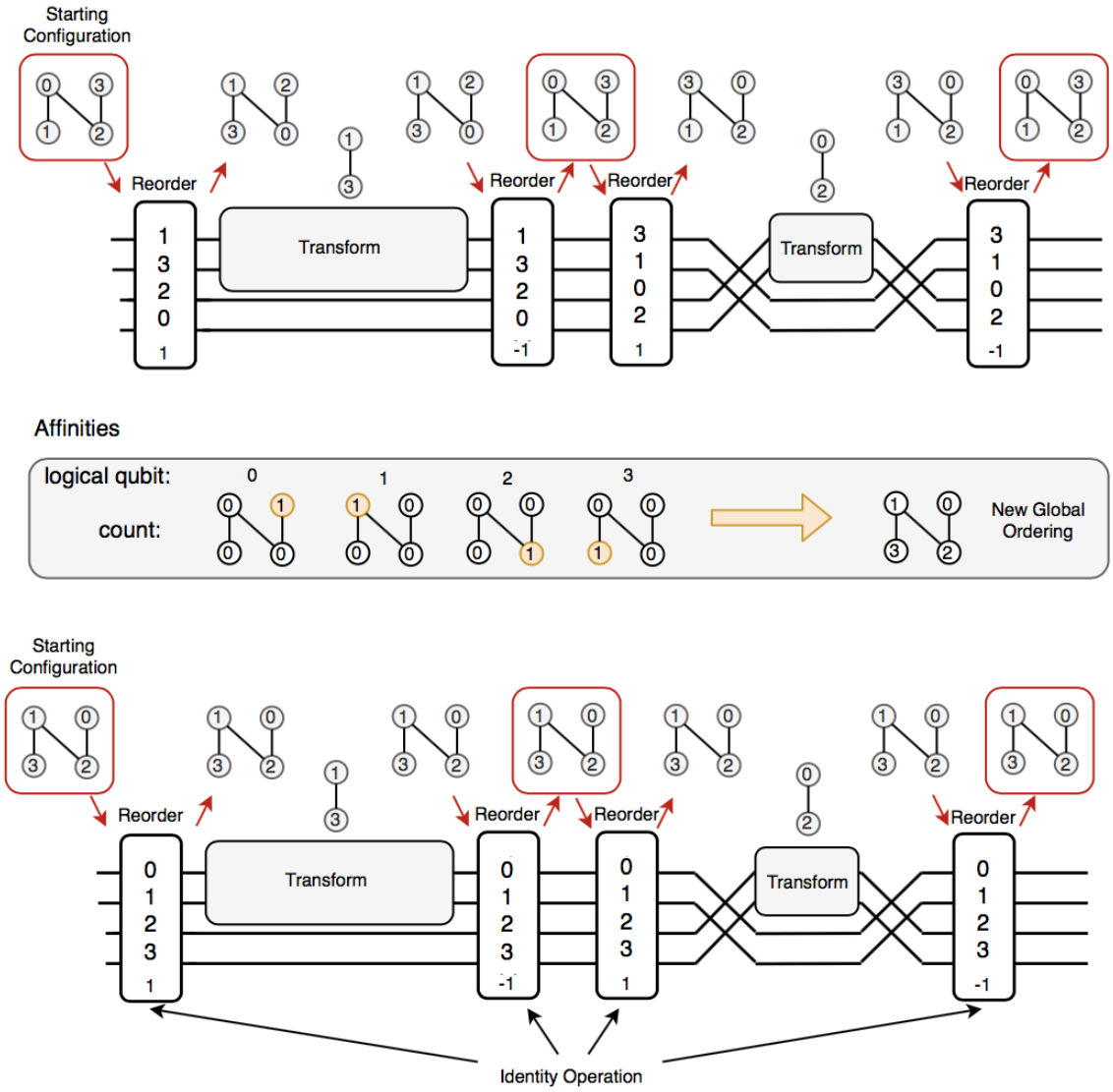


Figure 3.15: Permuting the starting configuration of an SPL formula. Transform placements are untouched but staging cost is reduced to 0.

3.3.5 Search

We now amend the optimization problem expressed in Section 2.4.1 by concretely instantiating the $\mathrm{Factorizations}$ operator with the aforementioned QSPIRAL stages. Recall the original problem formulation.

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

In order to implement the $\mathrm{Factorizations}$ operator, we searched over all possible rule trees. Given one such rule tree, we generated a factorization by applying the rule tree to the input in the breakdown phase, and then rewriting the resulting expression in the rewrite and global reordering phases. We evaluate the cost on the final circuit output, which is the result of the rewriting phase. Instantiating these operators as $\mathrm{Breakdown}$ and $\mathrm{Rewrite}$ respectively, we get the following minimization problem.

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

Where $circ$ is the high-level algorithm input, $arch$ is the hardware adjacency matrix, and rt is the rule tree that decomposes $circ$ into a gate-level expression. $\mathrm{Breakdown}$ applies the rule tree rt to $circ$ as described in Section 3.3.2, making sure to comply with the connectivity expressed in $arch$. $\mathrm{Rewrite}$ simplifies the SPL expression as described in Sections 3.3.3 and 3.3.4, converting the representation to a quantum circuit that can be expressed as QASM. Finally Cost is provided by the user, and in our case is the number of SWAP gates in the final circuit.

Viewed more simply, the rule tree rt is the varying parameter, and the following operator simply projects this onto the space of values which we are trying to minimize.

$$\text{fun } rt \mapsto \mathrm{Cost}(\mathrm{Rewrite}(\mathrm{Breakdown}(rt, circ, arch)))$$

This can be phrased as the following generic minimization problem, denoting the above function as F.

$$rt_{opt} = \arg\min_{rt} \mathrm{F}(rt)$$

This problem is a reasonably standard formulation, on which any number of traditional search techniques can be applied. Intuitively, the bounds on rt are such that rt must be a valid rule tree. After finding the optimal rule tree rt_{opt}, the final output circuit is the final value of the following expression.

$$\mathrm{Rewrite}(\mathrm{Breakdown}(rt_{opt}, circ, arch))$$

We solve this problem in QSPIRAL by performing a dynamic programming search to try numerous relevant rule trees, using heuristics to determine which paths to try and storing the results of subexpressions in a hashtable. We can then find the heuristically-best rule tree fairly efficiently; this memoization approach works quite well due to the nature of the breakdown phase, since we often end up decomposing the same formulas repeatedly, and can exploit parallelism due to the independent nature of the transform subexpressions.

3.3.6 Heuristics

Fine-tuning the approach described above, there are several optimizations that can be used to meet our efficiency goal. These are needed because it would be computationally expensive to try every possible hardware placement for each non-terminal in our algorithm. Specifically, when asked to decompose a qEmbed object, we should not search over *every* N-qubit permutation as this rapidly gives us too many rule trees to search over. We can eliminate likely-to-be inefficient mappings in several ways.

First, an unstructured search over all permutations is wasteful given our knowledge of the high-level transform. Since we embed transforms from the top-down, we have access to the highest-level description of the embedded algorithm, meaning we should be able to make intelligent decisions if we take the symmetry of the algorithm into account. A fixed embedding technique for the QFT is shown in later sections, but this can be done for any transform. A particularly efficient example concerns embedding a QHT, or the quantum Hadamard transform; since Hadamard transforms decompose into independent H gates, we can simply omit the reorder step entirely.

Additionally, since we take into account only the positions of the affected qubits in the global reordering stage, we need search only over arrangements of the k qubits the transform is applied to, and assign an arbitrary or pass-through ordering for the rest. This, as mentioned earlier, reduces the possible $N!$ reorderings to only $\frac{N!}{(N-k)!}$ reorderings.

We can also apply an optimization we call *group scheduling*. The key recognition behind this is that breaking a qCirc object into separate qEmbed objects for each transform could be wasteful if the preferred hardware arrangements for the transforms end up not conflicting. We can thus greedily attempt to schedule several transforms with the same hardware arrangement, and if breakdown fails due to a failure to meet connectivity requirements, we can then split the transforms up and try again until it succeeds. While this approach creates more work for larger transforms that are likely to always prefer their own reorderings, it can be efficient for long series of small transforms, as it reduces the number of objects we have to decompose.

Finally, every breakdown rule in SPIRAL is predicated on an arbitrary $is_applicable$ premise. By changing how $is_applicable$ is determined, one could implement fixed decision procedures to speed up the search or limit the possible span of rule trees. This approach is directly inspired by the field of constructive logic [42], where techniques like sequent calculus [29], inversion [45] and focusing [2] are used to lend speed and determinism to various methods of proof search.

3.3.7 System Outputs

The output of QSPIRAL is twofold. First, a fully simplified factorization of the input algorithm is provided which consists of the tensor and matrix product of symbolic matrices; this is the expression tree formulation presented in Chapter 2. Additionally, the backend QASM unparser can transform this mathematical representation into a program stream that can be loaded into most quantum circuit toolkits, as shown in Figure 3.16. It is worth noting that the QASM program stream is the same or similar to the format that is commonly used as *input* to existing toolkits. It should be apparent that most, if not all of our transformations would be impossible starting from such a reduced representation.

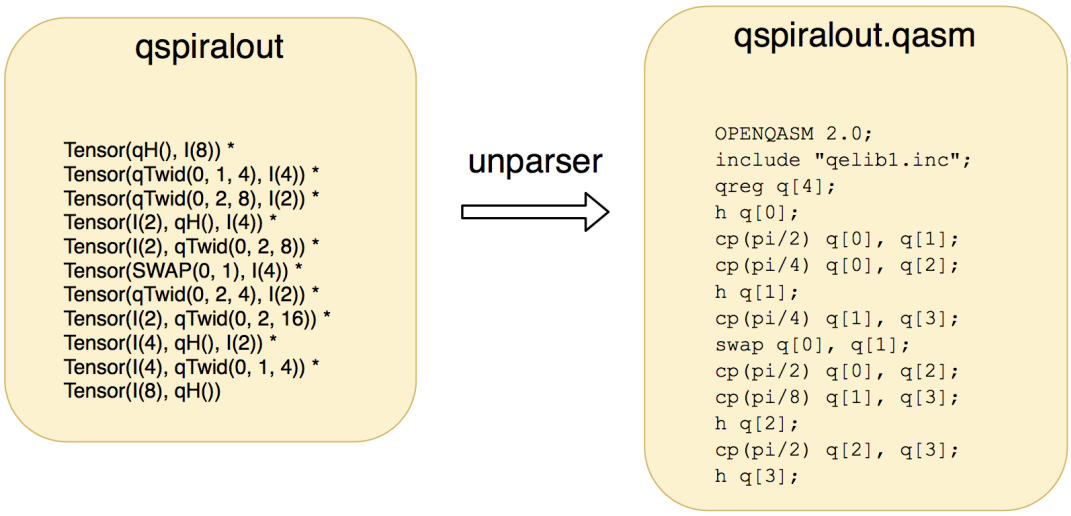


Figure 3.16: Unparsing QSPIRAL's symbolic matrix factorization; 4-qubit QFT factorization [left] 4-qubit QFT QASM program [right].

Along with the QASM code is a vector indicating the global reordering chosen for the circuit, allowing the user to know the input and output permutation of qubits. These can be reordered classically by simply measuring different qubits.

To solidify our view of quantum circuits as simply being matrix factorizations of the desired transform, we can expand out the $2^N \times 2^N$ transform matrix in SPIRAL for suitably small values of N. Quantum gates in QSPIRAL are expressed as symbolic matrices, and so by substituting in their definition, GAP can reconstruct the target transform matrix. As indicated by Figure 3.17, this is *same* transform matrix that is specified by the high-level algorithm syntax, accounting for a possible permutation introduced in the global reordering phase.

```
Expand_Matrix(
  Tensor(qH(), I(8)) *
  Tensor(qTwid(0, 1, 4), I(4)) *
  Tensor(qTwid(0, 2, 8), I(2)) *
  Tensor(I(2), qH(), I(4)) *
  Tensor(I(2), qTwid(0, 2, 8)) *
  Tensor(SWAP(0, 1), I(4)) *
  Tensor(qTwid(0, 2, 4), I(2)) *
  Tensor(I(2), qTwid(0, 2, 16)) *
  Tensor(I(4), qH(), I(2)) *
  Tensor(I(4), qTwid(0, 1, 4)) *
  Tensor(I(8), qH()) *
  Tensor(SWAP(0, 2), I(2)) *
  Tensor(I(2), SWAP(0, 2)) *
  Tensor(SWAP(0, 1), I(4))
);
```

=

```
Expand_Matrix(DFT(16));
```

Figure 3.17: Equivalent matrix expansion of 4-qubit QFT circuit (excluding vector normalization terms) [left] and a 16-point DFT [right].

This approach works for generic circuits, but the true benefits to be had are when compiling algorithms that have high-level symmetries. Thus, we take advantage of large bodies of Fourier transform literature in the next section to present a series of proposed scheduling heuristics for the quantum Fourier transform algorithm, and implement these in QSPIRAL.

3.4 Conclusions

In this chapter we outlined the complete QSPIRAL quantum code generator, a framework that effectively expresses and searches over a large space of circuits that compute the desired algorithm in order to find the best one with respect to some cost measure. We assume, due to reasons stated in the introductory chapter, that this cost measure contains the number of SWAP gates in some context, as much of the proposed framework implements a heuristic for efficiently placing operations in a manner that minimizes data movement.

QSPIRAL is a general compiler. This means that in the case that the desired circuit is not one that can easily be expressed by a high-level algorithmic symbol, we still accept gate-level input in the form of non-terminals that trivially decompose into the intended gates. This is a practical requirement, but providing gate-level input essentially bypasses most of the compiler stages since there are no global rewrites or special decomposition rules to try. Luckily, most useful algorithms will be able to leverage, at least partially, the high-level symbols we supply.

In the next section we take a particular look at one of these symbolic algorithms, the quantum Fourier transform, and showcase the applicability of classical FFT literature towards informing QFT circuit construction. This algorithm is an immensely important kernel and was an intuitive choice for study given SPIRAL's past successes with optimizing the FFT algorithm for classical architectures. We will present a series of breakdown heuristics that leverage the structure of the architecture to efficiently implement this algorithm in the QSPIRAL framework.

Chapter 4

Case Study: Quantum Fourier Transform

In Chapter 3 we discussed the SPIRAL implementation of a solver for the generalized optimization problem formalized in Chapter 2. Understandably, the real test for this system lies in its ability to generate effective circuits for important quantum algorithms. While our compilation approach must be general in order to allow arbitrary input programs, the current space of useful quantum algorithms is rather limited, meaning that we can study the structure of important quantum algorithms in order to implement effective compilation heuristics for them. This approach is not new. In fact, it is exactly what is done in classical SPIRAL to optimize known algorithms for various hardware architectures. Applying these techniques is novel in the quantum domain, however, because other frameworks do not have access to high-level algorithmic information at decomposition time.

One such quantum algorithm is Shor's algorithm [59] for integer factorization. In addition to smaller circuit elements, this algorithm relies heavily on the *quantum Fourier transform*, an implementation of the discrete Fourier transform in the quantum domain. This algorithm is of particular interest in this work because classical SPIRAL was originally built to optimize the FFT algorithm, and so the necessary intrinsics are already built-in. QSPIRAL is unique in being able to recognize that it is compiling a QFT kernel and hence can apply the high-level algorithmic manipulations described in this section. We intend for these heuristics to reduce the number of rule trees by both informing how we decompose the QFT and limiting our search over the hardware locations we want to compute it on.

In Section 4.1, we first motivate why the QFT is an important quantum algorithm. Section 4.2 provides a brief overview of the dataflow pattern of the FFT algorithm, and we directly relate this to the implementation of the QFT circuit in Section 4.3. In Section 4.4, we draw both from SPIRAL and parallel FFT research to develop heuristic algorithms for the quantum domain, an effort that has not yet widely been explored. By taking inspiration from existing FFT literature we are able to construct effective heuristics to make QFT optimization tractable in QSPIRAL and produce efficient results when compared to existing toolkits. Conclusions are drawn in Section 4.5; while we make very loose claims pertaining to optimality, we show that structured approaches taking advantage of algorithm symmetry can be a powerful tool.

4.1 Why is the QFT important?

We begin by motivating the quantum Fourier transform, or QFT, as a useful quantum algorithm. This algorithm simply computes a DFT on the joint state vector; an N-qubit QFT equates to a 2^N-point DFT.

The quantum Fourier transform was first described by Umesh Vazirani and Ethan Bernstein in 1993 [8], but found one of its first practical applications in 1994 with the advent of Shor's algorithm for factoring large numbers in polynomial time [59]. Shor's algorithm is inspired heavily by one proposed by Simon [61], but specifically leverages the QFT in order to power the period-finding portion of the circuit. Shor's algorithm is immensely important because it suggests that quantum computers could break the assumed-difficult problem at the core of RSA public-key encryption [56].

The QFT has since found uses in other algorithms, specifically as the driving kernel behind the quantum phase estimation circuit [43]. From these important applications, it is clear that the Fourier transform is an immensely important operator in the quantum world as well as in the classical one. Being able to run this algorithm efficiently on quantum hardware is therefore of importance.

4.2 The FFT Butterfly

The fast Fourier transform is an $O(n \log n)$ algorithm to compute the typically $O(n^2)$ discrete Fourier transform. The derivation of this algorithm leverages group-theoretic symmetries. As a brief overview, we can derive the algorithm as follows, starting with an N-point DFT.

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}; W_N = e^{-j2\pi/N} \tag{4.1}$$

We can split this summation into even and odd components.

$$X[k] = \sum_{n_{\text{even}}} x[n] W_N^{nk} + \sum_{n_{\text{odd}}} x[n] W_N^{nk} \tag{4.2}$$

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{(2r+1)k} \tag{4.3}$$

With some simplification we can rewrite this as two separate DFT computations of size $N/2$, with one multiplied by a twiddle factor W_N^k.

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_{N/2}^{2rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1] W_{N/2}^{2rk} \tag{4.4}$$

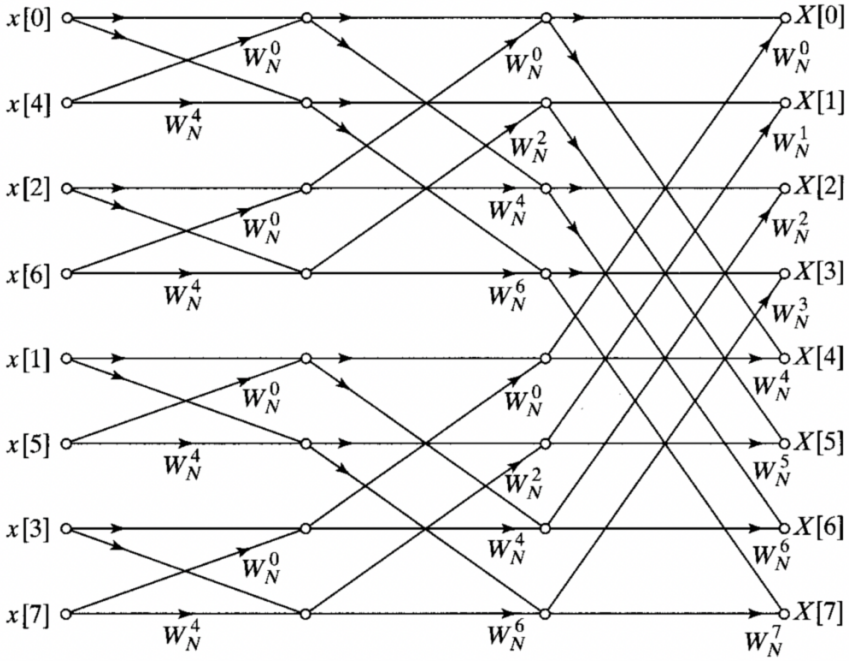


Figure 4.1: 8-point radix-2 DIT FFT algorithm. Sourced from [52].

Repeating this decomposition, we get the radix-2 decimation-in-time (DIT) FFT algorithm. The signal flow diagram of this algorithm is commonly expressed in terms of butterfly structures, as shown in Figure 4.1.

We can also derive a decimation-in-frequency version of this algorithm, which will ultimately result in the transpose of the above. We begin with the standard definition of the DFT.

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} \tag{4.5}$$

Dividing the time series in half, and merging summations, yields the following.

$$X[k] = \sum_{n=0}^{(N/2)-1} x[n] W_N^{nk} + \sum_{n=N/2}^{N-1} x[n] W_N^{nk} \tag{4.6}$$

$$X[k] = \sum_{n=0}^{(N/2)-1} x[n] W_N^{nk} + W_N^{(N/2)k} \sum_{n=0}^{(N/2)-1} x[n + (N/2)] W_N^{nk} \tag{4.7}$$

$$X[k] = \sum_{n=0}^{(N/2)-1} (x[n] + (-1)^k x[n + (N/2)]) W_N^{nk} \tag{4.8}$$

For $k = 2r$, or k even, we get the following expression.

$$X[k] = \sum_{n=0}^{(N/2)-1} (x[n] + (-1)^{2r} x[n + (N/2)]) W_N^{n2r} \tag{4.9}$$

$$X[k] = \sum_{n=0}^{(N/2)-1} (x[n] + x[n + (N/2)]) W_{N/2}^{nr} \tag{4.10}$$

This is the $N/2$ point DFT of $x[n] + x[n + (N/2)]$. Doing the same for the odd coefficients ($k = 2r + 1$) we get the following.

$$X[k] = \sum_{n=0}^{(N/2)-1} (x[n] + (-1)^{2r+1} x[n + (N/2)]) W_N^{n(2r+1)} \tag{4.11}$$

$$X[k] = \sum_{n=0}^{(N/2)-1} (x[n] - x[n + (N/2)]) W_N^n W_{N/2}^{nr} \tag{4.12}$$

This is the $N/2$ point DFT of $(x[n] - x[n + (N/2)])W_N^n$. Continuing to decompose in this fashion we get the transpose of the original signal flow graph. Hence, the DIF and DIT algorithms are simply transposes of one another. This relationship, as well as many others, holds in the quantum domain as well as in the classical one.

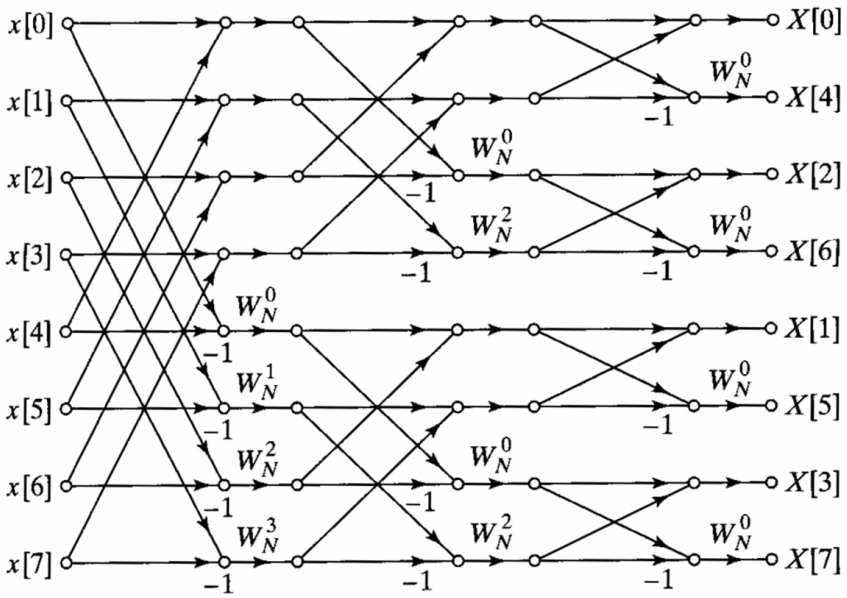


Figure 4.2: 8-point radix-2 DIF FFT algorithm. Sourced from [52].

The butterfly structures shown in Figure 4.2 are data transfers; in an optimized implementation of this algorithm, minimizing the communication cost between these streams is one of the key considerations. In a dataflow processor this would require scheduling relevant instructions

close together, and in a multicore system this would require the efficient mapping and reduction of various threads. The complex exponentials W_N^k, or twiddle factors, pose no problem as they can simply be applied locally via scalar multiplication.

The linear algebraic form of the decomposition we described above is the Cooley-Tukey [13] rule, as follows.

$$\mathrm{DFT}_n = (\mathrm{DFT}_k \otimes \mathrm{I}_m)\mathrm{T}_m^n(\mathrm{I}_k \otimes \mathrm{DFT}_m)\mathrm{L}_k^n; n = km \tag{4.13}$$

This rule expresses the butterfly topology in matrix form, and is showcased in Figure 4.3. By varying the values of k and m we can construct a variety of different decomposition schemes that correspond to algorithms beyond the standard radix-2.

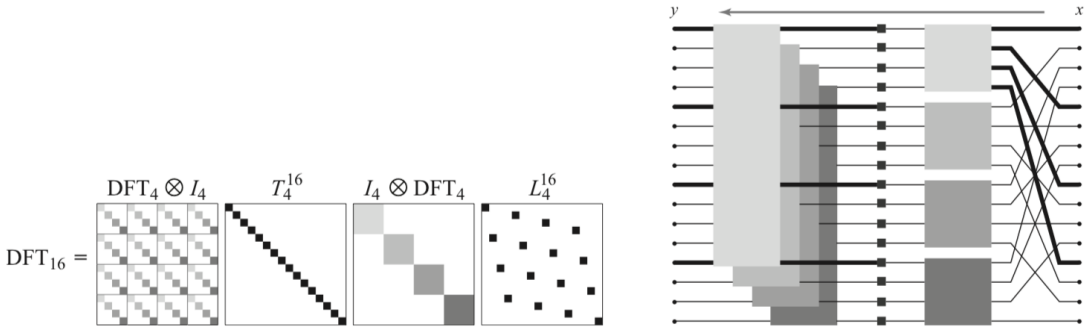


Figure 4.3: Cooley-Tukey decomposition in sparse matrix form [left], state vector dataflow representation [right]. Sourced from [25].

We can see that this existing formulation almost exactly suggests a way to construct a quantum circuit for this task, indicating that a QFT circuit effectively applies the aforementioned butterfly patterns to the state vector. We will derive this in the next section.

4.3 Classical-to-Quantum Translation

The building blocks of a QFT circuit are the same as its classical counterpart; the circuit is constructed of butterfly structures and twiddle factors. Interestingly, whereas the performance bottleneck in the classical realm is our ability to effectively implement the communication requirements of the butterfly structures, these can be implemented directly by a Hadamard gate and require no cross-qubit communication.

$$H = \frac{1}{\sqrt{2}}\mathrm{DFT}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{4.14}$$

Besides potentially the scaling factor (which is required to preserve vector length), the Hadamard gate is exactly the 2×2 DFT matrix, which is commonly known as the butterfly matrix. The

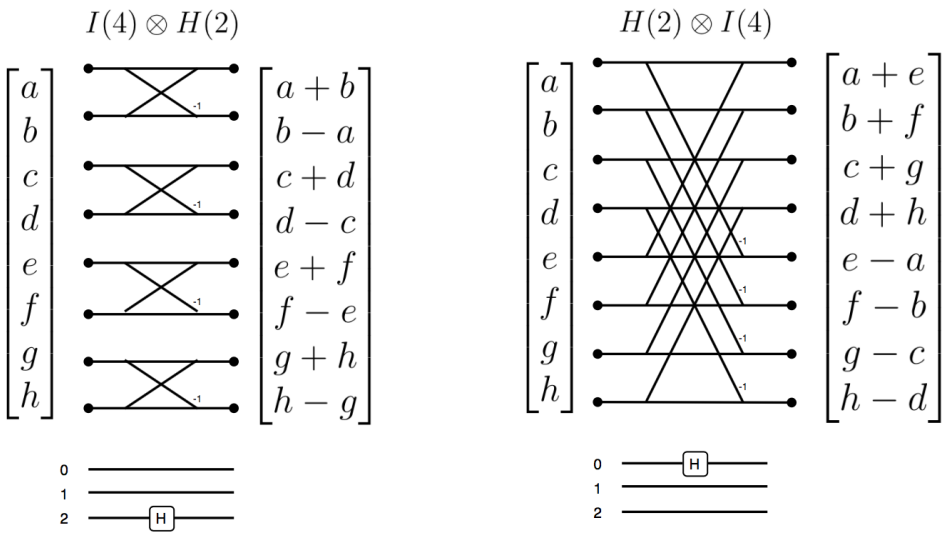


Figure 4.4: Implementing FFT butterflies with Hadamard gates. Different H placements yield different butterfly stages.

effect this gate has on the input vector is identical to the butterfly pattern needed to implement the various stages of the FFT algorithm, as shown in Figure 4.4.

Applying twiddle factors is a controlled operation, specifically a controlled rotation by a factor of π.

$$\text{Twiddle}(k) = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{j\pi}{k}} \end{bmatrix} \tag{4.15}$$

$$\text{Controlled_Twiddle}(k) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{j\pi}{k}} \end{bmatrix} \tag{4.16}$$

This means that the primary challenge in optimizing the QFT lies in satisfying the connectivity costs of the twiddle factor applications. In this sense, this is reversed from the classical domain, wherein the main bottleneck to performance was the communication cost of the butterfly network. We show in Figures 4.5 and 4.6 that the diagonal twiddle matrix T^n_m in the Cooley-Tukey algorithm can directly be implemented with controlled rotation gates; these stages are the primary target for optimization.

Just as the DIF and DIT butterfly networks are transposes of eachother in the classical domain, they can be transposed in the quantum domain as well (Figure 4.7). This is because the operation the quantum gates apply to the state vector is exactly the classical FFT butterfly, and thus the same symmetries apply.

Despite these similarities, while there are large bodies of FFT research pertaining to the development and implementation of various algorithms on parallel and vector machines, very little of this literature has been ported over the quantum realm. A significant barrier to this has

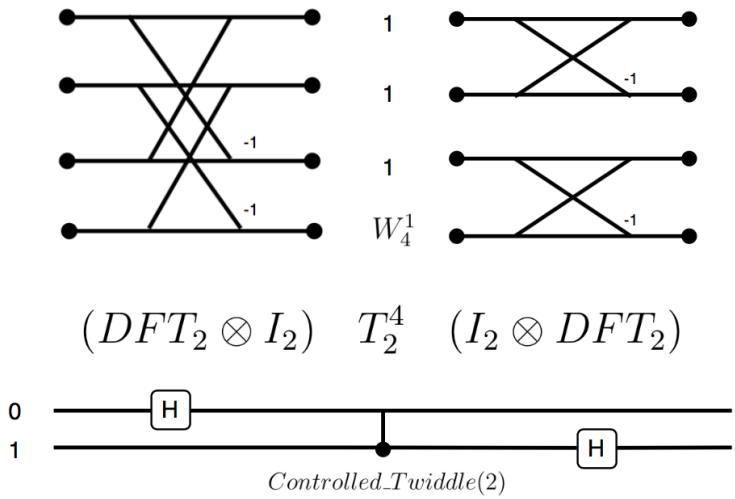


Figure 4.5: Implementing a 4-point FFT with Hadamard and controlled rotation gates.

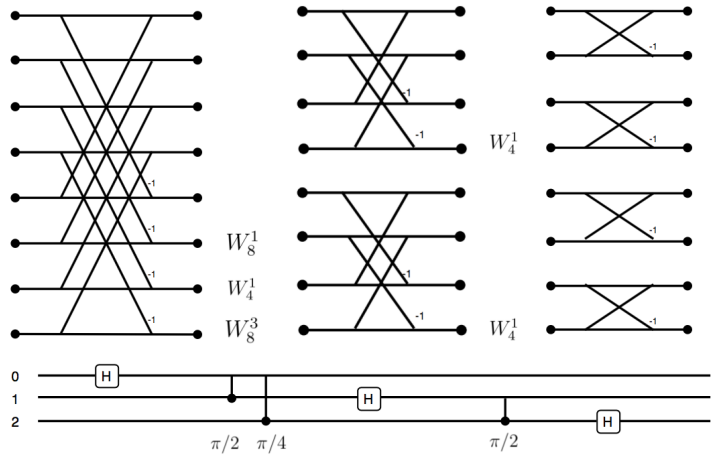


Figure 4.6: Implementing an 8-point FFT with Hadamard and controlled rotation gates.

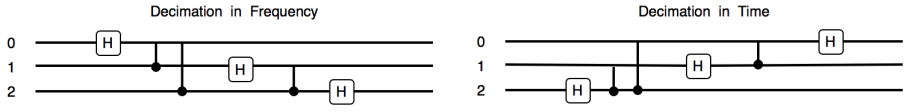


Figure 4.7: Transposing the DIF QFT [left] into the DIT QFT [right].

been the accessibility of quantum notation; parallels between the classical and quantum FFT do not become apparent until the circuit optimization problem has been restated in terms of pure linear algebra, as is done in Chapter 2. However, now that we have done so, we can leverage SPIRAL's library of FFT algorithms in order to provide valid circuit decomposition rules [25]. A trivial example of this is expanding our quantum circuit search beyond the radix-2 algorithm; the radix-4 algorithm suggests an alternative approach to scheduling individual quantum gates.

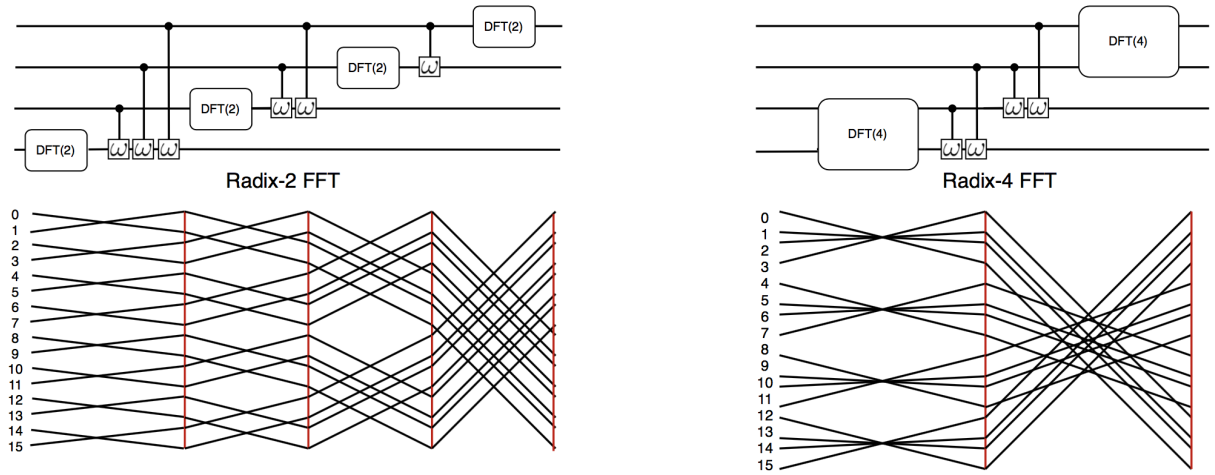


Figure 4.8: Comparison between radix-2 QFT [left] and radix-4 QFT [right].

In Figure 4.8, we show the exploration of various gate scheduling options by employing the Cooley-Tukey decomposition discussed in previous chapters (i.e. by varying k and m). Since QSPIRAL manipulates the QFT in terms of a high-level representation, and treats the object as more than an arbitrary string of gates, this and other decomposition identities can be applied.

4.4 QFT Optimization in SPIRAL

Now that we have provided a recipe for translating classical FFT structures into quantum-applicable algorithms, we leverage the well-studied parallel FFT to inform hardware mapping heuristics. Specifically, we will leverage the symmetries of a hypercube to inform how we partition and schedule qubits in our QFT decomposition. Our ability to apply these and any future QFT-specific heuristics we owe to the high-level approach taken in this thesis; if the input QFT circuit were represented as a flattened string of gates we would not be able to recognize and apply any of these algorithm-specific techniques. We will show that taking a structured and geometric approach to this problem has the potential to garner massive benefits over the traditionally-unstructured search procedure.

Hypercubes are popular in the classical FFT domain for several reasons. First, they are sparse architectures that require a maximum of $\log n$ edge traversals to reach any node in the graph. Additionally, the symmetry of the radix-4 FFT lends itself well to a division of the algorithm into planar sections, which a cube inherently does. These same reasons transfer to the quantum

domain. However, there is another reason why hypercubes are worth studying in a quantum-specific context. One of the most popular quantum topologies is a nearest-neighbor connected mesh, as exemplified by Google's Sycamore architecture [3]; this is due both to manufacturing concerns regarding higher-dimensional layouts and a consensus that mesh architectures could be useful for implementing the error-corrected qubits of the future [38]. While a mesh is certainly no hypercube, embedding a hypercube into a mesh is a well-studied problem, and so these heuristics end up relying primarily on known classical techniques that need only be adapted to a quantum environment.

In this section we present a general guide to our approach, and following from that, we discuss a simple algorithm that operates on qubits arranged directly in a hypercube. However, since quantum architectures are rarely built in the shape of a perfect hypercube, we then expand this algorithm to generalized topologies by performing an embedding of the hypercube onto the given qubit architecture (most commonly a mesh) and factoring our original algorithm onto this architecture. As we progress through the various QFT butterfly stages, this embedding will change to minimize edge costs, ensuring we only ever compute on qubits that are optimally located in a centralized cluster. Finally, we present a third heuristic, following from our general approach, that leverages a repeating diagonal handshaking pattern that can be scaled to larger hypercubes. We show in the next section that in leveraging these placement heuristics, QSPIRAL can often generate cheaper QFT circuits than can existing frameworks. We make no major optimality claims with respect to these heuristics; we simply to intend motivate the further pursuit of algorithm-specific heuristics in this space, and to show that our high-level algorithmic approach allows these heuristics to be applied.

4.4.1 General Approach

To inform the steps taken in the rest of this section, we begin by outlining our general approach towards developing the heuristics presented in this chapter.

We begin by selecting an algorithm, and from our knowledge of the dataflow patterns and symmetries of this algorithm, we select an optimized graph structure on which we would want to compute this algorithm. This graph can be any idealized qubit architecture that lends itself well towards computing the algorithm in question, and (as is at least the case for the Fourier transform) should often be directly informed by existing literature in the areas of parallel and scientific computing. The optimal qubit connectivity graph for any algorithm is trivially a fully-connected one, which is not immediately helpful since qubit architectures are rarely fully-connected. Therefore, we constrain our choice to graphs that are sparse enough to resemble real quantum architectures. Exact sparsity requirements are not formalized in this thesis, but since several aspects of computing in the classical domain (network topologies, bus interconnects, etc...) also value sparsity, it is our expectation that much of this research has already been done in classical literature to determine geometrically-inspired sparse graphs on which various algorithms can be efficiently computed. Once we select such an idealized qubit geometry, we can then develop a protocol for executing the desired algorithm directly on this geometry, a protocol that is also likely to be heavily inspired by existing literature in the classical domain.

However, whatever graph is ultimately chosen, these heuristics are not broadly useful if they can only be applied for that *specific* idealized architecture; this would imply that a separate

algorithm must be hard-coded for each new qubit layout. Therefore, we must have a protocol for factoring our ideal algorithm onto an arbitrary *real* qubit topology that does not necessarily match the architecture we assumed. We generalize our original algorithm, then, by lowering our ideal architecture to the real one; specifically, we perform an embedding of our idealized graph onto the real qubit topology. Once we have performed this embedding, we can then generally apply our original algorithm by treating each original step as an *abstract* operation, and translating these manipulations of the ideal graph into series of manipulations that achieve the same goal on the non-idealized real architecture; the less similar these two graphs, the larger incurred cost we would typically assume. For example, certain swaps on the ideal graph may require two or more swaps on the real architecture; similarly, neighboring qubits in the ideal graph may require data movement to become so in the real architecture. Translating these abstract operations into physical gates must be done to minimize the additional translation cost incurred by the embedding.

In our specific case, we develop heuristics for the quantum Fourier transform, and select the hypercube as an idealized structure on which to compute this algorithm. We leverage the Cooley-Tukey algorithm to develop a protocol for executing the QFT on a perfect hypercube (Section 4.4.2), perform an embedding of our idealized hypercube onto an arbitrary topology (Section 4.4.3) and provide a methodology for factoring our original hypercubic algorithm onto this non-ideal embedding (Section 4.4.4). Since a nearest-neighbor mesh is both a common quantum architecture and convenient layout on which to perform our hypercube embedding, we often assume our real architecture to be so in the following formalization. However, the approach presented can be generalized to any real architecture, and our embedding algorithm will lower the hypercube graph to any arbitrary geometry with the proper number of qubits. We repeat this process for an additional protocol in Section 4.4.5; this serves to illuminate the immensity of the design space and the wealth of additional research that this area could productively absorb.

4.4.2 Static Hypercube Algorithm

Let us start by assuming a hypercubic arrangment of qubits. We know by the structure of the FFT algorithm that qubit 0 will need to control the rotation of (i.e. *twiddle to*) every other qubit in the system with a higher index than itself. This pattern extends to all qubits, as seen in Figure 4.9, which displays a 4-qubit QFT circuit on a fully-connected architecture.

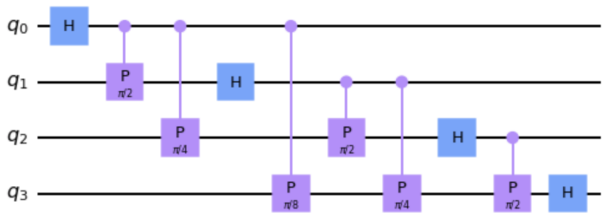


Figure 4.9: 4-qubit QFT circuit.

The structure of the algorithm introduces a natural dependency relation between a qubit and all preceding qubits; qubit k must be rotated by qubits $[0, \ldots, k-1]$. Reordering these twiddle

rotations is possible (due to the commutativity of diagonal matrix multiplication) as long as a qubit has been fully rotated *before* the Hadamard gate (i.e. the butterfly matrix) is applied; a qubit must also only rotate other qubits in the system *after* the butterfly has been applied to it. Permuting the gate schedule past these barriers changes the mathematical meaning of the circuit. We do not directly face these concerns, as we are not modifying a low-level circuit definition, but we do have to keep them in mind as we develop our decomposition technique.

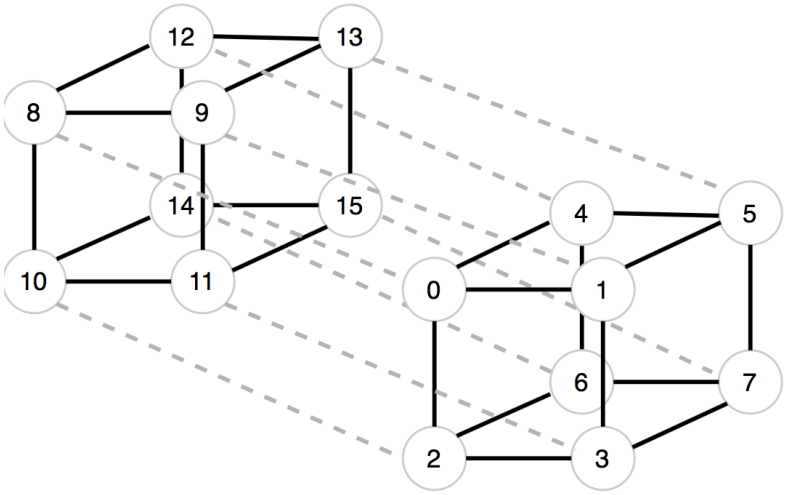


Figure 4.10: A 16-qubit hypercube.

Following the schedule suggested by the radix-2 DIF algorithm, on the architecture shown in Figure 4.10, we would first apply a Hadamard gate to qubit 0 and then proceed to rotate all other qubits in ascending order. However, the data movement required to handshake qubit 0 with all other qubits in the hypercube is very large. We must move qubit 0 adjacent to each of these qubits before we can apply the rotation, and so we are paying non-negligible data movement costs when we handshake with qubits across a diagonal. Specifically, handshaking qubit 0 with qubits 3, 5, 6, 9, 10 and 12 require traversing the diagonal of a 2-cube, 7, 11, 13 and 14 require traversing the diagonal of a 3-cube, and 15 requires traversing the full-dimensional diagonal of the 4-cube. Doing this sequentially for every qubit is very inefficient, seeing as qubit 0 will most likely be much closer to these qubits in the future after further permutations have taken place. Our goal is to design a permutation strategy that handshakes all necessary qubits in an acceptable order, but that minimizes the cost we pay for communication across these high-dimensional diagonals.

Deriving such a strategy, let us start with the $[0, 1, 2, 3]$ plane. Since 0 requires no rotation, we can directly apply the butterfly matrix to qubit 0 (a process we will call *progressing* for brevity), and twiddle to adjacent qubits 1, 2, 4 and 8. Since 1 need only be rotated by 0, we can then progress 1 and twiddle from 1 to qubits 5, 3 and 9. We have performed all possible twiddles from the current configuration, and so we swap qubits 0 and 1. Now we can twiddle from qubit 1 to qubits 8, 4 and 2, and from qubit 0 to 5, 9 and 3. Qubit 2 can now be progressed, and we twiddle from qubit 2 to qubits 3, 6 and 10. We finish progressing the qubits in the plane by progressing qubit 3 and twiddling from qubit 3 to qubits 7 and 11. This sequence continues in the following fashion.

1) $2 \Leftarrow swap \Rightarrow 3$
2) $2 = twiddle \Rightarrow 7, 11$
3) $3 = twiddle \Rightarrow 6, 10$
4) $0 \Leftarrow swap \Rightarrow 2, 1 \Leftarrow swap \Rightarrow 3$
5) $0 = twiddle \Rightarrow 7, 11$
6) $1 = twiddle \Rightarrow 6, 10$
7) $2 = twiddle \Rightarrow 5, 9$
8) $3 = twiddle \Rightarrow 4, 8$
9) $0 \Leftarrow swap \Rightarrow 1$
10) $2 \Leftarrow swap \Rightarrow 3$
11) $0 = twiddle \Rightarrow 6, 10$
12) $1 = twiddle \Rightarrow 7, 11$
13) $2 = twiddle \Rightarrow 4, 8$
14) $3 = twiddle \Rightarrow 5, 9$

This somewhat lengthy description of the swap sequence shown in Figure 4.11 has achieved the complete handshaking of plane $[0, 1, 2, 3]$ with planes $[4, 5, 6, 7]$ and $[8, 9, 10, 11]$.

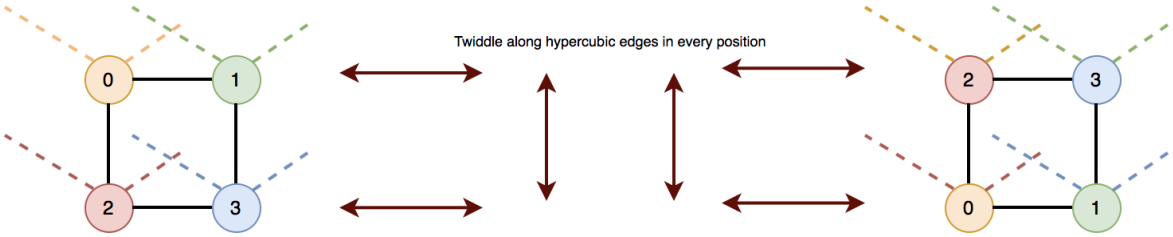


Figure 4.11: The 4-qubit planar swap sequence.

To complete the algorithm, we can repeat this same process to handshake plane $[4, 5, 6, 7]$ with plane $[12, 13, 14, 15]$, ignoring the backwards edges to plane $[0, 1, 2, 3]$. If we then swap planes $[0, 1, 2, 3]$ and $[4, 5, 6, 7]$ and repeat our planar swap sequence, we complete all necessary outbound rotations from cube $[0, 1, 2, 3, 4, 5, 6, 7]$ to cube $[8, 9, 10, 11, 12, 13, 14, 15]$. To finish, we can apply this algorithm recursively to perform an 8-qubit QFT on cube $[8, \ldots, 15]$. This yields a total of 35 swaps, which is cheaper than the upwards of 50 swaps needed to implement the direct radix-2. While the above sequence seems arbitrary, it actually follows directly from a particular decomposition of the FFT algorithm.

Inspecting closer, we recognize this sequence of operations corresponds quite closely with the Cooley-Tukey decomposition algorithm. Specifically, we can build up this circuit by first applying Cooley-Tukey to perform an even split at every iteration, and then modifying the order of the rotation gates in order to match the edges present in the hypercube. The original Cooley-Tukey circuit is shown in Figure 4.12, and the modified schedule is shown in Figure 4.13.

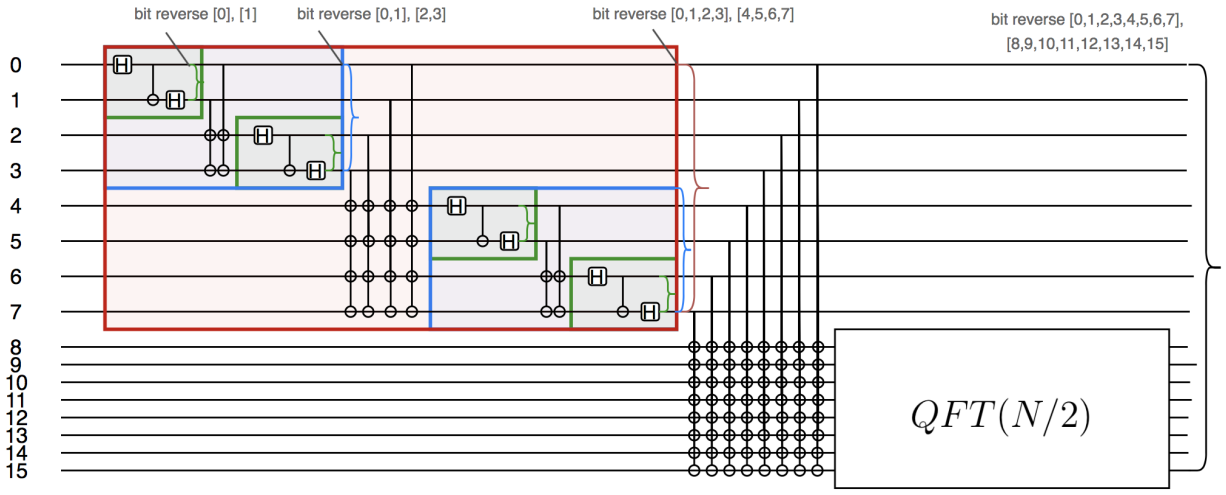


Figure 4.12: A 16-qubit QFT via Cooley-Tukey decomposition.

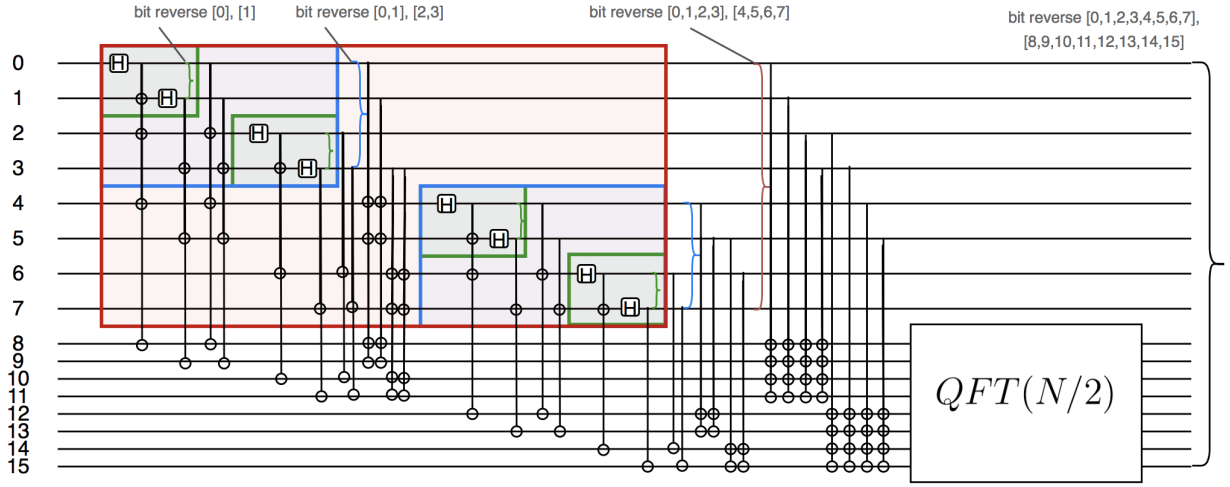


Figure 4.13: Modified 16-qubit Cooley-Tukey QFT, with hypercube-informed twiddle scheduling.

Figure 4.13 is no different than Figure 4.12; the only modification made was in adapting the gate schedule to take advantage of hypercubic neighbors in each position. Additionally, almost all of the swaps we described result directly from the final bit reversal of the output (a bit reversal in the quantum domain is simply a reversal of the qubit ordering). However, notice that on the diagonals of Figure 4.13 that some rotations are performed not strictly along hypercubic edges. By fully expanding these diagonals, as shown in Figure 4.14, we reach exactly the protocol described.

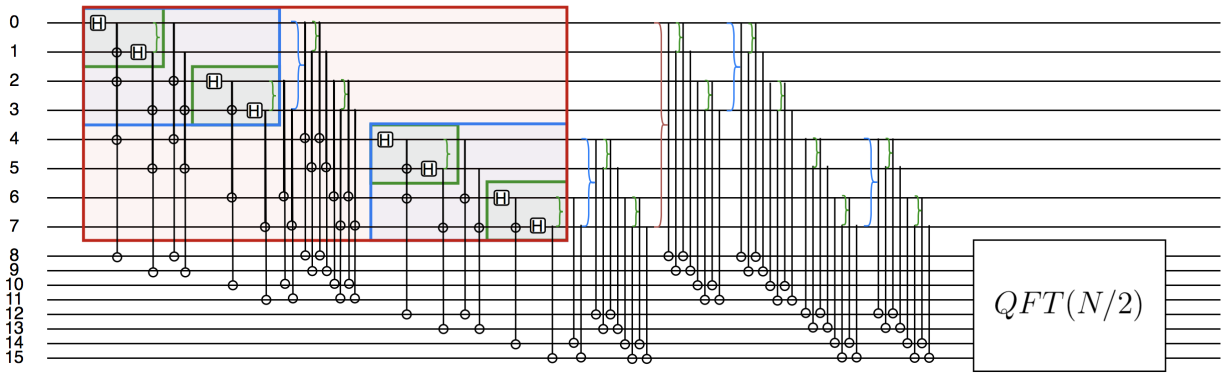


Figure 4.14: Expanded static hypercube QFT decomposition (final bit reversal omitted).

However, as stated before, quantum devices are rarely built in perfect hypercubes, meaning that this algorithm will rarely achieve the ideal 35 swaps for a 16-qubit QFT. In order to translate this heuristic algorithm into one usable for arbitrary or mesh topologies, we must perform an embedding of the hypercube onto our topology.

4.4.3 Mesh Embedding

Given the geometrically-inspired algorithm from the previous section, we now face the task of embedding the assumed hypercube onto an arbitrary input topology.

Inspecting the first plane in the hypercube, specifically the hardware locations $[0, 1, 2, 3]$, we notice that these qubits must control rotations to all other qubits in the system. This in turn suggests that these qubits should be clustered together in an area of the architecture that allows them to minimize the distance to all other qubits in the circuit. In other words, we attempt to place these 4 qubits in the center cluster of our architecture. Interestingly, searching for the center cluster of a graph can be made more efficient if the graph is *chordal* (which, interestingly, is useful for unrelated reasons in classical register allocation), but no such considerations are typically made when designing quantum hardware.

After the center plane is placed, increasing indices of qubits are placed around it in groups of 4, meaning we place one plane at a time. These additional qubits are placed to minimize their distance to any hypercubic neighbors that are already mapped in the architecture. For example, in placing the second plane consisting of qubits $[4, 5, 6, 7]$, we will be simultaneously minimizing 4's distance to $[0, 5, 6]$, 5's distance to $[4, 1, 7]$, 6's distance to $[2, 4, 7]$, and 7's distance to $[5, 3, 6]$. On a tie, we treat lower-dimensional edges as more important, meaning we successively build up

a centralized cluster in the architecture and avoid the edges of the connectivity graph wherever possible.

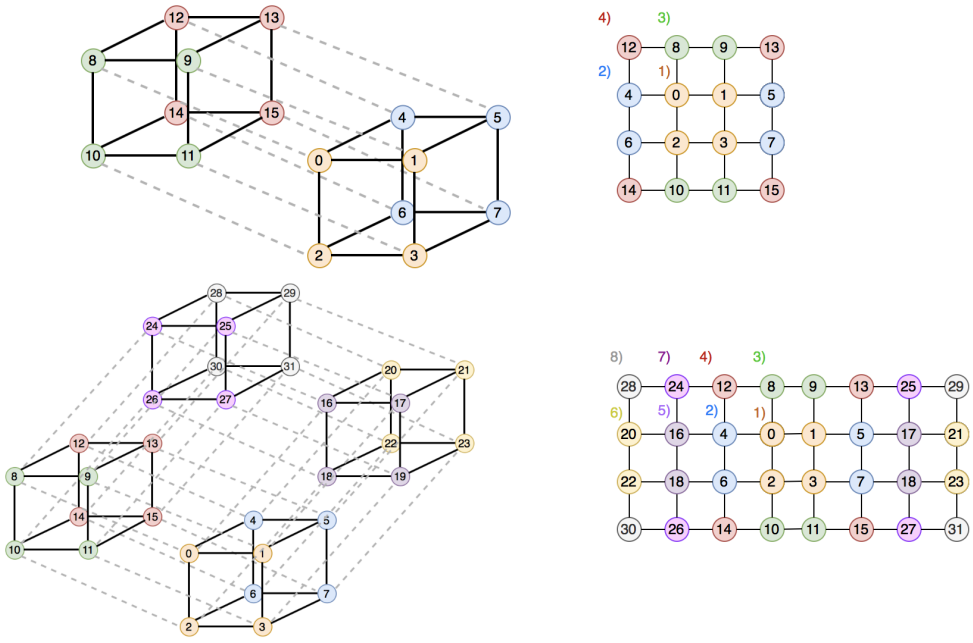


Figure 4.15: Embedding a 16-qubit hypercube into a 4×4 mesh topology [top], 32-qubit into 4×8 mesh [bottom].

For large architectures this placement algorithm becomes computationally intensive. Luckily, we can speed up the process by precomputing this embedding and storing it in a hashtable with the archictecture as the hash key. This is acceptable to us since hardware architectures change very slowly when compared to software algorithms. Additionally, if this needs to be further optimized in the future, this algorithm is a small black-box module which could be accelerated independently of the rest of the system. Regardless, once we have embedded the hypercube into the architecture, we have directly solved for the hardware locations that we wish to compute the QFT on and no longer need to search blindly over all other permutations.

Notice, however, that this embedding should arguably change throughout the stages of the QFT. Take as an example a swap performed between qubits 4 and 5 in the 4×4 16-qubit mesh in Figure 4.15. In the static hypercube algorithm we perform this swap assuming that it is cheap since 4 and 5 are neighbors in the hypercube. However, when this cube is flattened into a mesh, this no longer is the case. Ideally, if we wanted to take advantage of the hypercubic edges extending from plane $[4, 5, 6, 7]$, we would prefer the embedding to look like that shown in Figure 4.16.

Viewing the mesh embedding as a flattened version of the 16-qubit 4-cube, this new layout corresponds to a rotation of the original cube. Specifically, we have reflected across the third dimension, or rather the dimension in which qubits 0/4 and 8/12 share an edge.

When we finish with all qubits $[0, \ldots, 7]$ and wish to call our procedure recursively to implement an 8-qubit QFT on qubits $[8, \ldots, 15]$, we similarly want to rotate our mapping to put these

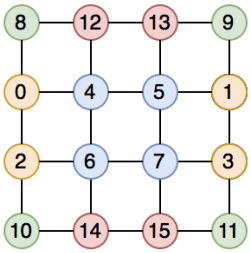


Figure 4.16: A 16-qubit hypercube embedding after performing a 3-dimensional cube rotation.

outermost qubits in the center of the mesh. At this stage in the algorithm we are done computing on the inner cube qubits $[0, \ldots, 7]$ and so we can rotate these qubits to the outside of the mesh without a need to ever recall them to the center. The ideal layout is shown in Figure 4.17.

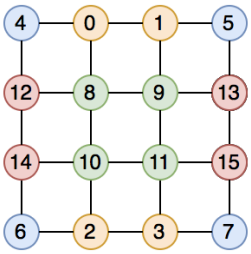


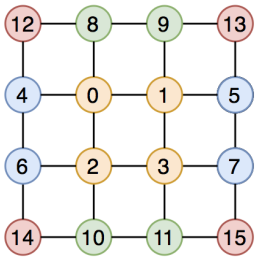
Figure 4.17: A 16-qubit hypercube embedding after performing a 4-dimensional cube rotation.

Extending these arguments, this means that we should continually rotate our hypercube embedding such that we never compute on non-centered qubits. This logic generalizes reasonably to non-mesh topologies since our embedding algorithm is agnostic to the input architecture. We will incorporate these rotations to form the *dynamic hypercube algorithm* that we discuss in the next section.

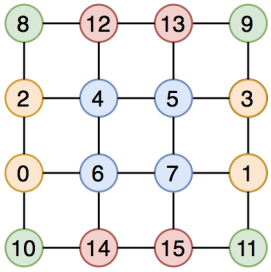
4.4.4 Dynamic Hypercube Algorithm

Following from aforementioned motivation we now modify the *static hypercube algorithm* by rotating the hypercube embedding throughout various stages of the QFT. The goal of these rotations is to make sure that we never compute on non-centered qubits. For an N qubit QFT, this means that we want the $N/2$-qubit hypercube that we are currently computing on to be centered. This extends to the $N/4$ hypercube within that, the $N/8$ hypercube within that, and so on. Since these rotations are not free we do not want to perform the same rotation twice, meaning that we want to finish computing on the currently-centered $N/2, N/4, N/8, ..., 4$-sized substructure before swapping it out of the center. By doing this properly, we achieve our goal of slowly pushing completed qubits farther from the center, leaving us with a monotonically non-increasing central cluster of relevant qubits.

Walking through the proposed algorithm, let us start with the 16-qubit mesh embedding shown in Figure 4.15.

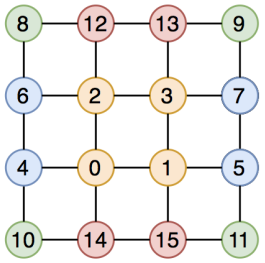


We can apply the swap sequence shown in Figure 4.11 to the $[0, 1, 2, 3]$ plane, handshaking the qubits in plane $[0, 1, 2, 3]$ with those in planes $[4, 5, 6, 7]$ and $[8, 9, 10, 11]$ in the process. In the static algorithm we would repeat this process directly on the $[4, 5, 6, 7]$ plane, but we understand this is inefficient since the $[4, 5, 6, 7]$ plane is not centered. Instead we perform a *rotation* on our mesh embedding to move these qubits and their neighbors into the center, and only then do we apply this swap sequence. This rotation is along dimension 3 ($3 = log_2 8$), and in the physical circuit, takes place after the 8-qubit diagonal.

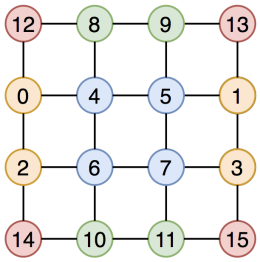


Applying the swap sequence shown in Figure 4.11 to qubits $[4, 5, 6, 7]$, we are able to handshake with plane $[12, 13, 14, 15]$. We do not handshake with plane $[0, 1, 2, 3]$ since those rotations are not needed in the QFT algorithm.

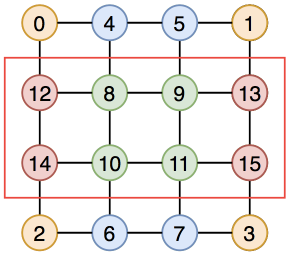
Continuing, we notice that we need to rotate from plane $[0, \ldots, 3]$ to plane $[12, \ldots, 15]$ and from plane $[4, \ldots, 7]$ to plane $[8, \ldots, 11]$. To achieve this, we swap the $[0, 1, 2, 3]$ plane with the $[4, 5, 6, 7]$ plane; this is not a rotation like before, but rather is just a permutation resulting from the Cooley-Tukey algorithm.



We can now twiddle from plane $[0, 1, 2, 3]$ to plane $[12, 13, 14, 15]$ with the same sequence of 6 swaps. To finish computing on our inner cube qubits $[0, \ldots, 7]$, we must re-rotate along dimension 3 ($3 = log_2 8$) to handshake plane $[4, 5, 6, 7]$ with plane $[8, 9, 10, 11]$.



To conclude, we perform a rotation along dimension 4 ($4 = log_2 16$, occurring after the 16-qubit diagonal) before calling this procedure recursively to implement an 8-qubit QFT on the now-centered outer cube qubits $[8, \ldots, 15]$.



The procedure outlined above is exactly that presented in the static algorithm, except with the addition of several rotations to slowly shift hypercubic substructures in and out of the center. Specifically, in terms of QFT stages, we rotate along the $log_2(N)^{th}$ dimension immediately after the N-qubit twiddle diagonal. The resulting circuit is shown in Figure 4.18.

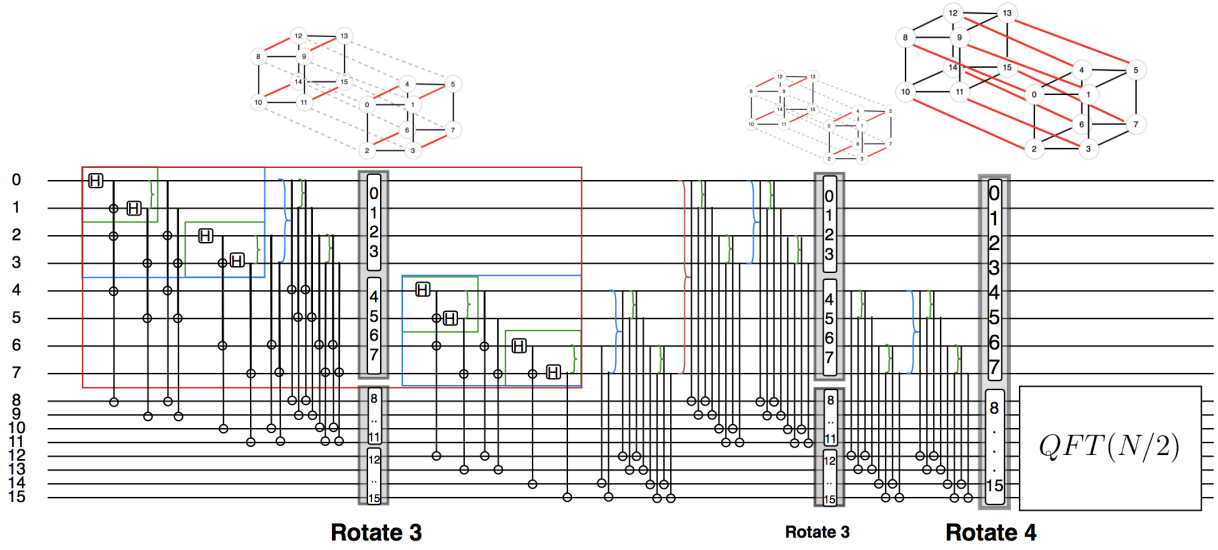


Figure 4.18: The 16-qubit dynamic hypercube QFT decomposition (static hypercube with additional embedding rotations).

In the 16-qubit example, All hypercubic edges extending from the center plane manifest as real edges in the mesh architecture. As the number of qubits increases this is no longer the case. Consider a 64-qubit QFT executed on a mesh; in the 64-qubit hypercube shown in Figure 4.19, qubit 0 should have edges extending to qubits 1, 2, 4, 8, 16 and 32, ordered canonically.

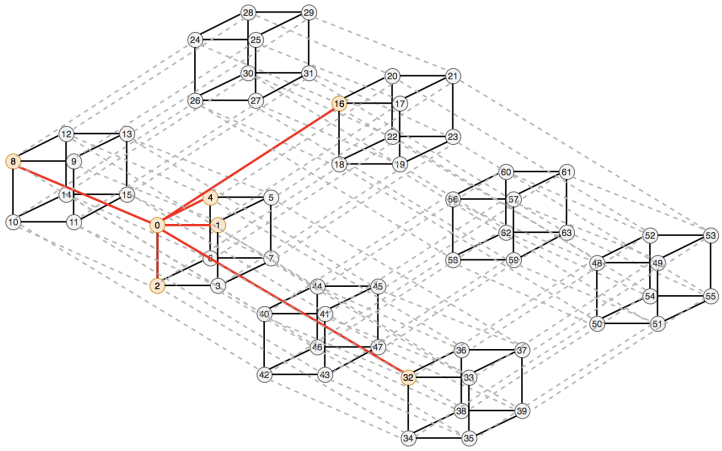


Figure 4.19: A 64-qubit hypercube. Qubit 0 has 6 neighbors with edges marked in red.

To handshake qubit 0 with all its hypercubic neighbors, then, we must do the following:

- $0 = twiddle \Rightarrow 1$
- $0 = twiddle \Rightarrow 2$
- $0 = twiddle \Rightarrow 4$
- $0 = twiddle \Rightarrow 8$
- $0 = twiddle \Rightarrow 16$
- $0 = twiddle \Rightarrow 32$

However, despite qubit 0 being neighbors with all these qubits in the hypercube, it will actually only be neighbors with qubits $[1, 2, 4, 8]$ since we can have at most 4 neighbors in the mesh. To implement these non-ideal edges we must temporarily bring qubit 0 out of the center to complete a tour on which qubit 0 is able to handshake with qubits 16 and 32. To do this, we compute the powers of the adjacency matrix in order to find the shortest path(s) from qubit 0 to qubits 16 and 32. After merging these paths in a manner identical to $\mathrm{Reorder}$ object cancellation, we obtain the optimal swap sequence that implements both of these compound edges at once. If there are multiple shortest paths, we try all combinations. In reality, this sequence can be expressed in SPIRAL as a subcircuit, and our solver will automatically implement this procedure.

4.4.5 Diagonal Hypercube Algorithm

Through developing the last two heuristics we have built up a series of useful abstractions for implementing efficient hypercubic QFT circuits. In this last heuristic, we similarly map an idealized algorithm onto a real architecture, but start with a different algorithm than the one presented

in Section 4.4.2. Specifically, we leverage the ring or torus structure of a hypercube (Figure 4.20), and attempt to copy a successful low-dimensional technique into higher dimensions. In lower dimensions this approach actually garners some improvement over the dynamic hypercube method presented above.

Given a ring of hypercubic structures labeled a, b, c, d, our goal is to twiddle all qubits in structure a to those in structures $[b, c, d]$, all qubits in structure b to those in structures $[c, d]$, and then call our algorithm recursively on the c, d substructure. Partitioning various hypercubes into ring topologies is seen below.

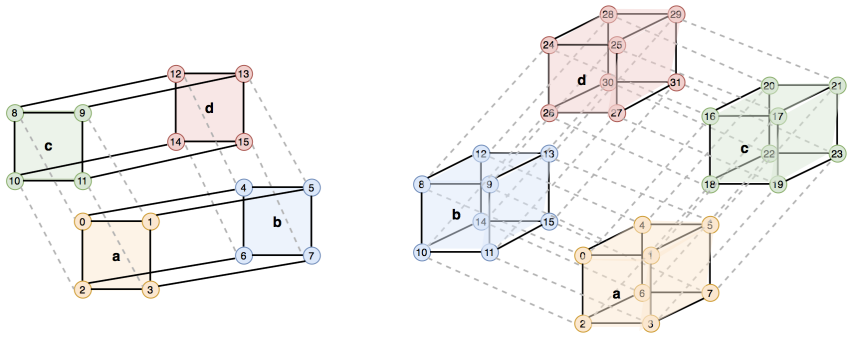


Figure 4.20: Partitioning of N-qubit hypercubes into rings of $N/4$-qubit hypercubes.

Recall that we are still embedding this hypercube onto a flattened graph. Therefore, structure a is the centermost $N/4$-qubit hypercube, $[a, b]$ is the centermost $N/2$-qubit hypercube, and $[c, d]$ is the outermost $N/2$-qubit hypercube. We will try to perform this operation in such a manner as to never have the same $N/4$-qubit hypercube occupy the center twice, and we must end with $[c, d]$ in the center $N/2$ positions before making our recursive call. With the standard ordering, however, we notice some unfortunate properties when embedding our architecture onto a mesh in Figure 4.21.

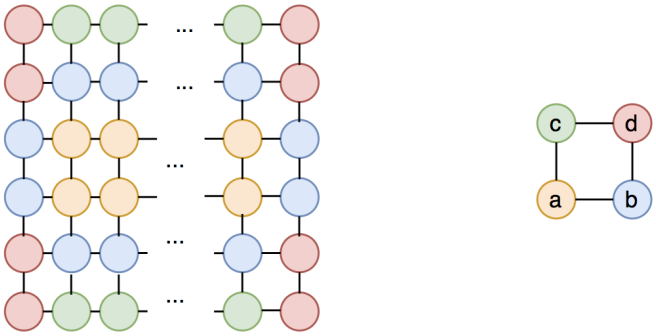


Figure 4.21: Division of N-qubit hypercube embedding into 4 $N/4$ regions. Partial flattening onto mesh topology [left], high-level view [right].

After twiddling from a to b and c, we must twiddle from structure a to structure d, which is separated from the center along the highest-dimensional diagonal. The most effective place

to put d is in b's position, since that will provide the shortest distance between the qubits in a and the qubits in d. However swapping b out of the center is not desirable, because the next $N/4$-sized substructure to occupy the middle must be b due to the constraints on twiddle gate ordering. We can get around this by simply *starting* with substructure b on the diagonal, as shown in Figure 4.22.

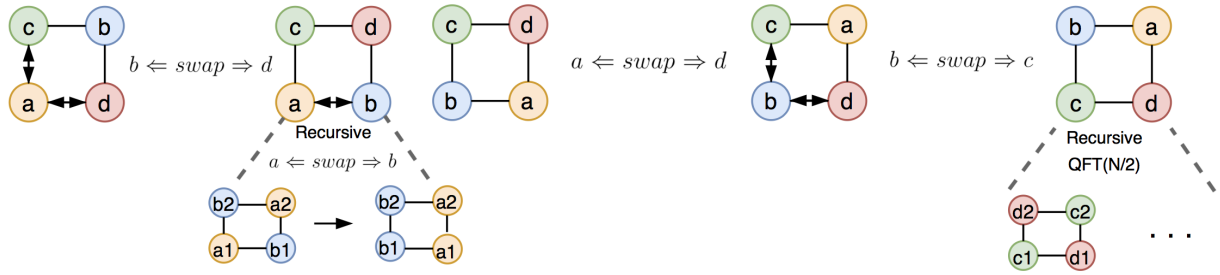


Figure 4.22: Twisted torus handshaking pattern.

With b on the diagonal, a can twiddle to the centermost d and c groups. We can then pull b into its proper location, untwisting the ring, and twiddle from a to b. Since a is now completely computed on, we can move b into the center and push a to the diagonal location. Now, we can twiddle from b to groups d and c, before shifting c into the central location and implementing a recursive $N/2$-sized QFT on the centered c and d groups.

We can leverage this pattern to implement a variant of the QFT heuristics previously proposed. We will perform the same hypercube embedding as done in the previous sections, except permuting the starting ordering such that the b and d qubits are swapped. We then can scale this simple a, b, c, d handshaking pattern to larger hypercubes. The complete sequence is shown in Figure 4.22.

While this heuristic is less directly inspired by Cooley-Tukey than the previous algorithm, it showcases the wealth of options available in determining the optimal placement patterns of algorithms for various architectures. Of note is that these problems lie completely in the realm of theoretical computer science, and not much of the problem requires any knowledge of quantum computing. This last point suggests that there are large bodies of classical literature that could directly inspire more heuristics like the three shown in this thesis.

4.4.6 Non-powers of two

The heuristics above assumed a power-of-two number of qubits. While making similar assumptions is not unusual in this space (radix-2, for example, requires a power-of-two-point transform), the above approach can be expanded.

For a QFT on a N qubits where $log_2 N \notin \mathbb{Z}$, QSPIRAL can decompose the QFT with other rules until one of the subcomponents is a power of two, in which case the hypercubic heuristics can be applied to only that subproblem. Since we search over all decompositions, this equates to finding the largest section of k qubits we can apply these heuristics to, applying our mesh embedding on the substructure represented by these k qubits. The other $N - k$ qubits can then be decomposed regularly, most likely according to the Cooley-Tukey rule.

It is important to note that this concerns *qubit* counts that are not powers of two. A QFT will always implement a power-of-two point transform, since an N-qubit QFT implements a 2^N-point DFT on the state vector.

4.5 Conclusions

In this chapter we outlined several possible heuristics for generating quantum Fourier transform circuits in the QSPIRAL framework. We were able to draw parallels between classical FFT literature and QFT circuitry, a process that required pattern matching between the classic signal flow diagrams of FFT algorithms and the effect that QFT circuits have on the input state vector. This allowed us to leverage large bodies of FFT literature in order to inspire our heuristics; we specifically chose a hypercube heuristic, and outlined the various ways these could inform how we decompose the input.

These heuristics fit seamlessly into the general QSPIRAL framework discussed in the previous chapter. If we decide to fire a specific heuristic based on the metadata we have available, we can partition the qubits according to the hypercube embedding and then apply either the dynamic or diagonal handshaking patterns discussed above. For qubit counts that are powers of two, this greatly accelerates the decomposition process as we do not need to search over any other decomposition rules for that symbol.

These heuristics make no claim to optimality; they are intended to exemplify our proposed compilation approach. Since QSPIRAL captures high-level information about the algorithm, there is no reason to treat a QFT as an arbitrary string of gates as do other frameworks. We have high-level metadata to work from, and thus we can implement a wide variety of mapping algorithms and fire each one when appropriate. If no heuristics apply, then QSPIRAL can simply backtrack to trying a search over rules such as Cooley-Tukey decomposition. A similar approach is taken by SPIRAL in the classical domain, and we believe it can be successful in the quantum domain. We indeed show a reasonable degree of success in the next chapter, in which we present and analyze our results.

Chapter 5

Evaluation

This chapter discusses the qualitative and quantitative evaluation of the QSPIRAL system. Specifically, we endeavour show the following key points.

- **Competitive results on general circuits.** QSPIRAL is able to produce competitive circuits, evaluated with respect to data movement costs, when invoked on gate-level test programs. To show this, we evaluate QSPIRAL on a series of benchmarks and provide a qualitative analysis of its performance.
- **Advantages on high-level input.** QSPIRAL, by leveraging the high-level symmetries of symbolic transforms, is often able to more effectively schedule these transforms on hardware. We show promising results for the QFT in particular. We also analyze the deficiencies of these results and where future work is needed.

In Section 5.1 we discuss the generalization of QSPIRAL to arbitrary inputs, and argue that while QSPIRAL accepts gate-level input, it is not our intended use case. In Section 5.2 we evaluate QSPIRAL's ability to generate QFT circuits from a high-level description, showing that we are able to produce results that are superior or comparable to those produced by Qiskit in a number of cases; we evaluate the performance of our heuristics on both mesh and non-mesh architectures. Conclusions are then drawn in Section 5.3.

5.1 Generalized Connectivity Satisfaction

To evaluate the QSPIRAL system, we must first show that the framework can be generalized to compile all inputs. We begin by showing that QSPIRAL produces circuits comparable to Qiskit's when invoked on gate-level input; this establishes that QSPIRAL's rewrite system is capable of performing peephole optimizations that are similar to those used by Qiskit. Next, we analyze these results and describe why, despite QSPIRAL's ability to handle these inputs, that the true potential of our system is in compiling and leveraging higher-level representations. We showcase this potential in the next section, and are able to reap significant benefits by leveraging high-level information.

While benchmarking suites such as QUEKO [62] and VOQC [35] exist, we choose, for a number of reasons, to form our own limited benchmarks for the purposes of this thesis. First, in order to more directly compare the modules in QSPIRAL and Qiskit respectively that solve

for connectivity, we wanted our benchmarks to consist only of controlled gates for simplicity (since single-qubit gates have no connectivity constraints). Additionally, for reasons analyzed in Section 5.1.2, QSPIRAL is relatively inefficient in compiling low-level circuits due to the lack of high-level symmetries to apply; without proper heuristics, QSPIRAL is not able to prune the search space effectively enough to give a reasonable execution time on longer gate-level expressions. Therefore, our benchmarking suite consists primarily of shorter circuits on smaller architectures, as seen in Figure 5.1. In order to efficiently compile these circuits, an optimized logical-to-physical mapping must be found, thereby leveraging QSPIRAL's ability to solve a generalized form of the connectivity satisfaction problem.

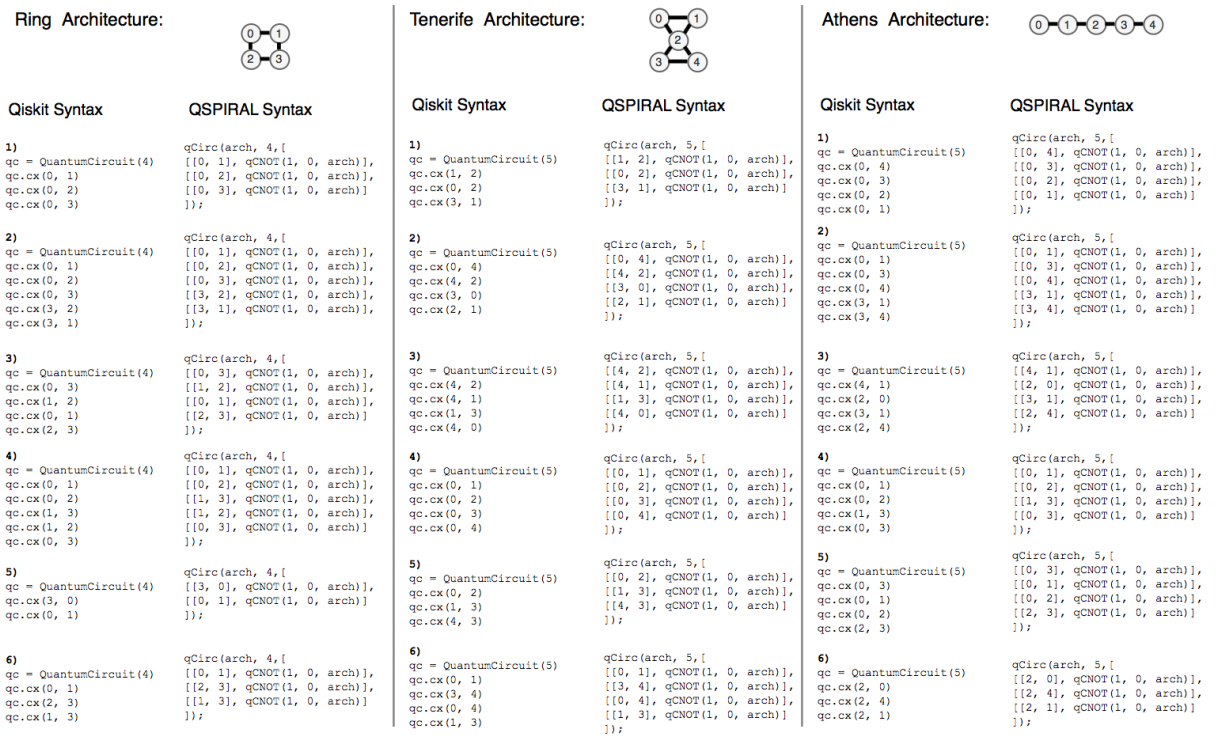


Figure 5.1: Benchmarks for evaluating generalized connectivity satisfaction.

5.1.1 Results

Figure 5.2 shows SWAP counts across various QSPIRAL compilation outputs, measured against those produced by different Qiskit optimization levels. Since the primary evaluation criterion in this work is the number of SWAP operations, the test circuits (Figure 5.1) are formulated as strings of controlled gates on sparse architectures. These circuits were chosen in order to isolate the particular compilation passes in Qiskit that satisfy connectivity, and compare their results against QSPIRAL's solution; since no high-level algorithmic symmetries can be taken into account, these systems are essentially applying two radically different heuristic algorithms to solve the same intractable problem. QSPIRAL's solver is shown to be competitive with Qiskit's on these circuits, and we would expect this trend to generalize to a much larger set of benchmarks.

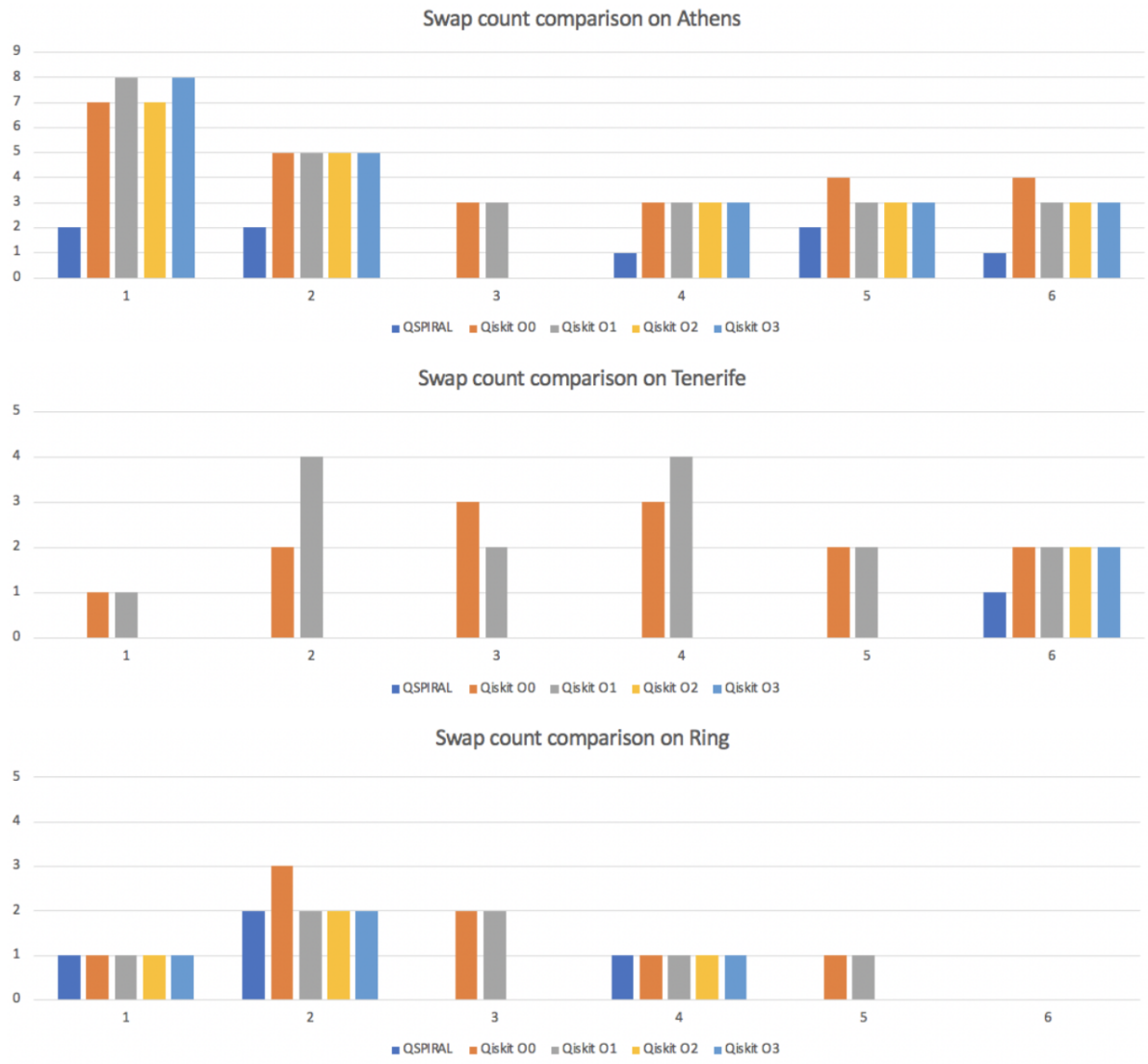


Figure 5.2: SWAP count on test circuits for three quantum architectures. X-axis is circuit number, in the order shown by Figure 5.1.

While the CNOT gates that make up these circuits could potentially be cancelled with the inserted swaps (recall that a SWAP gate can be implemented with 3 CNOT gates), we do not decompose the SWAP gates any further. This is done in order to more accurately compare the constraint satisfaction modules in both compilers. We could, however, exclude SWAP from our set of basis gates and minimize with respect to any of other gates or sets of gates, CNOT included; this only requires modifying the cost measure, as the rewrite phase (Section 3.3.3) will apply our library of gate-cancellation rules to simplify the formula accordingly. Since minimizing swaps is the primary motivation for this thesis, and there are still several simplification rules in Qiskit that have not yet been ported over as QSPIRAL rewrite rules, we feel justified in purely comparing SWAP gates. We believe QSPIRAL should be able to match Qiskit on these circuits for any cost measure if the large library of Qiskit gate identities were ported into QSPIRAL's rewrite system.

However, reimplementing Qiskit in SPIRAL is not the intent of this thesis. We describe in the next section why these results are not particularly consequential, as the true test of our system lies in QSPIRAL's ability to leverage algorithmic information to inform appropriate heuristics.

5.1.2 Analysis

As is clear by now, while QSPIRAL is competitive on our gate-level benchmarks, the system is not intended to take in programs expressed at that level. Providing input in terms of a linear series of gates equates to the input that traditional transpilers expect, and is suboptimal for reasons explained in the previous chapters. The novelty of our approach lies in the fact that we can leverage high-level algorithm specifications to construct optimized circuits directly; QSPIRAL is simply a mathematical engine that efficiently decomposes large algorithms into quantum circuit form. If the input transform consists only of low-level gates, such as a series of qCNOT nonterminals as shown in Figure 5.1, there are few such symmetries to apply. In this case, one might often be equivalently better off applying the local search procedure outlined by Qiskit, because there are essentially no high-level decomposition decisions to make.

There are not *no* decisions to make, and the results in the previous section are therefore still reasonably encouraging; QSPIRAL still manages to produce more efficient circuits than can Qiskit due to its ability to expand the global search space. Consider Figure 5.3, in which we exhaustively expand the search spaces for some of our test circuits and categorize with respect to SWAP count; the only heuristic rule employed is the one ensuring that controlled rotation gates are only mapped onto adjacent hardware qubits.

While gate counts vary wildly across the various possible decompositions, QSPIRAL is still able to find the most efficient formula and return it. One of the most promising features of QSPIRAL is its ability to generate a complete search space from the top-down, and *intentionally* prune this search space with heuristics; this is in stark contrast to peephole simplification methods, which cannot access any such search space, and are *structurally limited* to optimizing over a localized window. However, since QSPIRAL must then search over this global space, it relies heavily on having the proper heuristics to make this process tractable. These heuristics do not necessarily exist with gate-level input since there are few symmetries to apply. Therefore, the immense size of this search space currently limits QSPIRAL performance on low-level input.

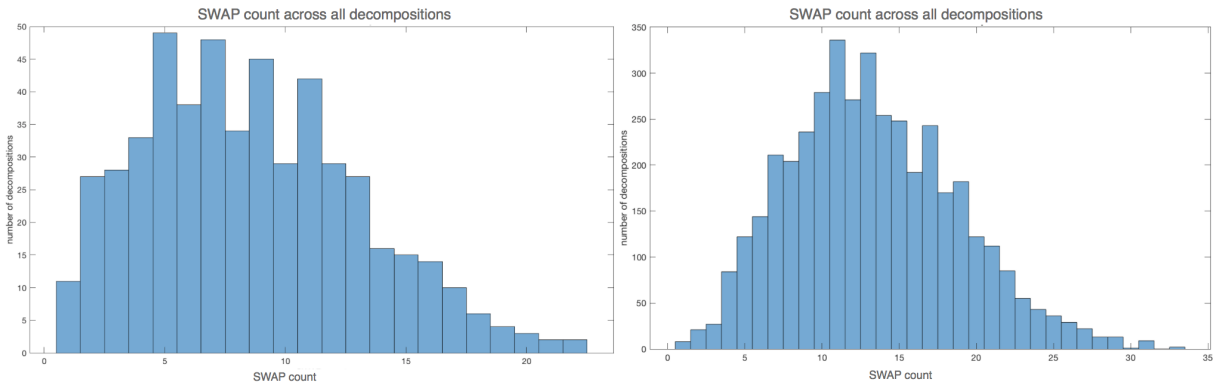


Figure 5.3: Search space histogram evaluated with respect to SWAP count: athens test circuit 6 [left], athens test circuit 4 [right]. Circuits resulting from all valid rule trees are plotted with swap count on the x-axis and circuit count on the y-axis.

Indeed, without *any* such heuristics, one would be forced to confront an intractable problem head-on, as shown in Figure 5.4.

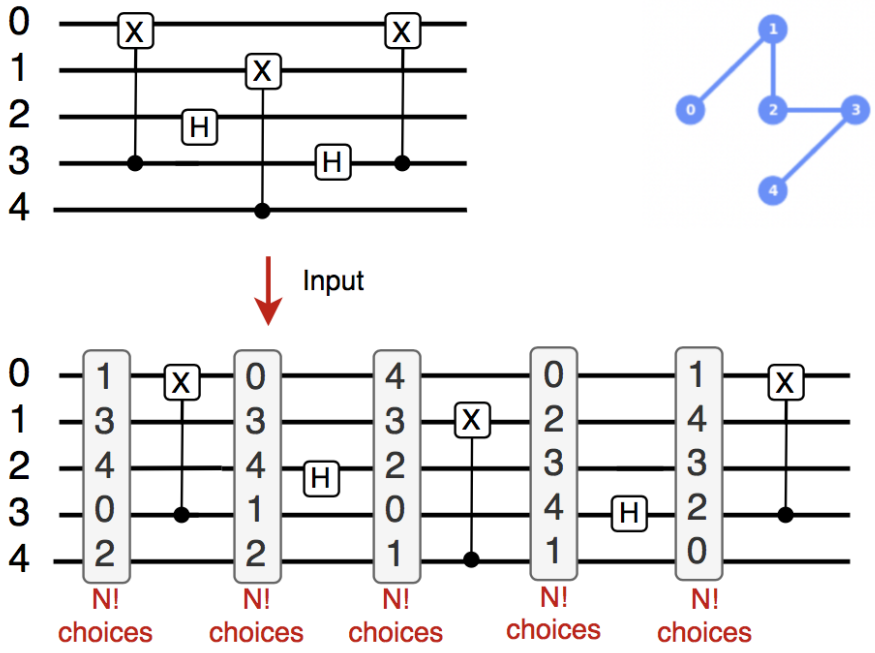


Figure 5.4: Naïve scheduling problem for low-level circuit input, with a reordering step between every gate.

One solution to this would be to introduce randomization, or to simply cap the number of decompositions we search over at some function of qubit count or circuit depth. In our case, besides preemptively pruning any reordering that violates connectivity (i.e. mapping a qCNOT non-terminal onto hardware qubits that do not immediately share an edge), we can apply the

qCirc group-scheduling heuristic (Section 3.3.6) to make this problem easier. However, QSPIRAL's factorization framework is still mostly bypassed since the input matrices are already in sparse form; we have equated circuit generation to sparse matrix decomposition, so many of our stages now have little to no effect.

The real benefits to be had with this type of system is in compiling algorithms expressed at a higher level of abstraction. Quantum programs are not, in our view, *circuits* so much as they are *linear transforms*. Shor's algorithm, for example, makes heavy use of the QFT, and Simon's a Walsh-Hadamard transform; both of these are better expressed as mathematical objects than circuits, since innumerable circuits can be used to implement both. Additionally, there is an identifiable set of linear transforms that is currently useful in the quantum domain, and arbitrary strings of gates rarely lead to any useful computation being done. We can circumvent the aforementioned complexity problem with high-level input because we only explore reorderings around functional unit boundaries, and even then only explore reorderings that make geometric sense given the architecture and the symmetries of the transform. We evaluate our implementation of one such high-level transform, the QFT, in the next section.

5.2 Quantum Fourier Transform

We now evaluate QSPIRAL's performance when executed on high-level algorithmic input. Specifically, we have targeted the QFT in this work. Other useful algorithms can be translated into QSPIRAL with the inclusion of additional decomposition rules and heuristics.

5.2.1 QFT on mesh architectures

The following statistics present QSPIRAL compilation results for the QFT algorithm on mesh architectures, measured against those from Qiskit on various optimization settings. Nearest-neighbor mesh architectures are relevant because they are immensely popular in cutting-edge quantum computers; they are also ideal for our QFT heuristics because hypercube structures embed well into meshes.

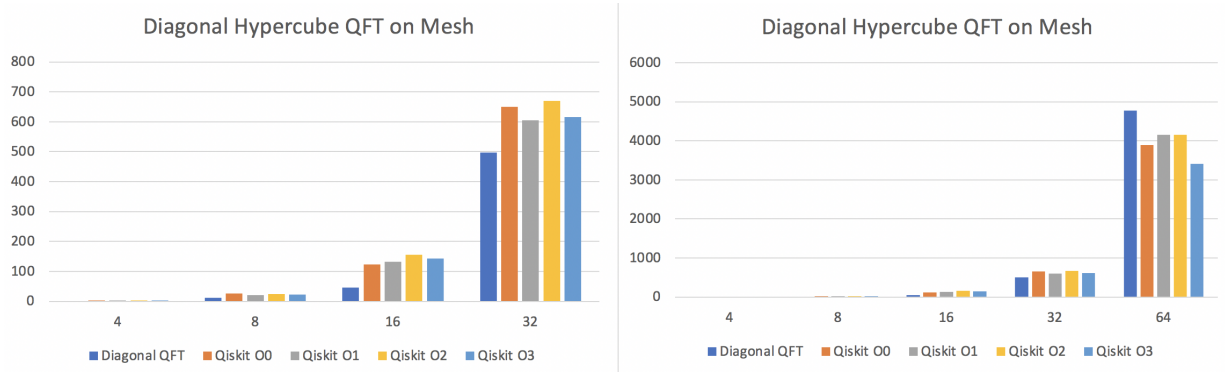


Figure 5.5: Diagonal hypercube QFT SWAP count evaluation on lattice: 4-32 [left], 4-64 [right]. Data is from Table 5.1.

Hypercube QFT SWAP count on lattice					
Qubits	Compiler	Optimization	Algorithm	Connectivity	SWAP Count
4	QSPIRAL	N/A	diagonal	2x2	1
4	QSPIRAL	N/A	dynamic	2x2	1
4	Qiskit	0	N/A	2x2	2
4	Qiskit	1	N/A	2x2	2
4	Qiskit	2	N/A	2x2	3
4	Qiskit	3	N/A	2x2	2
8	QSPIRAL	N/A	diagonal	2x4	11
8	QSPIRAL	N/A	dynamic	2x4	11
8	Qiskit	0	N/A	2x4	26
8	Qiskit	1	N/A	2x4	20
8	Qiskit	2	N/A	2x4	24
8	Qiskit	3	N/A	2x4	22
16	QSPIRAL	N/A	diagonal	4x4	45
16	QSPIRAL	N/A	dynamic	4x4	63
16	Qiskit	0	N/A	4x4	123
16	Qiskit	1	N/A	4x4	132
16	Qiskit	2	N/A	4x4	156
16	Qiskit	3	N/A	4x4	143
32	QSPIRAL	N/A	diagonal	4x8	497
32	QSPIRAL	N/A	dynamic	4x8	871
32	Qiskit	0	N/A	4x8	649
32	Qiskit	1	N/A	4x8	604
32	Qiskit	2	N/A	4x8	669
32	Qiskit	3	N/A	4x8	616
64	QSPIRAL	N/A	diagonal	4x8	4769
64	QSPIRAL	N/A	dynamic	4x8	7735
64	Qiskit	0	N/A	4x8	3893
64	Qiskit	1	N/A	4x8	4156
64	Qiskit	2	N/A	4x8	4149
64	Qiskit	3	N/A	4x8	3406

Table 5.1: Hypercube QFT SWAP count on lattice.

As shown in Figure 5.5, QSPIRAL realizes tangible data-movement savings for QFT sizes up to 32 qubits. At 64 qubits, results begin to degrade. This is due to the simplicity of our embedding; we assume in our idealized protocol that hypercube edges are cheaper than non-hypercube edges and implement data movement to ensure we only ever twiddle across a hypercube edge. For large meshes, however, many qubits are placed on the boundaries of the mesh and their hypercube edges become expensive to communicate across, violating our implicit assumption. In other words, the hypercube embedding degrades as qubits must be placed far from the center. We believe this bottleneck can be overcome with additional or modified heuristics.

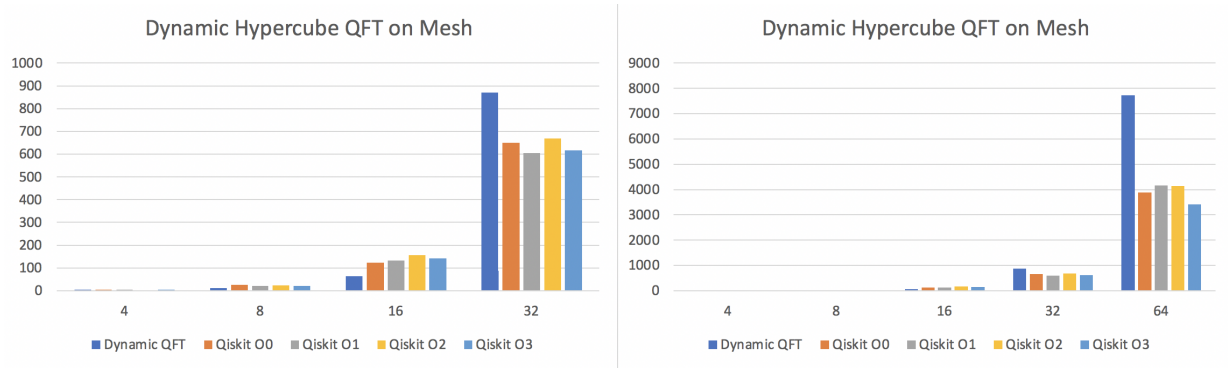


Figure 5.6: Dynamic hypercube QFT SWAP count evaluation on lattice: 4-32 [left], 4-64 [right]. Data is from Table 5.1.

As shown in Figure 5.6, the dynamic hypercube algorithm also requires fewer data movement operations for QFTs up to 16 qubits than does Qiskit on maximum optimization settings. The dynamic algorithm similarly degrades at larger sizes due to the same issues noted above. It degrades more rapidly due to the increasing costs of performing the cube rotations needed to keep the computation localized to the center of the mesh. We believe that this bottleneck can be addressed by modifying the rotation scheme, as there is commonly no need to rotate the entire hypercube when a rotation of just the centralized cluster would suffice.

Regardless, we show in these results that there is most likely a place for structured approaches in the future, and that massive gains can potentially be realized by forming appropriate decomposition heuristics for various algorithms in a variety of architecture contexts. The eventual goal of the QSPIRAL system, similar to that of the classical SPIRAL system, is to have a large library of these heuristics and fire each one when appropriate, on the entire input transform at once or on various decomposed subformulas; this is necessary because it is exceedingly rare that any one heuristic performs well in all situations. If no heuristics apply in a given situation, QSPIRAL can instead apply traditional Cooley-Tukey decomposition.

5.2.2 QFT on non-mesh architectures

We also evaluated our QFT decomposition heuristics on the non-mesh architectures shown in Figure 5.7, specifically IBM's athens, guadalupe, toronto, aspen, rochester and manhattan architectures; these were chosen since they are large departures from mesh architectures, and since they are IBM devices, it could be expected that Qiskit is built to compile circuits for them efficiently. We show that, despite these architectures being far from our ideal hypercube, our heuristics are still able to produce competitive results. Different heuristics could be developed for these architectures, and would most likely boast vastly *better* results; the QSPIRAL framework takes architectural information into account, so were these to be developed, they could be seamlessly integrated into our system.

As shown in Figure 5.8, the algorithms perform similarly to Qiskit on these non-mesh architectures despite the hypercube embedding being far from optimal. Additionally, our approach is visibly more deterministic; we get equivalent 16-qubit results on the toronto, manhattan and

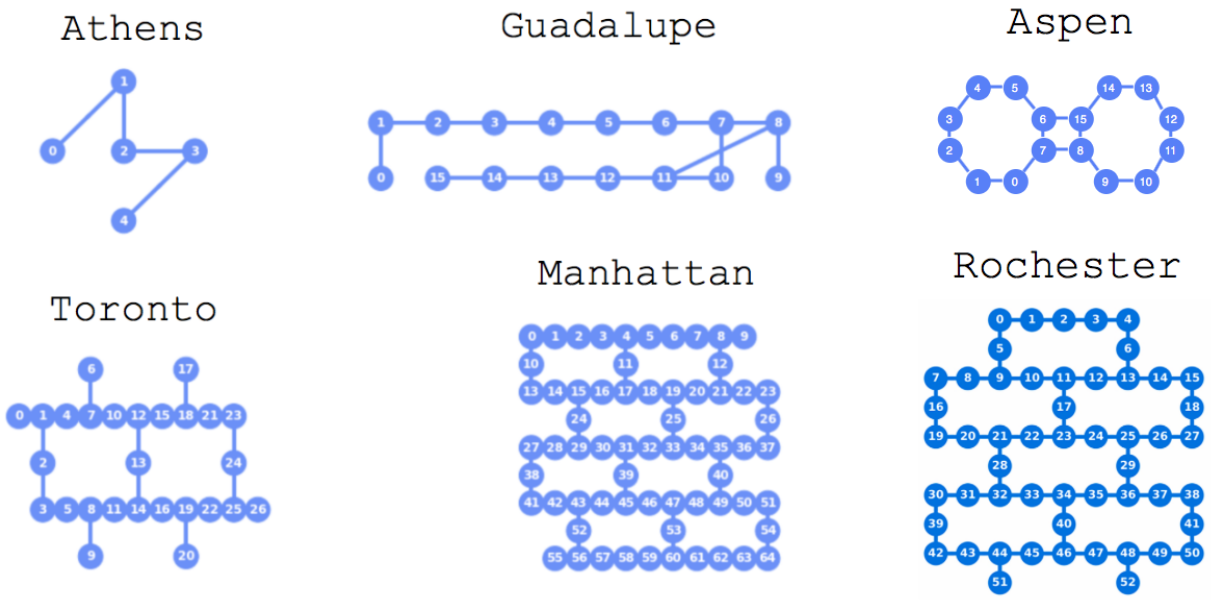


Figure 5.7: Chosen non-mesh architectures. Images sourced from Qiskit backend and adapted from [44].

rochester architectures, and equivalent 32-qubit results for the manhattan and rochester architectures. The same holds for 4-qubit results on the athens, toronto, manhattan and rochester architectures, and for the guadalupe and aspen architectures. These observations result from QSPIRAL being able to recognize and leverage the same optimal substructures within in each architecture. Following from this result, we expect that it could be possible to accelerate our hypercube embedding process (Section 4.4.3) by scanning an input architecture for substructures similar to those that QSPIRAL has encountered before; we could then reuse placements that have already been computed for that substructure. This could be useful if future large architectures have smaller, recognizable architectures embedded within them, such as the manhattan architecture containing several instances of the toronto architecture within it.

Despite being within a factor of Qiskit's performance, however, it is seen that the benefits of the algorithm are somewhat lost on architectures that depart massively from the assumed mesh topology. While this is unfortunate, it is not wholly unexpected; classical SPIRAL has many heuristics for computing algorithms on various hardware architectures, so there is no reason to believe QSPIRAL could not do the same, and fire the appropriate heuristic based on the algorithmic and adjacency matrix input. In this sense, these results serve not to dissuade further research in this space, but rather to motivate the expansion of this approach to more and larger architectures.

It is our expectation that our approach, with further work, will result in much more flexibility when attempting to tackle the scaling problem shown in Figure 1.1. By utilizing architectural and algorithmic inputs, we have more information to leverage when compared to existing approaches, and the benefits we *do* see prove that this algorithmic information can be useful if applied appropriately. QSPIRAL, due to its scope, has access to a strictly larger search space.

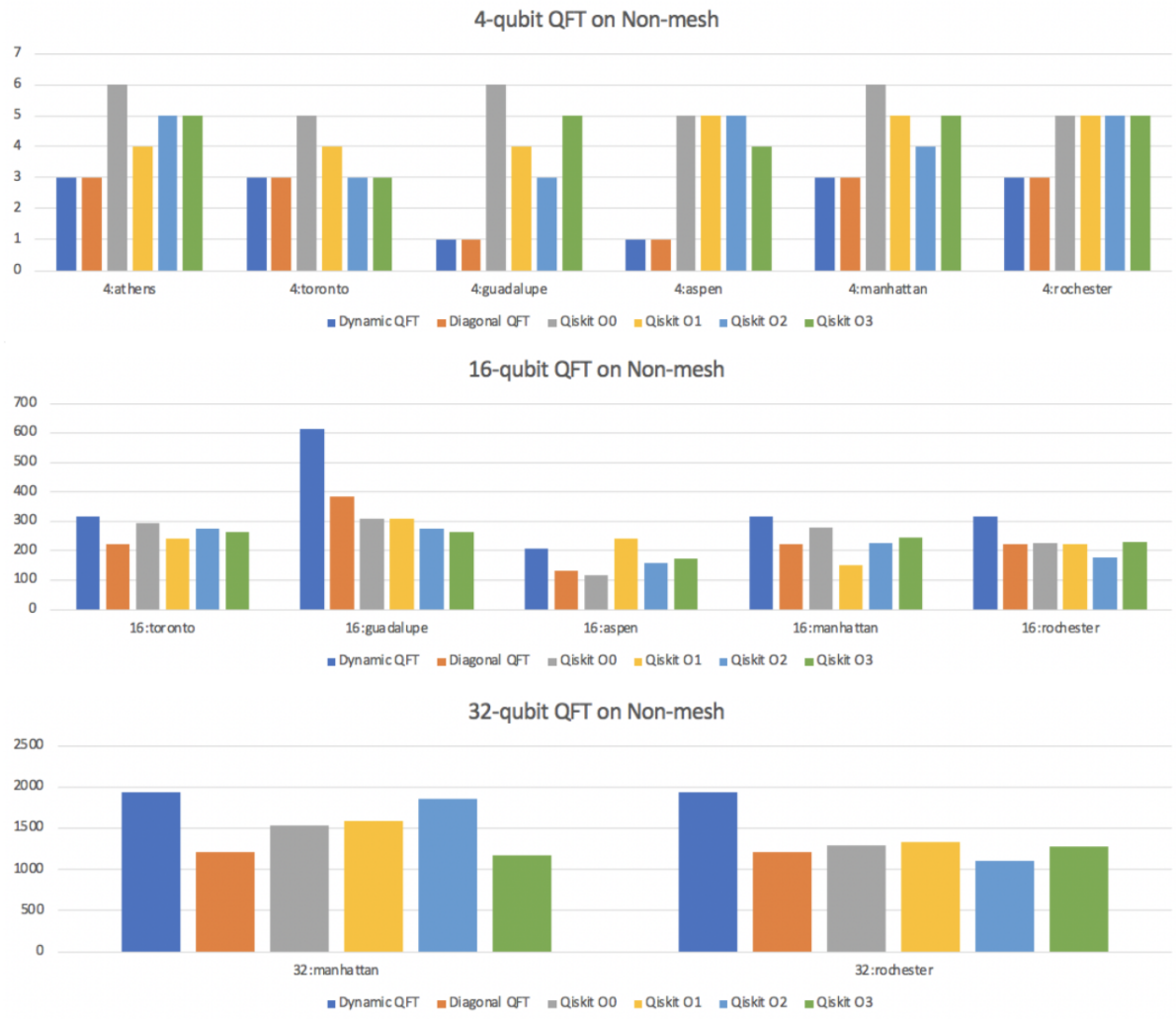


Figure 5.8: Hypercube QFT SWAP count evaluation on non-mesh. Data is from Tables 5.2, 5.3 and 5.4.

4-qubit QFT SWAP count on non-mesh				
Compiler	Optimization	Algorithm	Connectivity	SWAP Count
QSPIRAL	N/A	diagonal	athens	3
QSPIRAL	N/A	dynamic	athens	3
Qiskit	0	N/A	athens	6
Qiskit	1	N/A	athens	4
Qiskit	2	N/A	athens	5
Qiskit	3	N/A	athens	5
QSPIRAL	N/A	diagonal	toronto	3
QSPIRAL	N/A	dynamic	toronto	3
Qiskit	0	N/A	toronto	5
Qiskit	1	N/A	toronto	4
Qiskit	2	N/A	toronto	3
Qiskit	3	N/A	toronto	3
QSPIRAL	N/A	diagonal	guadalupe	1
QSPIRAL	N/A	dynamic	guadalupe	1
Qiskit	0	N/A	guadalupe	6
Qiskit	1	N/A	guadalupe	4
Qiskit	2	N/A	guadalupe	3
Qiskit	3	N/A	guadalupe	5
QSPIRAL	N/A	diagonal	aspen	1
QSPIRAL	N/A	dynamic	aspen	1
Qiskit	0	N/A	aspen	5
Qiskit	1	N/A	aspen	5
Qiskit	2	N/A	aspen	5
Qiskit	3	N/A	aspen	4
QSPIRAL	N/A	diagonal	manhattan	3
QSPIRAL	N/A	dynamic	manhattan	3
Qiskit	0	N/A	manhattan	6
Qiskit	1	N/A	manhattan	5
Qiskit	2	N/A	manhattan	4
Qiskit	3	N/A	manhattan	5
QSPIRAL	N/A	diagonal	rochester	3
QSPIRAL	N/A	dynamic	rochester	3
Qiskit	0	N/A	rochester	5
Qiskit	1	N/A	rochester	5
Qiskit	2	N/A	rochester	5
Qiskit	3	N/A	rochester	5

Table 5.2: 4-qubit QFT SWAP count on non-mesh.

16-qubit QFT SWAP count on non-mesh				
Compiler	Optimization	Algorithm	Connectivity	SWAP Count
QSPIRAL	N/A	diagonal	toronto	223
QSPIRAL	N/A	dynamic	toronto	317
Qiskit	0	N/A	toronto	294
Qiskit	1	N/A	toronto	241
Qiskit	2	N/A	toronto	276
Qiskit	3	N/A	toronto	262
QSPIRAL	N/A	diagonal	guadalupe	385
QSPIRAL	N/A	dynamic	guadalupe	615
Qiskit	0	N/A	guadalupe	307
Qiskit	1	N/A	guadalupe	309
Qiskit	2	N/A	guadalupe	276
Qiskit	3	N/A	guadalupe	262
QSPIRAL	N/A	diagonal	aspen	133
QSPIRAL	N/A	dynamic	aspen	207
Qiskit	0	N/A	aspen	119
Qiskit	1	N/A	aspen	240
Qiskit	2	N/A	aspen	160
Qiskit	3	N/A	aspen	174
QSPIRAL	N/A	diagonal	manhattan	223
QSPIRAL	N/A	dynamic	manhattan	317
Qiskit	0	N/A	manhattan	277
Qiskit	1	N/A	manhattan	150
Qiskit	2	N/A	manhattan	225
Qiskit	3	N/A	manhattan	245
QSPIRAL	N/A	diagonal	rochester	223
QSPIRAL	N/A	dynamic	rochester	317
Qiskit	0	N/A	rochester	228
Qiskit	1	N/A	rochester	221
Qiskit	2	N/A	rochester	178
Qiskit	3	N/A	rochester	229

Table 5.3: 16-qubit QFT SWAP count on non-mesh.

5.3 Conclusions

In this chapter we evaluate the QSPIRAL system for general inputs, as well as analyze the performance gains accrued from running the hypercube QFT heuristics outlined in the previous chapter. Our goal is not especially to evaluate the particular heuristics themselves, as much as to show that QSPIRAL is able to empower these heuristics and provides a framework for developing and employing them. If the approach taken by QSPIRAL is able to provide enough information as to enable *any* decent heuristics, that would be a great success in proving the utility of such a design.

32-qubit QFT SWAP count on non-mesh				
Compiler	Optimization	Algorithm	Connectivity	SWAP Count
QSPIRAL	N/A	diagonal	manhattan	1217
QSPIRAL	N/A	dynamic	manhattan	1937
Qiskit	0	N/A	manhattan	1532
Qiskit	1	N/A	manhattan	1588
Qiskit	2	N/A	manhattan	1858
Qiskit	3	N/A	manhattan	1171
QSPIRAL	N/A	diagonal	rochester	1217
QSPIRAL	N/A	dynamic	rochester	1937
Qiskit	0	N/A	rochester	1286
Qiskit	1	N/A	rochester	1331
Qiskit	2	N/A	rochester	1103
Qiskit	3	N/A	rochester	1281

Table 5.4: 32-qubit QFT SWAP count on non-mesh.

QSPIRAL performs reasonably well on an assortment of gate-level circuits, but as mentioned before, this is not the primary use case of the system. QSPIRAL is a compilation system that takes a novel viewpoint, namely that quantum circuits should be *generated* from a high-level algorithmic specification rather than handwritten as assembly and compiled.

QSPIRAL performs admirably well in certain cases when asked to decompose the quantum Fourier transform. Applying our breakdown heuristics, QSPIRAL is able to generate connectivity-compliant circuits that are significantly more optimized than those produced by Qiskit for various sizes. The dynamic and diagonal QFT results on meshes consisting of up to 32 qubits are especially impressive. We would expect this trend to continue; the fact that it does not implies that past 64 qubits our heuristics are not sufficient due to a degradation of the hypercube embedding. We expect, however, that additional breakdown rules or heuristics could overcome this challenge. The significant and deterministic improvements we see for smaller QFT sizes should be motivation for further developing our approach.

On trivially-small circuits, such as the 4-qubit QFT, we clearly see the benefits of applying algorithm-informed and structured approaches such as those employed in QSPIRAL. Even though a 4-qubit QFT on a 2×2 lattice can be easily implemented with a single swap, this solution is not often found by Qiskit. This is also a strong motivation for further developing our approach, as it could be unreasonable to expect scalable behavior from these existing approaches if reliability on smaller problems is still to be desired.

In the next chapter we discuss further work. Due to the encouraging results shown in this chapter, much of this work concerns expanding and refining our design rather than drastically modifying the scope of the project. However, there are other areas in this domain that QSPIRAL could be applied to, and we discuss those as well.

Chapter 6

Conclusions

In Section 6.1 we present a general overview of the topics discussed in this thesis. In Section 6.2 we discuss other applications of the QSPIRAL system within the quantum domain. We conclude with Section 6.3, in which we present closing remarks.

6.1 Overview

This thesis presented a novel approach for generating optimized quantum circuits. As an overview, we reassessed the quantum optimization problem by doing the following.

1. Discarding domain-specific notation for a more general formulation, and using the aforementioned formulation to draw useful parallels between quantum and classical algorithms
2. Formalizing circuit generation as a sparse matrix factorization task, using recursive rewrite rules to decompose the input *directly* from a high-level description instead of making local changes to a handwritten circuit
3. Casting the entire procedure as a generic search problem, over which we minimize data movement operations
4. Implementing a solver on an advanced computer algebra system founded solidly in group-theoretic principles

We started by covering the basic principles of quantum computing using notation that is familiar to the SPIRAL platform and allows us to more easily leverage relevant literature from other domains. The primary takeaway was that quantum circuits are simply sparse matrix factorizations of some overall $2^N \times 2^N$ unitary transform, and hence many decompositions and global rewrites of algorithms like the DFT directly applied to the quantum domain. Intuitively, applying these decomposition identities in practice required recognizing the characteristic transform of a circuit. However, we noticed that this transform matrix is impractical to expand from the circuit definition for large values of N, and therefore, existing approaches are often limited to making simple local changes to the program stream since those are all that can be done while ensuring unitary equivalence. In order to address this perceived insufficiency, we attempted to build a framework that would allow us to achieve tangible improvements by leveraging algorithm symmetries.

What we settled on was a system that takes in a high-level description of the target algorithm and summarily explores various relevant ways of decomposing this symbolic $2^N \times 2^N$ transform into the tensor and matrix product of quantum gates. These decompositions form mathematical expressions that can be simplified and evaluated based on a generic cost measure, in our case, the number of swaps. We leveraged a large library of breakdown rules that, when recursively applied, convert our symbolic input into one of these expressions. Searching over the possible rule applications equates to searching over algorithm decompositions, and by transitivity, the valid circuits that implement the transform. We pruned invalid decompositions by enforcing connectivity was met.

We implemented the aforementioned system (Figure 6.1) in SPIRAL, which utilizes an extremely similar technique to optimize algorithms for classical architectures. Particular care was taken to explain how qubits are partitioned at each breakdown step and how in-place computations can be executed on any subgroup of hardware locations in the overall architecture.

Recognizing that this system enables the application of powerful heuristics, and thus its utility is tied to the effectiveness of those heuristics, we implemented several for the QFT that were inspired by the classical parallel FFT. While these heuristics are not necessarily optimal, we showed significant data movement savings for moderately-sized QFT algorithms. Due to these promising results, we remain confident that expanding our library of heuristics might significantly increase our ability to scale algorithms to the potentially massive quantum hardware of the future.

6.2 Directions for Future Work

While the the general approach described in this thesis promises to be an effective methodology for framing and solving the quantum circuit optimization problem, there exist several improvements that could make this work more useful.

First, additional algorithms and heuristics should be implemented in QSPIRAL, particularly for important quantum kernels that aren't efficiently compiled with existing technologies. Besides the more traditional algorithms, the area of homomorphic encryption [28] could be of particular interest, especially since the number-theoretic transform (NTT) [37] is essentially a specialization of the discrete Fourier transform. Mapping these algorithms efficiently onto a quantum topology requires an analysis of the algorithm and why certain heuristics make it run quickly on a classical machine. A careful translation can then be done to port various breakdown rules and mapping strategies into the quantum domain. In this thesis we made an initial effort for the QFT by selecting an intermediate representation of the connectivity map (i.e. a hypercube) and lowering this idealized geometry onto a real architecture. We believe this approach warrants further study; other intermediate mappings besides the hypercube (e.g. butterflies, bus networks or trees) can be explored to address the inadequacies of our heuristics, and this general approach can be expanded to other transforms. Additional graph-theoretic methods of lowering these intermediate mappings onto real architectures, perhaps in several different stages, could also prove fruitful. Since many linear transforms share similar dataflow patterns, we expect that expanding this library of data-movement heuristics will be a valuable effort, especially since QSPIRAL should be able to recognize and reuse the heuristics appropriately. We believe that the formaliza-

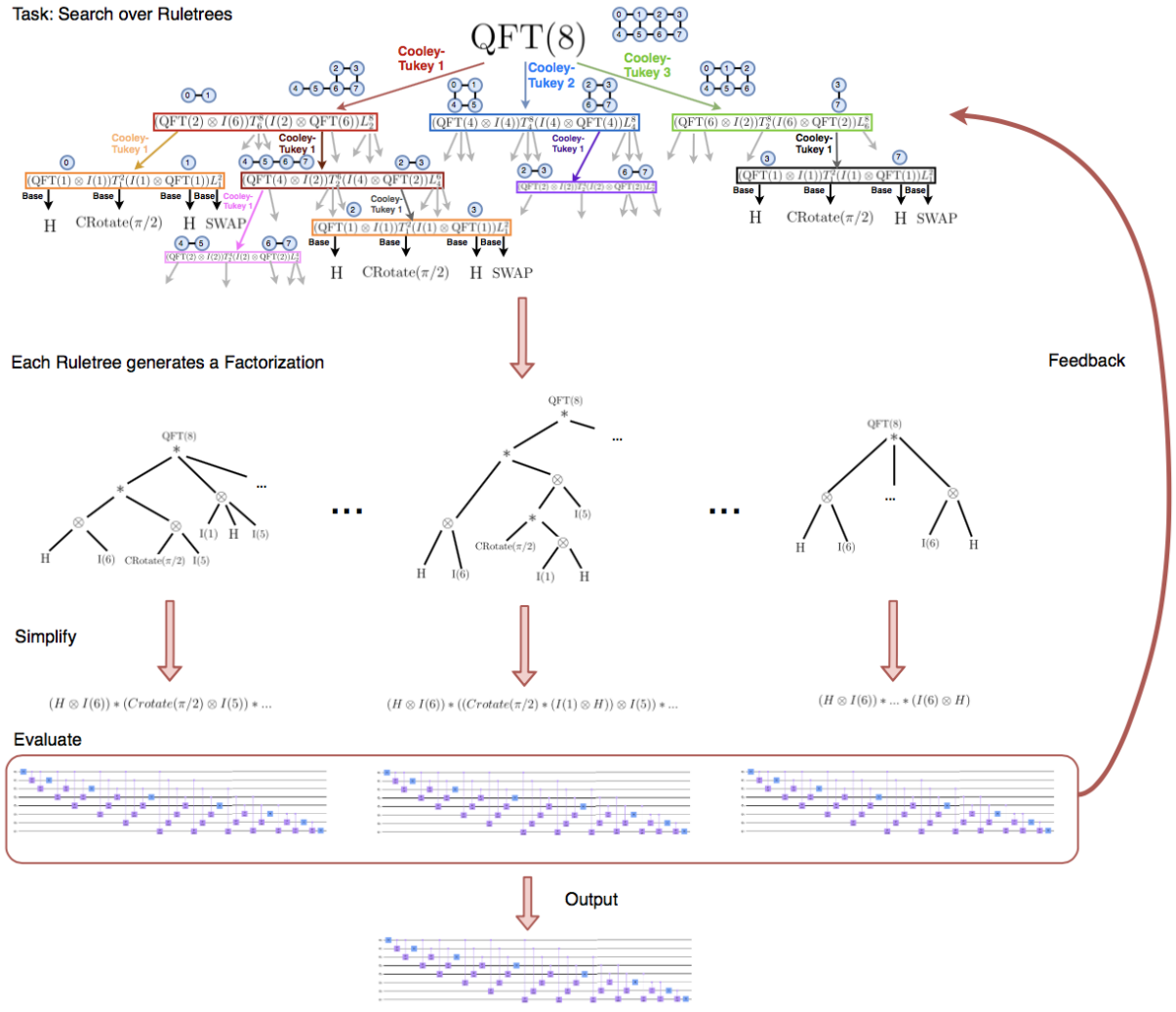


Figure 6.1: Full overview of quantum circuit generation procedure.

tion presented in this thesis does much to expose the quantum optimization problem to outside influence from other research domains such as parallel computing or graph/network theory, and that the approach presented could have massive untapped potential.

Additionally, since the current implementation of QSPIRAL is still inefficient in several respects, further profiling can be done to detect and address computational bottlenecks. Specifically, QSPIRAL could take advantage of further parallelism in order to run more efficiently on modern hardware. The unfortunate complexity problem detailed in Section 5.1.2 should also be addressed with further system optimizations.

Besides these expansions or refinements of the QSPIRAL framework, there exist a few additional areas in which this type of system could be applicable.

6.2.1 Proof of correctness

Proving the correctness of quantum circuits is a difficult task. The most exhaustive check of circuit correctness requires validating each entry in the $2^N \times 2^N$ unitary matrix that the circuit implements. However, expanding this matrix rapidly becomes infeasible as N increases. For values of N that are too large, circuit correctness can be verified experimentally by simulating the results repeatedly against a "known good" implementation. However, since measurement is a stochastic operator, this approach only guarantees with a high likelihood that the circuit is correct. Additionally, simulating large circuits also rapidly becomes untenable with increasing N. Besides this, simulation is too computationally expensive to include as an part of the compiler, which means that it becomes the added responsibility of the user to verify compilation output; this introduces an unwieldy and undesirable human dependency. Many of these verification challenges were encountered when developing the work shown in this thesis, particularly in validating the compilation results for larger circuits.

QSPIRAL promises a more effective way to verify circuit correctness. QSPIRAL starts with a high-level algorithm as input, which trivially implements itself correctly. Since QSPIRAL simply breaks down this large algorithm through divide-and-conquer decomposition rules, it should be possible to verify the correctness of each rule; therefore we would be guaranteed to have correct output.

Verifying circuit equivalence is a more difficult task, since it is nearly impossible to reconstruct the higher-level specification from the sparse decomposition. However, if both circuits were generated using QSPIRAL, it is conceivable that QSPIRAL could emit, alongside the generated code, a "proof" of the code in the form of the breakdown rule tree used to construct it. Given two such proofs, checking equivalence would only require verifying that the roots of these breakdown trees are the same, and that both are valid rule trees. The general idea of providing a proof alongside a compiled program is directly inspired by similar work in the area of computer security [49], albeit used here for a drastically different purpose.

6.2.2 Error-correcting Codes

While beyond the immediate scope of this work, there is currently a large effort underway to replace the NISQ quantum devices explored in this thesis with those supporting large numbers of

error-corrected qubits. Conceivably, error-corrected qubits would massively alleviate the concern of degrading quantum states.

Error-correction has been theoretically achieved by mapping one logical qubit onto several physical qubits [60]. While duplicating and polling the state in a manner similar to triple modular redundancy (TMR) techniques [64] is impossible due to the no-cloning theorem, it is not impossible to *spread* a single state across multiple physical qubits; by splitting one state across several real qubits, we can perform measurements and operations between these qubits to detect and correct errors as the circuit progresses, employing schemes inspired by more traditional error-correcting codes [32]. There is significant ongoing research into determining optimal codes and the quantum architectures that support them [23].

Reconciling this research with the work contained in this thesis, we believe there could be a role for SPIRAL to play in compiling circuits in this future era. First, by providing the chosen error-correcting code as input to QSPIRAL, we should be able to take this into account in the cost function and still be able to determine the best logical circuit for the specified hardware device. Alternatively, similarly to how we search over logical-to-physical mappings, QSPIRAL could conceivably assist in the generation of these codes and construct them alongside the program. We could possibly form them specifically for the desired architecture and algorithm pairing by searching over an additional layer of indirection.

6.3 Closing Remarks

The approach presented in this thesis is a novel approach, and to our knowledge this effort has not been duplicated elsewhere, excepting our previous publication [47]. The primary differentiation between our work and existing frameworks can be described as follows.

- **Approach**. Many existing quantum toolkits [1] take a program stream as input and perform lookahead searches over QASM instructions to minimally satisfy connectivity constraints. QSPIRAL takes an algorithm specification as input and generates a program implementing that algorithm. QSPIRAL is constrained by *what* transform a user specified but not *how* it was implemented; this mirrors the dichotomy between declarative and imperative programming languages [33]. QSPIRAL can thus leverage algorithmic information unavailable to other systems to inform heuristics and decomposition decisions; to other toolkits, a QFT might simply be an arbitrary string of gates.
- **Efficiency.** As circuit sizes increase, structured approaches will most likely be necessary in order to generate reasonable quantum programs. Peephole searches on large and sparse architectures will most likely perform worse than approaches that can recognize the architecture and apply known data-movement paradigms.
- **Accessibility.** QSPIRAL frames the circuit generation problem in terms of traditional computer science and linear algebra. This, in our opinion, is much more accessible to non-physicists. Additionally, differing syntax and representation is a barrier preventing many computing innovations from being transferred from classical to quantum computers.
- **Platform**. SPIRAL is a popular platform for generating optimized linear algebra libraries. Especially since so many of its systems directly translate to being useful in the quantum

domain, the framework itself is more accessible to signal processing experts and computer scientists who are likely to make major breakthroughs in this space. Additionally, SPIRAL implements *all* stages of the proposed system within the same framework, meaning the various stages of the compiler can be integrated to enable more intelligent search.

It is our hope that approaches inspired by or similar to the one described in this thesis could help create quantum compilers that are up to the task of generating efficient code for large devices when the need arises.

Bibliography

[1] H. Abraham et al., "Qiskit: An Open-source Framework for Quantum Computing," 2019. doi:10.5281/zenodo.2562110 1.1.2, 6.3

[2] J. Andreoli, "Logic programming with focusing proofs in linear logic," *Journal of Logic and Computation*, 1992. 3.3.6

[3] F. Arute et al., "Quantum Supremacy using a Programmable Superconducting Processor," *Nature*, vol. 574 (2019), pp. 505–510. 1.1.1, 2.1, 4.4

[4] J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proceedings of the International Conference on Information Processing*, UNESCO, 1959, pp.125-132. (document), 2.1

[5] J. S. Bell, "On the Einstein Podolsky Rosen Paradox," *Physics* vol. 1, no. 3, 1964, pp. 195-290. 2.2.2

[6] R. Bellman, *Dynamic programming*. Princeton, N.J: Princeton University Press, 2010. 1.2

[7] P. Benioff. "The computer as a physical system: A microscopic quantum mechanical hamiltonian model of computers as represented by turing machines," *Journal of Statistical Physics*, May 1980. 2.1

[8] E. Bernstein and U. Vazirani, "Quantum Complexity Theory," *SIAM J. Comput.*, 1997, pp. 1411–1473. First appeared in ACM STOC 1993. doi:10.1137/S0097539796300921. (document), 1.1, 4.1

[9] A.W. Burks, H.H. Goldstine and J. von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument (1946)," in *Perspectives on the computer revolution*, Z. W. Pylyshyn and L. J. Bannon, Ed. Norwood, NJ, United States: Ablex Publishing Corp., 1989, pp. 39-48. 2.1

[10] R. Ceselin, (28 August 2020), *Google's Sycamore processor mounted in a cryostat, recently used to demonstrate quantum supremacy and the largest quantum chemistry simulation on a quantum computer*[Online image], Phys.org, https://phys.org/news/2020-08-google-largest-chemical-simulation-quantum.html (document), 2.1

[11] A. Church, "An Unsolvable Problem of Elementary Number Theory," *American Journal of Mathematics*, vol. 58, no. 2, April 1936, pp. 345-363. 2.3

[12] A. Colmerauer and P. Roussel, "The birth of Prolog," in *Conference on the History of Programming Languages (HOPL-II), Cambridge, MA, USA, April 1993*, pp. 37–52. 3.2.2

[13] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex

Fourier Series," *Mathematics of Computation*, 1965, pp. 297-301. doi:10.1090/S0025-5718-1965-0178586-1. 4.2

[14] A. W. Cross, A. Javadi-Abhari, T. Alexander, N. de Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, J. Smolin, J. M. Gambetta and B. R. Johnson "OpenQASM 3: A broader and deeper quantum assembly language" [arxiv:2104.14722]. 2.1

[15] P. A. M. Dirac, "A new notation for quantum mechanics," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, issue 3, July 1939, pp. 416-418. doi:10.1017/S0305004100021162 2.2.1

[16] D. P. DiVincenzo, "Topics in quantum computers," arXiv preprint cond-mat/9612126 2.1

[17] L. Egan et al., "Fault-Tolerant Operation of a Quantum Error-Correction Code," 2021, arXiv:2009.11482 [quant-ph]. 1.1.1

[18] S. Egner, J. Johnson, D. Padua, M. Püschel and J. Xiong, "Automatic derivation and implementation of signal processing algorithms," *ACM SIGSAN Bulletin*, 2001. doi:10.1145/511988.511990 1.1.3

[19] S. Egner and M. Püschel, "Solving Puzzles related to Permutation Groups," *Proc. ISSAC*, 1998, pp. 186-193. 3.3.3

[20] A. Einstein, B. Podolsky and N. Rosen, "Can Quantum-Mechanical Description of Physical Reality be Considered Complete?" *Physical Review*, vol. 47, May 1935, pp. 777-780. 2.2.2

[21] S. C. Eisenstat, M. C. Gursky, M. H. Schultz and A. H. Sherman, "Yale Sparse Matrix Package," Yale University, New Haven, CT, 1997 3.3.1

[22] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, June 1982. 2.1

[23] A. G. Fowler et al., "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, 2012. doi:10.1103/PhysRevA.86.032324 6.2.2

[24] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe and J. M. F. Moura, "SPIRAL: Extreme Performance Portability," *Proceedings of the IEEE*, vol. 106, no. 11, 2018. (document), 1.2

[25] F. Franchetti and M. Püschel, "Fast Fourier Transform," in *Encyclopedia of Parallel Computing*, D. A. Padua (Editor). (document), 2.3.1, 4.3, 4.3

[26] F. Franchetti and M. Püschel et al. "Spiral," in *Encyclopedia of Parallel Computing*, 2011, D. A. Padua (Editor). (document), 3.2

[27] F. Franchetti, Y. Voronenko and M. Püschel, "A Rewriting System for the Vectorization of Signal Transforms," *Proceedings High Performance Computing for Computational Science (VECPAR) 2006*, LNCS 4395, pp. 363-377. 3.2.3

[28] C. Gentry, "Fully homomorphic encryption using ideal lattices," *Symposium on the Theory of Computing (STOC)*, 2009, pp. 169-178. 6.2

[29] G. Gentzen, "Untersuchungen über das logische Schließen," *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pp. 68–131, North-Holland, 1969. 2.4.2, 3.3.6

[30] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *28th Annual ACM Symposium on the Theory of Computing (STOC)*, pp. 212-219, 1996. arXiv:quant-ph/9605043 1.1

[31] G. G. Guerreschi, "Scheduler of quantum circuits based on dynamical pattern improvement and its application to hardware design," 2019, arXiv:1912.00035v1 [quant-ph]. 1.1.1

[32] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, 1950, pp. 147-160. 6.2.2

[33] R. Harper, "There Is Such A Thing As A Declarative Language, and It's The World's Best DSL," *Existential Type*. Updated July 22, 2013. [Blog]. Available:https://existentialtype.wordpress.com/2013/07/22/there-is-such-a-thing-as-a-declarative-language/, Accessed on: July 5, 2021. 6.3

[34] J. D. Hidary, *Quantum Computing: And Applied Approach*, Gewerbestrasse 11, 6330 Cham, Switzerland: Springer Nature Switzerland AG, 2019. 2.1

[35] K. Hietala et al., "A verified optimizer for Quantum circuits," *Proceedings of the ACM on Programming Languages*, vol. 5, January 2021, article 29, pp 1-29. doi:10.1145/3434318 5.1

[36] D. W. Hillis, "Nuts and Bolts," In *The Pattern On The Stone: The Simple Ideas That Make Computers Work*. New York, NY: Basic Books, 1998, ch. 1. pp. 1-4. 2.1

[37] S. Kim, W. Jung, J. Park and J. Ahn, "Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs," In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. Los Alamitos, CA, USA:IEEE Computer Society, October 2020, pp. 264-275. 6.2

[38] A. Yu. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, issue 1, January 2003, pp. 2-30. doi:10.1016/S0003-4916(02)00018-0 4.4

[39] H. T. Kung, "Why Systolic Architectures?" *Computer*, vol. 15, issue , 1982, pp. 37-26. doi:10.1109/MC.1982.1653825 2.1

[40] M. S. Lam, *A Systolic Array Optimizing Compiler*, New York, NY: Springer US, 1989. 2.1

[41] Y. Manin, "Computable and Non-Computable (in Russian)," Sovetskoye Radio, Moscow, 1980. 2.1

[42] P. Martin-Löf, "On the meanings of the logical constants and the justifications of the logical laws," Notes for three lectures given in Siena, Italy, 1996, Published in *Nordic Journal of Philosophical Logic*, April 1983. 3.3.6

[43] S. McArdle et al., "Quantum computational chemistry," *Rev. Mod. Phys.*, vol 92, issue 1, March 2020, doi:10.1103/RevModPhys.92.015003. 4.1

[44] D. McClure and J. Gambetta, "Quantum computation center opens." *IBM Research Blog*. Updated September 18, 2019. [Blog]. Available:https://www.ibm.com/blogs/research/2019/09/quantum-computation-center/, Accessed on: June 30, 2021. (document), 2.5, 5.7

[45] S. McLaughlin and F. Pfenning, "Imogen: Focusing the polarized inverse method for in-

tuitionistic propositional logic," I.Cervesato, H.Veith and A.Voronkov (Editors), in *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08), Doha, Qatar, November 2008*, Springer LNCS 5330. pp. 174–181. 3.3.6

[46] P. A. Milder, F. Franchetti, J. C. Hoe and M. Püschel, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Transactions on Design Automation of Electronic Systems*, Article 15, 2012. 3.2

[47] S. Mionis, F. Franchetti and J. Larkin, "Quantum Circuit Optimization with SPIRAL: A First Look," *Supercomputing (SC) 2020*. 6.3

[48] G. J. Mooney, G. A. L. White, C. D. Hill and L. C. L. Hollenberg, "Whole-device entanglement in a 65-qubit superconducting quantum computer," 2021, arXiv:2102.11521 [quant-ph]. 1.1.1

[49] G. C. Necula, "Proof-carrying code," *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Jan 1997, pp 106-119. doi:10.1145/263699.263712 6.2.1

[50] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge, United Kingdom: Cambridge University Press, 2010. 2.1, 2.3

[51] R. O'Donnell, Class Lecture, Topic: "Multi-Qubit Systems." 15-859BB, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September, 2018. 2.2.2

[52] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing, 3rd edition*, Upper Saddle River, NJ 07458: Pearson Higher Education Inc., 1989, pp. 723-741 (document), 1.1.3, 4.1, 4.2

[53] R. Prabhakar et al., "Plasticine: A Reconfigurable Architecture For Parallel Patterns," *ACM SIGARCH Computer Architecture News*, vol. 45, issue 2, May 2017, pp. 389–402. doi:10.1145/3140659.3080256 2.1

[54] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson and 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, pp. 232-275. (document)

[55] E. Reiffel and W. Polak, *Quantum Computing: A Gentle Introduction*. Cambridge, MA: MIT Press, 2011. (document)

[56] R.L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, issue 2, Feb 1978. doi:10.1145/359340.359342 (document), 4.1

[57] 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. (document), 1.2, 3.2

[58] E. Schrödinger, "An undulatory theory of the mechanics of atoms and molecules," *The Physical Review*, vol. 28, no. 6, December 1926. 2.1

[59] P. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994. doi:10.1137/S0097539795293172 1.1, 2.1, 4, 4.1

[60] P. Shor, "Scheme for reducing decoherence in quantum computer memory," *Physical Review A*, vol. 54, issue 4, 1995. doi:10.1103/PhysRevA.52.R2493 6.2.2

[61] D. R. Simon, "On the power of quantum computation," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, SFCS 1994, Washington, DC, USA, 1994,* IEEE Computer Society, pp. 116-124. 4.1

[62] B. Tan and J. Cong, "Optimality Study of Existing Quantum Computing Layout Synthesis Tools," *ICCAD '20: Proceedings of the 39th International Conference on Computer-Aided Design*, November 2020, article 137, pp. 1-9. doi:10.1145/3400302.3415620 5.1

[63] A. M. Turing, "On Computable Numbers, With An Application To The Entscheidungsproblem," 1936. 1.1.3

[64] J. von Neumann, "PROBABILISTIC LOGICS AND THE SYNTHESIS OF RELIABLE ORGANISMS FROM UNRELIABLE COMPONENTS," lectures delivered at the California Institute of Technology, January 4-15, 1952. Notes by R. S. Pierce, Caltech Eng. Library, QA.267.V6 6.2.2

[65] N. Wirth, "The Programming Language Pascal," *Acta Informatica*, vol. 1, 1971, pp. 35-63. 3.2.1

[66] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299 (1982), pp. 802-803. doi:10.1038/299802a0 2.3.2

[67] J. Xiong, J. Johnson, R. Johnson and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 298–308, 2001. 3.2.2