Compiler Management of Communication and Parallelism for Quantum Computation

Jeff Heckey
UC Santa Barbara
jheckey@ece.ucsb.edu

Shruti Patil
Princeton University
spatil@princeton.edu

Ali JavadiAbhari
Princeton University
ajavadia@princeton.edu

Adam Holmes
Cornell University
ah744@cornell.edu

Daniel Kudrow
UC Santa Barbara
dkudrow@cs.ucsb.edu

Kenneth R. Brown
Georgia Inst. of Technology
ken.brown@chemistry.gatech.edu

Diana Franklin
UC Santa Barbara
franklin@cs.ucsb.edu

Frederic T. Chong
UC Santa Barbara
chong@cs.ucsb.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

Abstract

Quantum computing (QC) offers huge promise to accelerate a range of computationally intensive benchmarks. Quantum computing is limited, however, by the challenges of decoherence: i.e., a quantum state can only be maintained for short windows of time before it decoheres. While quantum error correction codes can protect against decoherence, fast execution time is the best defense against decoherence, so efficient architectures and effective scheduling algorithms are necessary. This paper proposes the Multi-SIMD QC architecture and then proposes and evaluates effective schedulers to map benchmark descriptions onto Multi-SIMD architectures. The *Multi-SIMD* model consists of a small number of SIMD regions, each of which may support operations on up to thousands of qubits per cycle.

Efficient Multi-SIMD operation requires efficient scheduling. This work develops schedulers to reduce communication requirements of qubits between operating regions, while also improving parallelism.We find that communication to global memory is a dominant cost in QC. We also note that many quantum benchmarks have long serial operation paths (although each operation may be data parallel). To exploit this characteristic, we introduce Longest-Path-First Scheduling (LPFS) which pins operations to SIMD regions to keep data in-place and reduce communication to memory. The use of small, local scratchpad memories also further reduces communication. Our results show a 3% to 308% improvement for LPFS over conventional scheduling algorithms, and an additional 3% to 64% improvement using scratchpad memories. Our work is the most comprehensive software-to-quantum toolflow published to date, with efficient and practical scheduling techniques that reduce communication and increase parallelism for full-scale quantum code executing up to a trillion quantum gate operations.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03$15.00...$15.00.
http://dx.doi.org/10.1145/2694344.2694357

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles – cached memories, shared memory; D.2.8 [*Software Engineering*]: Metrics – performance metrics; D.3.2 [*Programming Languages*]: Language Classifications – design languages; D.3.4 [*Programming Languages*]: Processors – compilers, optimization, memory management

Keywords Quantum, LLVM, Compilers, Scheduling

1. Introduction

Quantum computing (QC) is a powerful paradigm with a potential for massive computational speedup. QC-based algorithms, in contrast to classical algorithms, traverse large solution spaces exponentially fast by manipulating *qubits*, which are superpositions of 0 and 1 states, instead of individual bits. Taking advantage of this, QC algorithms have been devised for important scientific problems that have high computational complexity. For example, recent work required 8400 MIPS-years to factor RSA-155 on a classical machine [4], while Shor's QC factoring algorithm offers the potential for a significant speedup on large problems due to a reduction in asymptotic complexity.

Although the QC algorithms we use as benchmarks offer dramatic reductions in computational complexity, it is also important to optimize their actual runtimes. Quantum bits (qubits) are highly error-prone, susceptible to decoherence over time, and require expensive Quantum Error Correcting Code (QECC) operations for reliable computation. Accelerating quantum computation is important because it allows the computation to "stay ahead of the errors".

In this paper, we propose the Multi-SIMD(k,d) architecture for quantum computation. Multi-SIMD approaches combine operation-level parallelism (with k different gates or operating regions) along with data-level parallelism (up to d qubits can be operated on within a single gate). We show that QC machines of this type are well-suited to the parallelism and computational needs of large-scale

QC benchmarks. In addition, we explore parallelism and locality scheduling techniques as a means to optimize the performance of quantum computations onto Multi-SIMD hardware.

A natural source of state degradation in this architecture is communication delay, due to the fact that qubits need to be moved between regions in order to undergo an operation or to interact with other qubits (to support data and instruction parallelism). QECC requires many *physical* qubits to represent a single *logical* qubit. This makes compute regions and memories large in area, which results in large communication distances. When we add support for operation- and data-level parallelism, distances become even larger.

These high communication costs push us towards a communication-centric scheduling model. Exploiting the "mostly-serial" nature (at the operation level) of many of our applications, we adopt a Longest-Path-First scheduling (LPFS) strategy that keeps operands in-place for computations. Small scratchpad memories can further avoid global communication by capturing some temporal locality. In fact, the scratchpad memories are critical to leveraging the locality generated by LPFS, since it is often the case that one operand stays in a region for the next operation, but other operands must temporarily be moved aside. To maintain scheduling quality while reducing analysis times, we have developed a hierarchical approach that performs fine-grained scheduling in leaf modules with higher-level coarse-grained scheduling to stitch them together. Our results show an increase in speedup when incorporating communication from 3% to 308% and when incorporating local memories from 3% to 64%. With both modifications, total speedups can reach 9.83X.

This paper makes the following contributions:

- To exploit parallelism while mitigating control complexity, we propose a novel Multi-SIMD(k,d) quantum architectural model. Ion trap technology with microwave control provides a practical basis for our model.
- We propose a hierarchical scheme to apply the scheduling algorithms to large-scale QC benchmarks. Through remodularization and application of the algorithms at coarse-grained and fine-grained levels, we show that our algorithms can effectively analyze large-scale QC benchmarks and generate schedules that are comparable to their estimated critical path lengths.
- We propose and evaluate communication-aware scheduling algorithms that reduce communication overheads while maximizing parallelism. We compare a traditional scheduling algorithm to the designed algorithm that considers the structure of quantum programs, and evaluate the speedup of both algorithms over sequential execution and naive communication models.
- We analyze the benefits of scheduling for local, scratchpad memories next to each compute region in reducing long communication delays to the global memory and overall execution speedup.

The rest of the paper is organized as follows: Section 2 discusses the technology and architectural model proposed. Section 3 discusses the design of the ScaffCC compiler and the program execution model. Section 4 discusses the scheduling algorithms in depth. Section 5 discusses the schedules found. Sections 6 and 7 discuss related work, future work, and conclusions of the paper.

2. Architectural Model

We propose Multi-SIMD(k,d), a family of quantum architectures motivated by practical constraints in *ion trap* technology. Multi-SIMD architectures support parallelism at both the operation and data levels. In this section, we describe the concept, feasibility and challenges of trapped ion quantum computation. Our proposed architectural model and the choice of its parameters are based on

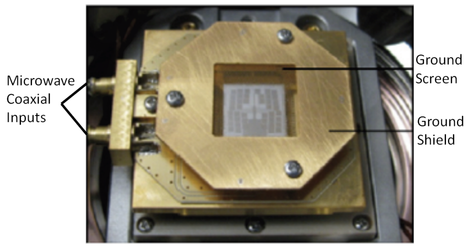

Figure 1: Microwave control for Ion Traps. Note the large off-chip coaxial connectors for the microwave signals. Figure is reproduced from [40] with permission.

ion trap QC, but are applicable to other technologies as discussed later.

2.1 Background on Quantum Computation

This section briefly describes background on aspects of QC most relevant to the work described here. In particular, one key constraint in quantum computation is the *no cloning theorem* [33], which states that a superposition state cannot be perfectly copied from one qubit to another qubit, without destroying the first one. Thus, in order to use the state of a qubit in another part of the architecture, that qubit must be physically moved. This can be costly on computation time, therefore in Section 2.3 we describe the use of a technique called *quantum teleportation*, which allows these movements to be hidden by pipelining them into the computation.

Another consequence of the no-cloning theorem, and also the fact that quantum operations have to be norm-preserving unitaries, is that these operations cannot have fan-in or fan-out, and all quantum computations must be reversible. This general property of quantum circuits means that normal, irreversible classical circuits cannot be directly written as quantum circuits, and we have to equip the compiler with a feature to convert classical operations to quantum ones. This is described in Section 3.1.

2.2 Ion Trap Technology

A trapped-ion system is currently the most promising candidate for realizing large-scale quantum computers. These include well characterized qubits, long decoherence, and the ability to initialize, measure and perform a universal set of gates [9, 27, 31, 38]. Experimental results with ion traps have demonstrated all the requirements of a viable quantum technology for realistic quantum computation [3, 6, 13, 37, 50], as laid down in the *DiVincenzo criteria* [12].

In trapped-ion systems, operations are typically performed by focusing lasers on each ion, which represents a physical qubit. Because these lasers must have precisely tuned relative phase, they can be extremely large in size, currently the size of a small room, limiting current visions for QCs to roughly 6-12 lasers.

However, recent experiments with microwaves demonstrated multi-qubit control that could scale from 10 to 100 ions for a single microwave emitter [40], like the example hardware shown in Fig. 1. Another level of fanout could create a small number of SIMD regions each capable of performing a different operation ("gate") on roughly 100 to 10000 qubits. Microwaves are a natural choice for driving single qubit gates and coupled with gradients they can also drive two-qubit gates [22, 36]. The signals for different regions can be tuned to cancel microwaves for some regions, effectively shielding some ions from neighboring signals [11].

Microwave control does not currently allow for qubit measurement, although schemes are under development. For this work, we assume laser-controlled measurement, which has better fanout

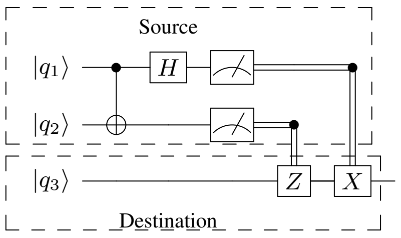

Figure 2: Communicating the state of q_1 using quantum teleportation: The EPR pair of q_2/q_3 is distributed prior to teleportaion, keeping q_2 near the communication source q_1, and using q_3 as the communication destination. By measuring the states of q_1 and q_2, and classically transmitting those measurement results (classical 0 and 1 bits), the state of q_3 takes on that of q_1 using none, one, or both of the X and Z operations at the target side. This completes the transmission of q_1 to q_3, while the state of q_1 is destroyed in the process.

characteristics than operation gates. This is because measurement requires only a single laser and the phase between multiple ions is not important.

All our analyses are performed at the logical level and ignore the lowest-level physical operations. The logical gate operations are actually sequences of microwave signals that operate on a group of physical qubits to produce a single logical qubit state. The encoding we assume is some form of concatenated code for QECC [17, 42, 43]. As the need for better reliability increases some of these codes can have an exponential overhead costs and resulting runtime increases. By keeping the runtimes below these boundaries, we can avoid these expenses.

2.3 Handling Data Movement and Teleportation

At a physical level, communication in the Multi-SIMD architecture is assumed to be achieved through *quantum teleportation* (QT), a phenomenon that makes transmission of exact qubit states possible. QT requires a pre-distribution of entangled Einstein-Podolsky-Rosen (EPR) pairs of qubits between the regions where communication will occur. EPR pairs are generated at the global memory, and distributed to the required regions. Fig. 2 illustrates the computations required. The communication cost is four times as high as a single quantum gate per move; in a naive movement model, this quintuples the actual compute cost because of repeatedly moving qubits between SIMD regions and the global memory. In many cases our compiler can schedule QT operations in parallel with the computation steps. It should be noted that in the context of our logical compiler, we treat all logical gates as having the same latency, which effectively means we assume the system will be clocked at the rate of the longest gate. More precise latencies can be found in [18]. For example, 2-qubit gates may physically take up to 10 times the latency of 1-qubits gates. Moreover, by choosing QT as the method of communication, we mask the latency of moving qubits around. This masking is possible by *pre-distribution* of these pairs before they are needed for the teleportation.

For a teleportation operation to occur, EPR pairs must be distributed to each SIMD region and global memory so that the sender and receiver each have one half of the pair. The distribution of such EPR pairs has been studied in detail in [49]. Since repeated movement of qubits on the physical fabric is error-prone, QT reduces quantum decoherence by communicating information through a classical channel. Our compiler schedules the pre-distribution of EPR pairs statically, as with other parts of the overall schedule. While QT has constant latency with communication distance, longer distances do imply higher EPR bandwidth requirements (larger communication channels to move enough EPR pairs throughout the architecture). To minimize EPR bandwidth requirements, future work will investigate distributed global memory and compiler algorithms for mapping to such a non-uniform memory architecture. Additionally, the issue of correctly dealing with the problem of decoherence at the physical level when moving EPR pairs, which may arise in long communication channels, is studied in [10].

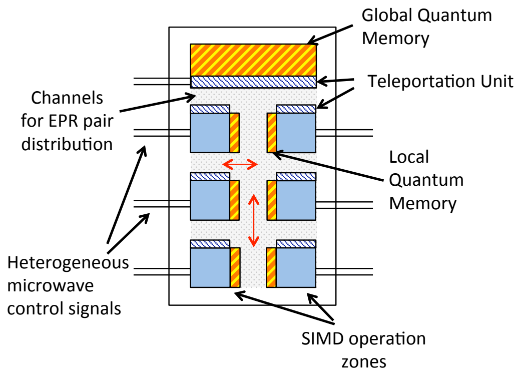

Figure 3: Block diagram of Multi-SIMD quantum architecture based on ion traps controlled by microwave technology. k operating regions each support quantum operations on d qubits simultaneously. Each operating region has a local memory for storing qubits. EPR qubit pairs are prepared near the global memory and distributed among regions via quantum teleportation.

2.4 The Multi-SIMD Architectural Model

Based on the ion trap technology with microwave signaling mechanism, we propose to build a scalable system with multiple independent operation regions, each offering gate-level parallelism. Fig. 3 shows the block diagram for our proposed architecture. Within each SIMD region, a large amount of data-level parallelism is available—100 - 10,000 qubits. In this manner, the architecture can support k independent regions each allowing d bits of SIMD computation, where k is limited by the number of microwave signals and the complexity of generating interference to shield neighboring regions. The value of d is limited by the ability to fan out the microwaves signal to all d qubits without disrupting the rest of the system.

These k SIMD regions serve two purposes. First, they allow active computation on qubits. Second, when idle, they can be used as passive short-term storage for qubits between compute operations. We also optionally provision each SIMD region with a small amount of scratchpad memory (or local memory) for temporary storage of qubits during intermediate computations on other qubits in the same region, similar to a traditional hierarchical memory system. During a scheduled run of the benchmark, active qubits must be moved between SIMD regions and idle qubits must be moved to the global memory. Storing some qubits in the local memories can potentially reduce overall communication cost to and from the global memory.

Note that the centralized global memory is a simplification that results from QT which makes the latency of long-distance communication constant. This constant global communication cost favors parallelism, since the serial version of our computations pay just as much to access global memory as our parallel versions do. In fact, parallelism breaks the computation into finer-grain chunks on separate regions. This reduces global memory accesses by leveraging local storage in regions and optional local memories, which is analogous to spatial computing approaches in classical computing [14, 45, 46].

Overall, efficient use of Multi-SIMD requires orchestration of qubits to maximize parallelism and to reduce qubit motion. Devising and evaluating effective scheduling techniques for Multi-SIMD is the focus of this paper.

2.5 Optimizing Communication using Local Memory

The communication overhead of four cycles per qubit movement in QT can be reduced even further by adding a local memory storage to each SIMD region. A local memory region bordering an operating region acts as a temporary store for a qubit that must be moved out of the SIMD region for one or more cycles, and moved back into the same region for the next computation on it. Due to the proximity of the local memory region, qubits can be moved back and forth physically using ballistic control. This eliminates the need for computational cycles required for teleportation to global memory reducing the communication overhead considerably. While the latency of ballistic transport of a qubit depends on the size of the operating region (or the distance to be traversed), note that even teleportation operation involves a similar ballistic latency for accomplishing the interaction of data qubits with an EPR qubit. In our experiments, we assume a one-cycle latency overhead for local communication.

2.6 Adaptability for Other Technologies

We use trapped-ion technology as a motivating example since its scaling properties and parameters are well understood. Our toolflow and the Multi-SIMD model, however, are not tied to any particular physical implementation and can be applied so long as the scaling properties hold. For example, superconducting qubits could be electronically tuned in and out of resonance in parallel, allowing SIMD operation in regions. Recent work with quantum dots using rotating frames and microwaves also could support Multi-SIMD in a manner very similar to trapped ions. A global operation can also be applied to a lattice of neutral atoms. Further, work by [5] utilized another SIMD approach based on electron spins on helium technology.

3. Computation and Compilation Framework

Our evaluation framework consists of a set of large-scale quantum benchmarks and a LLVM-based compiler toolchain designed to perform deep analysis at scale. Since large-scale quantum benchmarks can not be simulated on any classical computer, our compiler is also designed to estimate the performance and resource requirements of the code we generate.

3.1 ScaffCC Compiler Framework

Our scheduler implementation is built within ScaffCC, a compiler framework for synthesis of logical quantum circuits from a high-level language description [21]. As input, ScaffCC takes programs written in Scaffold, a C-like programming language for quantum computing [20]. Scaffold includes data types to distinguished quantum and classical bits and offers built-in functions for quantum gates. It also offers additional constructs for parallelism and control, such as parallel *for* loops and conditional statements based on the (probabilistic) state of qubits.

ScaffCC operates largely at the logical level, translating high-level quantum algorithms (benchmarks) described in Scaffold into the LLVM [26] intermediate representation and ultimately back-end-targeting QASM which is a technology-independent quantum assembly language [33, 44]. The basic data types in QASM are qubits and cbits, and the instruction set includes a universal set of gates (X, Y, and Z), and operations for measurement, as well as preparation in the states of $|0\rangle$ and $|1\rangle$. Quantum gates of Controlled NOT (CNOT), Hadamard (H) and Phase (S), referred to as the *Clifford* group of gates, along with the $\pi/8$ phase (T) gate form a minimum set for universal quantum computing. These logical gates are assumed to incorporate QECC sub-operations; no attempt in this paper is made to optimally schedule these sub-operations.

Many classical operations, such as $a + b = c$, cannot be expressed traditionally in reversible logic. To allow programmers to work in more familiar ways, we incorpoarate a tool called *Classical-To-Quantum-Gates* (CTQG) [21] into the compiler flow. This tool decomposes certain common programming syntax, such as arithmetic operations and if-then-else control structures, into QASM operations suitable for linking into the more quantum code.

The set of gates allowed in QASM is a subset of the full vocabulary allowed in Scaffold. To target just this subset (akin to an instruction set architecture in classical processors) the compiler uses a decomposition stage to translate arbitrary-angle *Rotation* gates and *Toffoli* gates into either approximate or precise equivalents. For Rotations, this is an approximate process implemented through the SQCT [25] toolbox. We perform our compiler analysis and scheduling on the quantum program after gate decomposition has been applied.

To incorporate parallelism/communication analysis and scheduling, we extend the ScaffCC compiler framework (based on the LLVM intermediate format) with new analysis passes. The current state-of-the-art in quantum programming is that nearly all of the program control flow, iteration counts, and gate operations can be determined classically, before program run time. As a result of this deeply-analyzable program control flow, most work on synthesis and optimization of quantum circuits has assumed a completely flat and unrolled circuit format [8, 29]. While *full* linearization was viable for early small QC benchmarks, quantum benchmarks have now progressed to the point where the benchmarks we study (Section 3.3) require $10^7$to 10^{12} operations or more, and thus this full unrolling is no longer tractable. We therefore incorporate a hierarchical scheduling technique, dividing the benchmark up along module boundaries into *leaf* and *non-leaf* modules. Leaf modules make no calls to other modules and are composed solely of QC primitive gates; non-leaves may use a mix of gates and submodule calls. Fine grained scheduling of the leaves are discussed in Sections 4.1 and 4.2. Coarse-grained parallel scheduling of the non-leaves is discussed in Section 4.3.

3.1.1 Leaf Module Flattening

Our hierarchical process allows scheduling of very large codes, but this comes at the cost of reduced parallelism being detected by the scheduler. Larger leaf modules provide more opportunities for good fine-grained scheduling, but when leaf modules are too large the scheduling time becomes unacceptably long. Here we look at how to flatten non-leaves into larger leaf modules optimally.

Fig. 4 illustrates a simple example of possible parallelism loss at module boundaries. Here, a program to be scheduled on a Multi-SIMD$(4, \infty)$ system, a Toffoli operation is followed by a dependent Toffoli operation i. *Toffoli(x,y,z)* is a three-qubit operation that flips the state of qubit z depending on states of x and y. Since qubits are subject to the no-cloning theorem any common operand in two operations presents a data dependency in quantum programs. In this case, the two gates share the input *a* and therefore the second is dependent on the first.

In the example, the Toffoli operation is decomposed into a circuit containing QASM-compatible gates. If these two modules were conjoined and scheduled together using fine-grained scheduling, they could be scheduled in 21 cycles, with $k = 2$. Maintaining modularity, on the other hand, serializes the resulting blackboxes due to the data dependency. At this granularity, we must execute

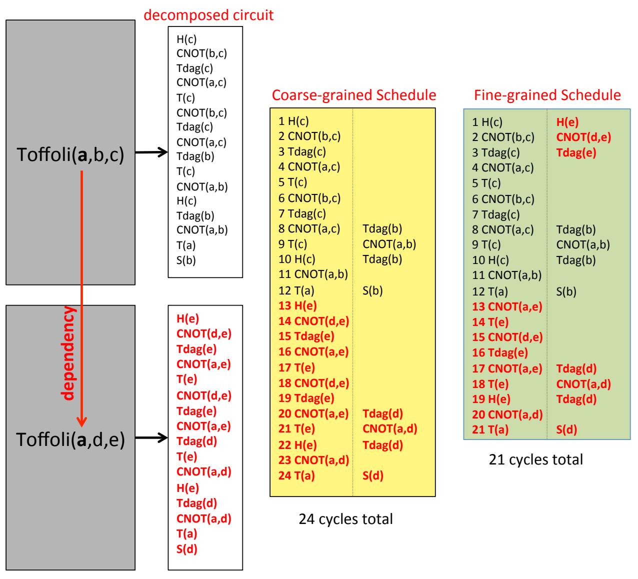

Figure 4: Scheduling of two dependent Toffoli operations with flattened and modular code. The Toffoli(x,y,z) operation is decomposed into gates supported by QASM. In the modular case, the scheduler treats the Toffoli operations as blackboxes and does not detect the inter-blackbox parallelism that exists between the decomposed circuits.

all of the scheduled sub-operations for *Toffoli(a,b,c)* before any of those for *Toffoli(a,d,e)*. In this case, the resulting $k = 2$ schedule for the two Toffoli operations is 24 cycles.

This example motivates our desire to choose the appropriate degree of modularity and flattening. To this end, we define and explore here a *Flattening Threshold (FTh)*. Namely, by performing resource estimation analysis on each module, we determine the number of gates within the module. If this is less than the flattening threshold, then the module is flattened, i.e. all the function calls contained within it are inlined. This results in leaf modules that consist of at most *FTh* operations.

The flattening threshold is determined by characterizing the initial modularity within a program. Fig. 5 shows the percentage of modules with gate counts falling within specified ranges. This reveals the proportion of modules that could be flattened for a certain *FTh*. Based on these and other experiments, the remainder of our studies use *FTh* set to 2 million operations. This flattened 80% or more of the modules contained within all benchmarks except *SHA-1*. For *SHA-1*, we used a flattening threshold for 3 million which flattened the entire benchmark.

3.2 Execution Model

The Multi-SIMD(k,d) model allows 1 to k discrete, simultaneous gates to be executed in a single *logical* timestep, each of which can be applied on 1 to d qubits apiece. For example, if 10 different qubits all require a CNOT operation applied to them, these can be positioned within a single operation region and the CNOT will be applied in a single timestep.

In this compiler, we assume that modules are blackboxes, so all active qubits are flushed to global memory during calls. This is mitigated by the fact that module calls are relatively infrequent and only cause a fixed overhead of a single teleportation cycle. It is also assumed that all ancilla qubits are generated by global and teleported to the SIMD region where they are needed.

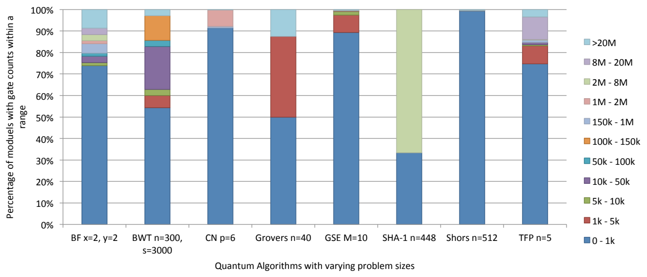

Figure 5: Histogram of gate counts represented as percentage of total modules in benchmarks. Using a flattening threshold of 2M operations, 80% or more modules were flattened for fine-grained scheduling for all benchmarks except SHA-1. For SHA-1, a flattening threshold of 3M was used to flatten the entire benchmark.

Teleporting data between SIMD regions also must be orchestrated by the scheduler so the quantum teleportation (QT) sub-operations shown in Fig. 2 can be scheduled. If a qubit physically residing in one region is scheduled in a different region in the next timestep, it is moved to that region. If no operation is scheduled on that qubit and that region is active in the next timestep, it is moved to a memory region. In later variations, a local memory is used to store qubits without teleporting them to global memory if they will be reused in the same region next.

Our execution models and evaluations assume that each gate operation takes 1 timestep. We also account for communication latencies. Some models of QC communication have a latency that varies proportionally to distance traveled, but QT approaches are distance insensitive. Rather, in QT, the bulk of the latency of each communication operation is the sequence of four qubit manipulation steps shown in Fig. 2. Their schedule can be integrated into the computation schedule for the program. In order to simplify timestep sequencing and accounting, each timestep is constrained to the longest operational time, for example 10μs for a CNOT [33].

3.3 Benchmarks

Finally, we also describe the quantum benchmarks our work studies. Promising quantum algorithms now exist for a range of problems such as factorization, searching, order and period finding, eigenvalue estimation, phase estimation and discrete logarithms [32]. We study eight quantum benchmarks of significant scale. Fully implemented in Scaffold, these are the largest and most sophisticated quantum computing programs produced to date. They are described below:

- Grover's Search (GS): Uses a quantum concept called *amplitude amplification* to search a database of 2^n elements [15]. The benchmark is parameterized by n.
- Binary Welded Tree (BWT): Uses quantum random walk algorithm to find a path between an entry and exit node of a binary welded tree [7]. The benchmark is parameterized by height of the tree (n) and a time parameter (s) within which to find the solution.
- Ground State Estimation (GSE): Uses quantum phase estimation algorithm to estimate the ground state energy of a molecule [48]. The benchmark is parameterized by the size of the molecule in terms of its molecular weight (M).
- Triangle Finding Problem (TFP): Finds a triangle within a dense, undirected graph [28]. The benchmark is parameterized by the number of nodes n in the graph.

- Boolean Formula (BF): Uses the quantum algorithm described in [2], to compute a winning strategy for the game of Hex. The benchmark is parameterized by size of the Hex board (x, y).
- Class Number (CN). A problem from computational algebraic number theory, to compute the class group of a real quadratic number field [16]. The benchmark is parameterized by p, the number of digits after the radix point for floating point numbers used in computation.
- Secure Hash Algorithm 1 (SHA-1): An implementation of the reverse cryptographic hash function [34]. The message is decrypted by using the SHA-1 function as the oracle in a Grovers search algorithm. The benchmark is parameterized by the size of the message in bits (n).
- Shor's Factoring Algorithm (Shors): Performs factorization using the Quantum Fourier Transform [41]. The benchmark is parameterized by n, the size in bits of the number to factor.

The benchmarks modularly describe the quantum circuit acting on logical qubits. The parameters dictate the problem size of the benchmarks. Our studied benchmarks ranged from 10^7 to 10^{12} gates with their various problem sizes.

4. Scheduling

Efficient scheduling of large-scale benchmarks requires a scalable approach and a delicate balance between maximizing parallelism and minimizing communication. We implement and study two different scheduling algorithms with different approaches to scheduling for parallelism with communication awareness. We also use a hierarchical, coarse-grained scheduler to limit final program size.

As discussed in Section 2.3, the primary means of communication within the MultiSIMD architecture is QT, which potentially requires an additional overhead of 4 timesteps per operation. If each operation timestep requires QT to receive its operands, then the full runtime could be 5X longer than the computation critical path. By adapting our schedulers to account for data movement costs and to reduce unnecessary movements, our runtimes remain much closer to the computation-only ideal.

Schedules are stored as a list of sequential timesteps. Each timestep consists of an array of $k + 1$ SIMD regions. The 0th region contains a list of the qubits that will be moved and their sources and destinations timestep. The remaining SIMD regions contain an unsorted list of operations to be performed in that region. Operations consist of an operation type (X, Z, CNOT, etc.), a list of qubit operands and a list of pointers to each operand's next operation.

4.1 Communication-Aware Scheduling with RCP

The Ready Critical Path (RCP) algorithm is a traditional parallel computing algorithm described in [51, 52]. The primary distinguishing feature is that it maintains a ready list instead of a free list for all tasks, only queuing the tasks that have all of their dependencies met. This prevents tasks from waiting for data during execution and reduces the overall run time. The ready list is a simple, unsorted list of operations.

Here RCP is extended to support the Multi-SIMD execution model by providing a priority scheduling mechanism that evaluates the operation type, movement cost, and *slack*. The operation type is used for grouping qubits to expose data parallelism. The movement cost indicates whether a bit needs to move (0) or is already in the present SIMD region (1); minimized movement is preferred. *Slack* is the graph distance between uses of a particular qubit, so if many operations need to occur before the next use that qubit does not need to be scheduled as soon; this is negatively correlated with the priority. The metrics can be multiplied by weights, w_op, w_dist,

```
Function rcp(Module my_module) is
    int simd, ts = 0;
    OPTYPE optype;
    RCP schedule;
    OP rcpq[] = my_module.top();
    while not rcpq.empty() do
        int simds[] = 1..k; while not (simds.empty() or
        rcpq.empty()) do
            {simd, optype} =
            getMaxWeightSimdOpType(rcpq, simds);
            schedule[ts][simd] = extract_optype(rcpq,
            optype);
            simds.delete(simds.find(simd));
        end
        updateRcpq(ts, rcpq); ts++;
    end
end
Function getMaxWeightSimdOpType(OP rcpq, list of int
simds) : {int, OPTYPE} is
    for each op in rcpq do
        optype_cnt[op.optype()]++
        for each qubit in op.args() do
            locs[op] —= 1 ¡¡ qubit.simd();
        end
        // foreach location
    end
    // foreach op for each simd in simds do
        for each op in rcpq do
            weight = w_op * op.optype() + w_dist *
            (locs[op] & (1 ¡¡ simd)) - w_slack * op.slack();
            if weight ¿ max then
                max = weight;
                max_simd = simd;
                max_optype = op.optype();
            end
        end
    end
    return {max_simd, max_optype};
end
Function updateRcpq(int ts, OP rcpq[], RCP schedule) is
    for each op in schedule[ts] do
        for each child in op.children() do
            if child.ready() then
                rcpq.push_unique(child);
            end
        end
        op.slack–;
    end
end
```

Algorithm 1: The RCP algorithm. At each timestep, it computes the relative weight for scheduling an operation type to each SIMD region based on the prevelence of that operation, the distance of each qubit from that SIMD region, and the graph distance to the next use of that qubit. The highest weighted operation type is then scheduled to its preferred SIMD region and the calculation is repeated until no more operations can be scheduled.

and w_slack respectively, though in this paper all weights are set to 1.

As shown in Algorithm 1, at each timestep the priority for each operation type in each SIMD region is calculated. A maximum priority is maintained along with the SIMD region and operation type. As the priority for each configuration is calculated, this maximum and its corresponding values are updated. When all configurations'

```
Function lpfs(DAG G, list of int simds, int l) : Schedule S is
    for $i$ in 0 to l-1 do
        // Get longest paths for allocated SIMD regions
        $simd[i]$ = getNextLongestPath(G.top);
    end
    $ready$ = $G$.top();
    while (! $ready$.empty() && ! $simd$.forall().empty() ) do
        // Schedule each time
        for $i$ in 0 to l-1 do
            // Schedule allocated SIMD regions
            if ($refill$ && $simd[i]$.empty()) then
                // Reuse SIMD region if it is out of
                operations
                $simd[i]$ = getNextLongestPath($ready$);
            end
            $op$ = $simd[i]$.pop();
            $S[time][i]$.push($op$);
            if ($opportunistic\_simd$) then
                // Schedule ready operations of the same
                type
                $S[time][i]$.push($ready$.getAllOps($op$.op_type));
            end
        end
        for $i$ in l to k-1 do
            // Schedule unallocated SIMD regions
            $optype$ = $ready$.top().op_type;
            $S[time][i]$.push($ready$.getAllOps($op$.op_type));
        end
        for $op$ in $S[time]$.forall().getAllOps() do
            // Update ready list
            $ready$.push($op$.getReadyChildren());
        end
        $ready$.uniq();
        $time$++;
    end
    return $S$;
end
```

Algorithm 2: The Longest Path First Scheduling algorithm. First, the longest paths for l allocated SIMD regions. At each timestep first schedule the allocated regions, then any unallocated regions are assigned from the free list. If the SIMD option is set, add data parallel operations to all regions. If refill is set, anytime a region completes its path find a new one from the current free list.

priority have been computed, the SIMD region and operation type with the highest priority is scheduled, the ready list is updated, and the SIMD region is removed from the list of available regions. This priority computation is repeated until all operations in the ready list are scheduled or all SIMD regions are occupied. The scheduler then updates the ready list with the operations that will be ready after the current timestep completes. The scheduler completes once the ready queue is empty.

4.2 Communication-Aware Scheduling with LPFS

Many of our benchmarks are highly serial, with an average critical path speedup of around 1.5x (Fig. 6). This serialization prevents the data dependencies from being accelerated significantly through parallelization, but it does provide ample opportunity for reducing expensive communication by optimizing the location of qubits throughout the execution. To this end, the Longest Path First Scheduling (LPFS) algorithm was developed (see Algorithm 2).

LPFS assigns l SIMD regions, where $l < k$, to be dedicated to computing l longest paths. These longest paths are statically assigned to those regions and so any qubits in their operational paths will have relatively few movements. This is especially useful when arbitrary rotations are decomposed, where a single qubit may have up to several thousand operations performed sequentially with no movement of the target qubit.

The longest path algorithm initially finds the critical path in the program DAG generated by LLVM. It starts at the top of the DAG and sets a tag at each node with the furthest distance from the top. After finding the largest depth at the bottom, the path is traced back and recorded. Multiple longest paths can be found sequentially, and at arbitrary positions in the execution based on the current free list. By being able to find multiple longest paths, further movement can potentially be reduced. All longest paths are returned as an ordered list of the operations to be performed in that path. These longest paths are stored in an array of size l for each statically scheduled region.

All operations that are not on any of the active longest paths are added to a free list. These are scheduled in any of the SIMD regions in the range $[l+1, k]$ based on their order in the free list. This order is not prioritized, but is instead populated based on the order of operations as they are scheduled, and their descendants.

Two additional options can be used to control LPFS, *SIMD* and *Refill*. *SIMD* scheduling allows any SIMD region dedicated to a longest path execution to execute other free list operations of the same type; this setting also allows any timesteps where that SIMD region would have to stall for dependencies to execute arbitrary operations from the free list as well, thus providing SIMD parallelism. The *Refill* option is for the case when multiple longest paths are used and they are of unequal lengths. Once the shorter path completes, the next longest un-executed path will be found and scheduled in the region that completed. Our experiments were run with l = 1, and both SIMD and Refill enabled.

4.3 Hierarchical Scheduling

To allow benchmarks to scale beyond tractable sizes, we employ a coarse-grained schedule which stitches together optimized schedules for leaf modules scheduled by RCP and LPFS in our call tree. The scheduler uses a simple, list-based approach to schedule leaf modules and operations in non-leaf modules with the goals of improving parallelism and/or reducing communication overheads. Algorithm 3 illustrates the overall approach.

Given a set of pre-scheduled leaf nodes, the coarse-grained scheduler completes the program scheduling by using a list scheduler to compose together a full schedule. Operations are assigned priorities based on criticality and scheduled in priority order. The quantum gate operations in non-leaf modules are scheduled along with invocations to other modules. The invoked modules have been previously scheduled by one of the fine-grained schedulers discussed next and are now treated as blackbox functions. Once the schedule for a module is determined, it is characterized as a blackbox with a length dimension equal to schedule length, and a width dimension equal to highest degree of parallelism found in the schedule.

To allow the coarse-grained scheduler to effectively parallelize the invoked blackboxes within the width (k) constraint, we consider flexible rectangular dimensions for each blackbox. During fine-grained scheduling of each module, multiple schedules are determined to find schedule lengths with widths between 1 to k. The coarse-grained scheduler is presented these blackboxes with multiple dimensions. When parallelizable modules are encountered, the combination of blackboxes that yields the minimal length subject to the width constraint is chosen. Algorithm 3 shows the pseudo-code for coarse-grained scheduling with flexible blackbox dimensions.

In a k-resource constrained schedule, the width for any invoked module is at most k. Any operations (other than invoked modules)

```
for each non-flat module do
    //Track schedule in terms of blackbox dimensions
    totalL = 0; totalW = 0; // total length and width
    currL = 0; currW = 0; //current length and width
    for each operation $F_i$ in a priority-ordered set of operations
    $\{F$: Priority($F_i$) $\geq$ Priority($F_j$) if $(i < j)\}$ do
        Check predecessors to find the earliest timestep $t_e$ in which
        $F_i$ can be scheduled
        Get width $W$ and length $L$ for $F_i$
        if $(t_e \leq totalL + currL)$ then
            // dependencies show that $F_i$ can be parallelized with
            previous schedule
            if $(currW + W \leq K)$ then
                //parallelize the operation $F_i$
                timestep($F_i$) = max($totalL$+1, $t_e$)
                $currW$ = $currW$+$W$
                $currL$ = max($currL$, timestep($F_i$)+$L$)
                $F_p$ = $\{F_p$,$F_i\}$ //Add to set of parallel functions
                in current schedule
            else
                //$k$-constraint would be violated if parallelized
                for set of functions $\{F_p$,$F_i\}$ do
                    Try all combinations of possible widths, and
                    compute length.
                end
                if one or more combinations found with
                combined width $\leq K$ then
                    Choose combination with smallest length.
                    $currW$ = Width of combination
                    $currL$ = Length of combination
                    $F_p$ = $\{F_p$,$F_i\}$ //Add to set of parallel
                    functions in current schedule
                else
                    //serialize $F_i$ due to $k$-constraint
                    $totalW$ = max($totalW$, $currW$)
                    $totalL$ = $totalL$ + $currL$
                    timestep($F_i$) = $totalL$+1
                    $currW$ = $W$; $currL$ = $L$
                    $F_p$ = $\{F_i\}$ //set of parallel functions in
                    current schedule
                end
            end
        else
            // serialize $F_i$ due to data dependency
            $totalW$ = max($totalW$, $currW$)
            $totalL$ = $totalL$ + $currL$
            timestep($F_i$) = $totalL$+1
            $currW$ = $W$; $currL$ = $L$
            $F_p$ = $\{F_i\}$ //set of parallel functions in current
            schedule
        end
    end
    //merge current box with total box dimensions
    $totalW$ = max($totalW$, $currW$)
    $totalL$ = $totalL$ + $currL$
    Store totalW and totalL in data structure
end
```

Algorithm 3: Hierarchical scheduling algorithm for k SIMD regions. Operations are scheduled in priority order similar to list scheduling. Flexible blackbox dimensions are considered for parallelizable modules to find the best combination for them.

encountered by the coarse-grained scheduler have an operation execution cost of 1 and a movement cost of 4.

4.4 Scheduling Communication using Local Memory

To reduce communication overheads, we study further optimizations using local memories attached to the SIMD regions. Once computations are mapped to SIMD regions and the necessary communication schedules are determined, some qubits can be moved to local memories using a lower overhead ballistic movement instead of teleportation to the global memory. Distinguishing a move as a local move or teleportation primarily depends on where its next computation is scheduled, but local memory regions may also be constrained by capacity. With prior information about the next computation and available space in the local memory, a move can be directed to a local or global memory by the scheduler.

When a qubit in a SIMD region (source) must be stored for a few cycles before being used in a *different* SIMD region (destination), there are multiple options for storage: the global memory, the source local memory, the destination local memory, or if idle, either of the two SIMD regions. With limited local memory capacity, holding qubits for other regions or holding additional qubits before they are required can both reduce the space available for qubits being currently operated upon. For simplicity, unless the source SIMD region is idle, we move such qubits to the global memory for storage.

After a computation or communication cycle, qubits which are not scheduled for the next operation are moved to global or scratchpad memory, while those which undergo further operations in the same region will stay in place. Physically, any qubits left behind from the teleportation of state can be reinitialized and reused as ancilla or EPR pairs.

The local moves are derived from the RCP or LPFS schedules on leaf modules. The schedules are updated to reflect the single cycle local move if its next operation location is in the same region and there is space in the local memory, otherwise the original teleportation movement is kept. If any SIMD regions in a timestep have a global move, the full four cycle move time is retained to avoid disrupting the schedule.

5. Results

We present three sets of results. First, we evaluate the amount of parallelism our schedulers can expose when communication is assumed to be zero-cost. Second, we demonstrate the benefits of communication-aware schedulers when accounting for communication costs to global memory. Finally, we evaluate the benefits of local scratchpad memories associated with each SIMD region.

In each set of results we look at all eight benchmarks under the LPFS and RCP scheduling algorithms. Both algorithms are run with k = 2 and 4 (the number of regions). For simplicity, we assume infinite d (number of qubits in a region). LPFS also uses Refill and SIMD options. Refill lets a region with a longest path schedule a new path if the current one completes, and SIMD allows non-path operations to be SIMD scheduled in a longest path region. All experiments were run at the logical level, omitting physical qubit sub-operations and error-correction, as these are constant multipliers to the overall runtimes within the range of runtimes involved.

5.1 Characterizing Logical Parallelism

In our first set of results, looking solely at the parallelism in the benchmarks, we are able to compare the speedups provided by a Multi-SIMD architecture against the theoretical maximum based on the estimated critical path. As seen in Fig. 6, all of the benchmarks except Shor's were able to achieve near-theoretical speedup on the Multi-SIMD architecture, at either k = 2 or 4. This is assisted by using $d = \infty$, which allows for clustering as many qubits as possible into SIMD regions. In Section 5.4 we explain the difference in trend for Shor's.

RCP speedups are lower than or equal to LPFS in every benchmark except TFP, in some cases tying at k = 4 with LPFS at k = 2. This is due to RCP being better suited to coarse-grained scheduling in classical functions, looking predominantly at data dependencies, then operation- and data-level parallelism, and finally slack

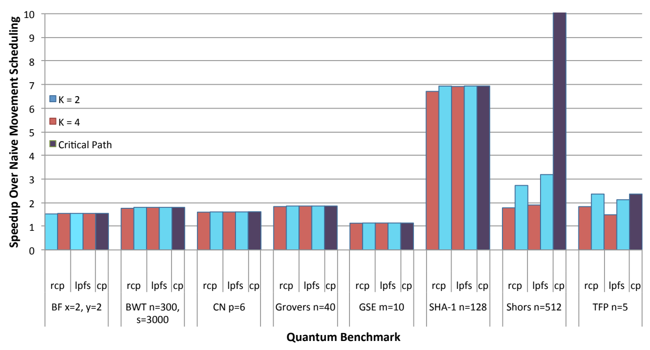

Figure 6: The speedup over sequential execution of each benchmark with each scheduling algorithm, compared to the estimated critical path. Almost all benchmarks, except Shor's, achieve near-complete speedup by $k = 4$.

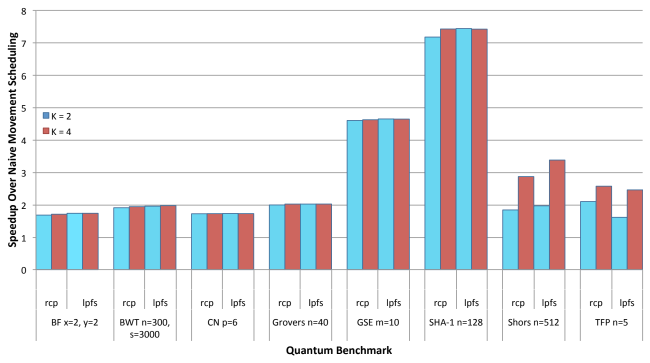

Figure 7: Speedups using a communication-aware scheduler over a sequential, naive movement model. All benchmarks show improvement over Fig. 6, with GSE showing the largest gains. Notice the renormalization of the y-axis in comparison to Fig. 6, as a result of introducing the communication costs.

based on graph distance. LPFS on the other hand focuses much more heavily on critical path prioritization.

TFP under RCP behaved interestingly, where all of the leaves scheduled by RCP had longer runtimes than those scheduled with LPFS. However, under the coarse-grained hierarchical scheduler the longer schedules were narrower, allowing for the blackboxes to be scheduled in parallel instead of in serial. This higher level parallelism allows for a shorter overall schedule.

5.2 Runtime Speedup with Data Movement Analysis

Fig. 7 shows all scheduling algorithm speedups over a naive movement model where data is moved between SIMD regions and global memory every timestep, effectively increasing the overall runtime by 5X. All benchmarks show some speedup over communication-unaware runtime models due to reduced movement. An average increase in speedup of 46% is seen across all benchmarks. The largest gain is seen in GSE (308%). The critical path was not used for a theoretical bound in these or the next results because we did not have a viable critical path model for incorporating movement.

Some benchmarks, such as BF, CN, Grovers, SHA-1 and Shor's, have a large number of highly dependent, serial operations. BF, CN and SHA-1 are composed of several CTQG modules, which produces unoptimized code that is highly locally serialized. This results in benchmarks that have a low degree of parallelism and cannot be well optimized. The interactions between data dependencies

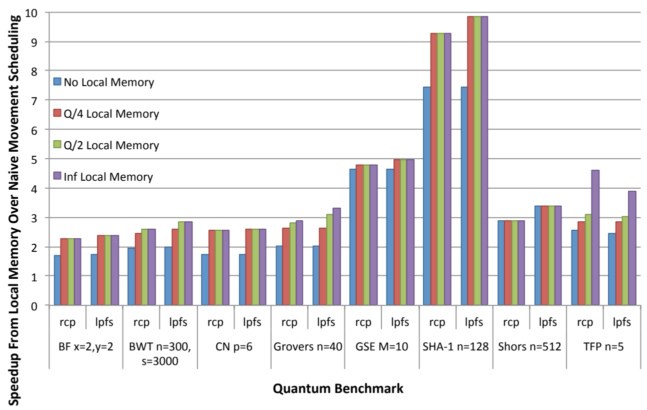

Figure 8: Speedups due to the addition of local memory to operating regions, computed over a sequential, naive movement model. Results are shown for a Multi-SIMD(4,∞) architecture.

also result in many small (1-2 qubit) moves between global memory and various SIMD regions that cannot be removed to improve performance.

GSE shows the largest gains due to its distinctive structure. The two key qubit registers containing the primary active qubits are rarely moved out of an SIMD region once they are in place and typically have long sequences of operations on the same qubits. This results in very few moves either between SIMD regions or memory.

5.3 Runtime Speedup with Local Memories

Fig. 8 illustrates the performance benefits of adding local memory to SIMD regions. To study the effectiveness of a small amount of local memory in each SIMD region, we experiment with varying memory sizes. For each benchmark, we limit the capacity of local memory to one-half and one-quarter of the minimum qubits Q required by the benchmark. Q is computed by scheduling the benchmark run sequentially, with maximal possible reuse of ancilla qubits across functions. Table 1 shows the value of Q determined for each benchmark. A parallel schedule of the benchmark may require more than Q qubits, thus the global memory may be much larger than the size of Q. The maximum benefit of local memory is demonstrated with an infinite capacity memory region.

Algorithms that are effective at reducing communication overheads do so by scheduling maximally interacting qubits in a single region. Adding local memory to SIMD regions raises the effectiveness of these algorithms since it allows the frequently interacting qubits to stay close. Thus, LPFS naturally sees higher benefits than RCP with local memory for most benchmarks.

These data movement based speedups show that, at least in some cases, it is possible to minimize the communications within the system to achieve a reasonable speedup (up to 9.82X in the case of SHA-1) in a practical system with physical limits on EPR distribution for teleportation and limited parallelism.

5.4 Sensitivity to d and k parameters:

The preceding results show that in most quantum benchmarks, data-level parallelism (d) achieved through microwave signaling is able to capture most of the parallelism, even with modest numbers of distinct SIMD regions. In fact, even though we practically assumed infinite amount of data-parallelism available in our SIMD regions, our other experiments have shown that decreasing this to below 32 qubits only causes marginal changes. This is important, since we have solely focused on the logical level, but we can see

Benchmark	Q
BF x=2, y=2	1895
BWT n=300, s=3000	2719
CN p=6	60126
Grovers n=40	120
GSE M=10	13
SHA-1 n=448	472746
Shors n=512	5634
TFP n=5	176

Table 1: The minimum number of qubits Q required by the benchmarks, computed with sequential execution and maximum reuse of ancilla qubits.

Rotation Operation	Primitive Operations Approximating Rotations
$R_z(q_1, \theta_1)$	$T(q_1) - S^\dagger(q_1) - H(q_1) - Z(q_1) - ...$
$R_z(q_2, \theta_2)$	$H(q_2) - Y(q_2) - X(q_2) - H(q_2) - ...$
...	...
$R_z(q_n, \theta_n)$	$S(q_n) - X(q_n) - T(q_n) - T^\dagger(q_n) - ...$

Table 2: Parallel rotations cannot be executed simultaneously on a hardware with primitive operations, unless there are enough SIMD regions to accommodate them.

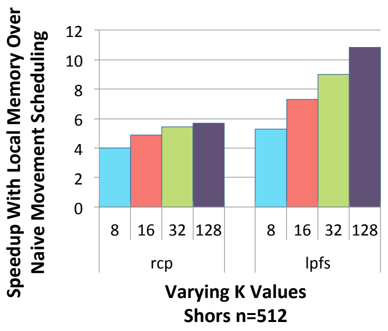

Figure 9: Shor's speedups as scheduled with a communication-aware scheduler on a Multi-SIMD architecture with local memories. High numbers of rotations cause long serial threads of operations to each execute on a separate SIMD region, thus getting better gains with higher k.

that even with added qubits for error correction, the architecture is able to cope and perform well.

Shor's showed a greater sensitivity to the number of SIMD regions available (k). The reason for this can be traced to the large number of rotation operations that exists in this code. These rotation operations can theoretically execute at the same time because they are on distinct qubits, except for the fact that practically they need to be decomposed into primitive, standard operations (as described in 3.1). This can prohibit the parallelization of operations unless more SIMD regions are created to accommodate them, as illustrated in Table. 2. Since many of these rotations were not inlined into the code, to keep the size manageable, they remain as blackboxes in the course-grained schedule. That causes the scheduler to allocate a separate region to each, effectively increasing the need for these regions. In Fig. 9, we show how the speedups vary with k for this benchmark.

6. Related and Future Work

This paper builds on several important previous studies relating to SIMD parallelism [5, 24, 39], ancilla preparation [19, 23], and quantum architecture [5, 30, 47]. Our work is the first to use a complete compiler infrastructure to discover this parallelism, allowing us to evaluate a non-trivial set of benchmarks (previous work focused almost exclusively on Shor's and Grover's, or other small quantum circuits). It is also the first to incorporate data movement analysis and optimizations within the compiler framework established.

Some prior work has explored optimization of execution latencies with SIMD architectures, but in a more limited context. Chi *et.al.* [5] proposed a SIMD architecture based on the technology of *electron spins on liquid helium*. For a quantum carry-lookahead adder circuit, they evaluated pipelining of ancilla preparation for CNOT and Toffoli gates to reduce latency, and optimization of width of SIMD regions to reduce area requirements. Our work builds on this model, with the implementation of a complete compiler and the study of a much larger and more diverse benchmark suite.

Schuchman $et.al.$ [39] identify a high-level parallelism pertaining to specific quantum tasks of *uncomputation* (analogous to garbage collection for qubits) and propose a multi-core architecture to minimize latency and expensive inter-core communication during their execution. This kind of parallelism fits well into our Multi-SIMD model; it can be easily extended to support the proposed multiple cores. Some degree of uncomputation already exists in the compiled code of our benchmarks and is naturally parallelized by our model, and more can be added in the future to reclaim unused qubits.

The SIMD regions in our architecture are well-suited for a commonly used class of error-correction codes known as concatenated codes [1, 42]. A new class of ensemble codes, known as surface codes [17], have the potential of lowering ECC overhead for very large problems. Future research will explore whether surface code operations are amenable to SIMD parallelism.

7. Conclusion

We have proposed a Multi-SIMD architectural target for this compiler that incorporates practical physical constraints in quantum computing technology. To fully exploit its computational power, our work has also developed a scalable compiler that uses deep analysis to create logical schedules that enable parallelism and minimize communication. Largely as a result of SIMD parallelism, efficient communication and fine-grained data locality, we achieve speedups over our sequential baseline of 2.3X to 9.8X. Since quantum error correction can have overhead exponential in program execution time [1, 35, 42], these speedups can be even more significant than they appear, because they offer important leverage in allowing complex QC programs to complete with manageable levels of QECC. The source for ScaffCC and benchmarks is available on GitHub at https://github.com/ajavadia/ScaffCC.git.

Acknowledgments

This work was funded in part by NSF PHY-1415537. Many researchers have contributed to our benchmarks, including John Black, Lukas Svec, Aram Harrow, Amlan Chakrabarti, Chen-Fu Chiang, Oana Catu and Mohammad Javad Dousti.

References

[1] P. Aliferis and A. Cross. Subsystem fault tolerance with the Bacon-Shor code. *arXiv preprint quant-ph/0610063*, 2006.

[2] A. Ambainis, A. M. Childs, B. W. Reichardt, R. Spalek, and S. Zhang. Any AND-OR formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a quantum computer. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07. IEEE Computer Society, 2007. ISBN 0-7695-3010-9.

[3] B. Blinov, D. Moehring, L.-M. Duan, and C. Monroe. Observation of entanglement between a single trapped atom and a single photon. *Nature*, 428(6979):153–157, 2004.

[4] S. Cavallar, B. Dodson, A. Lenstra, W. Lioen, P. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, and P. Zimmermann. Factorization of a 512–bit RSA modulus. In B. Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67517-4.

[5] E. Chi, S. A. Lyon, and M. Martonosi. Tailoring quantum architectures to implementation style: a quantum computer for mobile and persistent qubits. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07. ACM, 2007. ISBN 978-1-59593-706-3.

[6] J. Chiaverini, D. Leibfried, T. Schaetz, M. Barrett, R. Blakestad, J. Britton, W. Itano, J. Jost, E. Knill, C. Langer, R. Ozeri, and D. J. Wineland. Realization of quantum error correction. *Nature*, 432 (7017):602–605, 2004.

[7] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03. ACM, 2003. ISBN 1-58113-674-9.

[8] I. Chuang. Quantum architectures: qasm2circ. URL `http://www.media.mit.edu/quanta/qasm2circ/`.

[9] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Lett.*, 74:4091–4094, May 1995.

[10] D. Copsey, M. Oskin, F. Impens, T. Metodiev, A. Cross, F. T. Chong, I. L. Chuang, and J. Kubiatowicz. Toward a scalable, silicon-based quantum computing architecture. *Selected Topics in Quantum Electronics, IEEE Journal of*, 9(6):1552–1569, 2003.

[11] D. P. L. A. Craik, N. M. Linke, T. P. Harty, C. J. Ballance, D. M. Lucas, A. M. Steane, and D. T. C. Allcock. Microwave control electrodes for scalable, parallel, single-qubit operations in a surface-electrode ion trap, August 2013. URL `http://arxiv.org/pdf/1308.2078.pdf`.

[12] D. P. DiVincenzo. The physical implementation of quantum computation. *Fortschritte der Physik*, 48(9-11):771–783, 2000. ISSN 1521-3978.

[13] J. Garcıa-Ripoll, P. Zoller, and J. Cirac. Speed optimized two-qubit gates with laser coherent control techniques for ion trap quantum computing. *Phys. Rev. Lett*, 91(15):157901, 2003.

[14] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the TRIPS computer system. In *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Mar. 2009.

[15] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, 1996. ISBN 0-89791-785-5.

[16] S. Hallgren. Fast quantum algorithms for computing the unit group and class group of a number field. In H. N. Gabow and R. Fagin, editors, *STOC*. ACM, 2005. ISBN 1-58113-960-8.

[17] C. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14 (12):123011, 2012.

[18] N. Isailovic. *An investigation into the realities of a quantum datapath*. PhD thesis, University of California, Berkeley, 2010.

[19] N. Isailovic, M. Whitney, Y. Patel, and J. Kubiatowicz. Running a quantum circuit at the speed of data. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 177–188. IEEE Computer Society, 2008.

[20] A. JavadiAbhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, F. Chong, M. Martonosi, M. Suchara, K. Brown, M. Pedram, and T. Brun. Scaffold: Quantum programming language. Technical report, Princeton University, 2012.

[21] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. Chong, and M. Martonosi. ScaffCC: A framework for compilation and analysis of quantum computing programs. *ACM International Conference on Computing Frontiers (CF 2014)*, May 2014.

[22] M. Johanning, A. Braun, N. Timoney, V. Elman, W. Neuhauser, and C. Wunderlich. Individual addressing of trapped ions and coupling of motional and spin states using RF radiation. *Phys. Rev. Lett.*, 102: 073004, Feb 2009.

[23] N. C. Jones, R. Van Meter, A. G. Fowler, P. L. McMahon, J. Kim, T. D. Ladd, and Y. Yamamoto. Layered architecture for quantum computing. *Physical Review X*, 2(3):031007, 2012.

[24] J. Kim, S. Pau, Z. Ma, H. McLellan, J. Gates, A. Kornblit, R. E. Slusher, R. M. Jopson, I. Kang, and M. Dinu. System design for large-scale ion trap quantum information processor. *Quantum Information & Computation*, 5(7):515–537, 2005.

[25] V. Kliuchnikov, D. Maslov, and M. Mosca. SQCT: Single qubit circuit toolkit. URL `https://code.google.com/p/sqct/`.

[26] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[27] D. Leibfried, B. DeMarco, V. Meyer, D. Lucas, M. Barrett, J. Britton, B. J. WM Itano, C. Langer, and D. T Rosenband. Experimental demonstration of a robust, high-fidelity geometric two ion-qubit phase gate. *Nature*, 422(6930):412–415, 2003.

[28] F. Magniez, M. Santha, and M. Szegedy. Quantum algorithms for the triangle problem. In *Proceedings of the Sixteenth annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 1109–1117, 2005. ISBN 0-89871-585-7.

[29] T. Metodi, D. Thaker, A. Cross, F. T. Chong, and I. L. Chuang. Scheduling physical operations in a quantum information processor. *Proceedings for the SPIE Defense & Security symposium, Orlando, FL, April*, 2006.

[30] T. S. Metodi, D. D. Thaker, and A. W. Cross. A quantum logic array microarchitecture: Scalable quantum data movement and computation. In *MICRO*, pages 305–318. IEEE Computer Society, 2005. ISBN 0-7695-2440-0.

[31] C. Monroe, D. Meekhof, B. King, W. Itano, and D. Wineland. Demonstration of a fundamental quantum logic gate. *Physical Review Letters*, 75(25):4714, 1995.

[32] M. Mosca. Quantum algorithms. In R. A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 7088–7118. Springer New York, 2009. ISBN 978-0-387-75888-6.

[33] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[34] N. I. of Standards and Technology. *FIPS PUB 180-4: Secure Hash Standard (SHS)*. U.S. Department of Commerce, 2012.

[35] M. Oskin, F. Chong, and I. Chuang. A practical architecture for reliable quantum computers. *Computer*, 35(1):79–87, 2002. ISSN 0018-9162.

[36] C. Ospelkaus, U. Warring, Y. Colombe, K. R. Brown, J. M. Amini, D. Leibfried, and D. J. Wineland. Microwave quantum logic gates for trapped ions. *Nature*, 476:181–184, 2011.

[37] M. Riebe, H. Häffner, C. Roos, W. Hänsel, J. Benhelm, G. Lancaster, T. Körber, C. Becher, F. Schmidt-Kaler, and D. James. Deterministic quantum teleportation with atoms. *Nature*, 429(6993):734–737, 2004.

[38] F. Schmidt-Kaler, H. Häffner, M. Riebe, S. Gulde, G. P. Lancaster, T. Deuschle, C. Becher, C. F. Roos, J. Eschner, and R. Blatt. Realiza-

tion of the Cirac–Zoller controlled-NOT quantum gate. *Nature*, 422 (6930):408–411, 2003.

[39] E. Schuchman and T. N. Vijaykumar. A program transformation and architecture support for quantum uncomputation. *SIGARCH Comput. Archit. News*, 34(5):252–263, Oct. 2006. ISSN 0163-5964.

[40] C. M. Shappert, J. T. Merrill, K. R. Brown, J. M. Amini, C. Volin, S. C. Doret, H. Hayden, C.-S. Pai, and A. W. Harter. Spatially uniform single-qubit gate operations with near-field microwaves and composite pulse compensation. *New Journal of Physics*, 15(083053), 2013.

[41] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.

[42] A. Steane. Error correcting codes in quantum theory. *Physical Review Letters*, 77(5):793–797, 1996.

[43] A. M. Steane. Active stabilization, quantum computation, and quantum state synthesis. *Phys. Rev. Lett.*, 78:2252–2255, Mar 1997.

[44] K. Svore, A. Aho, A. Cross, I. Chuang, and I. Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1):74–83, 2006. ISSN 0018-9162.

[45] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007. ISSN 0734-2071.

[46] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 2–. IEEE Computer Society, 2004. ISBN 0-7695-2143-6.

[47] D. D. Thaker, T. S. Metodi, A. W. Cross, I. L. Chuang, and F. T. Chong. Quantum memory hierarchies: Efficient designs to match available parallelism in quantum computing. In *ISCA*, pages 378–390. IEEE Computer Society, 2006. ISBN 0-7695-2608-X.

[48] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics*, 109(5):735, 2010.

[49] M. G. Whitney, N. Isailovic, Y. Patel, and J. Kubiatowicz. A fault tolerant, area efficient architecture for shor's factoring algorithm. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 383–394, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0.

[50] D. Wineland, C. Monroe, W. Itano, B. King, D. Leibfried, D. Meekhof, C. Myatt, and C. Wood. Experimental primer on the trapped ion quantum computer. *Spectroscopy*, 7:8, 1998.

[51] T. Yang and A. Gerasoulis. PYRROS: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437, 1992.

[52] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Comput.*, 19(12):1321–1344, Dec. 1993. ISSN 0167-8191.