SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation

Yongshan Ding[1], Xin-Chuan Wu[1], Adam Holmes[1,2], Ash Wiseth[1],
Diana Franklin[1], Margaret Martonosi[3], and Frederic T. Chong[1]

[1]*Department of Computer Science, University of Chicago, Chicago, IL 60615, USA*
[2]*Intel Labs, Intel Corporation, Hillsboro, OR 97124, USA*
[3]*Department of Computer Science, Princeton University, Princeton, NJ 08544, USA*

***Abstract*—Compiling high-level quantum programs to machines that are size constrained (i.e. limited number of quantum bits) and time constrained (i.e. limited number of quantum operations) is challenging. In this paper, we present SQUARE (Strategic QUantum Ancilla REuse), a compilation infrastructure that tackles allocation and reclamation of scratch qubits (called ancilla) in modular quantum programs. At its core, SQUARE strategically performs uncomputation to create opportunities for qubit reuse.**

Current Noisy Intermediate-Scale Quantum (NISQ) computers and forward-looking Fault-Tolerant (FT) quantum computers have fundamentally different constraints such as data locality, instruction parallelism, and communication overhead. Our heuristic-based ancilla-reuse algorithm balances these considerations and fits computations into resource-constrained NISQ or FT quantum machines, throttling parallelism when necessary. To precisely capture the workload of a program, we propose an improved metric, the "active quantum volume," and use this metric to evaluate the effectiveness of our algorithm. Our results show that SQUARE improves the average success rate of NISQ applications

by 1.47X. Surprisingly, the additional gates for uncomputation create ancilla with better locality, and result in substantially fewer swap gates and less gate noise overall. SQUARE also achieves an average reduction of 1.5X (and up to 9.6X) in active quantum volume for FT machines.

***Index Terms*—quantum computing, compiler optimization, reversible logic synthesis**

I. Introduction

Thanks to recent rapid advances in physical implementation technologies, quantum computing (QC) is seeing an exciting surge of hardware prototypes from both academia and industry [1]–[4]. This phase of QC development is commonly referred as the Noisy Intermediate-Scale Quantum (NISQ) era [5]. Current quantum computers are able to perform on the order of hundreds of quantum operations (gates) using tens to hundreds of quantum bits (qubits). While modest in scale, these NISQ machines are large and reliable enough to perform some computational tasks. Looking beyond the NISQ era, quantum computers will ultimately arrive at the Fault-Tolerant (FT) era [6], [7], where quantum error correction is implemented to ensure operation fidelity is met for arbitrarily large computations.

One major challenge, however, facing the QC community, is the substantial resource gap between what quantum computer hardware can offer and what quantum algorithms (for classically intractable problems) typically require. Space and time resources in a quantum computer are extremely constrained. Space is constrained in the sense that there will be a limited number of qubits available, often further complicated by poor connectivity between qubits. Time is also constrained because qubits suffer from decoherence noise and gate noise. Too many successive operations on qubits results in lower program success rates.

Due to space and time constraints, it is critical to find efficient ways to compile large programs into programs (circuits) that minimize the number of qubits and sequential operations (circuit depth). Several options have been proposed [8]–[14]. Among the options, one approach not yet well studied is to coordinate allocation and reclamation of qubits for optimal reuse and load balancing [15]. Reclaiming qubits, however, comes with a substantial operational cost. In particular, to obey the rules of quantum computation, before recycling a used qubit, additional gate operations need to be applied to "undo" part of its computation.

In this paper, we propose the first automated compilation framework for such strategic quantum ancilla reuse (SQUARE) in modular quantum programs that could be readily applied to both NISQ and FT machines. SQUARE is a compiler that automatically determines places in a program to perform such *uncomputation* in order to manage the trade-offs in qubit savings and gate costs and to optimize for the overall resource consumption.

Optimally choosing reclamation points in a program is crucial in minimizing resource consumption. This is because reclaiming too often can result in significant time cost (due to more gates dedicated to uncomputation). Likewise, reclaiming too seldom may require too many qubits (e.g. fail to fit the program in the machine). For example, Figure 1 shows how qubit usage changes over time for the modular exponentia-

Corresponding author: yongshan@uchicago.edu

tion step in Shor's algorithm [16]. Unfortunately, finding the optimal points in a program for reclaiming qubit could get extremely complex [17], [18]. An efficient qubit reuse strategy will play a pivotal role in enabling the execution of programs on resource-constrained machines.

To precisely estimate the workload of a computational task, we propose a resource metric called "*active quantum volume*" (AQV) that evaluates the "liveness" of qubit during the lifetime of the program, which we will formally introduce in Section III-B. This is inspired by the concept of "quantum volume" introduced by IBM [19], a common measure for the computational capability of a quantum hardware device, based on parameters such as number of qubits, number of gates, and error probability. AQV is a metric that measures the volume of resource required by a program when executing on a target hardware, which can therefore serve as an minimization objective for the allocation and reclamation strategies.

The contributions of our work are:

- We present a heuristic-based compilation framework, called SQUARE, for optimizing qubit allocation and reclamation in modular reversible programs. It leverages the knowledge of qubit locality as well as program modularity and parallelism.
- We introduce a resource metric, active quantum volume (AQV), that calculates the "liveness" of qubits over the lifetime of a program. This new metric allows us to quantify the effectiveness of various optimization strategies, as well as to characterize the volume of resources consumed by different computational tasks.
- Our approach fits computations into resource-constrained NISQ machines by strategically reusing qubits. Surprisingly, adding gates for uncomputation can *improve* the fidelity of a program rather than impair it, as it creates ancilla with better locality, leading to substantially fewer swap gates and thus less gate noise. SQUARE improves the success rate of NISQ applications by 1.47X on average.
- Our approach has broad applicability from NISQ to FT machines. SQUARE achieves an average reduction of 1.5X (and up to 9.6X) in active quantum volume for FT systems.

The rest of the paper is organized as follows: Section II briefly discusses the basics of quantum computation and compilation of reversible arithmetic to quantum circuits, as well as related work in both classical and quantum compilation. Section III illustrate the central problem of allocation and reclamation of ancilla tackled in this paper and the general idea of our solution. Section IV describes in detail the techniques that make up our proposed algorithm. Section V evaluates the performance of the algorithm on an array of benchmarks under the NISQ and FT architectures. Finally, Section VI summarizes and then highlights challenges awaiting satisfactory solutions.

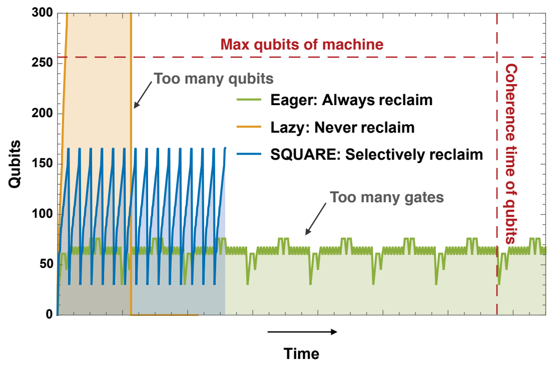


Fig. 1: Qubit usage over time for Modular Exponentiation. The shaded area under the curve corresponds to the active quantum volume of this application. The blue curve, representing a balance between qubit reclamation and uncomputation, has the lowest area and is the best option.

II. Background and Related Work

A. Basics of Quantum Computing

Quantum computers are devices that harness quantum mechanics to store and process information. For this paper, we highlight three of the basic rules derived from the principles of quantum mechanics:

- *Superposition rule:* A quantum bit (qubit) can be in a quantum state of a linear combination of 0 and 1: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex amplitudes satisfying $|\alpha|^2 + |\beta|^2 = 1$.
- *Transformation rule:* Computation on qubits is accomplished by applying a unitary quantum logic gate that maps from one quantum state to another. This process is *reversible* and *deterministic*.
- *Measurement rule:* Measurement or readout of a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ collapses the quantum state to classical outcomes: $|\psi'\rangle = |0\rangle$ with probability $|\alpha|^2$ and $|\psi'\rangle = |1\rangle$ with probability $|\beta|^2$. This is *irreversible* and *probabilistic*.

1) Reversibility constraints.: The above three rules give rise to the potential computing power that quantum computers possess, but at the same time, they impose strict constraints on what we can do in quantum computation. For example, the transformation rule implies that any quantum logic gate we apply to a qubit has to be *reversible*. The classical `AND` gate in Figure 2 is *not* reversible because we cannot recover the two input bits based solely on one output bit. To make it reversible, we could introduce a scratch bit, called *ancilla*, to store the result out-of-place, as in controlled-controlled-`NOT` gate (or `Toffoli` gate) in Figure 2. Note that we use the terminology "ancilla" in its most general sense–it is not limited to error correction ancilla, but rather, any (physical or logical) qubits used as scratch space for computation. As the arithmetic complexity scales up when tackling difficult computational problems, we quickly see extensive usage of ancilla bits in our circuits due to this *reversibility constraint*.

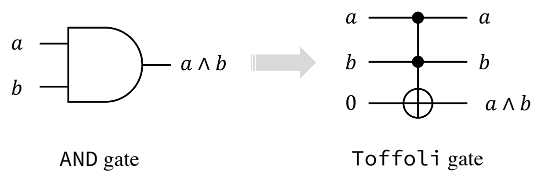


Fig. 2: Circuit diagram for the irreversible `AND` gate and the reversible `Toffoli` gate.

B. Synthesizing Reversible Arithmetic

For small arithmetic logic, algorithms exist to directly synthesize reversible circuits from the truth-table of the desired function [20]–[22] and with templates [23]. This typically works well for small low-level combinational functions, but not for functions with internal states [24]. As the complexity of the arithmetic in an algorithm scales up, *modularity* quickly becomes convenient, and in many cases necessary. That is, to construct high-level arithmetic, we need to build up from small modular subroutines.

In reversible logic synthesis and optimization, besides making our circuit for the reversible function contain as few gates as possible, we would also like to minimize the amount of scratch memory (i.e. number of ancilla bits) used in the circuit. Fortunately, there is a way to recycle ancilla bits for later reuse. For a circuit that makes extensive use of scratch memory, managing the allocation and reclamation of the ancilla bits becomes critical to producing an efficient implementation of the function.

1) Role of reversible arithmetic in quantum algorithms.: Reversible arithmetic plays a pivotal role in many known quantum algorithms. The advantage of quantum algorithms is thought to stem from their ability to pass a superposition of inputs into a classical function at once, whereas a classical algorithm can only evaluate the function on single input at a time. Many quantum algorithms involve computing classical functions, which must be embedded in the form of reversible arithmetic subroutines in quantum circuits. For example, Shor's factoring algorithm [16] uses classical modular-exponentiation arithmetic, Grover's searching algorithm [25] also implements its underlying search problem as an oracle subroutine, and the HHL algorithm for solving linear system of equations contains an expensive reciprocal step [26]. These reversible arithmetic subroutines are typically the most resource-demanding computational components of the entire quantum circuit.

C. Compiling Quantum Circuits to Target Architecture

As discussed above, there are several options for obtaining a synthesized classical reversible circuit. The next step is to compile it down to a sequence of instructions that a quantum machine recognizes and natively supports, that is to resolve *architectural constraints*. This means considering the following two aspects:

1) *Instruction set.* There are certain quantum logic gates that are supported in a given device architecture. In most cases, this gate set is "Clifford+T" gates, comprised of the `CNOT` gate, `NOT` gate (or `X` gate), Hadamard gate (or `H` gate) and `T` gate. This is a common set for most of today's gate-based quantum hardware prototypes, as well as for large-scale fault-tolerant machines (e.g. with surface code error correction). Given a classical reversible circuit, we can replace each gate with its quantum counter-part. In particular, `NOT` gates and `CNOT` gates can be directly implemented as quantum gates. For `Toffoli` gates, algorithms exist that decompose them into a sequence of Clifford+T gates [27]–[31]. At lower level, some instruction sets are proposed to offer direct control over the target hardware [32].

2) *Qubit communication.* Multi-qubit quantum gates are implemented by interacting the operand qubits with one another. At the physical level, building large-scale quantum machines with all-to-all qubit connectivity is shown to be extremely challenging. The latest effort from IonQ [33] offers a machine with 11 fully-connected qubits using trapped-ion technology. Superconducting machines, for instance those by IBM [3] and Rigetti [34], typically have *much* lower connectivity. Any scalable proposal would involve an architecture of limited qubit connectivity and a model for resolving long-distance interactions. As a consequence, interacting qubits that are not directly connected would induce communication costs.

1) Difference between NISQ and FT machines.: Depending on the topology of the architecture and the model for resolving two-qubit interactions, communication costs will differ. In the context of a NISQ machine, the most frequently used approach to resolve a long-distance two-qubit gate is through swaps, where two (physical) qubits are moved closer by performing a chain of swap gates that connects them. Each SWAP gate consists of three `CNOT` gates. The time to complete a swap chain is proportional to the length of the chain. In a FT machine, a logical qubit is encoded by a number of physical qubits. A logical operation is specified by a sequence of physical operations on its physical qubits. For instance, for surface code implementation, physical qubits form a 2D grid with every data qubit connected to its four nearest neighbors through stabilizer ancillas. In essence, a logical operation is defined by specifying how the stabilizer ancillas interact with the data qubits. In particular, a logical two-qubit gate can be defined by braiding[1] which creates a path between logical qubits, where the stabilzer ancillas along the path do not interact with their neighbors [37], [38]. Although it can extend to arbitrary length and shape in constant time, two braids are not allowed to cross. We refer interested readers to [7], [39] for excellent tutorial.

Although our proposed SQUARE approach is designed to optimize for compiling large quantum algorithms onto medium- to large-scale systems with hundreds or thousands of qubits, we demonstrate that NISQ machines can benefit

[1]The focus of this study is on braiding, but other schemes such as lattice surgery [35], [36] exists for resolving two-qubit interactions on surface code, which may expose different communication tradeoffs.

from SQUARE optimizations significantly as well. As such, Section V will include experiments that sweep a large range of system sizes (from tens to thousands of qubits), assuming architectures with their appropriate communication models (e.g. swaps and braiding). The one key difference between swaps and braids is that the time to complete a swap chain is proportional to the length of swap, whereas the time to complete a braid is proportional to the number of crossings with other braids.

D. Reclaiming Ancilla Qubits via Uncomputation

Reclaiming qubits is the process of returning them to their original $|0\rangle$ state for future reuse. Due to entanglement, this process could be costly; ancilla qubits that are entangled with data qubits will alter the data qubits' state if they are reset or measured. Fortunately, *uncomputation*, introduced by Bennett [40], is the process for undoing a computation in order to remove the entanglement relationship between ancilla qubits and data qubits from previous computations. Figure 3 (left pane) illustrates this process. In that circuit diagram, the U_f box denotes the circuit that computes a classical function f. The garbage produced at the end of U_f is cleaned up by storing the output elsewhere and then undoing the computation.

This uncomputation approach has two potential limitations: firstly, if uncomputation is not done appropriately, we need to pay for the additional gate cost, and secondly, it only works if the circuit U_f implements classical reversible logic - i.e. can be implemented with `Toffoli` gate alone, optionally with `NOT` gate and `CNOT` gate. Quantum algorithms contain non-classical gates such as Hadamard gate, phase gate and T gate; this work focuses on the part in quantum algorithms that computes classical functions (usually arithmetics) which can be implemented without those gates. As discussed in Section II-B1, classical reversible logic plays a large part of most quantum algorithms.

Related work on optimization of qubit allocation and reclamation in reversible programs dates back to as early as [17], [41], where they propose to reduce qubit cost via fine-grained uncomputation at the expense of increasing time. Since then, more [18], [42]–[44] have followed in characterizing the complexity of reclamation for programs with complex modular structures. Recent work in [24], [45] show that knowing the structure of the operations in U_f can also help identify bits that may be eligible for cleanup early. A more recent example [46] improves the reclamation strategy for straight-line programs using a SAT-based algorithm. Some of the above work emphasizes on identifying reclamation opportunities in a flat program, whereas our focus is on coordinating multiple reclamation points in a larger modular program.

E. Reclaiming Qubits via Measurement and Reset

If ancilla qubits have already been disentangled from the data qubits, we can directly reclaim them by performing a measurement and reset. We can save the number of qubits, by moving measurements to as early as possible in the program, so early that we can reuse the same qubits after measurement for other computation. Prior art [47], [48] has extensively studied this problem and proposed algorithms for discovering such opportunities.

This measurement-and-reset (M&R) approach also has limitations: firstly, a near-term challenge for NISQ hardware is to support fast qubit reset. Without it, reusing qubits after measurement could be costly or, in many cases, unfeasible. The state-of-the-art technique for resetting a qubit on a NISQ architecture is by waiting long enough for qubit decoherence to happen naturally, typically on the order of milliseconds for superconducting machines [3], significantly longer than the average gate time around several nanoseconds. FT architectures have much lower (logical) measurement overhead (that is roughly the same as that of a single gate operation), and thus are more amenable to the M&R approach. Secondly, qubit rewiring as introduced in [48] works only if measurements can be done early in a program, which may be rare in quantum algorithms – measurements are absent in many program (such as arithmetic subroutines) or only present somewhere deep in the circuit. M&R of a qubit is allowed only after all entangled results are no longer needed, whereas uncomputation can be done partially for any subcircuit. As such, unlike the uncomputation approach, M&R does not *actively create* qubit reuse opportunities.

F. Related Work in Classical Compilation

Some similarities can be seen in register allocation in classical computing. In that setting, we assign program variables to a limited number of registers in the CPU for fast access. Variables that are not stored in register may be moved to and from RAM, as a process called "register spilling". The analysis of live variable and register reuse can be very similar to that of qubits. For instance, our heuristic-based methodology is inspired by register allocation in GPU/distributed systems where communication cost needs to be minimized, and by the technique "rematerialization" that reduces the register pressure (i.e. number of registers in use at any point in time) by recomputing some variables instead of storing them to memory. But the trade-offs in qubit allocation and reclamation are unique, which we will introduce as "recursive recomputation" and "qubit reservation" in Section III-C. Finding the optimal strategy for register allocation, and similarly for qubit reuse, is known to be a hard problem [42], [49]. Luckily, we are able to transfer some general insights from the rich history of classical register allocation optimization to solve the problems in qubit allocation and reclamation.

The connection made between qubit reuse and classical register allocation [50]–[52] allows us to inherit some of the intuitions from a wealth of classical literature. Nonetheless, the uncomputation/reuse/locality trade-offs we face are fairly unique. Indeed, rematerialization [53] is very much like qubit reclamation, in that they both aim to lower active registers/qubits at the expense of computation, yet it does not exhibit the same exponential recomputation cost, nor is the increase in the live-range of variables from the recomputing step the same as qubit reservation caused by not uncomputing. We

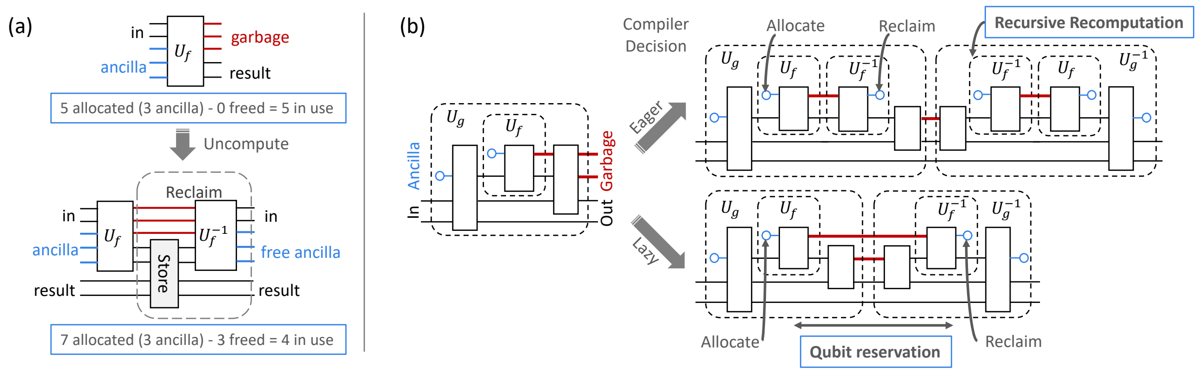


Fig. 3: (a) Ancilla qubit reclamation via uncomputation. Each horizontal line is a qubit. Each solid box contains reversible gates. Qubits are highlighted red for the duration of being garbage. (b) Illustration for Eager and Lazy strategies with their respective issues – recursive recomputation and qubit reservation. Each dashed box denotes a function call containing the enclosed gates. The allocation and reclamation points have been marked as blue circles in the circuit.

also gained general insights from numerous techniques in code scheduling [54], [55], and thread-level parallelism [56].

III. Key Idea and Motivation

This paper focuses primarily on reusing qubits via uncomputation, and discusses the significance of our proposed strategy in current noisy intermediate-scale quantum (NISQ) and future fault-tolerant (FT) architectures. Prior work such as [24] follows two basic strategies: "Eager" cleanup and "Lazy" cleanup, as illustrated in Figure 3.

Baseline 1 "Eager": Recursive Recomputation. Eager reclaims qubits at the end of every function. In the example of Figure 3, Eager performs uncomputation at the end of both U_f and U_g. When reclaiming ancilla qubits in such programs with nested functions, the uncompute step of the caller would have to repeat *everything* inside of its callee, including the callee's uncompute step. This hierarchical structure will consequently lead to re-computation of the callees, as marked in Figure 3. More formally, for a hierarchical program with ℓ levels, in the worst case, recomputation causes the number of steps to increase by a factor of 2^ℓ. We call this exponential blowup phenomenon "recursive recomputation". That is why the 2-level program in Figure 3 has roughly 4 times more steps as the original circuit. This factor will play a crucial role in our heuristic design.

Baseline 2 "Lazy": Qubit Reservation. Lazy reclaims qubits only at the top-level function. In Figure 3, this means only U_g is uncomputed, but not U_f. Lazy can sometimes be a preferred strategy because it avoids the wasted recomputation[2]. In other words, it is sometimes beneficial to temporarily leave the garbage of callees, and uncompute the garbage by their callers. This is equivalent to *inlining* the callee into the caller, and letting the caller handle the reclamation of all ancilla qubits. However, with the benefit of the avoided recomputation comes the cost of "qubit reservation". The ancilla qubits from callee are *reserved* or *blocked out* from any reuse until the end of the caller. This can be seen at the bottom right of the example in Figure 3. The garbage qubit from U_f stays as garbage until almost the end of U_g^{-1}, whereas in the Eager case, it is cleaned up right away.

[2]There are exceptions, such as recursive Fourier sampling, where recomputation cannot be avoided and is required for correctness [57].

A. Overview of SQUARE Algorithm

Most existing qubit reuse algorithms [17], [24], [41] emphasize on the asymptotic qubit savings, and commonly make an ideal assumption that machines have all-to-all qubit connectivity (i.e. no locality constraint). Since all qubits are considered identical, a straightforward way to keep track of qubits is to maintain a global pool, sometimes referred to as the *ancilla heap*. Ancilla qubits are pushed to the heap when they are reclaimed, and popped off when they are allocated, for instance in a last-in-first-out (LIFO) manner. In this ideal model, we can simply track qubit usage by counting the total number of fresh qubits ever allocated during the lifetime of a program. However, leading proposals of NISQ and FT quantum architectures have far stricter locality constraints.

Our Strategic QUantum Ancilla REuse (SQUARE) algorithm is highly motivated by the lesson that communication can be a determinant factor for qubit allocation and reclamation. Take the NISQ architecture as an example. We make the following two novel observations. Firstly, *same algorithm needs different strategies for different machine connectivity.* In Figure 5, a benchmark named Belle (whose details can be found in Section V-A) prefers Eager strategy on a 2-D lattice topology (with swaps), but Lazy strategy on a fully-connected topology (without swaps). Secondly, and most counter-intuitively, *adding uncomputation gates can improve overall circuit fidelity, if done properly*. With careful allocation and reclamation, the expense of additional uncomputation gates is compensated by the reduction of communication cost. This is because uncomputation allows us to create ancilla with

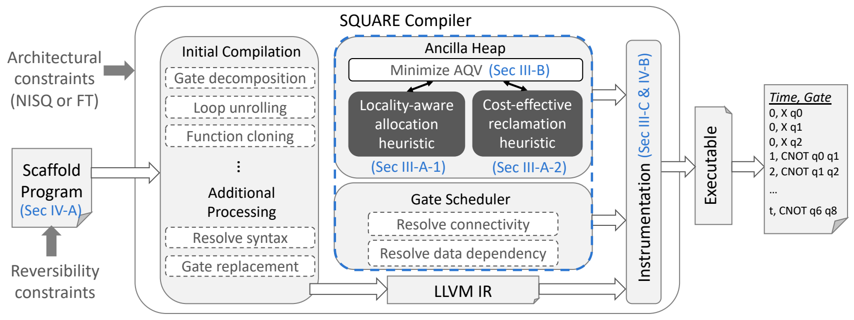


Fig. 4: Our Strategic Quantum Ancilla Reuse (SQUARE) compilation flow. SQUARE takes as input a Scaffold [58] program (see sample code in Figure 6) and produces an executable that simulates the dynamics of qubit allocation/reclamation and gate scheduling, which can then prints out an optimized schedule of quantum gate instructions.

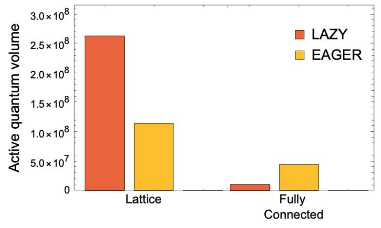


Fig. 5: Locality constraint changes the desired reclamation strategy. Results are based on a synthetic benchmark "Belle". Lower active quantum volume (defined in Section III-B) is better. Belle performs better on a lattice machine with Eager strategy, while preferring Lazy when operating on a fully-connected machine.

better locality, resulting in fewer swap gates and less overall gate noise.

Now we discuss how SQUARE finds the strategies for allocation and reclamation.

1) Locality-Aware Allocation (LAA).: We present the Locality-Aware Allocation (LAA) heuristic in the SQUARE algorithm that prioritizes qubits according to their locations in the machine. At a high level, LAA chooses qubits from the ancilla heap by balancing three main considerations – communication, serialization, and area expansion.

When deciding which qubits to allocate and reuse, our heuristic-based algorithm assigns priorities to all qubits. The priorities are weighted not only by the communication overhead of two-qubit interactions but also by their potential impact to the parallelism of the program. Reusing qubits adds data dependencies to a program and thus serializes computation (which is similar to how reusing register names could lead to false data dependencies and serialization), but not reusing qubits expands the area of active qubits and thus increases the communication overhead between them. Recall from Section II-C1, communication have different tradeoffs under NISQ and FT architectures. We will make this distinction in terms of our heuristics clearer in Section IV-C.

2) Cost-Effective Reclamation (CER).: The Cost-Effective Reclamation (CER) heuristic makes uncomputation decisions with a simple *cost-benefit analysis*: at each potential reclamation point, we estimate and compare two quantities:

- C_1: cost of uncomputation and reclaiming ancilla qubits;
- C_0: cost of no uncomputation and leaving garbage qubits.

CER balances the cost of recursive recomputation and qubit reservation as discussed in Section II-D. To do so, we need an efficient way to accurately estimate the C_1 and C_0 quantities.

In particular, the decision of child function affects not only the cost of itself, but also the cost of parent function. If a child function decides to uncompute, the additional gate costs need to be duplicated should its parent decide to uncompute as well. This was illustrated in Section III-C as the phenomenon we called "recomputation". Thus, we should take the level of the function into account when we make the decision. The total cost of a uncomputation, C_1, can be expressed as:

$$C_1 = N_{active} \times G_{uncomp} \times S \times 2^{\ell} \tag{1}$$

where N_{active} is the number of active qubits, G_{uncomp} is the number of gates for uncomputation (including those in all children functions), ℓ is the level of the child function in the program call graph, and S is the communication factor. Details can be found in Section IV-D.

Now, suppose a function does *not* uncomputing/reclaiming ancilla, the next chance to reclaim them is when its parent function uncomputes. Thus, we want to estimate the cost of holding the ancilla live until the parent's uncompute block is executed. The cost, C_0, can be approximated as:

$$C_0 = N_{anc} \times G_p \times S \times \sqrt{(N_{active} + N_{anc})/N_{active}} \tag{2}$$

where N_{anc} is the number of ancillae held by the function, G_p is the number of gates from the current function to the parent's uncompute function. The term under the square root sign captures the effect of 'area expansion", which we will discuss in greater detail in Section IV-D.

B. Active Quantum Volume

To accurately estimate the workload of a program, we define the active quantum volume (AQV) of a program as:

$$V_A = \sum_{q \in Q} \sum_{(t_i, t_f) \in T_q} (t_f - t_i)$$

where Q is the set of all qubits in the system, and T_q is a sequence of pairs $\{(t_i^0, t_f^0), (t_i^1, t_f^1), \ldots, (t_i^{|T_q|-1}, t_f^{|T_q|-1})\}$. Each pair corresponds to a qubit usage segment, that is we denote t_i^k and t_f^k as the allocation time and reclamation time of the k^{th} time that qubit q is being used, respectively. AQV is high when a large number of qubits stay "live" (in-use) during the execution; thus, the higher the AQV, the more costly it is to execute on that target machine.

The key to this metric is in the term "active". In particular, we exclude the time that a reclaimed qubit spends in the heap from volume calculation, because it has been restored to the $|0\rangle$ state (ground state), which does not suffer from the decoherence noise as an excited state does. Hence, AQV serves as a *minimization objective* in SQUARE. There are a few practical advantages for using AQV over other resource metrics:

1) AQV is a better measure of the exposure to errors than the space$\times$time metric (i.e. number of qubits times circuit depth) [39], [59]. The more time a qubit stays live, the more susceptible it is to noise from its surroundings. We show lower AQV yields higher success rate in Section V-C.
2) Unlike qubit count, gate count, or circuit depth, AQV allows us to more accurately model "liveness" of qubits on a machine (i.e. which qubits are actively carrying information and performing computation as opposed to staying in ground state unused). [60] and [61] shows that keeping a preferred subset of qubits live can boost program success rate.
3) IBM's quantum volume (QV) [19] characterizes the amount of computational resource a quantum device offers, AQV measures the portion of resource being actively utilized by a program on the device.

C. Compilation Tool Flow of SQUARE

In a nutshell, our SQUARE compilation algorithm takes as input a Scaffold program [58] and produces an optimized schedule of all of its quantum instructions. This is accomplished through what is known as the "instrumentation-driven" approach, also used in [62], which allows us to pre-simulate the control flow in a quantum program. This works because all inputs are known at compile time for most quantum programs, so we can use their known control flow to simulate resource usage.

Figure 4 illustrates in detail the compilation flow for SQUARE. It consists of three main components: 1) an easy-to-use syntactical construct compatible within the Scaffold language, 2) a qubit allocation heuristic, and 3) a qubit reclamation heuristic.

Under the hood, an input program first goes through an initial compilation step, where each `Allocate()` and `Free()` instruction is replaced by a classical function call (such as in C/C++) that implements the heuristic algorithm. Each quantum gate is replaced by a classical function that resolves the connectivity constraints of its operand qubits and then schedules the gate to the earliest time step possible. As a result, we have obtained an executable for the classical control flow of the quantum program. The compiler maintains an ancilla heap (i.e. pool of reclaimed qubits) that stores all the reclaimed ancilla qubits. Future allocations can therefore choose to pop from the ancilla heap or initialize brand new qubits. One of the key contributions of our work is a heuristic that makes such decisions.

D. Complexity of SQUARE

SQUARE is a heuristic-based greedy algorithm. It makes allocation and reclamation decisions as they appear in program order. As a result, it takes time that scales linearly to the number of reclamation points in a program. Consider a program with nested functions – all decisions in the callees are made prior to that of the caller, so when deciding for the caller function, the cost of uncomputation is deterministic and easy to estimate. On the flip side, we could end up in a sub-optimal situation where callee's decisions are made neglecting the potential burden for uncomputing its caller.

The computational complexity of qubit reclamation via uncomputation has been studied. It has been shown that, for programs with linear sequential dependency graph, we can use the reversible pebbling game to approach this problem [46]. However, finding the optimal points in a program with hierarchical structure is PSPACE complete [42]. For a program with ℓ levels and d callees per function, there can be as many as d^ℓ possible reclamation points in the worst case. We could be dealing with 2^{d^ℓ} different combinations of reclamation decisions. So clearly, the naïve way for finding the optimal strategy by exhaustively enumerating all possible decisions is far from efficient.

IV. Implementation Details of SQUARE

In this section, we describe the implementation details of the components of SQUARE algorithm, including the expressive syntactical construct in the Scaffold programming language [58] that exposes the optimization opportunities, the instrumentation-driven LLVM [62] that translates the quantum program into a classical executable, and details of the locality-aware allocation heuristics and the cost-effective reclamation heuristics that we left out from Section III-A.

A. Syntactical Construct

In order to express the opportunities for qubit allocation and reclamation optimizations, we augment the high-level Scaffold [58] programming language with an additional syntactical construct: *Compute-Store-Uncompute Code Blocks*. As shown in Figure 6, the keywords "Allocate" and "Free" are used to express the locations of qubit allocation and reclamation

```
#include "qalloc.h"

void fun1(qbit* in, qbit* out) {
  qbit anc[1];
  Allocate(anc, 1);
  Compute {
    Toffoli(in[0], in[1], in[2]);
    CNOT(in[2], anc[0]);
    Toffoli(in[1], in[0], anc[0]);
  }
  Store {
    CNOT(anc[0], out[0]);
  }
  Uncompute{
    // Invoke Inverse() to populate
    // Or write out explicitly:
    Toffoli(in[1], in[0], anc[0]);
    CNOT(in[2], anc[0]);
    Toffoli(in[0], in[1], in[2]);
  }
  Free(anc, 1);
}

int main () {
  qbit new[4]; // declare name
  Allocate(new, 4); // allocate qubits
  fun1(new, &new[3]);
  return 0;
}
```

Fig. 6: Format of *compute-store-uncompute* construct for qubit allocation and reclamation. Shown here an example function (`fun1`) that allocates and reclaims an ancilla qubit.

respectively. To enable automation in the optimizations, the compiler needs additional information about the code structure. By writing a `Compute` code block, the program now has explicitly specified the set of instructions that belong to forward computation. Optionally, programmer can choose to automatically generate the content of the `Uncompute` block by invoking `Inverse()`.

Under the hood, the compiler will replace each `Allocate` and `Free` instruction with an invocation to our heuristic algorithms. Depending on the reclamation decision, it will either execute or skip the uncomputation step accordingly.

B. Instrumentation-Driven Compilation

In this section, we illustrate a number of advantages of the instrumentation-drive approach over the conventional pass-drive approach used in most quantum compilers.

The traditional pass-driven approach for compiling and optimizing quantum programs is done by sending a high-level quantum program through multiple layers of transformations, each of which completes a different task. For instance, we have transformations to resolve classical control structures (e.g. loop unrolling and module inlining), explore circuit optimizations (e.g. commutativity and parallelism), satisfy architectural constraints (e.g. qubit connectivity), assign qubit mappings, and perform gate scheduling, etc. One of the potential limitations in this approach is that each transformation performs independently, and in some cases even conflicts with each other [54]. So it is very hard to jointly optimize for some correlated problems such as mapping and scheduling.

Algorithm	Description
Eager	Reclaim qubits whenever possible, as shown in Section III.
Lazy	Only reclaim qubits from the top level in the program call graph, as shown in Section III.
SQUARE	Combines Locality-aware allocation (LAA) and Cost-effective reclamation (CER). See Section III-A.

TABLE I: List of compiler configurations.

Techniques such as feedback loops could in some cases work well in practice.

Two main reasons that the instrumentation-driven approach may be a more natural fit for our purpose are: the dynamic nature of our optimization and compilation time scalability. Recall from Section III-C, our compilation tool flow produces an executable that allows us to dynamically optimize for the allocation and reclamation of qubits in reversible programs with parallel and modular structures. In the next section, we illustrate the details of our heuristic algorithms and how they are integrated in the compilation tool flow.

C. Allocation Policy Details

The allocation policy is most concerned about the communication overhead of two-qubit operations in a program.

- Under NISQ architecture, communication between two qubits is accomplished by move one qubit to another via a series of swaps. So swap distance is a direct measure of the locality. The higher the distance, the longer it takes for a chain of swaps to complete.
- The concept of locality can be trickier in a FT architecture. Communication is accomplished via braiding. Braids can have arbitrary length or shape, but they are not allowed to cross. As [37] shows, average braid length and average braid spacing are both strongly correlated with the number of braid crossings. So we can reduce communication overhead by moving interacting qubits closer and moving non-interacting qubits far apart.

When there are fewer qubits available than requested (due to either the maximum qubit constraint or a shortage in the ancilla heap), we mark the allocation as pending, and proceed to schedule all non-dependent, parallel computation and reclamation. Allocation requests are not fulfilled until sufficient ancilla qubits have been reclaimed.

D. Reclamation Policy Details

The reclamation policy dictates what and when ancilla qubits get recycled. The decisions rely heavily on three main considerations: qubit savings, uncomputation gate count, and communication overhead. In Section III-A2, we have discussed how SQUARE balances between qubit savings and gate count. Now we present further details on how to estimate the communication factor in Equation 1 and 2.

- NISQ architecture: We use the average swap-chain length per gate as the estimate for S. This is obtained from the history of swap chains during the compile time simulation – we keep a running average of the number of swaps for the gates we scheduled, and use it as an estimate for the subsequent gates in the same module.

- Fault-tolerant (FT) architecture: We use average braiding conflicts per gate as the estimate for S. The communication latency due to braid routing is estimated (similarly as [37]) by factoring in the average braid length, average braid spacing, and number of crossings.

Since "qubit reservation" causes the active qubit area to be expanded, leading to higher communication overhead, the multiplicative factor $\sqrt{(N_{active} + N_{anc})/N_{active}}$ aims to estimate the swap or braid length increase due to the expansion.

Algorithm 1 and 2 are pseudo-code of SQUARE, implementing LAA and CER respectively. Procedures under namespace LLVM are functions that operate on the LLVM IR. In particular, *get_interact_qubits()* obtains the set of qubits with which the allocated qubits interact by looking ahead in the code block. *gen_uncompute_block()* and *rm_uncompute_block()* conditionally expands or deletes the code block under `Uncompute{}` (as shown in Figure 6). *closest_qubit_in_heap()* and *closest_qubit_new()* look for available qubits to reuse from the heap and from new qubits, respectively. Both functions return the candidate qubits and scores. The scores are calculated based on the communication, serialization, and area expansion, as described in Sec IV-C. We select the qubits with minimum scores until the requested n qubits are allocated.

Algorithm 1 Allocate: *Locality-Aware Allocation*

```
Input: Number of qubits $n$
Output: Set of qubits $\mathcal{S}$
 1: $\mathcal{I} \leftarrow$ LLVM::get_interact_qubits()
 2: $\mathcal{S} \leftarrow \emptyset$;
 3: for $i \leftarrow 1$ to $n$ do
 4:     $q_1, score_1 \leftarrow$ closest_qubit_in_heap($\mathcal{I}$)
 5:     $q_2, score_2 \leftarrow$ closest_qubit_new($\mathcal{I}$)
 6:     if $score_1 \leq score_2$ then
 7:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{q_1\}$
 8:     else
 9:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{q_2\}$
10:     end if
11: end for
```

Algorithm 2 Free: *Cost-Effective Reclamation*

```
Input: Number of qubits $n$, Set of qubits $\mathcal{S}$
 1: $C_1 \leftarrow$ cost of uncomputation
 2: $C_0 \leftarrow$ cost of no uncomputation
 3: if $C_1 \leq C_0$ then
 4:     LLVM::gen_uncompute_block()
 5:     heap_push($n, \mathcal{S}$)
 6: else
 7:     LLVM::rm_uncompute_block()
 8:     LLVM::transfer_to_parent($n, \mathcal{S}$)
 9: end if
```

V. Evaluation

A. Benchmarks

Table II lists the QC benchmarks and brief description in our study. These benchmarks are reversible arithmetic functions or applications that use ancilla qubits. Since ancilla qubits are expensive in both NISQ and FT architectures, it is crucial to reuse ancilla qubits and improve the success rate of a program. The first 4 benchmarks (RD53, 6SYM, 2OF5, and ADDER4) are small arithmetic functions suitable for executing on NISQ systems (10 - 100 qubits). The rest of the benchmarks are medium to large functions that are more demanding in computational resources than current NISQ systems can offer. The number of qubits they use, for instance, is on the order of hundreds or thousands. For the last three benchmarks, we construct random synthetic circuits (Jasmine, Elsa, and Belle) with different characteristics in their program structures. In particular, a benchmark is parameterized by the size and shape of its program call graph using 5 variables: number of nested levels, max number of callees per function, max number of input qubits per function, max number of ancilla qubits per function, maximum number of gates per function.

Name	Description
RD53	Input weight function with 5 inputs and 3 outputs.
6SYM	Function with 6 inputs and 1 output.
2OF5	Output is 1 if number of 1s in its input equals two.
ADDER4	4-bit in-place controlled-addition[3].
Jasmine-s	Small and shallow instance of synthetic benchmark Jasmine.
Elsa-s	Small and shallow instance of synthetic benchmark Elsa.
Belle-s	Small and shallow instance of synthetic benchmark Belle.
ADDER32	32-bit in-place controlled-addition.
ADDER64	64-bit in-place controlled-addition.
MUL32	32-bit out-of-place controlled-multiplier.
MUL64	64-bit out-of-place controlled-multiplier.
MODEXP	Modular exponentiation function[4].
SHA2	Cryptographic hash function[5].
SALSA20	Stream cipher core function[6].
Jasmine	Shallowly nested synthetic function[7].
Elsa	Heavy workload and shallowly nested synthetic function.
Belle	Light workload and deeply nested synthetic function.

TABLE II: Characteristics of benchmark programs.

B. Experimental Setup

All compilation experiments are carried out on Intel Core i7-3960X (3.3GHz, 64GB RAM), implemented in the quantum compiler framework ScaffCC [58] version 4.0. Noise simulations use Intel E5-2680v4 (28-core, 2.4GHz, 64GB RAM), performed using the IBM `Qiskit` software [66]. Table I lists the ancilla reuse algorithms in our study. *Eager* and *Lazy* are two baselines that appear commonly in prior work. *SQUARE* is our Strategic QUantum Ancilla REuse algorithm.

[3]The adders are based on the Cucarro adder [63], [64].

[4]Modular exponentiation is an important subroutine used in Shor's factoring algorithm [16].

[5]SHA2 contains multiple rounds of in-place modular additions and bit rotations, based on the implementation from [24]. When used as an oracle in Grover's algorithm [25], we can find hash collisions more efficiently, and thereby reduce the security of the hash function.

[6]Salsa20 involves 20 rounds of 4 parallel modules. Each module modifies 4 words with modular additions, XOR operations, and bit rotations. The Salsa20 stream cipher uses the Salsa20 core function to encrypt data. [65] Salsa family functions have been popularly adopted for TLS in places like the Chrome browser and OpenSSH.

[7]Qubits and gates are randomly assigned.

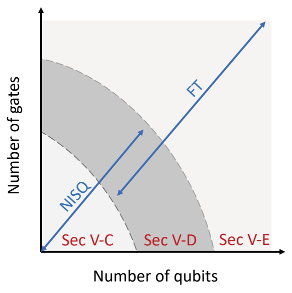


Fig. 7: QC architecture boundary.

The rest of the section are divided up into three main parts (Figure 7) – experimental results on NISQ architecture (Section V-C) with 2D lattice of physical qubits and nearest-neighbor connectivity as commonly used in [3], [67], [68], on NISQ-FT boundary architecture (Section V-D) with same architecture model but on larger benchmarks, and on FT architecture (Section V-E) with surface code error corrected logical qubits [37], [69].

C. NISQ Experiments

Although SQUARE was initially designed to improve the performance of large-scale applications, we find that reclaiming ancilla reduces program footprint and thus swap count due to communication on NISQ machines. In this section, we give analytic and noise simulation results that quantify the fidelity gains from this reduced swap count. To make noise simulation tractable, we focus on small benchmarks and introduce small versions of our 3 synthetic benchmarks as in Table II.

1) AQV Analysis: For our NISQ benchmarks, Figure 8a and Table III show the characteristics of the compiled QC programs with different compiling policy. With the Eager compiling policy, the programs use the fewest qubits, but it may cost too many gates to reuse the ancilla qubits. SQUARE finds the balance between qubit uses and gate costs. We show the AQV comparison in Figure 8a. The AQV is reduced when we apply LAA that allocates the closest qubits, reducing the number swaps. When full SQUARE is applied, AQV is further reduced because of reduction in uncompute cost.

2) Program Success Rate by Analytical Model: Program success rates in our evaluation are estimated by a worst-case analysis using qubit decoherences and gate errors. Multiplying the single-qubit/two-qubit gate success rates and the probability of qubit coherence from Table IV, we observe an average improvement by 1.47X w.r.t Eager and 1.07X w.r.t. Lazy. With strategic uncompuptation by SQUARE, programs use fewer qubits and improve overall chance of success. In reality, this worst case analysis may neglect program structures and noise cancellation. Results are even more positive in the next section where we perform noise simulation.

3) Noise Simulations: All simulations in our evaluation use IBM `Qiskit Aer` simulator [66] with noise models from the `qiskit.providers.aer.noise` library – `depolarize_noise` for single-qubit and two-qubit gate noises, and `thermal_relaxation` for T1/T2 relaxations to account for qubit decoherence. Table IV shows the parameters in our simulation, compared against those in real devices. Figure 8c shows the results from simulation; each data point is obtained from 8192 shots of noisy circuit simulation. We use total variation distance d_{TV}, to compare measurement outcomes of noisy circuits with those of ideal ones; it's a common measure for QC experiments [71]–[73]. We observe that SQUARE achieves lowest distance for almost all benchmarks compared to Eager or Lazy.

Benchmarks	Policy	# Gates[a]	# Qubits	Circuit Depth	# Swaps
RD53	Lazy	536	19	395	462
	Eager	1064	10	878	633
	SQUARE	932	11	635	370
6SYM	Lazy	648	19	456	654
	Eager	1293	11	1279	1247
	SQUARE	1078	12	731	520
2OF5	Lazy	708	18	723	759
	Eager	1410	8	2374	1728
	SQUARE	1176	10	952	385
ADDER4	Lazy	656	18	787	725
	Eager	1184	12	1139	748
	SQUARE	920	14	715	421
Jasmine-s	Lazy	275	16	232	73
	Eager	1226	5	1055	327
	SQUARE	510	8	427	128
Elsa-s	Lazy	163	15	787	725
	Eager	501	8	438	163
	SQUARE	254	13	223	85
Belle-s	Lazy	220	14	202	69
	Eager	712	6	574	113
	SQUARE	294	9	266	89

[a] Here # Gates does not include swap gates (listed in a separate column).

TABLE III: NISQ benchmarks compilation results.

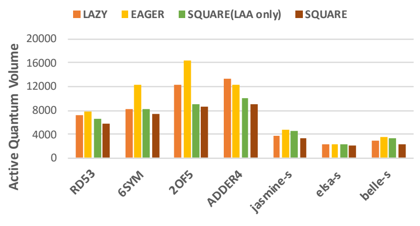
	# Qubits	ϵ_{single}	ϵ_{two}	T1 (μs)	T2 (μs)
IBM-Sup [3], [70]	20	$< 1\%$	$< 2\%$	55	60
IonQ-Trap [33]	79	$< 1\%$	$< 2\%$	$> 10^6$	$> 10^6$
Our Simulation	< 20	0.1%	1%	50	70

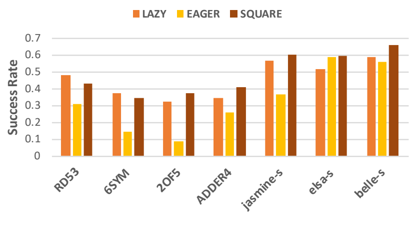
TABLE IV: Error rates on real devices and noise models on our simulation.

4) Applicability of SQUARE to NISQ Machines: Table III and Figure 8b together show the impact of uncomputation on circuit fidelity. SQUARE finds a balanced middle-ground between qubit savings and gate costs by strategically uncomputing its functions. Surprisingly, when comparing Lazy with SQUARE, the additional gates for uncomputation *reduces* the total number of operations, thanks to a substantial reduction in swap gates, as ancilla qubits with better locality are actively reclaimed and reused. Uncomputation also dis-entangles garbage qubits from output qubits, preventing noise from propagating. Furthermore, SQUARE retains most of the qubit savings as Eager does. Overall, SQUARE achieves high success rate using fewer qubits than Lazy.

D. NISQ-FT Boundary Experiments

The boundary between NISQ and fault-tolerant architectures are far from clear. For completeness, we analyzed the performance of the SQUARE algorithm assuming medium-scale machines (with 100-10000 qubits) is built without error


(a) Active quantum volume. (Lower AQV is better.)


(b) Worst-case analytical model. (Higher success rate is better.)

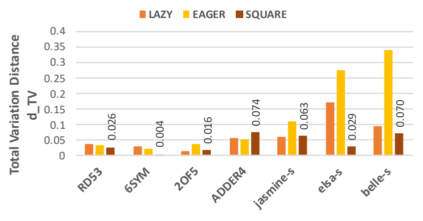

(c) Realistic noise simulation using IBM `Qiskit Aer` simulator. (Lower total variation distance is better.)

Fig. 8: Impact of SQUARE optimizations on NISQ applications. All benchmarks use fewer than 20 qubits; SQUARE stands out as a strategy that uses substantially fewer qubits while maintaining high application success rate.

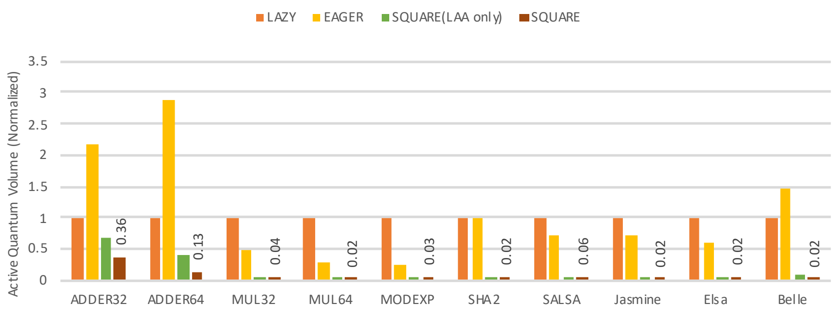


Fig. 9: AQV results on medium-scale non-error-corrected quantum systems. Numbers on the chart correspond to the normalized AQV values of the SQUARE algorithm.

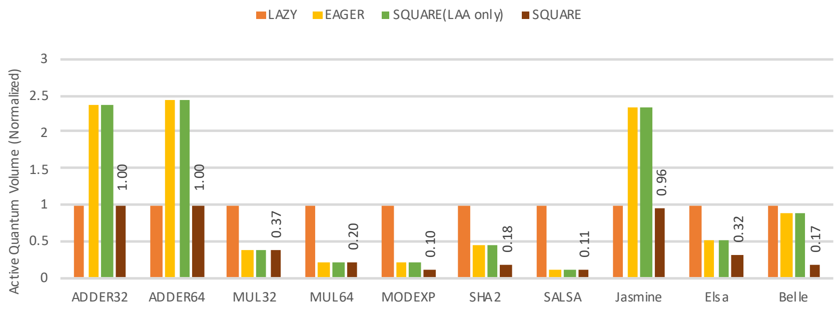


Fig. 10: AQV results on fault-tolerant quantum systems.

correction. Figure 9 shows the AQV results with different compiling policies, and the normalized AQVs of SQUARE are labeled. We observe significant AQV savings by SQUARE, reducing the AQV by a factor of 6.9X on average when compared to the Lazy policy.

E. *Fault-Tolerant (FT) Experiments*

The FT experiments share the set of benchmarks used in the NISQ-FT experiments, but use braiding for communication. To do so, we build and integrate a braid simulator in SQUARE to precisely calculate the communication overhead for executing a program on a surface-code error-corrected architecture.

Following prior work [37]–[39], we assume logical qubits on the surface are laid out in a 2-D array, with sufficient distance between qubits. The separation between qubits serves as channels, allowing other qubits to braid through. So in our simulator, we associate one site per qubit and channels wide enough for a single qubit to braid through. Furthermore, different single-qubit gates have different time cost.

We substitute the swap-chain generation procedure in the SQUARE's gate scheduler with a braid generation procedure. In particular, when a `CNOT` gate is scheduled, we first find a route between the operand qubits, and then check if it crosses with other ongoing braids. It is queued until its route has been cleared.

As shown in Figure 10, SQUARE significantly reduces AQV in all applications under the FT system environment. Comparing to Lazy policy, SQUARE achieve 44.08% AQV reduction on average, and up to 89.66% reduction.

VI. Conclusion

We have presented an automated compilation tool flow that manages the allocation and reclamation of qubits in reversible program with modular structures. We choose a dynamic heuristic-based approach to tackle the challenges, proving how we can use the knowledge of qubit locality and program structure to our advantage to efficiently compile high-level arithmetic for a resource-constrained machine. That is accomplished by SQUARE via cost-effective uncomputation. In this process, we introduce a resource metric, AQV, that quantifies the amount of resource utilized by a given computational task. It allows us to measure and compare the effectiveness of various compiler optimization designs.

The core of our optimization tool flow is the allocation and reclamation heuristics, which predict the cost of uncomputation based on information such as qubit savings, gate overheads, potential reuse, and decisions in children modules in the program call graph. Our methodology is shown to be effective on a suite of benchmarks, including common arithmetic functions and synthetic programs with arbitrary structures. We evaluate SQUARE on NISQ systems and FT systems. The results show that our study has practical value for not only current NISQ devices but also future FT systems.

Our work bridges qubit reclamation and classical register allocation, which allows us to adapt ideas in the heuristic design from classical literature. Much remains to be explored – what other intuitions from classical compilation can be used to optimize qubit allocation and reclamation.

This work relies on heuristics to seek a balance between minimal ancilla usage and minimal gate complexity. It remains an interesting open problem on whether the strategy can achieve information-theoretical lower bound asymptotically. Such asymptotic analysis has been explored at gate level (such as for multi-control not gate) or for some small arithmetic functions (such as adders); it would be a natural extension to study the asymptotic behavior of various uncomputation strategies at systems level (e.g. with communication costs).

Acknowledgment

This work is funded in part by EPiQC, an NSF Expedition in Computing, under grants CCF-1730449/1730082; in part by STAQ, under grant NSF Phy-1818914; and in part by DOE grants DE-SC0020289 and DE-SC0020331. We thank Kenneth Brown, Isaac Chuang, Yipeng Huang, Ali Javadi-Abhari, and Frank Mueller for valuable discussions. We also thank the anonymous reviewers for their useful comments.

References

[1] S. Debnath, N. M. Linke, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe, "Demonstration of a small programmable quantum computer with atomic qubits," *Nature*, vol. 536, no. 7614, p. 63, 2016.

[2] "A Preview of Bristlecone, Google's New Quantum Processor," https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html, accessed: 2019-03-29.

[3] "IBM Announces Advances to IBM Quantum Systems and Ecosystem," https://www-03.ibm.com/press/us/en/pressrelease/53374.wss, accessed: 2019-03-29.

[4] "CES 2018: Intel's 49-Qubit Quantum Supremacy," https://spectrum.ieee.org/tech-talk/computing/hardware/intels-49qubit-chip-aims-for-quantum-supremacy., accessed: 2019-03-29.

[5] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[6] C. H. Bennett, D. P. DiVincenzo, J. A. Smolin, and W. K. Wootters, "Mixed-state entanglement and quantum error correction," *Physical Review A*, vol. 54, no. 5, p. 3824, 1996.

[7] D. Gottesman, "An introduction to quantum error correction and fault-tolerant quantum computation," in *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, vol. 68, 2010, pp. 13–58.

[8] K. Bertels, I. Ashraf, R. Nane, X. Fu, L. Riesebos, S. Varsamopoulos, A. Mouedenne, H. Van Someren, A. Sarkar, and N. Khammassi, "Quantum computer architecture: Towards full-stack quantum accelerators," *arXiv preprint arXiv:1903.09575*, 2019.

[9] E. T. Campbell and M. Howard, "Unified framework for magic state distillation and multiqubit gate synthesis with reduced resource cost," *Physical Review A*, vol. 95, no. 2, p. 022316, 2017.

[10] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, p. 180, 2017.

[11] L. E. Heyfron and E. T. Campbell, "An efficient quantum compiler that reduces t count," *Quantum Science and Technology*, vol. 4, no. 1, p. 015004, 2018.

[12] A. Paler, I. Polian, K. Nemoto, and S. J. Devitt, "Fault-tolerant, high-level quantum circuits: form, compilation and description," *Quantum Science and Technology*, vol. 2, no. 2, p. 025003, 2017.

[13] D. S. Steiger, T. Häner, and M. Troyer, "Projectq: an open source software framework for quantum computing," *Quantum*, vol. 2, no. 49, pp. 10–22 331, 2018.

[14] D. Wecker and K. M. Svore, "Liqui—¿: A software design architecture and domain-specific language for quantum computing," *arXiv preprint arXiv:1402.4467*, 2014.

[15] M. Soeken, T. Haener, and M. Roetteler, "Programming quantum computers using design automation," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 137–146.

[16] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[17] C. Bennett, "Time/space trade-offs for reversible computation," *SIAM Journal on Computing*, vol. 18, no. 4, pp. 766–776, 1989.

[18] E. Knill, "An analysis of bennett's pebble game," *arXiv preprint math/9508218*, 1995.

[19] L. S. Bishop, S. Bravyi, A. Cross, J. M. Gambetta, and J. Smolin, "Quantum volume," *Quantum Volume. Technical Report*, 2017.

[20] D. Große, R. Wille, G. W. Dueck, and R. Drechsler, "Exact multiple-control toffoli network synthesis with sat techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 703–715, 2009.

[21] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conference, 2003. Proceedings*. IEEE, 2003, pp. 318–323.

[22] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Hierarchical reversible logic synthesis using luts," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 78.

[23] D. Maslov, G. W. Dueck, and D. M. Miller, "Toffoli network synthesis with templates," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 807–817, 2005.

[24] A. Parent, M. Roetteler, and K. M. Svore, "Reversible circuit compilation with space constraints," *arXiv preprint arXiv:1510.00377*, 2015.

[25] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 1996, pp. 212–219.

[26] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical review letters*, vol. 103, no. 15, p. 150502, 2009.

[27] N. Abdessaied, M. Amy, M. Soeken, and R. Drechsler, "Technology mapping of reversible circuits to clifford+ t quantum circuits," in *Multiple-Valued Logic (ISMVL), 2016 IEEE 46th International Symposium on*. IEEE, 2016, pp. 150–155.

[28] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits,"

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 32, no. 6, pp. 818–830, 2013.

[29] V. Kliuchnikov, D. Maslov, and M. Mosca, "Fast and efficient exact synthesis of single qubit unitaries generated by clifford and t gates," *arXiv preprint arXiv:1206.5236*, 2012.

[30] D. Maslov, "Advantages of using relative-phase toffoli gates with an application to multiple control toffoli optimization," *Physical Review A*, vol. 93, no. 2, p. 022311, 2016.

[31] J. Welch, A. Bocharov, and K. M. Svore, "Efficient approximation of diagonal unitaries over the clifford+ t basis," *arXiv preprint arXiv:1412.5608*, 2014.

[32] X. Fu, L. Riesebos, M. Rol, J. van Straten, J. van Someren, N. Khammassi, I. Ashraf, R. Vermeulen, V. Newsum, K. Loh *et al.*, "eqasm: An executable quantum instruction set architecture," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 224–237.

[33] "IonQ harnesses single-atom qubits to build the world's most powerful quantum computer," https://ionq.co/news/december-11-2018, accessed: 2019-03-29.

[34] "Rigetti Announces Its Hybrid Quantum Computing Platform and a 1m Prize," https://techcrunch.com/2018/09/07/rigetti-announces-its-hybrid-quantum-computing-platform-and-a-1m-prize/, accessed: 2019-03-29.

[35] C. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, no. 12, p. 123011, 2012.

[36] D. Litinski and F. v. Oppen, "Lattice surgery with a twist: Simplifying clifford gates of surface codes," *Quantum*, vol. 2, 2018.

[37] Y. Ding, A. Holmes, A. Javadi-Abhari, D. Franklin, M. Martonosi, and F. Chong, "Magic-state functional units: Mapping and scheduling multi-level distillation circuits for fault-tolerant quantum architectures," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 828–840.

[38] A. Javadi-Abhari, P. Gokhale, A. Holmes, D. Franklin, K. R. Brown, M. Martonosi, and F. T. Chong, "Optimized surface code communication in superconducting quantum computers," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 692–705.

[39] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, no. 3, p. 032324, 2012.

[40] C. H. Bennett, "Logical reversibility of computation," *IBM journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973.

[41] H. Buhrman, J. Tromp, and P. Vitányi, "Time and space bounds for reversible simulation," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2001, pp. 1017–1027.

[42] S. M. Chan, M. Lauria, J. Nordstrom, and M. Vinyals, "Hardness of approximation in pspace and separation results for pebble games," in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. IEEE, 2015, pp. 466–485.

[43] M. P. Frank and T. F. Knight Jr, "Reversibility for efficient computing," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1999.

[44] B. Komarath, J. Sarma, and S. Sawlani, "Pebbling meets coloring: Reversible pebble game on trees," *Journal of Computer and System Sciences*, vol. 91, pp. 33–41, 2018.

[45] M. Amy, M. Roetteler, and K. M. Svore, "Verified compilation of space-efficient reversible circuits," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 3–21.

[46] G. Meuli, M. Soeken, M. Roetteler, N. Bjorner, and G. De Micheli, "Reversible pebbling game for quantum memory management," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 288–291.

[47] A. Paler, A. G. Fowler, and R. Wille, "Faster manipulation of large quantum circuits using wire label reference diagrams," *Microprocessors and Microsystems*, vol. 66, pp. 55–66, 2019.

[48] A. Paler, R. Wille, and S. J. Devitt, "Wire recycling for quantum circuit optimization," *Physical Review A*, vol. 94, no. 4, p. 042337, 2016.

[49] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 283–298.

[50] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 428–455, 1994.

[51] G. Chaitin, "Register allocation & spilling via graph coloring," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 98–105.

[52] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 895–913, 1999.

[53] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 311–321, 1992.

[54] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 88–98.

[55] S. S. Pinter, "Register allocation with instruction scheduling," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 248–257. [Online]. Available: http://doi.acm.org/10.1145/155090.155114

[56] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for gpus," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 395–406. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830813

[57] S. Aaronson, "Quantum lower bound for recursive fourier sampling," *Quantum Information & Computation*, vol. 3, no. 2, pp. 165–174, 2003.

[58] A. Javadi-Abhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: A framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/2597917.2597939

[59] A. Holmes, Y. Ding, A. Javadi-Abhari, D. Franklin, M. Martonosi, and F. T. Chong, "Resource optimized quantum architectures for surface code implementations of magic-state distillation," *Microprocessors and Microsystems*, vol. 67, pp. 56–70, 2019.

[60] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019.

[61] S. S. Tannu and M. K. Qureshi, "A case for variability-aware policies for NISQ-era quantum computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019.

[62] J. Heckey, S. Patil, A. JavadiAbhari, A. Holmes, D. Kudrow, K. R. Brown, D. Franklin, F. T. Chong, and M. Martonosi, "Compiler management of communication and parallelism for quantum computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 445–456. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694357

[63] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," *arXiv preprint quant-ph/0410184*, 2004.

[64] I. L. Markov and M. Saeedi, "Constant-optimized quantum circuits for modular multiplication and exponentiation," *arXiv preprint arXiv:1202.6614*, 2012.

[65] D. J. Bernstein, "The salsa20 family of stream ciphers," in *New stream cipher designs*. Springer, 2008, pp. 84–97.

[66] H. Abraham, I. Y. Akhalwaya, G. Aleksandrowicz, T. Alexander, G. Alexandrowics, E. Arbel, A. Asfaw, C. Azaustre, AzizNgoueya, P. Barkoutsos, G. Barron, L. Bello, Y. Ben-Haim, D. Bevenius, L. S. Bishop, S. Bosch, S. Bravyi, D. Bucher, F. Cabrera, P. Calpin, L. Capelluto, J. Carballo, G. Carrascal, A. Chen, C.-F. Chen, R. Chen, J. M. Chow, C. Claus, C. Clauss, A. J. Cross, A. W. Cross, S. Cross, J. Cruz-Benito, C. Culver, A. D. Córcoles-Gonzales, S. Dague, T. E. Dandachi, M. Dartiailh, DavideFrr, A. R. Davila, D. Ding, J. Doi, E. Drechsler, Drew, E. Dumitrescu, K. Dumon, I. Duran, K. EL-Safty, E. Eastman, P. Eendebak, D. Egger, M. Everitt, P. M. Fernández, A. H. Ferrera, A. Frisch, A. Fuhrer, M. GEORGE, J. Gacon, Gadi, B. G. Gago, J. M. Gambetta, A. Gammanpila, L. Garcia, S. Garion, J. Gomez-Mosquera, S. de la Puente González, I. Gould, D. Greenberg, D. Grinko, W. Guan, J. A. Gunnels, I. Haide, I. Hamamura,

V. Havlicek, J. Hellmers, Ł. Herok, S. Hillmich, H. Horii, C. Howington, S. Hu, W. Hu, H. Imai, T. Imamichi, K. Ishizaki, R. Iten, T. Itoko, A. Javadi-Abhari, Jessica, K. Johns, T. Kachmann, N. Kanazawa, Kang-Bae, A. Karazeev, P. Kassebaum, S. King, Knabberjoe, A. Kovyrshin, V. Krishnan, K. Krsulich, G. Kus, R. LaRose, R. Lambert, J. Latone, S. Lawrence, D. Liu, P. Liu, Y. Maeng, A. Malyshev, J. Marecek, M. Marques, D. Mathews, A. Matsuo, D. T. McClure, C. McGarry, D. McKay, D. McPherson, S. Meesala, M. Mevissen, A. Mezzacapo, R. Midha, Z. Minev, A. Mitchell, N. Moll, M. D. Mooring, R. Morales, N. Moran, P. Murali, J. Müggenburg, D. Nadlinger, G. Nannicini, P. Nation, Y. Naveh, P. Neuweiler, P. Niroula, H. Norlen, L. J. O'Riordan, O. Ogunbayo, P. Ollitrault, S. Oud, D. Padilha, H. Paik, S. Perriello, A. Phan, M. Pistoia, A. Pozas-iKerstjens, V. Prutyanov, D. Puzzuoli, J. Pérez, Quintiii, R. Raymond, R. M.-C. Redondo, M. Reuter, J. Rice, D. M. Rodríguez, M. Rossmannek, M. Ryu, T. SAPV, SamFerracin, M. Sandberg, N. Sathaye, B. Schmitt, C. Schnabel, Z. Schoenfeld, T. L. Scholten, E. Schoute, J. Schwarm, I. F. Sertage, K. Setia, N. Shammah, Y. Shi, A. Silva, A. Simonetto, N. Singstock, Y. Siraichi, I. Sitdikov, S. Sivarajah, M. B. Sletfjerding, J. A. Smolin, M. Soeken, I. O. Sokolov, SooluThomas, D. Steenken, M. Stypulkoski, J. Suen, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, S. Thomas, M. Tillet, M. Tod, E. de la Torre, K. Trabing, M. Treinish, TrishaPe, W. Turner, Y. Vaknin, C. R. Valcarce, F. Varchon, A. C. Vazquez, D. Vogt-Lee, C. Vuillot, J. Weaver, R. Wieczorek, J. A. Wildstrom, R. Wille, E. Winston, J. J. Woehr, S. Woerner, R. Woo, C. J. Wood, R. Wood, S. Wood, J. Wootton, D. Yeralin, R. Young, J. Yu, C. Zachow, L. Zdanski, C. Zoufal, Zoufalc, azulehner, bcamorrison, brandhsn, chlorophyll zz, dan1pal, dime10, drholmie, elfrocampeador, faisaldebouni, fanizzamarco, gruu, kanejess, klinvill, kurarrr, lerongil, ma5x, merav aharoni, ordmoj, sethmerkel, strickroman, sumitpuri, tigerjack, toural, vvilpas, welien, willhbang, yang.luh, yelojakit, and yotamvakninibm, "Qiskit: An open-source framework for quantum computing," 2019.

[67] A. Holmes, S. Johri, G. G. Guerreschi, J. S. Clarke, and A. Matsuura, "Impact of qubit connectivity on quantum algorithm performance," *arXiv preprint arXiv:1811.02125*, 2018.

[68] A. Shafaei, M. Saeedi, and M. Pedram, "Qubit placement to minimize communication overhead in 2d quantum architectures," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014, pp. 495–500.

[69] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. C. White, J. Mutus, A. G. Fowler, B. Campbell *et al.*, "Superconducting quantum circuits at the surface code threshold for fault tolerance," *Nature*, vol. 508, no. 7497, p. 500, 2014.

[70] "Coherence times are obtained from daily calibration data from ibm q experience."

[71] C. Bădescu, R. O'Donnell, and J. Wright, "Quantum state certification," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, 2019, pp. 503–514.

[72] A. Lund, M. J. Bremner, and T. Ralph, "Quantum sampling problems, bosonsampling and quantum supremacy," *npj Quantum Information*, vol. 3, no. 1, pp. 1–8, 2017.

[73] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.