Q-GPU: A Recipe of Optimizations for Quantum Circuit Simulation Using GPUs

Yilun Zhao[1], Yanan Guo[2], Yuan Yao[1], Amanda Dumi[3], Devin M Mulvey[3], Shiv Upadhyay[3],
Youtao Zhang[1], Kenneth D Jordan[3], Jun Yang[2], and Xulong Tang[1]
[1]*Department of Computer Science, University of Pittsburgh, Pittsburgh, USA*
[2]*Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, USA*
[3]*Department of Chemistry, University of Pittsburgh, Pittsburgh, USA*
{*yiz178,yag45,yuy56,amandadumi,dmm219,shivupadhyay,zhangyt,jordan,juy9,tax6*}*@pitt.edu*

***Abstract*—In recent years, quantum computing has undergone significant developments and has established its supremacy in many application domains. While quantum hardware is accessible to the public through the cloud environment, a robust and efficient quantum circuit simulator is necessary to investigate the constraints and foster quantum computing development, such as quantum algorithm development and quantum device architecture exploration. In this paper, we observe that most of the publicly available quantum circuit simulators (e.g., QISKit from IBM, QDK from Microsoft, and Qsim-Cirq from Google) suffer from slow simulation and poor scalability when the number of qubits increases. To this end, we systematically investigate the deficiencies in quantum circuit simulation (QCS) and propose Q-GPU, a framework that leverages GPUs with comprehensive optimizations to allow efficient and scalable QCS. Specifically, Q-GPU features i) proactive state amplitude transfer, ii) zero state amplitude pruning, iii) delayed qubit involvement, and iv) lossless non-zero state amplitude compression. Experimental results across nine representative quantum circuits indicate that Q-GPU significantly reduces the execution time of the state-of-the-art GPU-based QCS by 71.89% (3.55× speedup). Q-GPU also outperforms the state-of-the-art OpenMP CPU implementation, the Google Qsim-Cirq simulator, and the Microsoft QDK simulator by 1.49×, 2.02×, and 10.82×, respectively.**

***Keywords*-Quantum Circuit Simulation; GPU**

I. Introduction

Quantum computing is a promising computing paradigm that has the potential to solve problems that cannot be handled by classical computers in a feasible amount of time [9]. In the past decade, there has been steady progress towards building a large quantum computer. The number of qubits in a real quantum machine has increased from 14 in 2011 [42] to 127 in 2021 [28]. IBM promises a 1000 qubits quantum machine by the year 2023 [27]. Despite this rapid development, current quantum computing is still positioned in the Noisy Intermediate-Scale Quantum (NISQ) era where the public has limited access to reliable quantum machines. Thus, quantum circuit simulation (QCS) toolsets provide an essential platform satisfying many needs, e.g., developing algorithms with a large number of qubits, validating and evaluating newly proposed quantum circuits, and design space exploration of future quantum machine architectures.

In general, QCS is challenging as it is both compute-intensive and memory-intensive [20], [38]. The reasons are: i) fully and accurately tracking the evolution of quantum system through classical simulation [16] requires storing all the quantum state amplitudes, which carries a memory cost that grows exponentially as the number of qubits in the simulated quantum circuit increases, and ii) applying a gate within a quantum circuit requires a traversal of all the stored state amplitudes, leading to an exponentially increase in computational complexity. Modern GPUs have been used to fuel QCS in high-performance computing (HPC) platforms. Specifically, when applying a gate to a n-qubit quantum circuit, the 2^n state amplitudes are evenly divided into groups, and each group of amplitudes is updated independently in parallel by GPU threads. However, the promising parallelism of GPUs is diminished by the limited GPU memory capacity. For example, simulating a quantum circuit with 34 qubits requires 256 GB of memory for state amplitudes, exceeding the memory capacity of any modern GPUs.

There exist several prior works optimizing QCS, including multi-GPU supported simulation [32], [35], OpenMP and MPI based CPU simulation [24], [48], [55], and CPU-GPU collaborative simulation [17], [18]. Most of these works focus on distributed simulation while failing to exploit the tremendous parallelism provided by GPUs due to the memory constraints. In particular, our characterization indicates that the state-of-the-art GPU-based simulation [17], [18] suffers from low GPU utilization when the number of qubits in the quantum circuit is large. As a result, most state amplitudes are stored and updated on the CPU, failing to take advantage of the GPU parallelization. Moreover, the static and unbalanced allocation of state amplitudes introduces frequent amplitude exchange between CPU and GPU, which introduces significant data movement and synchronization overheads.

In this paper, we aim to achieve high-performance, scalable, and general-purpose QCS using modern GPUs. We propose *Q-GPU*, a framework that significantly enhances the simulation performance for practical quantum circuits. Q-GPU leverages GPUs as the main execution engine and is featured with end-to-end optimizations to fully take advantage of the rich computational parallelism in GPUs, while minimizing the amount of data movement between CPU and GPU. First, instead of statically assigning state amplitudes on the GPU and CPU as done in prior works [18], Q-GPU dynamically allocates groups of state amplitudes on the GPU and proactively exchanges the state amplitudes between CPU and GPU. Doing so maximizes the overlap of data transfer between CPU and GPU, thereby improving the GPU utilization and reducing the GPU idleness. Second, rather than using a single data compression algorithm to compress all state amplitudes [55], we handle zero-valued and non-zero state amplitudes differently and separately. For zero amplitudes, we develop reordering algorithms to greedily select gates that involve the least number of qubits. This is built upon the observation that the less qubits that are involved by any gates, the more zero-valued state amplitudes can be pruned. For non-zero amplitudes, we implement efficient lossless data compression on GPU to further reduce data transfer. This paper makes the following contributions:

- We use the popular IBM QISKit-Aer with its state-of-the-art CPU-GPU implementation [3], and conduct an in-depth characterization of QCS. We observe that the simulation performance degrades significantly as the number of qubits increases due to the severe under-utilization of GPU parallelism. To exploit GPU parallelism, we implement dynamic state amplitude allocation that allows the GPU to update state amplitudes. However, such an intuitive implementation even consumes more execution time due to the massive data movement between CPU and GPU.
- We propose Q-GPU, a framework comprising end-to-end optimizations to mitigate the data movement overheads and unleash the GPU capability in QCS. Specifically, Q-GPU is featured with the following major optimizations: i) dynamic state amplitudes allocation and proactive data exchange between CPU and GPU, ii) dynamic zero state amplitude pruning, iii) dependency-aware quantum gate reordering to enlarge the potential of pruning, and iv) efficient GPU-supported lossless compression for non-zero amplitudes.
- We evaluate the proposed Q-GPU framework using nine practical quantum circuits. Experimental results indicate that in all circuits tested, Q-GPU significantly improves the QCS performance. On average, it achieves 3.55× speedup over the baseline. We also compare Q-GPU with Google Qsim-Cirq [54] and Microsoft QDK [41], and results show that Q-GPU approach achieves 2.02× and 10.82× speedups over Qsim-Cirq and QDK, respectively.
- We also test Q-GPU in both PCIe and NVLink connected multi-GPU environments and the results show 2.97× and 2.98× speedups over the state-of-the-art multi-GPU QCS. The Q-GPU simulator is released at: *https://github.com/Zhaoyilunnn/q-gpu*.

II. Background

A. Quantum Basics

Quantum computation is built upon the *quantum bit* or *qubit* for short [44]. A qubit is a two-level quantum system defined by two computational orthonormal basis states $|0\rangle$ and $|1\rangle$. A quantum state $|\psi\rangle$ can be expressed by any linear combination of the basis states.

$$|\psi\rangle = a_0|0\rangle + a_1|1\rangle, \tag{1}$$

where a_0 and a_1 are complex numbers whose squares represent the probability amplitudes of basis states $|0\rangle$ and $|1\rangle$, respectively. Note that we have $|a_0|^2 + |a_1|^2 = 1$, meaning that after measurement, the readout of state $|\psi\rangle$ is either $|0\rangle$ or $|1\rangle$, with probabilities $|a_0|^2$ and $|a_1|^2$, respectively. The states of a quantum system are generally represented by *state vectors* as

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}. \tag{2}$$

For an n-qubit system, there are 2^n state amplitudes. Then, the quantum state $|\psi\rangle$ can be expressed as a linear combination

$$|\psi\rangle = a_{0\ldots00}|0\ldots00\rangle + a_{0\ldots01}|0\ldots01\rangle + \cdots + a_{1\ldots11}|1\ldots11\rangle. \tag{3}$$

Similarly, the state of a n-qubit system can also be represented by a state vector with 2^n dimensions as

$$|\psi\rangle = a_{0\ldots00}\begin{bmatrix}1\\0\\ \vdots\\0\end{bmatrix} + a_{0\ldots01}\begin{bmatrix}0\\1\\ \vdots\\0\end{bmatrix} + \cdots + a_{1\ldots11}\begin{bmatrix}0\\0\\ \vdots\\1\end{bmatrix} = \begin{bmatrix}a_{0\ldots00}\\a_{0\ldots01}\\ \vdots\\a_{1\ldots11}\end{bmatrix}. \tag{4}$$

Quantum computation describes changes occurring in this state vector. A quantum computer is built upon a *quantum circuit* containing *quantum gates* (or quantum operations), and a quantum algorithm is described by a specific quantum circuit. A quantum gate that acts on k qubits is represented by a $2^k \times 2^k$ unitary matrix.

Let us consider a 2-qubit system with a Hadamard gate/operation operating on qubit 0. A Hadamard gate can be represented as

$$H \equiv \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}. \tag{5}$$

Then the state vector of this 2-qubit system is updated through

$$\begin{bmatrix}a'_{00}\\a'_{01}\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}a_{00}\\a_{01}\end{bmatrix}, \tag{6}$$

$$\begin{bmatrix}a'_{10}\\a'_{11}\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}a_{10}\\a_{11}\end{bmatrix}. \tag{7}$$

For an n-qubit system, when a H gate is applied to qubit j the amplitudes are transformed as [15]:

$$\begin{bmatrix}a'_{\times\cdots\times0_j\times\cdots\times}\\a'_{\times\cdots\times1_j\times\cdots\times}\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}\begin{bmatrix}a_{\times\cdots\times0_j\times\cdots\times}\\a_{\times\cdots\times1_j\times\cdots\times}\end{bmatrix} \tag{8}$$

Therefore, the indices of every pair of amplitudes have either 0 or 1 in the jth bit, while all other bits remain the same[1]. Note that each pair of amplitudes can be updated in parallel.

B. Quantum Circuit Simulation (QCS)

There are several approaches to simulating a quantum circuit, each featuring different advantages and drawbacks. We summarize the three most widely used approaches below.

- **Schrödinger style simulation:** Schrödinger simulation describes the evolution of a quantum system by tracking its quantum state. It tracks the transformations of the state vector according to Equation 8. Note that one can also track the density matrix $\rho = |\psi\rangle\langle\psi|$, which is useful when measurement is required during simulation [16], [32]. In this work, we only consider quantum measurements at the end of circuits.
- **Stabilizer formalism:** Simulation based on the stabilizer formalism is efficient for a restricted class of quantum circuits [1], [16], [44]. Specifically, stabilizer circuits (a.k.a Clifford circuits) can be simulated in $O(poly(n))$ space and time costs. Rather than tracking the state vector, the quantum state is uniquely represented and tracked by its stabilizers, which is essentially a group of operators derived from the Clifford group. A detailed description can be found in [1].
- **Tensor network:** Tensor network simulators are useful when a single or few amplitudes of the full state vector are being updated as tensor networks [26], [37], [39], [40]. For example, one type of tensor network that are extremely common are matrix product states (MPS). When applied to a single amplitude in Equation 3, the resulting state resembles a long string of matrix multiplications

$$\begin{aligned} |\psi\rangle &= \sum_{j_0 \cdots j_{n-1} j_n} a_{j_0 \cdots j_{n-1} j_n} |j_0 \dots j_{n-1} j_n\rangle \\ &= \sum_{j_0 \cdots j_{n-1} j_n} Tr[A^{j_0} \dots A^{j_{n-1}} A^{j_n}] |j_0 \dots j_{n-1} j_n\rangle \end{aligned} \tag{9}$$

  The matrices A (rank-2 tensors) in Equation 9 can be thought of as a decomposition of the full coefficient tensor a. Despite the restriction of returning a limited number of amplitudes, tensor networks states are efficient as they compress the dimension of the problem from $O(2^n)$ to $O(nd^2)$ where d is the dimension of the individual tensors in Equation 9.

Among all these simulation methods, Schrödinger style simulation is widely used as the mainstream simulation method [4], [15], [18], [20], [24], [34], [48], [49], [52], [55]. Also, industrial quantum circuit simulators such as IBM QISKit [3], Google Qsim-Cirq [24], [52] and Microsoft QDK [41] use full state vector simulations. Currently, there is no GPU support in Qsim-Cirq and Microsoft QDK. Therefore, we build Q-GPU using QISKit-Aer as it has the state-of-the-art GPU support.

[1]"$\times$" can be 0 or 1; the "$\times$" in the same position of $a_{\times\times\times\times\times\times\times 0}$ and $a_{\times\times\times\times\times\times\times 1}$ are the same.

Table I: List of quantum circuit benchmarks.

Abbrv.	Application
`hchain`	Linear hydrogen atom chain [53]
`rqc`	Random quantum circuit [9]
`qaoa`	Quantum approximate optimization algorithm [19]
`gs`	Graph state [25], [31]
`hlf`	Hidden linear function [11]
`qft`	Quantum Fourier transform [30]
`iqp`	Instantaneous quantum polynomial-time [12], [13]
`qf`	Quadratic form [21]
`bv`	Bernstein-Vazirani algorithm [8]

III. Characterization of QCS

A. Quantum Circuit Benchmarks

In this paper, we target 9 representative quantum circuits as listed in Table I.

- `hchain`: This circuit is a representative quantum chemistry application which describes a system of hydrogen atoms arranged linearly [2], [7], [23], [43], [50]. This circuit incorporates increased circuit depth and an early entanglement in terms of total operations.
- `rqc`: The random quantum circuit from Google [9], [10] is used to represent the quantum supremacy compared to classical computers.
- `qaoa`: Quantum approximate optimization is a promising quantum algorithm in the NISQ era that produces approximate solutions for combinatorial optimization problems [19].
- `gs`: This circuit is used to prepare graph states [5] that are multi-particle entangled states. Examples include many-body spin states of distributed quantum systems that are important in quantum error correction [36].
- `hlf`: This benchmark circuit solves the 2D hidden linear function problem [11].
- `qft`: The quantum Fourier transform circuit [30] is the quantum analog of the inverse discrete Fourier transform. It is an important function in Shor's algorithm [51].
- `iqp`: The instantaneous quantum polynomial circuit provides evidence that sampling the output probability distribution of a quantum circuit is difficult when using classical approaches [12], [13].
- `qf`: This circuit implements a quadratic form on binary variables encoded in qubit registers. It is used to solve the quadratic unconstrained binary optimization problems [21].
- `bv`: This circuit implements the algorithm to solve Bernstein-Vazirani problem [8].

B. Baseline QCS

In this paper, we use the popular IBM QISKit-Aer simulator. We consider the state-of-the-art GPU-supported simulation [18] in QISKit-Aer as our baseline. We run all simulations on a server with dual 10-core Intel Xeon Silver 4114 CPUs at 2.2 GHz, 384 GB of memory, and an NVIDIA

P100 GPU with 16 GB of memory connected through PCI-e. We have also evaluated Q-GPU using NVIDIA A100 and V100 in Section V-D, and multi-GPU in Section V-E. We use CUDA v10 and Nvprof [45] to conduct our characterization. The baseline simulates quantum computations by iteratively executing gate on the state vector. The actual computations are mainly vector-matrix multiplications in the form of Equation 8. The simulation in QISKit-Aer has three key steps: 1) state vector partitioning, 2) static chunk allocation, and 3) reactive chunk exchange.

Step 1: State vector partitioning: QISKit-Aer baseline first partitions the state vectors into "chunks". Chunk is the granularity used in the simulator to update the state vector. For illustrative purposes, let us assume we have a 7-qubit circuit, i.e., there are in total 2^7 different state amplitudes from $a_{0000000}$ to $a_{1111111}$. All the states are stored in a vector (i.e., the state vector), and this state vector is partitioned into 8 chunks. Each chunk contains 16 state amplitudes as shown in Figure 1. The three most significant bits are used to index the chunks, and the remaining 4 bits are offsets within a chunk.

Step 2: Static chunk allocation: After partitioning, these chunks are allocated in GPU memory based on the capacity. As illustrated in Figure 1, if a GPU can only store 3 chunks, the remaining 5 chunks will be stored in the host CPU memory. For example, when 64 GB memory is needed to simulate 32 qubits, the first 16 GB is allocated in the P100 GPU and the remaining 48 GB is in the CPU memory.

Step 3: Reactive chunk exchange: During circuit simulation, a chunk exchange between the GPU and the CPU arises when the requested state amplitudes are not locally available on the GPU. In QISKit-Aer, the chunk exchange between the CPU and the GPU is triggered on-demand. That is, when both the chunks on the CPU and the GPU are involved in one state update, the corresponding CPU chunks are transferred to GPU for updating. After the operation, the updated chunks are transferred back to the CPU. Note that, the amount of data exchange in the following scenarios depends on the qubits in the specific gate simulation.

- **Case 1: All the indices of the qubits involved in the current gate are smaller than the chunk size:**. For example, a gate on qubit 0 requires amplitudes $a_{\times\times\times\times\times\times0}$ and $a_{\times\times\times\times\times\times1}$ (see Equation 8). In this case, each chunk can be updated independently without requiring extra data movement.
- **Case 2: Some indices of qubits involved in the current gate are outside the chunk boundary:** In this scenario, let us assume there is a gate that operates on q_6, thereby the required pairs of amplitudes are $a_{\times0\times\times\times\times\times}$ and $a_{\times1\times\times\times\times\times}$. However, as depicted in Figure 1, none of the chunks contains a pair of required amplitudes, i.e., the computation for updating amplitudes involves more than one chunk. Specifically, to update the pairs of amplitudes, we need ($chunk_0$, $chunk_2$), ($chunk_1$, $chunk_3$), …, and ($chunk_5$, $chunk_7$). However, ($chunk_1$, $chunk_3$) involves one chunk on the GPU and one chunk on the CPU. In this scenario, data exchange is required. In the baseline QISKit-Aer simulation, the requested chunks are always copied from CPU to GPU. That is, in the example above, the CPU copies $chunk_3$ to GPU. After the $chunk_3$ is updated together with $chunk_1$, it is copied back to the CPU memory.

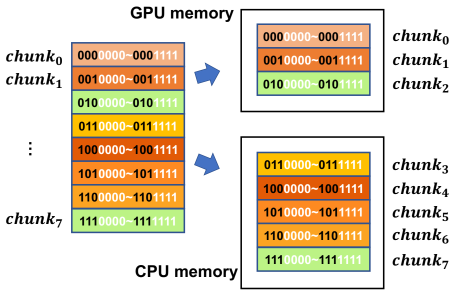


Figure 1: Example of baseline execution where the state vector is statically partitioned and allocated on CPU and GPU.

Note that as the GPU memory capacity is less compared to the CPU host memory, a large number of chunks are statically allocated on CPU memory when the number of qubits is large. For instance, on the P100 GPU with 16 GB memory, we observe from experiments that when simulating a circuit that has 34 qubits, the state vector is divided into 8192 chunks, 496 chunks are allocated on GPU, while the remaining 7696 chunks are all on CPU. Therefore, one can expect that most of the time, the CPU does the state amplitude update without taking advantage of the GPU acceleration.

C. *Characterization and Observations*

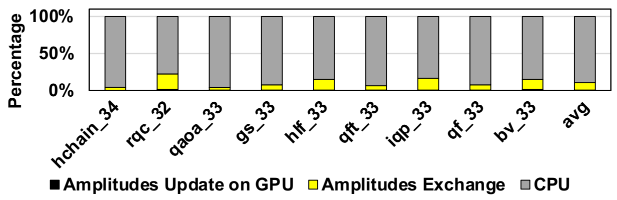


Figure 2: Baseline execution time breakdown.

In this section, we quantify the simulation performance of the baseline QISKit-Aer. We first study the scalability when the number of qubits increases. We observe that, if there are less than 30 qubits in the circuit, the baseline GPU is much faster than compared CPU-based simulation (e.g. 9.67× speedup for 29-qubit circuits on average), since the entire state vector fits in the P100 GPU memory and there is no need for data exchange and synchronization. However, the baseline GPU performance significantly drops when the

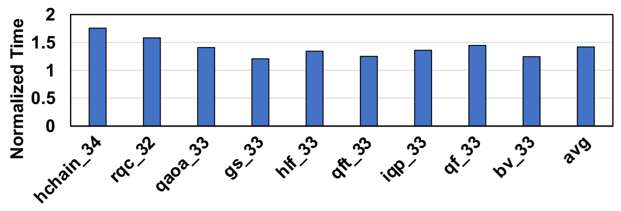


Figure 3: Normalized execution time of naive approach.

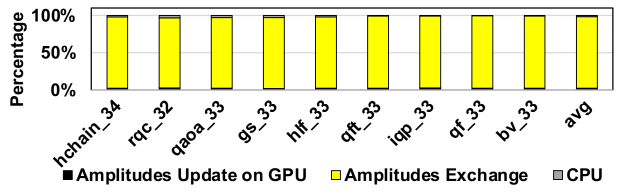


Figure 4: Execution time breakdown of naive approach.

number of qubits is larger than 30. GPU becomes worse than CPU when the number of qubits reaches 32. For example, we observe a factor of 1.8× slowdown for `qft_33` [2].

To investigate the reason for this slowdown, we show the breakdown of the execution time in Figure 2. One can observe that, on average, 88.89% of the execution is spent on the CPU, indicating that the GPUs are not properly used in the baseline execution for large number qubit circuits. Moreover, the overheads required for amplitudes exchange and synchronization occupies 10.29% of the average execution time, and the computation time of GPU only occupies 0.82% of total time on average. *In other words, most of the computation is performed by the CPU and the GPU is idle due to the static state chunk allocation in the baseline QCS execution.* In Figure 6, Ⓘ depicts the execution timeline of the baseline.

D. Will a Naive Optimization Work?

To improve the GPU utilization, an intuitive optimization is to dynamically allocate the chunks and transfer the chunks to GPU for updates. We implemented the dynamic state vector chunk allocation in QISKit-Aer baseline. Figure 3 depicts the execution time of the naive optimization normalized to the baseline execution. Surprisingly, none of the quantum circuits we studied show improvements when using dynamic allocation. To further investigate the reason, we break down the execution time and show the results in Figure 4. As one can observe, while CPU execution time significantly reduces, the data movement dominates, indicating that the GPU is severely underutilized waiting for data. Figure 6 Ⓘⓘ shows the naive execution timeline. Therefore, naive dynamic allocation alone does not deliver good QCS performance. More sophisticated end-to-end optimizations are required to systematically improve the QCS performance and scalability.

[2]In this paper, we use n in the circuit name (e.g., `circ_n`) to represent a circuit with n qubits.

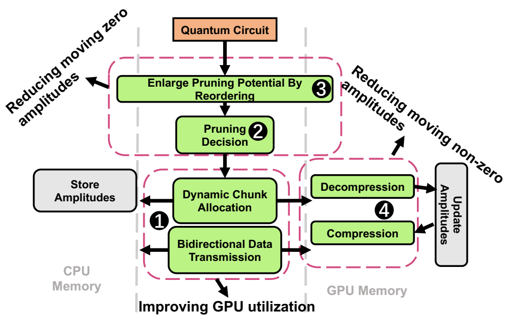

Quantum Circuit
Enlarge Pruning Potential By Reordering
❸
Pruning Decision
❷
Reducing moving zero amplitudes
Reducing moving non-zero amplitudes
Store Amplitudes
Dynamic Chunk Allocation
❶
Bidirectional Data Transmission
Decompression
❹
Compression
Update Amplitudes
CPU Memory
GPU Memory
Improving GPU utilization

Figure 5: High level overview of Q-GPU.

IV. Q-GPU

In this paper, we propose *Q-GPU*, a framework that features several end-to-end optimizations. Figure 5 depicts the high-level overview of Q-GPU. (❶) Q-GPU performs proactive state amplitude transfer to fully utilize the bi-directional data transfer bandwidth between CPU and GPU (Section IV-A). (❷) Before copying state amplitudes to GPU, Q-GPU performs dynamic redundancy elimination that prunes zero state amplitudes to avoid unnecessary data movements (Section IV-B). (❸) Q-GPU features a compiler-assisted, dependency-aware quantum gate reordering to enlarge the potential of pruning, i.e., the number of zero amplitudes (Section IV-C). (❹) Q-GPU implements a GPU-supported, lossless amplitude compression to further reduce the data transfer caused by non-zero state amplitudes with minimal runtime overheads (Section IV-D).

A. Proactive State Amplitudes Transfer

In the naive execution, one reason behind the poor GPU utilization is the sequential state amplitude transfer between CPU and GPU. Specifically, when the GPU finishes updating all local chunks, those chunks are first copied back to CPU memory before the CPU can transfer the next batch of un-updated chunks to the GPU. This restriction is reasonable in the scenarios when particular chunks are involved in consecutive updates, since the chunks being copied from the GPU's memory cannot be overwritten during the copying. In other words, data movements are synchronized to avoid data conflicts. However, if the subsequent chunks from the CPU are not copied to the same memory locations on the GPU where current chunks are stored, such data conflict does not exist. As a result, one can transfer the chunks simultaneously from the CPU to the GPU and from the GPU to the CPU.

In our work, Q-GPU leverages CUDA streams to enable concurrent and bi-directional chunk copy to fully utilize the available bandwidth between the CPU and GPU. To avoid potential data conflict, Q-GPU implements two CUDA streams and partitions the GPU memory into two halves. One stream is responsible for the first half partition that acts as a buffer holding the chunks the GPU is currently updating. The other stream is responsible for the second half partition that acts as a buffer for "prefetching" the next chunks for the GPU to update. The two memory partitions work as "circular buffers" to feed the GPU with the required chunks. These two streams can potentially overlap and execute concurrently.

Figure 6 illustrates the timeline of the baseline and each of our optimizations. The proposed proactive state amplitude transfer (III) achieves A cycles savings compared to the baseline (I).

B. *Pruning Zero State Amplitudes*

While overlapping improves the bandwidth utilization, the total amount of amplitudes that are transferred remains unchanged. To reduce the data movement, we observe that there exist a considerable amount of zero state amplitudes that do not need to be updated during simulation. Thus, those zero state amplitudes can be pruned before transferring the chunks.

Source of zero amplitudes: Let us assume there are n qubits, the initial states are usually set as $|0\rangle^{\otimes n}$ in the general QCS, indicating that all qubits have zero probability of being measured as $|1\rangle$. Hence, all state amplitudes are zeros, except for $a_{0_1 0_2 \ldots 0_n}$ which is 1. As the state of a particular qubit is unchanged until an operation is being applied on it, its state remains $|0\rangle$ until that operation happens. For instance, if a particular qubit q_k is $|0\rangle$, all the state amplitudes $a_{\times \cdots \times 1_k \times \cdots \times}$ are zeros since q_k has zero probability to be measured as $|1\rangle$. In general, if m of n-qubits are not involved, amplitudes $a_{\times 0_{k_1} \times 0_{k_2} \cdots \times 0_{k_m} \times \times}$ are possible to be non-zero values, whereas the remaining amplitudes are guaranteed to be zero values, i.e., $2^n - 2^{n-m}$ amplitudes are zero values. Therefore, even if only one qubit is not involved, then half of the state amplitudes are zeros.

Pruning potential: To investigate the potential of pruning, Table II lists the number of total operations and the number of operations before all qubits are involved. For circuits like `iqp`, we can expect a significant reduction of data movement after pruning since many qubits are not involved until the very end of execution. However, for `qft` and `qf`, all qubits are involved at the beginning of execution, diminishing the potential benefits of pruning. We also use `hchain_10` as an example and plot the distribution of state amplitudes after each operation (i.e., quantum gate) being applied. Figure 7 shows the state amplitudes distribution after 0, 30, 60

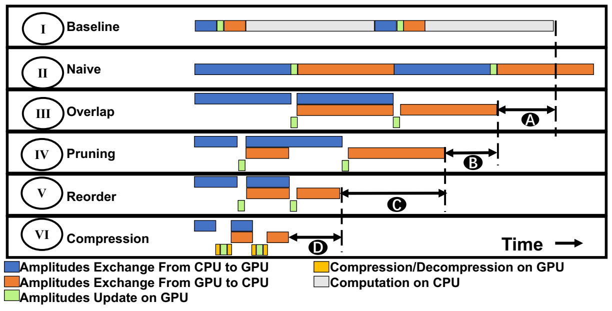


Figure 6: Timeline showing the benefits of each optimization in Q-GPU.

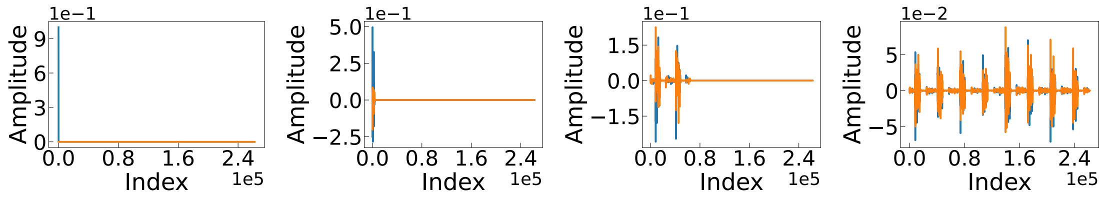
Table II: The number of total operations and the number of operations before all qubits are involved for all circuits with 34 qubits.

Circuit	Total number of operations	Number of operations before all qubit involvement	Percentage
`hchain`	1786	272	15.23%
`rqc`	124	54	43.55%
`qaoa`	754	19	2.51%
`gs`	37	16	43.24%
`hlf`	48	16	33.33%
`qft`	184	13	7.07%
`iqp`	146	132	90.41%
`qf`	222	16	7.21%
`bv`	134	34	25.37%

and 90 operations. One can observe that a large portion of state amplitudes are zeros at the beginning of the simulation (Figure 7a). During simulation, the amplitudes are gradually updated to non-zero values since more qubits are involved (Figures 7b to 7d).

In general, let us assume we have an operation involving m states, if all of the states are zero, these m states remain zeros after applying any operation. As a result, we do not need to transfer the zero state amplitudes to the GPU as their values will not change. Therefore, one can reduce the data movement between CPU and GPU by pruning the zero state amplitudes.

Pruning Mechanism: To this end, Q-GPU uses bits in a binary string as flags to indicate whether a qubit has been involved after a set of gate operations (denoted as $involvement$ in Algorithm 1). Initially, all the bits in $involvement$ are set to 0. When q_k is involved, the kth bit in $involvement$ is set to 1. Recall that the state vector is partitioned into chunks, the index of a chunk, i.e., $iChunk$, determines whether a chunk will be transferred or not. To compare $iChunk$ with flag bits in $involvement$, we define $iChunk'$ as the left-shifted $iChunk$ to align with $involvements$. When $iChunk'$ is larger than $involvement$, it indicates that at least one bit of $iChunk'$ is 1 and the corresponding flag bit in $involvement$ is 0. In this situation, the corresponding qubit (i.e., indexed by this flag bit) has not been involved by any operation. As such, we skip the remaining chunks and stop the iteration (line 5). On the other hand, if $iChunk'$ is smaller than or equal to $involvement$, the redundancy within a chunk is determined by $iChunk'$ & $involvement$ (line 7). For a qubit whose corresponding bit in $iChunk'$ is 1, if it has already been involved by previous operations, its corresponding bit in $involvement$ is also 1. Therefore, for all the qubits that is 1 in $iChunk'$, if all of them have already been involved by previous operations, $iChunk'$ & $involvement$ results in $iChunk'$ itself. Otherwise, all the state amplitudes within this chunk are zeros, and we can prune this chunk. Moreover, the $chunkSize$ here is dynamically determined rather than a statically fixed value, which enhances the benefit of the strategy discussed above. Specifically, we select $chunkSize$



(a) (b) (c) (d)

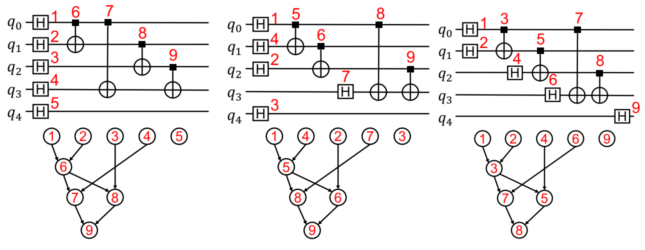
Figure 7: State amplitudes distribution of `hchain_10`, after 0, 30, 60 and 90 operations from (a) to (d). Blue and orange lines denote real and imaginary parts of an amplitude respectively.

Algorithm 1: Pruning zero state amplitudes.

```
Variable list:   N             Total chunks number in CPU,
                 involvement   Flag indicating which qubits are involved
1  /* Determine chunkSize by locating the least
      non-zero bit of involvement                                   */
2  chunkSize, N = getChunkSize(involvement)
3  for iChunk ← 0 to N − 1 do
4      iChunk′ = iChunk << chunkSize
5      if iChunk′ > involvement then
6          break
7      if iChunk′&involvement ≠ iChunk′ then
8          continue
9      /* Amplitudes update                                          */
10     ...
11 updateInvolvement(involvement)
```

by finding the least non-zero bit of *involvement*. This is useful, especially at the beginning of the simulation where many state amplitudes are zeros. For instance, assuming we have an 8-qubit circuit and the *involvement* flag is 00000011 at the early execution stage, the *chunkSize* is dynamically set to 2, which has fewer zeros within a chunk compared to a larger chunk. The *involvement* flag bits are updated according to the qubits involved in each operation (line 11). In Figure 6, the proposed pruning mechanism ((IV)) further saves ((B)) cycles over (III).

C. Reordering to Delay Qubit Involvement



(a) (b) (c)

Figure 8: A walk-through example to illustrate the reordering benefits using `gs_5`. The red numbers denote the operation orders before and after reordering.

In order to enlarge the potential of pruning such that more state amplitudes are zeros during simulation, we propose compiler-assisted, dependency-aware quantum operation reordering to delay the involvement of qubits. Specifically, when applying a gate, we choose the one that incurs the minimum number of additional qubits to be involved with those qubits that have been already involved by previous operations. For example, Figure 8a shows the `gs_5` circuit in the original execution order. The first five gates are H gates, where each gate applies to an individual qubit. As a result, once these gates have been applied, all the five qubits are involved. The next operation is a CNOT gate applied to qubits q_0 and q_1 ($CNOT_6$). All the state amplitudes are likely to be non-zero because the qubits are involved by the H gates. Therefore, applying this CNOT gate requires updating all the non-zero amplitudes in the state vector, leading to moving and traversing the entire state vector on the GPU. However, the $CNOT_6$ can be executed before some of the H gates without violating the circuit semantics. This gate reordering allows more zero state amplitudes (fewer data movements) when simulating the $CNOT_6$ gate. It is also important to emphasize that any reordering must obey the gate dependencies. For instance, $CNOT_6$ and $CNOT_7$ cannot be reordered due to the dependency on q_0.

To this end, we propose a compiler-assisted optimization to reorder the gate sequence with the goal of delaying the qubit involvement. Specifically, gates that are applied on different qubits in a quantum circuit can be executed independently in any order and the execution sequences of these independent gates does not affect the final simulation result [20], [29], [33]. This provides us the opportunity to reorder the independent gates. We use a directed acyclic graph (DAG) to represent the gate dependency in a circuit. Based on the DAG, we reorder the independent gates such that the simulation sequence involves the minimum number of new qubits when simulating each gate. Specifically, we investigate two heuristic strategies: 1) greedy reordering, and 2) forward-looking reordering.

Greedy reordering: greedy reordering traverses the DAG in topological order and greedily selects the gate (i.e., node in the DAG) that introduces the minimum number of new qubits to the list of updated qubits. The details of this method are illustrated in Algorithm 2. First, gates without predecessors in the DAG can be executed at the first steps

Algorithm 2: Quantum operation reorder.

```
Input :   $DAG$         A DAG representing circuit dependencies.
Output:   $gatesList$   List of gates after reordering,
1  $gatesList = [\,]$
2  $exeList = [\,]$
3  /* First we build $DAG$ and push gates without
      predecessors to an execution list                  */
4  for $g$ in $DAG$ do
5      if $g$.numPredecessors() == 0 then
6          $exeList$.append($g$)
7  /* Then we traverse DAG in topological order and
      greedily decides the execution order of the gates
      */
8  while $exeList \neq \emptyset$ do
9      $nextGate = NULL$
10     $minCost = 0$
11     for $g$ in $exeList$ do
12         $cost = g$.getCost()
13         if $cost < minCost$ then
14             $minCost = cost$
15             $nextGate = g$
16     $exeList$.erase($nextGate$)
17     $gatesList$.append($nextGate$)
18     for $g$ in $nextGate$.descendants() do
19         $g$.numPredecessors() = $g$.numPredecessors() $- 1$
20         if $g$.numPredecessors() == 0 then
21             $exeList$.append($g$)
```

and are put into $exeList$. Second, we traverse the gates in $exeList$ and find the one that introduces the minimum number of newly involved qubits (lines 8 to 15). Then, we remove this gate from $exeList$ and append it to the list of re-ordered gates. Third, we traverse the descendants of this gate and if a descendant does not have any predecessors other than this current gate, it will be added to $exeList$ (lines 18 to 21). The second and the third steps are repeated until $exeList$ is empty. In the rest of this section, we use Figure 8a as the example to illustrate how we perform reordering. At first, the $exeList$ is [g_1, g_2, g_3, g_4, g_5]. Since each of these five gates involves one new qubit, we randomly select one gate among them to start simulation. In this example, g_1 is selected as the starting gate. After traversing all its descendants, no new gates can be added into $exeList$. Next, the $exeList$ becomes [g_2, g_3, g_4, g_5]. In the next three steps, we randomly select g_3, g_5 and g_2 since no new gates can be executed and all gates in $exeList$ have equal priority. Then the $exeList$ becomes [g_4, g_6]. At this time, $involvedQubits$ is [q_0, q_1, q_2, q_4]. Therefore, g_4 involves one new qubit (q_3), whereas g_6 will not introduce any new qubits since it acts on q_0 and q_1 that are already in the involved list. Therefore, we will greedily select g_6 to execute since it involves the least new qubits. One can follow these reordering steps to reach the new ordering shown in Figure 8b. As a result, the number of involved qubits at each step is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 4 \rightarrow 4 \rightarrow 5 \rightarrow 5 \rightarrow 5$. Since the baseline is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 5 \rightarrow 5$, the final involvement is delayed by two steps. However, a better solution for reordering is to select g_2 and g_6 in the second and the third step, since applying these two gates only adds one qubit to $exeList$, while applying g_3 and g_5 adds two. Thus greedy reordering may miss the optimal choice.

Algorithm 3: Cost calculation in forward-looking reordering.

```
Input :   $g$               Gates from $exeList$,
          $exeList$         List of gates that are executable,
          $involvedQubits$  Set of involved qubits.
Output:   $cost$            Potential involved qubits after executing $g$.
1  $costCurrent = 0, costLookAhead = 0$
2  /* First we compute additional qubits that will be
      acted on by executing current gate                 */
3  for $q$ in $g$.qubits() do
4      if $q$ not in $involvedQubits$ then
5          $costCurrent = costCurrent + 1$
6          $involvedQubits$.insert($q$)
7  $exeList$.erase($g$)
8  for $g'$ in $g$.descendants() do
9      if $g'$.numPredecessors() == 1 then
10         $exeList$.push($g'$)
11 /* Then we traverse current $exeList$ and compute the
      cost of selecting a gate that involve least
      additional qubits                                  */
12 for $g''$ in $exeList$ do
13     $curCostLookAhead = 0$
14     for $q'$ in $g''$.qubits() do
15         if $q'$ not in $involvedQubits$ then
16             $curCostLookAhead = curCostLookAhead + 1$
17     if $curCostLookAhead < costLookAhead$ then
18         $costLookAhead = curCostLookAhead$
19 $cost = costCurrent + costLookAhead$
20 return $cost$
```

Forward-looking reordering: To address the deficiency in greedy-reordering, we propose Forward-looking reordering that looks ahead of all the equal-priority gate candidates before making a decision. We implemented a $cost$ counter to determine the priority of the gates in $exeList$. In *greedy reordering*, the $cost$ is simply computed by counting new involved qubits (lines 3-6 in Algorithm 3). In contrast, the computation of $cost$ in forward-looking reordering is more sophisticated as it consists of two components: $costCurrent$ and $costLookAhead$ (line 1). The $costCurrent$ is the same with the $cost$ used in greedy reordering. Again, we use the example in Figure 8a to illustrate Algorithm 3. Initially, the $exeList$ is also [g_1, g_2, g_3, g_4, g_5]. We take g_1 as an example to explain the computation of $costLookAhead$. First, we assume g_1 has already been executed. Then, the $costCurrent$ is 1 and $involvedQubits$ becomes [q_0] (lines 3-6). Since no descendants of g_1 can be executed, the $exeList$ becomes [g_2, g_3, g_4, g_5] (lines 7-10). Then, we traverse the $exeList$. For each gate in $exeList$, we compute the cost of selecting this gate by counting the new involved qubits (lines 12-18) and selecting the least cost as $costLookAhead$. Now, executing any gate in $exeList$ will involve one new qubit, thus $costLookAhead$ is computed as 1 (lines 12-16). Similarly, one can find that all gates at the first step have equal priority. For the purpose of illustration, we assume g_1 is randomly selected. Then the $exeList$ becomes [g_2, g_3, g_4, g_5]. Although all gates still have equal $costCurrent$, we can find that g_2 has the least $costLookAhead$. The reason is that, when we assume

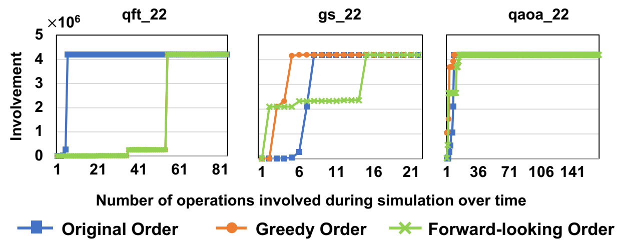


Figure 9: Qubit *involvement* during simulation in three representative circuits.

executing g_2 and look ahead from g_2, we find that executing g_6 introduces no new qubits. In contrast, look ahead after executing other gates will introduce new qubits. Therefore, g_2 is selected to be executed in the next step. Following this procedure, we get the result of forward-looking reorder as shown in Figure 8c: the *involvement* at each step becomes $1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4 \rightarrow 4 \rightarrow 4 \rightarrow 5$. Compared to greedy reordering, we further delay the final involvement by two steps. Note that, *exeList* and *involvedQubits* in Algorithm 3 are just copies of the original ones, thus their original values are not changed.

Reorder effectiveness: To assess the performance of the reordering algorithms discussed above, we implement them to reorder the original operation sequences for all benchmark circuits that have 22 qubits and plot the *involvement* (Algorithm 1 in Section IV-B) after each gate has been applied. For the purpose of illustration, we depict the results of three representative benchmark circuits in Figure 9, where the *involvement* is defined in section IV-B and used in Algorithm 1. For each order, i.e., original order, greedy-reorder, and forward-looking reorder, the "speed" of reaching the maximum *involvement* indicates the pruning potential. We observe that, forward-looking reordering results in the largest pruning potential, while greedy reordering only works for `qft_22` and even results in less pruning potential than baseline for `gs_22`. Particularly, for `gs_22` and `qft_22`, forward-looking reordering effectively delays the involvement of qubits. Thus, we can expect the pruning potentials of these circuits to be enlarged by forward looking reordering. However, for `qaoa_22`, none of the reordering algorithms work due to the prevalent dependencies among the gates. Refering back to Figure 6, when reordering (Ⓥ) is employed, we can prune more chunks, which saves additional Ⓒ cycles compared to (Ⅳ).

Note that, pruning and reordering do not affect the simulation results nor introduce error to quantum circuits. This is because a quantum state vector is partitioned into groups and each group is updated independently in parallel (as we discussed in Section II-A). A group of amplitudes is a 1xN vector, and a quantum gate is an NxN matrix. Thus, the actual computation involves multiple parallel 1xN vectors – NxN matrix multiplications. If a 1xN vector contains all zeros, it will stay unchanged after being multiplied with any matrix, thus can be "pruned" safely (i.e., we leave these all-zero 1xN vectors on CPU w/o transferring them to the GPUs and update them). Moreover, the reordering does not affect the simulation results since we do not violate dependencies among gates. The greedy reordering involves light-weight overheads and is effective in the circuits containing less dependent gates (e.g., `qft`).

D. Compressing Non-zero State Amplitude

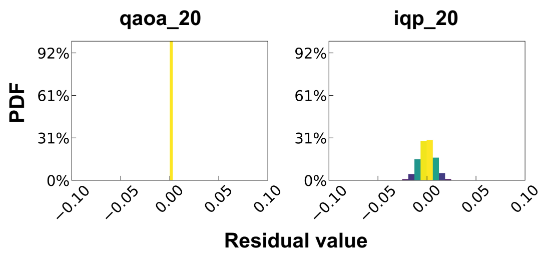


Figure 10: Residual distributions for `qaoa_20` and `iqp_20`.

Compressibility: While pruning removes the zero state amplitudes, those non-zero amplitudes still cause data movement overheads especially for circuits that do not have large pruning potentials (e.g., `qaoa` in Figure 9). In order to reduce the data movement caused by non-zero state amplitudes, we investigate the potential compressibility and propose a GPU-supported efficient lossless data compression in Q-GPU. Specifically, we observe that many non-zero entries within a state vector, after each operation, have similar amplitude values. In other words, there is a significant "spatial" similarly among consecutive state amplitudes in the state vector. To demonstrate the compressibility, we use `qaoa_20` and `iqp_20` as examples and show the residuals by subtracting the consecutive state amplitudes. As one can observe from Figure 10, for `qaoa_20`, most of the residuals are zero or very close to zero, indicating a potential for residual-based compression. However, `iqp` will be less compressible due to more diverse distribution.

Compression Strategy: We use the GFC algorithm [47] in Q-GPU. We implement the GFC as GPU kernels to perform the compression in parallel, thereby reducing the compression and decompression overheads. As shown in Figure 11, we divide a chunk into segments and each segment is compressed/decompressed by a single warp. Multiple warps compress multiple segments independently and concurrently. We empirically choose the number of segments to match the GPU parallelism such that the GPU is properly utilized. To compress/decompress a single segment, we further partition a segment into micro-chunks. Each micro-chunk contains 32 amplitudes to match the warp size. Each warp iteratively computes residuals between consecutive micro-chunks within a segment and encodes the residuals into compressed formats. For example, thread$_j (0 \leq j \leq 31)$ of a warp is responsible to subtract data double$_j$ in current micro-chunk$_k$ and corresponding data double$_i$ in previous

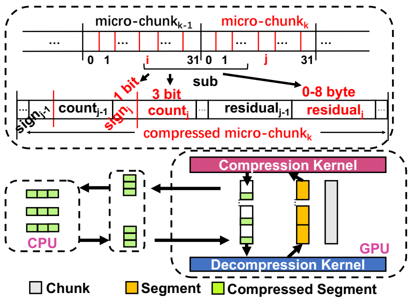


Figure 11: Overview of compression in Q-GPU.

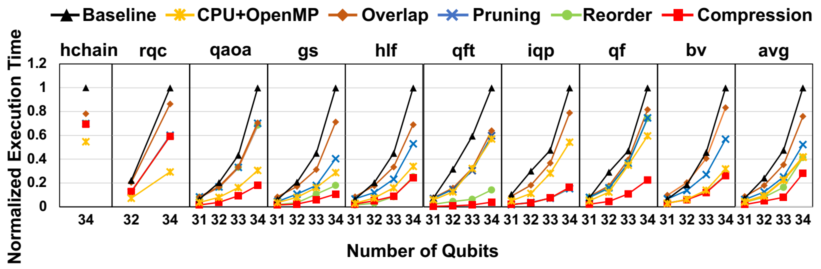


Figure 12: Normalized execution time for circuits with different number of qubits (the lower the better).

micro-chunk$_{k-1}$ to get the residual value. Then, it encodes leading zeros of residual into a 4-bit prefix, where one bit is used to record the sign and the other three bits for counting the bytes of leading zeros. Other threads in the warp execute the same steps on different data and communicate through shared memory to compute the size of compressed micro-chunks. As shown in figure 11, the compression is performed on the GPU after updating the chunk before copying it to the CPU. The compressed segments are transferred to the CPU instead of the original state chunks. The CPU keeps the compressed segments and copies the compressed segments to the GPUs upon request. Once the chunks are copied to the GPU, the amplitudes are decompressed, updated, and then compressed. As can be seen from Figure 6, compression (ⓥⓘ) saves Ⓓ cycles over ⓥ and introduces negligible overhead. Later, in section V, we quantify the overheads incurred by the compression and decompression procedures.

V. Experimental Evaluation

In this section, we evaluate Q-GPU using the nine circuits in Table I. We implement Q-GPU by substantially extending IBM QISKit-Aer. For all experiments, the default optimizations in QISKit-Aer are turned on in both baseline and Q-GPU evaluation. We test *six different versions* of execution for all quantum circuit benchmarks:

- *Baseline:* This version is the default QISKit-Aer (v0.7.0) with state-of-the-art GPU support [3]. As illustrated in Section III-B, state amplitudes are statically allocated on the CPU and the GPU in this version.
- *Naive:* This version is the intuitive implementation discussed in Section III-D, which dynamically allocates state amplitudes to GPU. The performance of this version is dominated by expensive data movements.
- *Overlap:* This version implements the first optimization – proactive state amplitude transfer – in Q-GPU. It is built upon the *Naive* version and its details are discussed in Section IV-A.
- *Pruning:* This version adds the proposed pruning mechanism (Section IV-B) to *Overlap*. By skipping the data movement of zero state amplitudes, the amount of data movement is reduced.
- *Reorder:* In this version, we implement *forward-looking reorder* algorithm (Section IV-C) to enlarge the potential for pruning. This reordering is performed by a simple compiler pass integrated in the Q-GPU.
- *Compression/Q-GPU:* In this version, all optimizations are employed with the proposed lossless compression.

A. Overall Performance

Figure 12 shows the overall performance and scalability among the six versions for all nine quantum circuits. The y-axis in the figure denotes the normalized execution time to the *Baseline* version. From the figure, one can make the following observations. First, Q-GPU significantly reduces the execution time of QCS across all the circuits. Specifically, *Overlap*, *Pruning*, *Reorder*, and *Compression/Q-GPU* bring 24.03%, 47.69%, 58.60%, and 71.89% execution time reductions over the baseline execution for the largest number of qubits (i.e., 34 qubits) that can run on our platform. Second, the scalability of QCS performances is significantly improved by, “breaking”, the memory limitation in GPUs. The average performance reduces execution time by 71.89% over baseline (3.55× speedup) for 34 qubits. Although we can only simulate up to 34 qubits due to the CPU memory limitation (384 GB) in our system (Section III-C), one can infer from the trend that Q-GPU is scalable to large circuit sizes. Third, Q-GPU achieves different performances for different circuits. Specifically, for `gs`, `qft`, `qaoa` and `iqp`, higher execution time reduction is observed, whereas for `hchain` and `rqc`, less execution time reduction is observed. This is because, for `hchain` and `rqc`, reordering cannot enlarge the pruning potential due to dependent gates. Their amplitude residuals also have disperse distribution (similar to `iqp` in Figure 10). Thus, either *Reorder* or *Compression* improves little for these two benchmarks. Finally, for different circuits, a certain version may not have the same improvement. For example, *Overlap* version generates a similar execution time reduction in all circuits tested. However, for *Pruning*, *Reorder*, and *Compression*, the runtime reduction is different between different circuits. For example, *Pruning* and *Reorder* improve little for `qaoa` and `qf` because these two circuits do not have much potential of pruning the zero amplitudes. That is, their qubits get

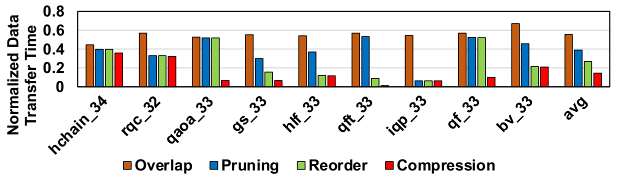


Figure 13: Normalized data transfer time (the lower the better).

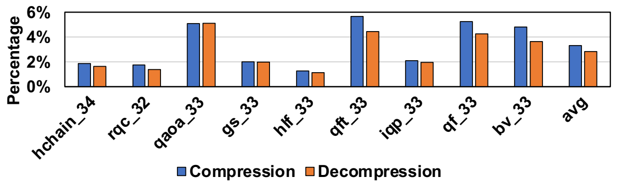


Figure 14: Compression and decompression overheads.

involved quickly with dependent operations. However, `qaoa` achieves significant benefits by compression as there is great potential of compressibility. (discussed in Section IV-D). We also compared Q-GPU with QISKit-Aer CPU-OpenMP in Figure 12. For 34-qubit circuits, the proposed Q-GPU achieves 32.68% average execution time reduction (1.49$\times$ speedup) over the CPU-OpenMP. Q-GPU has less speedup for `hchain` and `rqc` and cannot beat CPU-OpenMP version. As explained earlier in this paragraph, this is because the pruning potential and the compressibility are low in both circuits where Q-GPU is less effective.

To further understand the execution time reduction, Figure 13 plots, for each version, the data transfer time. In this figure, the y-axis represents the data transfer time normalized to the *Naive* version. Clearly, one can observe a step-wise data transfer time reduction in the versions of Q-GPU. First, *Overlap* uniformly reduces the data transfer time by an average of 44.56%. Note that, the savings generated in *Overlap* are independent of circuit types. For *Pruning* and *Reorder*, the reduction of data transfer time varies in different circuits. This is because the number of zero state amplitudes and the potential of pruning heavily relies on the circuit type. For example, `qaoa`, `qft`, and `qf` get all qubits involved at early stage of simulation. Hence, pruning is less effective for these circuits compared to others. Also, as discussed in Section IV-C, *Reorder* has little effect on `hchain`, `rqc`, `qaoa`, and `qf` due to dependent operations in these circuits. Therefore, *Reorder* delivers similar data transfer time reduction with *Pruning* for these circuits. However, for those circuits with less dependent operations, *Reorder* significantly reduces their data transfer time by enlarging the pruning potential. For circuits like `qaoa`, `gs`, `qft`, and `qf`, *Compression* effectively reduces the data movement by leveraging the spatial similarity discussed in Section IV-D.

We also quantify the computation time of compression and decompression in Figure 14. Overall, the compression and decompression overhead is 3.31% and 2.84% of the execution time. Potentially one may further optimize the compression and decompression by overlapping them on GPU, but we found the overhead is negligible compared to the significant reduction in execution time that we achieved. We also want to emphasize that the execution times reported in Figure 12 have all the sources of overhead included.

B. Roofline Analysis

We conduct a GPU roofline analysis to show the compute intensity and memory intensity of QCS on NVIDIA Tesla V100 GPU with 16 GB memory. We use NVIDIA Nsight Compute [46] to collect the metrics and calculate the arithmetic intensity and FLOPS of GPU for the QCS applications. We use `qft` and `iqp` as examples and the results are shown in Figure 15. As one can observe from the figure, QCS is memory-bound since all points are located under the line. When the number of qubits is small (less than or equal to 29 qubits) and the whole state vector can fit into GPU memory, the execution reaches the bound of FLOPS. However, when the simulation significantly exceeds the GPU memory capacity (larger than or equal to 31 qubits , the baseline suffers from very low FLOPS. In contrast, the naive approach improves the FLOPS, but the arithmetic intensity is also reduced. The proposed Q-GPU has much higher FLOPS than the baseline and naive approach.

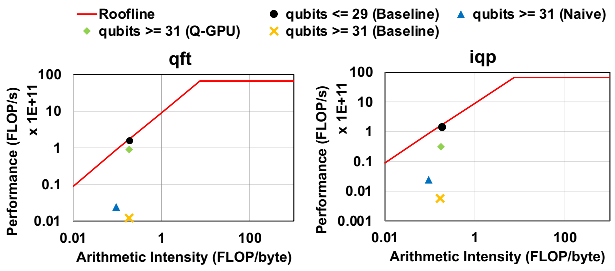
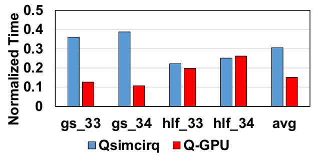


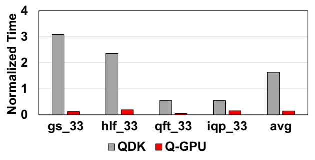
Figure 15: Roofline analysis of Q-GPU using `qft` and `iqp`.

C. Comparison with Other Simulators

We next compare Q-GPU with Google Qsim-Cirq v0.8.0 plus Cirq v0.9.2 [54] and Microsoft QDK v0.15 [41]. Note that both Qsim-Cirq and QDK have OpenMP enabled and we observe that they use all available CPU threads during execution. To perform the simulation on Qsim-Cirq, we need to first transform our circuit benchmarks into OpenQASM programs [14], and then import these programs to Qsim-Cirq for simulation. Unfortunately, not all the transformed circuits can run on Qsim-Cirq due to the lack of support for some gates (e.g., the "cp" gate cannot be recognized by Qsim-Cirq) used in the benchmarks. As a result, we test `gs` and `hlf` on Qsim-Cirq. Similarly, to run those benchmarks on QDK, we have to further convert the OpenQASM programs to "qsharp", i.e., the quantum language used in Microsoft, and the conversion only succeeded for `qft`, `iqp`, `hlf`, and `gs`. The normalized execution time results are shown in



(a) Comparison with Google Qsim-Cirq.



(b) Comparison with Microsoft QDK.

Figure 16: Comparisons of Q-GPU with Google Qsim-Cirq v0.8.0 and Microsoft QDK v0.15.

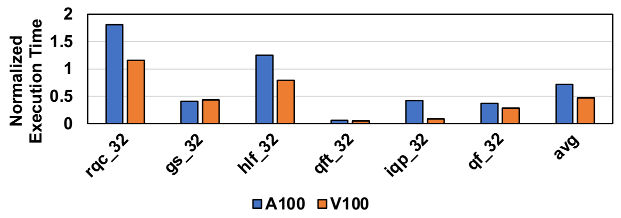


Figure 17: Normalized execution time on NVIDIA Tesla A100 and V100.

Figure 16. On average, Q-GPU achieves 50.50% execution time reduction (2.02× speedup) over Qsimcirq, and 90.76% execution time reduction (10.82× speedup) over QDK.

D. Performance on NVIDIA V100 and A100

We want to emphasize that Q-GPU is not bound to any particular GPU architecture. To investigate how Q-GPU performs on different GPU architectures, we run Q-GPU on i) a server with an 32-GB NVIDIA V100 GPU (the corresponding CPU is a 8-core Intel Xeon 6133 CPU and the host memory is 80 GB), and ii) a server with an 40-GB NVIDIA A100 GPU (the corresponding CPU is a 12-core virtual CPU and the host memory is 85 GB). As shown in Figure 17 Q-GPU achieves on average 27.05% and 53.24% execution time reductions on A100 and V100, respectively. Since A100 has large device memory and the host memory is limited, Baseline A100 has higher GPU utilization and performs better in some circuit benchmarks (`hchain_34` and `qaoa_32` exceed server memory and fail to execute).

E. Multi-GPU Performance

We next evaluate Q-GPU in multi-GPU systems. We use the Qiskit-Aer multi-GPU support as our baseline [18]. As shown in Figure 18, Q-GPU employs multi-GPU as follows: consider a 7-qubit circuit with an operation acting on q_5. We first assign all state amplitudes to CPU memory. Then we partition state chunks on the CPU into 4 groups and assign them to two GPUs, i.e., G0 and G1. In the example, $group_0$ and $group_2$ are assigned to G0 and $group_1$ and $group_3$ are assigned to G1. In other words, Q-GPU assigns amplitudes to multiple GPUs in a round-robin fashion using groups. When the GPUs finish the computation, the chunks assigned to each GPU are copied back to the CPU host memory. We evaluate Q-GPU using two multi-GPU servers. Server-1 has a 32-core CPU with 208 GB main memory and four NVIDIA Tesla P4 GPUs (8 GB memory per GPU)

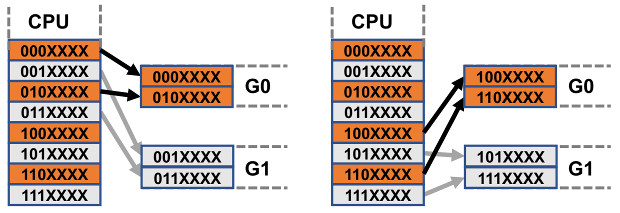


Figure 18: Multi-GPU execution in Q-GPU.

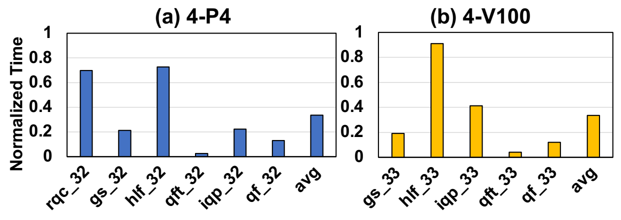


Figure 19: Normalized execution time on multi-GPU platforms.

connected through PCIe. Server-2 has the same CPU and four NVIDIA Tesla V100 GPUs (16 GB memory per GPU) connected through NVLink. Figure 19 shows the execution time normalized to baseline Qiskit multi-GPU execution. Note that, the circuits running on V100 have larger number of qubits since the V100 memory capacity is larger. As one can observe, Q-GPU achieves 66.38% (2.97× speedup) and 66.46% (2.98× speedup) execution time reductions on server-1 and server-2, respectively. This is because, the majority of the data movement is between CPU and GPUs in the multi-GPU quantum simulation. The cross GPU data movement is limited and does not dominate the execution time.

F. Deep Circuit

We next study how Q-GPU performs in deep circuits. Note that Q-GPU has limited improvements if the circuits have many dependent gates. However, for deep circuits with independent gates, Q-GPU is able to yield appealing improvements. To this end, we test Q-GPU using Google deep circuit (denoted as `grqc`) which is generated under rules defined in [9] and two random deep circuits (denoted as `rqc_31` and `rqc_32`). Table III shows the number of operations and the performances of the three deep circuits. As one can observe, pruning and reordering in Q-GPU can achieve 41.47% execution time reduction on Google deep circuit, and on average 17.69% execution time reduction on two random deep circuits. This indicates the effectiveness of Q-GPU in deep circuits.

Table III: Effectiveness of pruning and reordering for deep random circuits.

Circuit	Total number of operations	Overlap (s)	Reorder (s)	Execution time reduction (%)
`grqc_32`	7241	42875.22	25095.02	41.47%
`rqc_31`	598	1646.08	1349.94	17.99%
`rqc_32`	623	3396.02	2805.54	17.39%

VI. Related Works

To the best of our knowledge, Q-GPU is the first work that systematically optimizes quantum circuit simulation on GPUs.

Prior works have focused on QCS optimizations on different platforms, from readily available devices to cloud environments [20], [24], [26], [48], [49], [52], [55], [57]. Thomas et al. [24] simulated 45-qubits circuit using 8,192 nodes. They optimized single node performance by using automatic code generation and optimization of compute kernels. Edwin et al. [48] claimed to simulate more than 49 qubits by partitioning quantum circuits to "subcircuits" and delay their entanglements. In [55], the authors proposed lossy data compression to reduce the memory requirement of simulating large-scale quantum circuits. Aneeqa et al. [20] focused on fully exploiting single CPU performance for simulating a large number of qubits. The developed algorithm aims to reorder circuits such that more gates can be clustered to reduce the number of gates. Compared with prior works, Q-GPU is built upon several interrelated system optimizations to fully take advantage of GPUs and mitigate data movement overheads. While existing works [20], [29] leverage the commutation of gates for clustering of gates, the reordering of Q-GPU is another independent stage that has a completely different purpose. It greedily selects gates that involve the least number of qubits, so as to enlarge pruning potential. In contrast, the reordering algorithms in [20], [29] selects gates that involve many qubits that diminish the pruning potential. Additionally, rather than using lossy data compression [55] to reduce memory consumption, Q-GPU provides a hybrid approach with pruning and compression to address zero and non-zero amplitudes separately. As such, it significantly reduces amplitudes exchange without losing fidelity, making Q-GPU a more general simulator that can support any quantum application.

There are also several works that utilize GPUs to accelerate QCS [4], [6], [18], [22], [32], [35], [56]. Most of these works have limited capability in simulating large quantum circuits due to the limited memory capacity of GPUs. Ang et al. [32] proposed a multi-GPU centric QCS framework that tracks the density matrix. However, their framework is still limited by the aggregated memory capacity of multi-GPUs. For a single-node, they can only simulate up to 14 qubits on an NVIDIA V100 GPU. Doi et al. [18] proposed to enable simulation using a GPU even when the required memory exceeds the GPU memory capacity. In summary, compared to prior works, Q-GPU breaks the GPU memory capacity limitation, i.e., it is able to simulate 34 qubits which require 256 GB memory on a 16 GB memory GPU, and fully takes advantage of GPU parallelization. The fundamental design innovation behind this is to dynamically and proactively transfer the state amplitudes through end-to-end optimizations to minimize the data movement overheads caused by state amplitudes transfer.

VII. Concluding Remarks

In this paper, we propose Q-GPU, a framework tailored with GPU optimizations to effectively improve the quantum circuit simulation performance for quantum circuits with a large number of qubits. The Q-GPU breaks the memory limitation of GPUs and delivers scalable simulation performance based on the proposed end-to-end system optimizations. Experimental results across nine representative quantum circuits indicate that Q-GPU achieves on average 71.89% execution time reduction (3.55× speedup) over the state-of-the-art GPU-based QCS on a single GPU. It also outperforms the most recent OpenMP CPU implementation and other publicly available quantum simulators.

Acknowledgements

The authors sincerely thank the anonymous HPCA reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2011146, a startup funding from the University of Pittsburgh, and the Pitt momentum seeding grant.

References

[1] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Physical Review A*, vol. 70, no. 5, p. 052328, 2004.

[2] W. A. Al-Saidi, S. Zhang, and H. Krakauer, "Bond breaking with auxiliary-field quantum monte carlo," *The Journal of Chemical Physics*, vol. 127, no. 14, p. 144101, 2007. [Online]. Available: https://doi.org/10.1063/1.2770707

[3] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen, J. M. Chow, A. D. Córcoles-Gonzales, A. J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S. D. L. P. González, E. D. L. Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I. F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B. G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, Łukasz Herok, H. Horii, S. Hu, T. Imamichi, T. Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Y. Maeng, M. Marques, F. J. Martín-Fernández, D. T. McClure, D. McKay, S. Meesala, A. Mezzacapo, N. Moll, D. M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L. J. O'Riordan, H. Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, A. R. Davila, R. H. P. Rudy, M. Ryu, N. Sathaye, C. Schnabel, E. Schoute, K. Setia, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J. A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, K. Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J. A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I. Y. Akhalwaya, and C. Zoufal, "Qiskit: An Open-source Framework for Quantum Computing," Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2562111

[4] A. Amariutei and S. Caraiman, "Parallel quantum computer simulation on the gpu," in *15th International Conference on System Theory, Control and Computing*. IEEE, 2011, pp. 1–6.

[5] H. Aschauer, W. Dür, and H.-J. Briegel, "Multiparticle entanglement purification for two-colorable graph states," *Physical Review A*, vol. 71, no. 1, p. 012319, 2005.

[6] A. Avila, A. Maron, R. Reiser, M. Pilla, and A. Yamin, "Gpu-aware distributed quantum simulation," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 860–865.

[7] A. Baiardi and M. Reiher, "The density matrix renormalization group in chemistry and molecular physics: Recent developments and new challenges," *The Journal of Chemical Physics*, vol. 152, no. 4, p. 040903, Jan. 2020. [Online]. Available: http://aip.scitation.org/doi/10.1063/1.5129672

[8] E. Bernstein and U. Vazirani, "Quantum complexity theory," *SIAM Journal on computing*, vol. 26, no. 5, pp. 1411–1473, 1997.

[9] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven, "Characterizing quantum supremacy in near-term devices," *Nature Physics*, vol. 14, no. 6, pp. 595–600, 2018.

[10] A. Bouland, B. Fefferman, C. Nirkhe, and U. Vazirani, "On the complexity and verification of quantum random circuit sampling," *Nature Physics*, vol. 15, no. 2, pp. 159–163, 2019.

[11] S. Bravyi, D. Gosset, and R. König, "Quantum advantage with shallow circuits," *Science*, vol. 362, no. 6412, pp. 308–311, 2018.

[12] M. J. Bremner, R. Jozsa, and D. J. Shepherd, "Classical simulation of commuting quantum computations implies collapse of the polynomial hierarchy," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 467, no. 2126, pp. 459–472, 2011.

[13] M. J. Bremner, A. Montanaro, and D. J. Shepherd, "Average-case complexity versus approximate simulation of commuting quantum computations," *Physical review letters*, vol. 117, no. 8, p. 080501, 2016.

[14] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," *arXiv preprint arXiv:1707.03429*, 2017.

[15] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito, "Massively parallel quantum computer simulator," *Computer Physics Communications*, vol. 176, no. 2, pp. 121–136, 2007.

[16] Y. Ding and F. T. Chong, "Quantum computer systems: Research for noisy intermediate-scale quantum computers," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 1–227, 2020.

[17] J. Doi and H. Horii, "Cache blocking technique to large scale quantum computing simulation on supercomputers," in *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 2020, pp. 212–222.

[18] J. Doi, H. Takahashi, R. Raymond, T. Imamichi, and H. Horii, "Quantum computing simulator on a heterogenous hpc system," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 85–93.

[19] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[20] A. Fatima and I. L. Markov, "Faster schrödinger-style simulation of quantum circuits," *arXiv preprint arXiv:2008.00216*, 2020.

[21] A. Gilliam, S. Woerner, and C. Gonciulea, "Grover Adaptive Search for Constrained Polynomial Binary Optimization," *Quantum*, vol. 5, p. 428, Apr. 2021. [Online]. Available: https://doi.org/10.22331/q-2021-04-08-428

[22] E. Gutierrez, S. Romero, M. A. Trenas, and E. L. Zapata, "Simulation of quantum gates on a novel gpu architecture," in *International Conference on Systems Theory and Scientific Computation*. Citeseer, 2007.

[23] J. Hachmann, W. Cardoen, and G. K.-L. Chan, "Multireference correlation in long molecules with the quadratic scaling density matrix renormalization group," *The Journal of Chemical Physics*, vol. 125, no. 14, p. 144101, Oct. 2006. [Online]. Available: http://aip.scitation.org/doi/10.1063/1.2345196

[24] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126947

[25] M. Hein, W. Dür, J. Eisert, R. Raussendorf, M. Van den Nest, and H.-J. Briegel, "Entanglement in graph states and its applications," in *Quantum Computers, Algorithms and Chaos*. IOS Press, 2006, pp. 115–218.

[26] C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan *et al.*, "Classical simulation of quantum supremacy circuits," *arXiv preprint arXiv:2005.06787*, 2020.

[27] IBM. (2020) Ibm's roadmap for scaling quantum technology. [Online]. Available: https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/

[28] IBM. (2021) Ibm quantum breaks the 100-qubit processor barrier. [Online]. Available: https://research.ibm.com/blog/127-qubit-quantum-processor-eagle

[29] R. Iten, R. Moyard, T. Metger, D. Sutter, and S. Woerner, "Exact and practical pattern matching for quantum circuit optimization," *arXiv preprint arXiv:1909.05270*, 2019.

[30] A. JavadiAbhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "Scaffcc: Scalable compilation and analysis of quantum programs," *Parallel Computing*, vol. 45, pp. 2–17, 2015.

[31] D. E. Koh, "Further extensions of clifford circuits and their classical simulation complexities," *Quantum Information and Computation*, vol. 17, pp. 262–282, 2015. [Online]. Available: http://www.rintonpress.com/journals/qic/index.html

[32] A. Li, O. Subasi, X. Yang, and S. Krishnamoorthy, "Density matrix quantum circuit simulation via the bsp machine on modern gpu clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[33] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for nisq-era quantum devices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1001–1014.

[34] R. Li, B. Wu, M. Ying, X. Sun, and G. Yang, "Quantum supremacy circuit simulation on sunway taihulight," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 805–816, 2019.

[35] Z. Li and J. Yuan, "Quantum computer simulation on gpu cluster incorporating data locality," in *International Conference on Cloud Computing and Security*. Springer, 2017, pp. 85–97.

[36] D. A. Lidar and T. A. Brun, *Quantum error correction*. Cambridge university press, 2013.

[37] D. Lykov, R. Schutski, A. Galda, V. Vinokur, and Y. Alexeev, "Tensor network quantum simulator with step-dependent parallelization," *arXiv preprint arXiv:2012.02430*, 2020.

[38] I. L. Markov, A. Fatima, S. V. Isakov, and S. Boixo, "Massively parallel approximate simulation of hard quantum circuits," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[39] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM Journal on Computing*, vol. 38, no. 3, pp. 963–981, 2008.

[40] A. McCaskey, E. Dumitrescu, M. Chen, D. Lyakh, and T. Humble, "Validating quantum-classical programming models with tensor network simulations," *PloS one*, vol. 13, no. 12, p. e0206704, 2018.

[41] Microsoft, "Qdk." [Online]. Available: https://github.com/microsoft/qdk-python

[42] T. Monz, P. Schindler, J. T. Barreiro, M. Chwalla, D. Nigg, W. A. Coish, M. Harlander, W. Hänsel, M. Hennrich, and R. Blatt, "14-qubit entanglement: Creation and coherence," *Physical Review Letters*, vol. 106, no. 13, p. 130506, 2011.

[43] M. Motta, C. Genovese, F. Ma, Z.-H. Cui, R. Sawaya, G. K.-L. Chan, N. Chepiga, P. Helms, C. Jiménez-Hoyos, A. J. Millis *et al.*, "Ground-state properties of the hydrogen chain: dimerization, insulator-to-metal transition, and magnetic phases," *Physical Review X*, vol. 10, no. 3, p. 031058, 2020.

[44] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," 2002.

[45] NVIDIA. (2020) Profiler user's guide. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/

[46] NVIDIA. (2021) Nsight compute. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[47] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a gpu," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–7.

[48] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, T. Magerlein, E. Solomonik, and R. Wisnieff, "Breaking the 49-qubit barrier in the simulation of quantum circuits," *arXiv preprint arXiv:1710.05867*, vol. 15, 2017.

[49] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, and R. Wisnieff, "Leveraging secondary storage to simulate deep 54-qubit sycamore circuits," *arXiv preprint arXiv:1910.09534*, 2019.

[50] Y. Shen, X. Zhang, S. Zhang, J.-N. Zhang, M.-H. Yung, and K. Kim, "Quantum implementation of the unitary coupled cluster for simulating molecular electronic structure," *Phys. Rev. A*, vol. 95, p. 020501, Feb. 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.95.020501

[51] 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.

[52] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik, "qhipster: The quantum high performance software testing environment," *arXiv preprint arXiv:1601.07195*, 2016.

[53] N. H. Stair, R. Huang, and F. A. Evangelista, "A Multireference Quantum Krylov Algorithm for Strongly Correlated Electrons," *Journal of Chemical Theory and Computation*, vol. 16, no. 4, pp. 2236–2245, Apr. 2020. [Online]. Available: https://pubs.acs.org/doi/10.1021/acs.jctc.9b01125

[54] Q. A. team and collaborators, "qsim," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4023103

[55] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–24.

[56] P. Zhang, J. Yuan, and X. Lu, "Quantum computer simulation on multi-gpu incorporating data locality," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2015, pp. 241–256.

[57] A. Zulehner and R. Wille, "Advanced simulation of quantum computations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 848–859, 2018.