BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram

Shui Jiang
The Chinese University of Hong Kong
Shatin, Hong Kong
sjiang22@cse.cuhk.edu.hk
University of Wisconsin-Madison
Madison, WI, United States
sjiang285@wisc.edu

Yi-Hua Chung
University of Wisconsin-Madison
Madison, WI, United States
yihua.chung@wisc.edu

Chih-Chun Chang
University of Wisconsin-Madison
Madison, WI, United States
chih-chun.chang@wisc.edu

Tsung-Yi Ho
The Chinese University of Hong Kong
Shatin, Hong Kong
tyho@cse.cuhk.edu.hk

Tsung-Wei Huang
University of Wisconsin-Madison
Madison, WI, United States
tsung-wei.huang@wisc.edu

Abstract

Quantum circuit simulation (QCS) plays an important role in the designs and analysis of a quantum algorithm, as it assists researchers in understanding how quantum operations work without accessing expensive quantum computers. Despite many QCS methods, they are largely limited to simulating *one input* at a time. However, many simulation-driven quantum computing applications, such as testing and verification, require simulating multiple inputs to reason a quantum algorithm under different scenarios. We refer to this type of QCS as *batch quantum circuit simulation* (BQCS). In this paper, we present `BQSim`, a GPU-accelerated batch quantum circuit simulator. `BQSim` is inspired by the state-of-the-art *decision diagram* (DD) that can compactly represent quantum gate matrices, but overcomes its limitation of CPU-centric simulation. Specifically, `BQSim` uses DD to optimize a quantum circuit for reduced BQCS computation and converts DD to a GPU-efficient data structure. Additionally, `BQSim` employs a task graph-based execution strategy to minimize repetitive kernel call overhead and efficiently overlap kernel execution with data movement. Compared with three state-of-the-art quantum circuit simulators, `cuQuantum`, `Qiskit Aer`, and `FlatDD`, `BQSim` is 3.25×, 159.06×, and 311.42× faster on average.

CCS Concepts: • **Computer systems organization** → **Quantum computing**; • **Computing methodologies** → **Parallel algorithms**; • **Software and its engineering** → **Scheduling**.

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3715984

Keywords: Quantum Circuit Simulation; Decision Diagram; GPU; Task Graph

ACM Reference Format:
Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3715984

1 Introduction

Quantum computing has the potential to tackle certain problems (e.g., cryptography [15], portfolio optimization [19]) that are classically intractable. Among various QC applications [9, 39, 52, 59, 68], *quantum circuit simulation* (QCS) [13, 17, 20, 22, 26, 29, 32, 34, 37, 47, 48, 56, 57, 60, 68, 69, 71, 72] plays a crucial role as it allows researchers to simulate a quantum algorithm on a classical computer without the need to access high-cost quantum computers. However, QCS is computationally demanding because it requires large time and space complexity to compute state amplitudes of qubits. For example, simulating an n-qubit quantum state involves computing 2^n state amplitudes. Furthermore, modern quantum circuits can incorporate thousands of quantum gates to approach quantum advantages [33], making it increasingly challenging to complete QCS within a reasonable runtime.

To mitigate these challenges, existing QCS works have proposed various strategies. For example, [17, 26, 32, 56, 60, 67] introduced multi-threaded QCS algorithms through state partition. [34, 37, 68–70] leveraged SIMD and GPU parallelism to further accelerate QCS. Additionally, [13, 16, 22, 30, 32, 47, 60, 68] optimized the quantum circuit using gate fusion and pattern matching to reduce the amount of QCS computation.

While these strategies can improve the performance of QCS, a fundamental challenge remains unsolved: existing

simulators are largely limited to the *strong scaling* within a *single input* (i.e., state vector). That is, they focus on improving runtime by increasing the number of cores under a single input. However, many simulation-driven quantum applications, such as verification [9], testing [62–64], and state analysis [25, 33, 41], require simulating *multiple inputs* to understand the behavior of a quantum circuit under different scenarios. In these applications, hundreds to thousands of batches of inputs are fed into a quantum circuit for simulation, which we refer to as *batch quantum circuit simulation* (BQCS). While an intuitive approach to BQCS is to fork multiple single-input QCS processes, this type of embarrassingly parallel strategy fails to leverage the substantial data parallelism in BQCS, which could significantly benefit from GPU acceleration.

However, designing a GPU-accelerated batch quantum circuit simulator is non-trivial. First, single-input QCS has been computationally expensive because of superposition [14] and entanglement [24]. Extending it to multiple-input BQCS will further exacerbate this challenge. Second, while many data structures [21, 29, 48, 50, 66, 72] can efficiently represent quantum circuits, most of them are CPU-centric. For example, state-of-the-art decision diagram (DD) [32, 48, 72] has shown promising results in representing gate matrices in a compact graph structure, but all DD-based QCS works are limited to CPU parallelism. Third, extending these CPU-centric data structures to GPU is not straightforward, because GPU has distinct performance and memory models from CPU, requiring specially designed kernel algorithms and memory layout to make the most of GPU parallelism.

To overcome these challenges, we introduce `BQSim`, a GPU-accelerated batch quantum circuit simulator. `BQSim` is inspired by and built upon the state-of-the-art DD-based simulator, `FlatDD` [32], but overcomes its limitation of CPU-only parallelism. We summarize our technical contributions below:

- We introduce a novel gate-fusion algorithm that reduces the amount of BQCS computation by leveraging DD to explore gate matrix sparsity and regularity.
- We introduce a GPU kernel to convert DD into a GPU-efficient data structure that can achieve low thread divergence and efficient memory access patterns in BQCS.
- We introduce a task graph-based execution strategy to minimize repetitive kernel call overhead and efficiently overlap kernel execution with data movement.

We evaluated the performance of `BQSim` on a set of commonly used quantum circuits selected from `MQT-Bench` [51]. Compared with three state-of-the-art simulators, `cuQuantum` [7], `Qiskit Aer` [31], and `FlatDD` [32], `BQSim` is 3.25×, 159.06×, and 311.42× faster on average, respectively. The source code is available at https://github.com/IDEA-CUHK/BQSim.

2 Background

In this section, we first provide an overview of QCS. Next, we introduce DD, an efficient data structure for QCS. Finally, we discuss gate fusion, a key quantum circuit optimization technique.

2.1 Quantum Circuit Simulation

The goal of QCS is to derive the final state after applying all quantum gates in a given quantum circuit to an initial state [32]. A quantum gate and a state are expressed using a *gate matrix* and a *state vector*, respectively. Applying a quantum gate to a state is equivalent to multiplying the state vector by the gate matrix. For example, if we apply a single-qubit Hadamard gate M to a single-qubit input state $|\psi\rangle = |0\rangle$, the resulting state $|\psi'\rangle$ is calculated in Equation 1.

$$|\psi'\rangle = M \cdot |\psi\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\cdot\begin{pmatrix}1\\0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} \tag{1}$$

Equation 1 can be extended to circuits with multiple qubits through *Kronecker product* [31]. However, when simulating circuits with $n > 1$ qubits, we do not have to construct the full $2^n \times 2^n$ gate matrices. Instead, we can update the amplitudes of the state vectors directly [7, 31, 32, 56, 60]. For example, if we apply a single-qubit gate $U = (u_{ij})$ to the k-th qubit of a state vector $|\psi\rangle = (a_i)^T$, the operation to obtain the resulting state vector $|\psi'\rangle = (a'_i)^T$ is calculated in Equation 2.

$$\begin{pmatrix}a'_{*\cdots*0_k*\cdots*}\\ a'_{*\cdots*1_k*\cdots*}\end{pmatrix} = \begin{pmatrix}u_{00} & u_{01}\\ u_{10} & u_{11}\end{pmatrix}\cdot\begin{pmatrix}a_{*\cdots*0_k*\cdots*}\\ a_{*\cdots*1_k*\cdots*}\end{pmatrix} \tag{2}$$

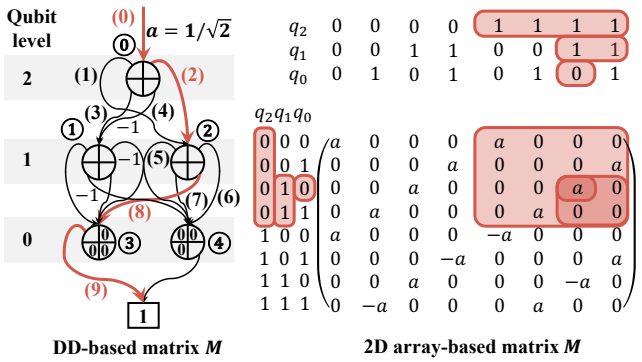
On the other hand, if we apply a controlled gate $V = (v_{ij})$ to a state vector $|\psi\rangle = (a_i)^T$, the resulting state vector $|\psi'\rangle = (a'_i)^T$ is calculated in Equation 3, where c represents the control qubit and t is the target qubit.

$$\begin{pmatrix}a'_{*1_c*\cdots*0_t*\cdots*}\\ a'_{*1_c*\cdots*1_t*\cdots*}\end{pmatrix} = \begin{pmatrix}v_{00} & v_{01}\\ v_{10} & v_{11}\end{pmatrix}\cdot\begin{pmatrix}a_{*1_c*\cdots*0_t*\cdots*}\\ a_{*1_c*\cdots*1_t*\cdots*}\end{pmatrix} \tag{3}$$

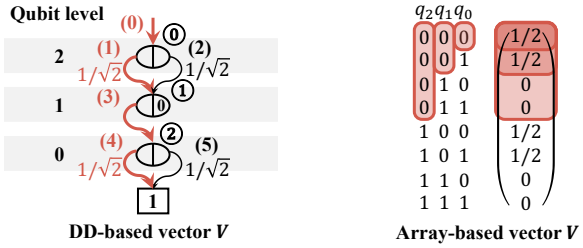
Various types of QCS exist, such as unitary simulation [72], state vector simulation [31], tensor network simulation [66], and density matrix simulation [38]. In this paper, we focus on full-state simulation, which provides the complete set of ideal state amplitudes for each quantum gate, offering a comprehensive view of the quantum circuit's behavior.

2.2 Decision Diagram

DD [48, 72] is a state-of-the-art data structure that efficiently represents quantum gate matrices and state vectors by exploring their regularity and sparsity. Figure 1 shows DD examples for gate matrix M and state vector V. In Figure 1a, we store M in DD, where nodes represent sub-matrices in M, and edges track the values of elements in M. Edge (0) points to the root node ⓪ with an edge weight of $a = 1/\sqrt{2}$.



(a) Matrix M represented in DD (left) and 2D array (right).



(b) Vector V represented in DD (left) and array (right).

Figure 1. Storing array-based matrix and vector using DDs. Unless specified otherwise, all edges have a weight of one.

All edge weights are uniquely determined via *normalization* [48, 72]. Node ⓪ represents the entire M, with four outgoing edges ((1), (2), (3), and (4)) pointing to the four equally partitioned sub-matrices: $M[0:4][0:4]$, $M[0:4][4:8]$, $M[4:8][0:4]$, and $M[4:8][4:8]$. In a DD, each qubit level corresponds to a qubit, ordered from the most to the least significant. Node ⓪ is placed on qubit level 2 because the size of all its sub-matrices are $2^2 \times 2^2$. Sub-matrices $M[0:4][0:4]$ and $M[0:4][4:8]$ are identical, while sub-matrices $M[4:8][0:4]$ and $M[4:8][4:8]$ are opposites. Consequently, $M[0:4][0:4]$ and $M[0:4][4:8]$ can share node ②, and $M[4:8][0:4]$ and $M[4:8][4:8]$ can share node ① with opposite incoming edge weights. Likewise, the sub-matrices represented by nodes ① and ② can be further equally partitioned into $2^1 \times 2^1$ sub-matrices. For example, edges (5), (6), (7), and (8) point to sub-matrices $M[0:2][4:6]$, $M[0:2][6:8]$, $M[2:4][4:6]$, and $M[2:4][6:8]$, respectively. Among these, $M[0:2][4:6]$ and $M[2:4][6:8]$ are identical, and $M[0:2][6:8]$ and $M[2:4][4:6]$ are identical as well. Therefore, edges (5) and (8) point to the shared node ③, and edges (6) and (7) point to the shared node ④. Node ③ represents matrix $\left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)$. Among the four outgoing edges of node ③, the upper-left edge points to the constant-one node, and the other three edges are set to constant zero. The same applies to node ④.

The matrix value at an index pair is the product of the edge weights along the corresponding path in DD. For example, for $M[2][6]$, we have $q_1 = 1$ and $q_2 = q_0 = 0$ for its row index 2; and $q_2 = q_1 = 1$ and $q_0 = 0$ for its column index 6. Therefore, node ⓪, at qubit level 2, chooses edge (2) (i.e., $q_2 = 0$ for row and $q_2 = 1$ for column), pointing to node ②, which represents $M[0:4][4:8]$ (shaded in red in Figure 1a). Node ② then chooses edge (8) (i.e., $q_1 = 1$ for both row and column), which points to node ③, representing $M[2:4][6:8]$. Finally, node ③ chooses edge (9) (i.e., $q_0 = 0$ for both row and column), pointing to the constant-one node. The value of $M[2][6]$ is the product of the weights of edges (0), (2), (8), and (9) (red thick edges in Figure 1a), resulting in $M[2][6] = 1/\sqrt{2}$.

We can represent state vector V in Figure 1b using DD in a similar fashion. Root node ⓪ represents the entire V, with its two outgoing edges (i.e., (1) and (2)) pointing to the equally partitioned sub-vectors $V[0:4]$ and $V[4:8]$. Node ⓪ is placed on qubit level 2 because the length of its sub-vectors is 2^2. Since sub-vectors $V[0:4]$ and $V[4:8]$ are identical, they can share node ① with equal edge weights of $1/\sqrt{2}$, determined by normalization. After dividing the common edge weight $1/\sqrt{2}$, we can further partition the sub-vector represented by node ① (i.e., $\left(1/\sqrt{2}\ \ 1/\sqrt{2}\ \ 0\ \ 0\right)^T$) into $\left(1/\sqrt{2}\ \ 1/\sqrt{2}\right)^T$ and $\left(0\ \ 0\right)^T$. The former is represented by node ②, while the latter is set to the constant-zero edge. Finally, both of node ②'s outgoing edges point to the constant-one node with equal edge weights of $1/\sqrt{2}$.

The vector value at a certain index is the product of the edge weights along the corresponding edge path in DD. For example, for index 0, we have $q_2 = q_1 = q_0 = 0$. Therefore, node ⓪ chooses edge (1) (i.e., $q_2 = 0$), leading to node ①, which represents $V[0:4]$ (shaded in red in Figure 1b). Next, node ① chooses edge (3), leading to node ②, which represents $V[0:2]$. Finally, node ② chooses edge (4), pointing to the constant-one node. The value of $V[0]$ equals the product of the weights of edges (0), (1), (3), and (4) (red thick edges in Figure 1b), resulting in $V[0] = 1/2$.

DD can effectively compress the computation of quantum gate matrices and state vectors. For instance, computing the DD-based matrix M in Figure 1a involves only 26 edges and six nodes, compared to 64 nodes in 2D array-based M. DD-based matrix-vector and matrix-matrix multiplications are performed using depth-first-search (DFS), where the multiplication is partitioned into many smaller multiplications among sub-matrices and sub-vectors. The results of repeated multiplications are reused via caches [48, 71, 72].

2.3 Gate Fusion

To efficiently simulate large circuits containing thousands of gates (e.g., QNN [33]), simulators often fuse multiple gate matrices into a single equivalent gate matrix [13, 22, 32, 47, 60] to reduce the amount of QCS computation. Among various gate-fusion algorithms, array-based gate fusion [13, 22, 60],

which fuse gates represented in 2D arrays, has proven effective. However, array-based gate fusion becomes inefficient as the size of the fused gate matrix grows exponentially with the number of qubits. For example, a 2D array-based ten-qubit fused gate matrix would be 1024×1024, making further fusion difficult. In contrast, recent DD-based gate fusion [32] offers a more efficient approach by representing gate matrices in DD, which explores the sparsity and regularity of gate matrices. DD-based gate fusion avoids redundant zeros and repeated sub-matrices, enabling the fusion of additional gates without significant overhead, thereby effectively reducing the amount of QCS computation.

3 Algorithm

In this section, we discuss the technical details of our `BQSim` algorithm. As shown in Figure 2, `BQSim` consists of three stages: BQCS-aware gate fusion, DD-to-ELL conversion, and task graph-based execution.

In stage ① (Section 3.1), we reduce the amount of BQCS computation by leveraging DD to explore gate matrix sparsity and regularity. However, extending the fused gates in DD to GPU is non-trivial, because GPU has distinct performance and memory models from CPU. Therefore, in stage ② (Section 3.2), we convert the fused gates from DD to a GPU-efficient data structure, ELL [36, 53], to achieve low thread divergence and efficient memory access patterns in BQCS. Finally, in stage ③ (Section 3.3), we introduce a task graph-based execution strategy to simulate the fused gates in ELL across multiple input batches. This strategy can minimize repetitive kernel call overhead and efficiently overlap kernel execution with data movement.

3.1 BQCS-aware Gate Fusion

BQCS-aware gate fusion is performed using DD to explore gate matrix sparsity and regularity. However, existing DD-based gate fusion [32] is limited to optimizing CPU-based single-input QCS. To extend this gate-fusion algorithm to GPU-accelerated BQCS, we first define and calculate the BQCS cost of each gate (Section 3.1.1), and then fuse all the gates in a circuit based on their BQCS costs (Section 3.1.2).

3.1.1 BQCS Cost for Each Gate. As we shall discuss, `BQSim` eventually models BQCS as ELL-based sparse-matrix multiplication (spMM) (Section 3.3.1). Since the core computation of matrix multiplication is multiply-accumulate (MAC) operations [10], we measure the BQCS cost using the number of MAC operations (i.e., #MAC). For any gate matrix represented in ELL, the #MAC needed to compute every state amplitude is the same, which is equal to the maximum number of non-zero elements per row (NZR) of the gate matrix (Section 3.2). Therefore, we can define the BQCS cost of a gate matrix as its maximum NZR.

Finding the maximum NZR of an $n-$qubit gate matrix is challenging due to its 2^n rows. To address this issue, we observe that the NZR values across the rows exhibit a regular pattern, making them efficient to process using DD. Consequently, we store the NZR of all rows in a NZR vector (NZRV) and use DD to represent the NZRV.

Figure 3 shows the algorithm to find the DD-based NZRV of a DD-based gate matrix M. We use DFS to traverse the nodes in M, where each node represents a sub-matrix. We recursively find the NZRV of each node (i.e., sub-matrix), and derive the NZRV of M. To avoid repeated computations, the NZRV of each node is stored in a map T. In the map T shown in Figure 3, only the first column (i.e., node) and the last column (i.e., DD-based NZRV) are stored in memory, while the two middle columns are included only for illustration. For each node, we traverse its outgoing edges from top to bottom, left to right. Starting with node ⓪, we move through node ① to arrive at node ②, which represents $\left(\begin{smallmatrix}1 & 0\\ 0 & 0\end{smallmatrix}\right)$, whose NZRV is $\left(\begin{smallmatrix}1\\ 0\end{smallmatrix}\right)$. Then, we proceed to node ③, whose NZRV is $\left(\begin{smallmatrix}1\\ 1\end{smallmatrix}\right)$. Returning to node ①, the NZRVs of the top and bottom rows are simply those of nodes ② and ③. We can concatenate the NZRVs of nodes ② and ③ using native DD operation `DDConcatenate` [72] (i.e., T[①]=`DDConcatenate`(T[②], T[③])), yielding $(1\ 0\ 1\ 1)^T$ in array representation. Similarly, the NZRV of node ④ is $(1\ 1\ 1\ 0)^T$. For the root node ⓪, which represents M, the NZRVs of the top and bottom rows are calculated by summing the NZRVs of nodes ① and ④, using native DD operation `DDAdd` [72]. Then, we concatenate the NZRVs of the top and bottom rows (i.e., T[⓪]=`DDConcatenate`(`DDAdd`(T[①], T[④]), `DDAdd`(T[④], T[①]))). Finally, the maximum NZR of M (i.e., 2) is obtained through DFS traversal of T[⓪] (the DD-based NZRV of M).

3.1.2 BQCS Cost-based Gate Fusion. Figure 4 illustrates the three steps in BQCS cost-based gate fusion. In steps ① and ②, we fuse DDs with low BQCS costs (i.e., 1 and 2) to reduce the total number of gates used in step ③, where more expensive DD multiplications are performed through greedy fusion. We fuse gates by multiplying their DD-based gate matrices using native DD operation `DDMultiply` [72].

In step ①, we fuse consecutive diagonal or permutation gates, all of which have a BQCS cost of 1. The resulting fused gate remains a diagonal or permutation gate with a BQCS cost of 1. As a result, the overall BQCS cost is reduced from the sum of individual costs to a single cost of 1. For example, in Figure 4, gates M_2, M_3, M_6, and M_7 in G, along with the resulting fused gates M_{32} and M_{76} in F_1, all have a cost of 1. In step ②, we fuse every two consecutive gates, each with a cost of 2, resulting in a fused gate with a BQCS cost of 4. For instance, in Figure 4, gates M_1 and M_0 in F_1 each have a cost of 2, while the fused gate M_{10} in F_2 has a cost of 4. Although the overall BQCS cost remains unchanged (i.e., $2 + 2 = 4$), simulating a single fused gate instead of two reduces memory loads and stores, thereby improving BQCS

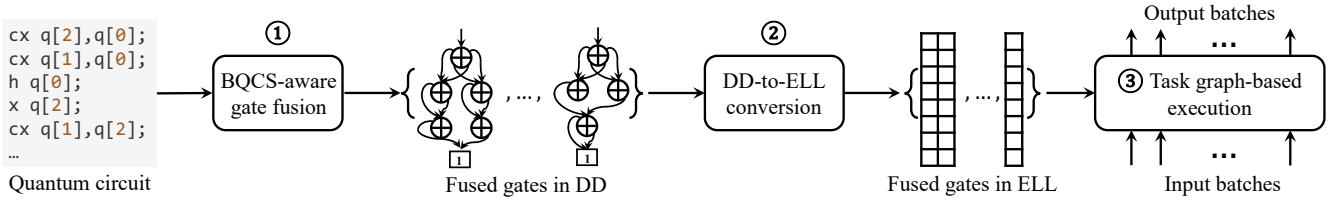


Figure 2. Overview of `BQSim` algorithm. ① We apply BQCS-aware gate fusion to reduce the amount of BQCS computation by leveraging DD to explore gate matrix sparsity and regularity. ② We convert the fused gates from DD to ELL, to achieve low thread divergence and efficient memory access patterns in BQCS. ③ We introduce a task graph-based execution strategy, which simulates the fused gates in ELL across multiple input batches. This strategy can minimize repetitive kernel call overhead and efficiently overlap kernel execution with data movement.

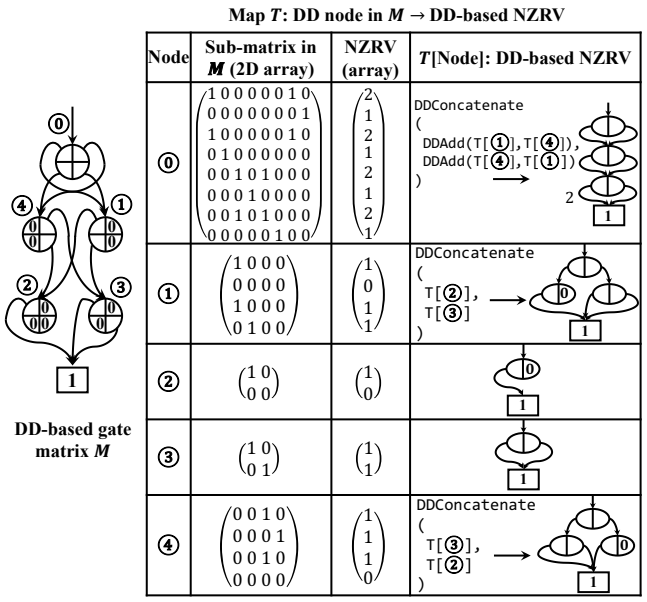


Figure 3. Determining the NZRV of a DD-based gate matrix M. Each DD node in M represents a sub-matrix, and map T is used to track the NZRV of each sub-matrix. Unless specified otherwise, all edge weights are equal to one.

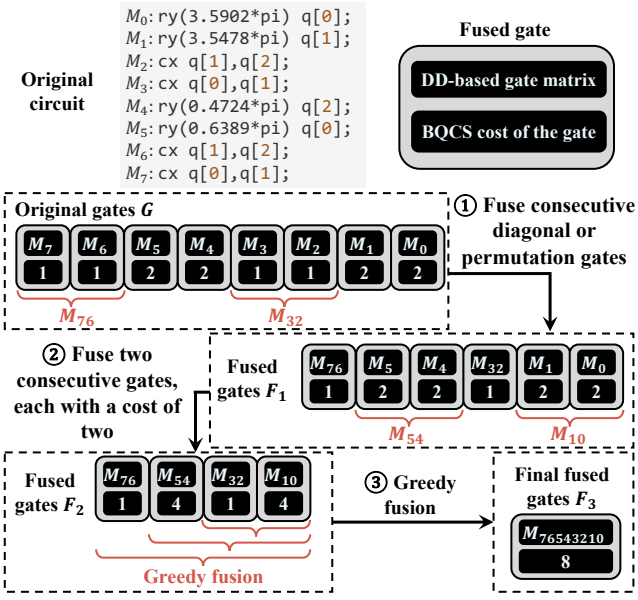


Figure 4. Steps of our BQCS cost-based gate fusion. ① Fuse consecutive diagonal or permutation gates. ② Fuse every two consecutive gates, each with a cost of two. ③ Apply greedy fusion to the result of the previous step.

performance. Finally, in step ③, we apply the greedy fusion algorithm from `FlatDD` [32] to the fusion result of step ② (i.e., F_2 in Figure 4). This algorithm fuses gates only when the resulting gate has a lower BQCS cost. For example, the BQCS costs of gates M_{10} and M_{32} are 4 and 1, respectively, while gate M_{3210} has a cost of 4. Since the fusion reduces the overall cost (i.e., $4 < 4 + 1$), we fuse M_{10} and M_{32} into M_{3210}.

3.2 DD-to-ELL Conversion

The goal of DD-to-ELL conversion is to convert the fused gates from DD to a GPU-efficient data structure, ELL [36, 53]. We choose ELL over other sparse formats (e.g., CSR, COO) because the NZRs of quantum gate matrices are distributed roughly uniformly across rows, which is best suited for ELL [8]. To demonstrate this uniformity, we calculate the average coefficient of variation (CV) of NZRs in gate matrices used for BQCS-aware gate fusion across four quantum circuits [6, 51], as shown in Table 1. CV quantifies the relative variability of a series of values [5], where a lower CV indicates lower variability and greater uniformity [18, 55]. In Table 1, the $\overline{CV}$ values for VQE, QNN, and TSP circuits are zero, indicating that NZRs are uniform across all rows for all gates. While the quantum supremacy circuit exhibits a nonzero $\overline{CV}$, its value of 0.0328 remains low, indicating high uniformity in practice [55, 65]. This uniformity allows for balanced partitioning of GPU threads across rows in the ELL format, resulting in low thread divergence and efficient memory access patterns in BQCS.

Table 1. The average coefficient of variation (CV) of NZRs in gate matrices used for BQCS-aware gate fusion across four quantum circuits: Quantum supremacy ($n = 12$), VQE ($n = 16$), QNN ($n = 17$), and TSP ($n = 16$).

Circuits	Supremacy ($n = 12$)	VQE ($n = 16$)	QNN ($n = 17$)	TSP ($n = 16$)
$\overline{CV}$ of NZR	0.0328	0	0	0

As shown in Figure 7a, the DD-based gate matrix M is converted to the ELL-based M. The 2D array-based M included in Figure 7a is only for illustration and is not stored in memory. For each row in the 2D array-based M, which corresponds to a combination of DD edges in the DD-based M, ELL stores the non-zero elements in the ELL value matrix and their column indices in the ELL column index matrix. Each row in the ELL value matrix corresponds to a state amplitude, and the #MAC needed to compute that amplitude is the number of values in the row, which is the number of columns in the ELL value matrix. This number is determined by the maximum NZR of gate matrix M, as rows with fewer non-zero elements than the maximum NZR are padded with zeros (e.g., rows 1 and 5 in Figure 7a).

An intuitive approach for DD-to-ELL conversion is traversing all DD edge combinations to generate the ELL value and column index matrices on CPU (i.e., CPU-based conversion). However, CPU-based conversion has exponential time complexity, as there are 2^n rows for an n-qubit gate matrix. To address this issue, we leverage GPU to convert the rows in parallel. While GPU-based conversion is typically faster, Figure 5a shows that for certain gates, CPU-based conversion can outperform GPU-based conversion. Additionally, Figure 5b shows that CPU-based conversion can outperform GPU-based conversion more significantly as the number of DD edges increases. This is because more DD edges introduce more branches, resulting in greater GPU thread divergence and memory inefficiency.

Therefore, to optimize the conversion performance, we introduce a hybrid approach that adaptively selects either GPU-based or CPU-based conversion based on the number of DD edges. We refer to this approach as *hybrid conversion*. If a DD has more edges than a specified threshold τ, we use CPU-based conversion; otherwise, we use GPU-based conversion. The best selection strategy of τ depends on the DD structure and the hardware used for BQCS. Since there is no universally optimal value, we parameterize it for different applications. In Section 3.2.1, we introduce DD storage on GPU (i.e., GPU-based DD). Then, in Section 3.2.2, we introduce a GPU kernel to convert the GPU-based DD to ELL.

3.2.1 GPU-based DD. Figure 6 shows a GPU-based DD using an edge array and a node array. Each edge contains a weight and a pointer to the node it connects to, while each

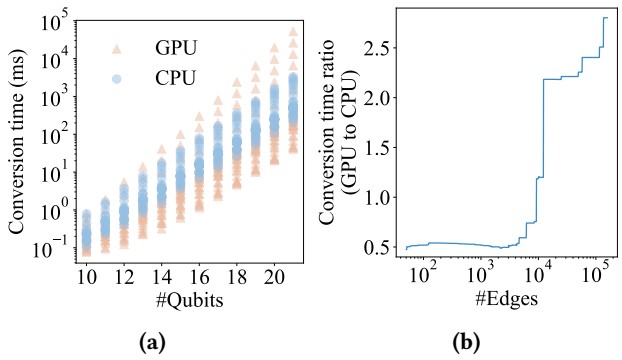


Figure 5. Comparison of GPU-based and CPU-based DD-to-ELL conversion. The data are collected by converting the gates of various circuits. (a) GPU-based and CPU-based conversion time vs. the number of qubits. (b) GPU-based to CPU-based conversion time ratio vs. the number of edges.

node contains its qubit level and pointers to four outgoing edges. Special cases, such as pointers to the constant-one node and the constant-zero edge, are represented by null pointers (i.e., $\varnothing$). The edges and nodes are stored in the edge array and the node array, respectively. For example, in Figure 6, edge (0) in the edge array has weight w_0 and points to node ⓪ in the node array. Node ⓪, located at qubit level 2, points to four edges in the edge array: (1), (8), (12), and (13). The connections between the edge array and the node array efficiently forms the DD structure on GPU.

3.2.2 DD-to-ELL Conversion on GPU. We describe the conversion algorithm using both Algorithm 1 (GPU-based conversion CUDA kernel) and an example in Figure 7. For an $n-$qubit gate, the kernel is launched with a number of blocks equal to the number of rows in the ELL representation, with each block generating a single row. The number of threads in each block is equal to the number of qubits. For example, in Figure 7a, block 6 reads the red edges in the DD-based gate matrix M (corresponding to row 6 of the 2D array-based M), and writes the result to row 6 of the ELL value and column index matrices.

In Algorithm 1, each block runs an independent DFS to generate a row in the ELL representation. Since recursion is not efficient on GPU, we implement DFS using iteration with a custom stack for tracking edges: *edge_stack*. To track the traversal direction for a node at each qubit level, we use two arrays: $left\_right$ and *up_down* (line 1). These arrays, along with the stack, are stored in GPU shared memory to reduce memory latency [54]. Initially, $left\_right$ is set to 0 (i.e., left) for all qubit levels, while *up_down* is initialized based on the block index (lines 2-5). For example, the *up_down* array for block 6 is {1,1,0} (i.e., {down, down, up}). We initialize the stack pointer *stack_ptr* to 0, and push edge (0) onto *edge_stack* (line 7). We initialize current gate matrix value *val* and column index *col* to 1 and 0, respectively (line 8).

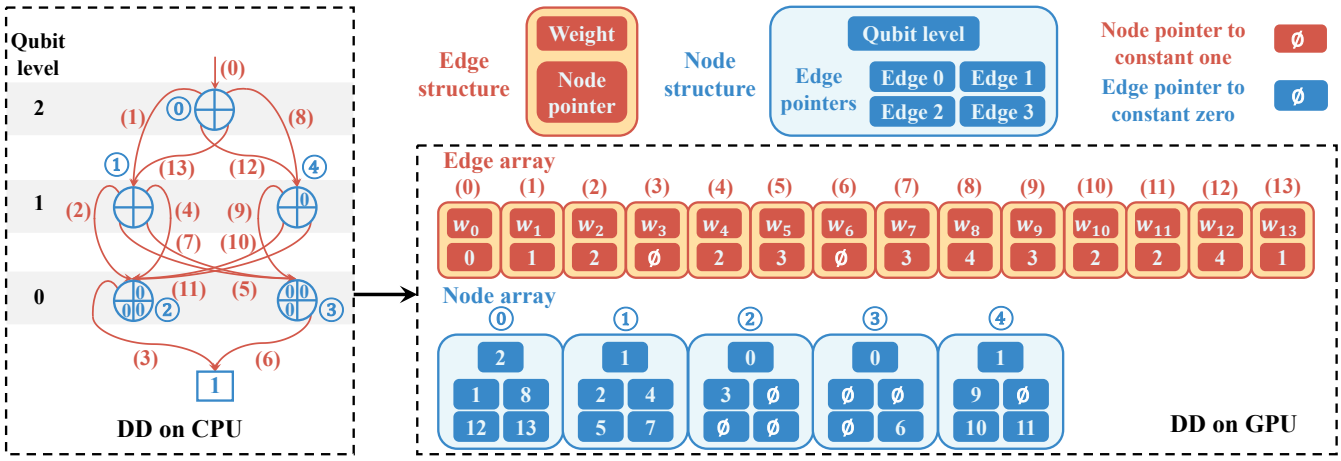
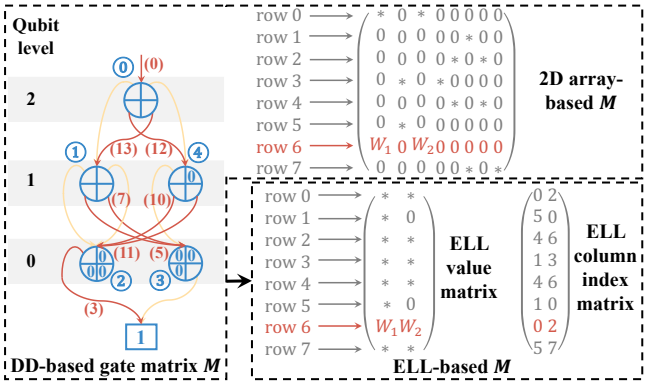
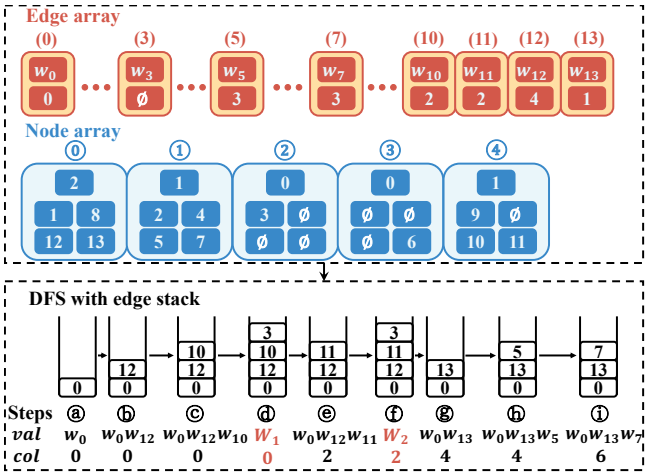


Figure 6. GPU-based DD using a node array and an edge array.



(a) The ELL representation of a DD-based gate matrix M.



(b) Example: Block 6 traverses M (GPU-based DD representation) in DFS to generate row 6 in the ELL representation.

Figure 7. Converting a gate matrix M from DD to ELL. $W_1 = w_0w_{12}w_{10}w_3$. $W_2 = w_0w_{12}w_{11}w_3$.

Next, recursion begins and continues until the edge stack is empty. We first find the current edge and its corresponding node. If the edge is the constant-zero edge, *edge_stack* pops this edge and returns. If the node is the constant-one node, the block writes the results to the ELL value and column index matrices, and *edge_stack* pops this edge and returns (lines 9-17). If $left\_right[stack\_ptr]$ equals two, this means that the left and right directions of this node has already been traversed (line 18). Since this node is fully explored, we reset $left\_right[stack\_ptr]$ to 0 for other nodes at this qubit level and pop this edge. We also restore variables *val* and *col* (lines 19-21). If the node is not fully explored, we visit its unvisited edges, push the edge onto *edge_stack*, and update *val* and *col* as described in `FlatDD` [32] (lines 22-28).

In Figure 7b, block 6 traverses the GPU-based DD in DFS. From step ⓐ to step ⓓ, block 6 moves through edges (0), (12), (10) and (3), reaching the constant-one node. At step ⓓ, block 6 writes $val = W_1 = w_0w_{12}w_{10}w_3$ and $col = 0$ to row 6 of the ELL value and column index matrices, respectively. From step ⓓ to step ⓕ, edges (3) and (10) are popped from the edge stack and edges (11) and (3) are pushed onto it. At the same time, *val* is updated by dividing it by w_3 and w_{10}, and then multiplying it by w_{11} and w_3, resulting in W_2. *col* is updated by adding 2^1. At step ⓕ, block 6 writes $val = W_2 = w_0w_{12}w_{11}w_3$ and $col = 2$ to row 6 of the ELL value and column index matrices, respectively.

3.3 Task Graph-based Execution

To simulate a quantum gate on a batch of state vectors, we call the BQCS kernel (Section 3.3.1) on the GPU. While the overhead of invoking a kernel is typically small, this overhead will become significant when simulating large circuits with many gates and batches due to repetitive kernel calls. Moreover, the CPU-GPU data movement overhead for batch inputs and outputs also accumulates as the number of batches increases.

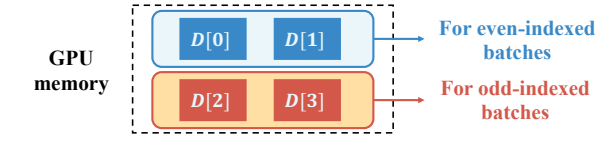
Algorithm 1 GPU kernel to convert DD to ELL

Input: *Edges*: edge array, *Nodes*: node array, V: ELL value matrix, C: ELL column index matrix, $MNZR$: maximum NZR of the gate matrix for conversion

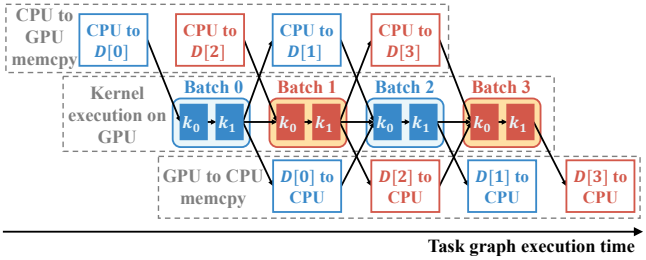
```
1:  shared edge_stack[n], left_right[n], up_down[n]
2:  tid = threadIdx.x; bid = blockIdx.x
3:  left_right[tid] = 0
4:  up_down[n − 1 − tid] = bid&(1 << tid)
5:  __syncthreads()
6:  if tid == 0 then
7:    stack_ptr = 0; edge_stack[stack_ptr] = 0
8:    val = 1; col = 0; idx = 0
9:    while stack_ptr ≥ 0 do
10:     edge_ptr = edge_stack[stack_ptr]
11:     if edge_ptr == ∅ then
12:       stack_ptr − −; continue
13:     node_ptr = Edges[edge_ptr].node_pointer
14:     if node_ptr == ∅ then
15:       C[bid · MNZR + idx] = col
16:       V[bid · MNZR + idx] = val · Edges[edge_ptr].
          weight
17:       stack_ptr − −; idx + +; continue
18:     if left_right[stack_ptr] == 2 then
19:       left_right[stack_ptr] = 0; stack_ptr − −
20:       val = val/Edges[edge_ptr].weight
21:       col = col − (1 << Nodes[node_ptr].qubit_lv)
22:     else
23:       child_idx = 2·up_down[stack_ptr]+left_right
          [stack_ptr]
24:       left_right[stack_ptr] + +
25:       val = (left_right[stack_ptr] == 1) ? val ·
          Edges[edge_ptr].weight : val
26:       col = col + (left_right[stack_ptr] − 1) · (1 <<
          Nodes[node_ptr].qubit_lv)
27:       edge_stack[stack_ptr] = Nodes[node_ptr].e-
          dge_pointers[child_idx]
28:       stack_ptr + +
```

To address these issues, we describe the kernel execution and the data movement as tasks and organize them into a *task graph* based on their dependencies. By delegating the scheduling of task graph to a GPU runtime, such as CUDA Graph [43, 61], SYCL Graph [11], or HIP Graph [3], we can minimize repetitive kernel call overhead while overlapping kernel execution with data movement. In Section 3.3.1, we introduce the BQCS kernel, and in Section 3.3.2, we define the task dependencies.

3.3.1 BQCS Kernel. The BQCS kernel is an ELL-based spMM kernel, where the gate matrix, represented in ELL, multiplies a matrix that represents a batch of state vectors. Since task graph-based execution imposes no restrictions on the kernel, any ELL-based spMM kernel can be easily integrated into the task graph as the BQCS kernel. In this paper, we implement a custom BQCS kernel based on previously studied ELL-based spMM algorithms [45, 46].



(a) Memory buffers in GPU memory.



(b) Dependencies in our task graph for BQCS execution.

Figure 8. Memory buffers and task dependencies.

3.3.2 Task Dependencies. To overlap the execution of BQCS kernels with data movement tasks across batches, we use multiple *memory buffers* [42, 44] to store different BQCS inputs and outputs on GPU at the same time. As shown in Figure 8a, we use two buffers ($D[0]$ and $D[1]$) to store the inputs and outputs of even-indexed batches, and another two buffers ($D[2]$ and $D[3]$) to store the inputs and outputs of odd-indexed batches. Specifically, for batch I_B, the input and output of kernel I_k are determined by $D[2(I_B\%2) + (\lfloor I_B/2 \rfloor \cdot (L+1)+I_k)\%2]$ and $D[2(I_B\%2)+(\lfloor I_B/2 \rfloor \cdot (L+1)+I_k+1)\%2]$, respectively. Here, L is the total number of BQCS kernel calls per batch.

Figure 8b shows the dependencies of our task graph for BQCS execution across multiple batches. There are four batches, each with two BQCS kernels, k_0 and k_1. Before simulating batch 0, its input is copied from the CPU to $D[0]$. Kernel k_0 reads from $D[0]$ (i.e., $D[2(0\%2)+(\lfloor 0/2 \rfloor \cdot (2+1)+0)\%2]$) and writes to $D[1]$. Then, k_1 reads from $D[1]$ and writes back to $D[0]$. Meanwhile, the CPU copies the input for the coming batch 1 to $D[2]$ (i.e., $D[2(1\%2) + (\lfloor 1/2 \rfloor \cdot (2+1) + 0)\%2]$). After batch 0 finishes, batch 1 begins. Similarly, kernel k_0 reads from $D[2]$ and writes to $D[3]$ (i.e., $D[2(1\%2) + (\lfloor 1/2 \rfloor \cdot (2+1) + 1)\%2]$), and then k_1 reads from $D[3]$ and writes to $D[2]$. Meanwhile, the result from batch 0 (i.e., $D[0]$) is copied back to the CPU, and the input of batch 2 is copied from the CPU to $D[1]$ (i.e., $D[2(2\%2) + (\lfloor 2/2 \rfloor \cdot (2+1) + 0)\%2]$). Both of these copies must finish before batch 2 starts. The same process applies to the remaining batches.

4 Experimental Results

We evaluated the performance of BQSim on 16 medium and large quantum circuits selected from MQT-Bench [51], which are representative of different quantum applications including machine learning and portfolio optimization. In Section 4.2, we compare the runtime of BQSim with three baselines. We then evaluate the performance of the three stages of BQSim separately: BQCS-aware gate fusion (Section 4.3), DD-to-ELL conversion (Section 4.4), and task graph-based execution (Section 4.5). In Section 4.6, we demonstrate the scalability of BQSim over increasing batch sizes. In Section 4.7, we evaluate the power consumption of BQSim. In Section 4.8, we study the runtime breakdown of the three stages of BQSim. Finally, in Section 4.9, we perform an ablation study to evaluate how each stage contributes to the performance of BQSim.

For each simulation run, we randomly generate 200 input batches, each of 256 inputs (i.e., state vectors), based on the popular BQCS settings [33, 62, 63]. We set the threshold τ to 2000, which can yield decent performance on our machine. All experiments are conducted on a Ubuntu 22.04.3 LTS machine with 16 Intel i7-11700 CPU cores at 2.50 GHz, one RTX 48 GB A6000 GPU, and 128 GB RAM. We implement our task graph using Taskflow [27, 28], which provides a high-level C++ wrapper over CUDA Graph. For data with exponential differences, we measure the average in geometric mean. We validate BQSim by comparing our simulation results with the baselines, where we observe identical state amplitudes in the output.

4.1 Baselines

Given the page limit and many simulators, it is impossible to compare BQSim with all of them. Instead, we consider three representative simulators as our baselines, cuQuantum [7], Qiskit Aer [31], and FlatDD [32], for two reasons: (1) All the baselines are open-source [1, 2, 4], allowing us to fairly study and reason their results. (2) All the baselines are highly optimized. cuQuantum and Qiskit Aer are GPU-accelerated QCS libraries. FlatDD is a recent simulator that has successfully parallelized DD using manycore CPU. Both Qiskit Aer and FlatDD also provide powerful gate-fusion algorithms.

cuQuantum only provides gate-level BQCS via the custatevecApplyMatrixBatched API [12]. To simulate the whole circuit, we apply this API iteratively to each quantum gate. Since Qiskit AER and FlatDD do not support BQCS at all, we run eight processes to simulate eight input states in parallel. This configuration of process-level parallelism provides the best throughput performance on our machine. Additionally, each FlatDD process uses 16 CPU threads at which its performance saturates on our machine.

4.2 Overall Comparison

Table 2 compares the overall runtime of BQSim with cuQuantum, Qiskit Aer, and FlatDD. Unless otherwise specified, all runtimes are measured in milliseconds (ms), and we terminate the runs that take longer than 24 hours.

BQSim is faster than cuQuantum, Qiskit Aer, and FlatDD on all quantum circuits. On average, BQSim is 3.25×, 159.06×, and 331.42× faster than cuQuantum, Qiskit Aer, and FlatDD, respectively. The largest speed-up is observed when compared with FlatDD, which is limited to CPU parallelism. While Qiskit Aer has GPU acceleration, BQSim significantly outperforms Qiskit Aer due to its lack of support for BQCS. Although GPU-based cuQuantum supports gate-level BQCS, BQSim is still 3.25× faster than cuQuantum. We attributes this speed benefit to our BQCS-aware gate fusion and task graph-based execution, which reduce the amount of BQCS computation and provide scheduling-level enhancements.

The speed-up of BQSim over the baselines become more remarkable when simulating large quantum circuits. For example, when simulating the 16-qubit VQE of 78 gates, BQSim is 2.46× faster than cuQuantum; when simulating the larger 16-qubit Portfolio optimization of 424 gates, BQSim is 5.10× faster than cuQuantum.

Table 2 presents results for circuits with up to 21 qubits. However, BQSim can scale to larger qubit counts by adjusting batch sizes and leveraging multiple GPUs. For instance, the batch of state vectors can be partitioned across multiple GPUs where each GPU can afford a larger qubit count. This indeed highlights another advantage of BQSim: the circuit is optimized once into a reusable simulation task graph that can run different batches on multiple GPUs.

4.3 Evaluation of BQCS-aware Gate Fusion

In Table 3, we evaluate the effectiveness of BQSim's BQCS-aware gate fusion by comparing the resulting #MAC among BQSim and the baselines.

BQSim outperforms cuQuantum, Qiskit Aer, and FlatDD across all quantum circuits in terms of #MAC. BQSim achieves the largest improvement compared with cuQuantum, which is 10.76× on average. This is because cuQuantum does not fuse gates. While Qiskit Aer incorporates a powerful gate-fusion algorithm, BQSim is still 3.85× faster on average. This is because Qiskit Aer's gate fusion is limited to array-based gate matrix representation, which computes redundant zeros and repeated sub-matrices (Section 2.3). Although FlatDD efficiently avoid such redundancy by representing gates with DD, BQSim is still 1.23× faster on average, because FlatDD's gate-fusion algorithm is limited to optimizing CPU-based single-input QCS.

4.4 Evaluation of DD-to-ELL Conversion

Figure 9 compares the runtimes of GPU-based, CPU-based, and hybrid DD-to-ELL conversions across five circuits with different numbers of qubits: QNN ($n = 17, 19, 21$), VQE ($n = 16$), and TSP ($n = 16$). For QNN ($n = 17, 19, 21$), hybrid conversion yields the best performance. For instance, on QNN of $n = 17$, GPU-based and CPU-based conversions are 1.96×

Table 2. Comparison of runtimes among BQSim (ours), cuQuantum, Qiskit Aer, and FlatDD on 16 circuits representative of different quantum applications.

Circuits			Runtime (ms)				Speed-up of BQSim over		
Name	#Qubits (n)	#Gates	cuQuantum	Qiskit Aer	FlatDD	BQSim (ours)	cuQuantum	Qiskit Aer	FlatDD
QNN	17	934	246280	1663228	24195648	**24218**	10.17×	68.68×	999.08×
	19	1158	1181539	5491441	>24 h	**113254**	10.43×	48.49×	>762.89×
	21	1406	5598394	20428647	>24 h	**517621**	10.82×	39.47×	>166.92×
VQE	12	58	1433	394267	167565	**884**	1.62×	446.00×	189.55×
	14	68	5901	470945	576705	**2495**	2.37×	188.76×	231.14×
	16	78	24619	874623	2442323	**10026**	2.46×	87.24×	243.60×
Portfolio opt.	16	424	56934	1035447	3393370	**11159**	5.10×	92.79×	304.09×
	17	476	122784	1755908	6979064	**24551**	5.00×	71.52×	284.27×
	18	531	264992	3135291	15009161	**51675**	5.13×	60.67×	290.45×
Graph state	16	32	18424	872669	1056870	**9822**	1.88×	88.85×	107.60×
	18	36	75305	2923585	4537635	**39611**	1.90×	73.81×	114.55×
	20	40	308446	10285365	20036118	**157555**	1.96×	65.28×	127.17×
TSP	9	94	245	373035	130619	**138**	1.78×	2703.15×	946.51×
	16	171	36083	886423	3986412	**16435**	2.20×	53.94×	242.56×
Routing	6	39	51	363760	54736	**31**	1.65×	11734.19×	1765.68×
	12	81	1628	392998	240627	**666**	2.44×	590.09×	361.30×

Table 3. Comparison of #MAC (smaller is better) among BQSim, cuQuantum, Qiskit Aer, and FlatDD on 16 circuits representative of different quantum applications.

Circuits			#MAC				Improvement vs.		
Name	#Qubits (n)	#Gates	cuQuantum	Qiskit Aer	FlatDD	BQSim (ours)	cuQuantum	Qiskit Aer	FlatDD
QNN	17	934	489684992	60162048	19791872	**17301504**	28.30×	3.48×	1.14×
	19	1158	2428502016	266338304	87556096	**77594624**	31.30×	3.43×	1.13×
	21	1406	11794382848	1178599424	383778816	**343932928**	34.29×	3.43×	1.12×
VQE	12	58	950272	360448	348160	**253952**	3.74×	1.42×	1.37×
	14	68	4456448	1703936	1589248	**1277952**	3.49×	1.33×	1.24×
	16	78	20447232	7864320	7602176	**6029312**	3.39×	1.30×	1.26×
Portfolio opt.	16	424	111149056	92798976	8912896	**8388608**	13.25×	11.06×	1.06×
	17	476	249561088	210763776	19267584	**17825792**	14.00×	11.82×	1.08×
	18	531	556793856	475004928	39845888	**37748736**	14.75×	12.58×	1.06×
Graph state	16	32	8388608	4194304	2293760	**2097152**	4.00×	2.00×	1.09×
	18	36	37748736	18874368	10223616	**9437184**	4.00×	2.00×	1.08×
	20	40	167772160	83886080	45088768	**41943040**	4.00×	2.00×	1.08×
TSP	9	94	192512	81920	95232	**55296**	3.48×	1.48×	1.72×
	16	171	44826624	19660800	17432576	**12582912**	3.56×	1.56×	1.39×
Routing	6	39	9984	3840	4736	**3072**	3.25×	1.25×	1.54×
	12	81	1327104	540672	499712	**393216**	3.38×	1.38×	1.27×

and 1.56× slower than hybrid conversion. This is because some gates have simpler DD structures, better suited for the GPU, while others are more complex with many branches, which run more efficiently on the CPU. Hybrid conversion outperforms both by adaptively selecting either GPU-based or CPU-based conversion depending on the structural complexity of each DD. For VQE of $n = 16$ and TSP of $n = 16$, GPU-based and hybrid conversions both deliver the best performance. We attribute this result to the simpler DD structures in these circuits, which are more efficient on GPU, and hybrid conversion always selects GPU-based conversion for optimal performance.

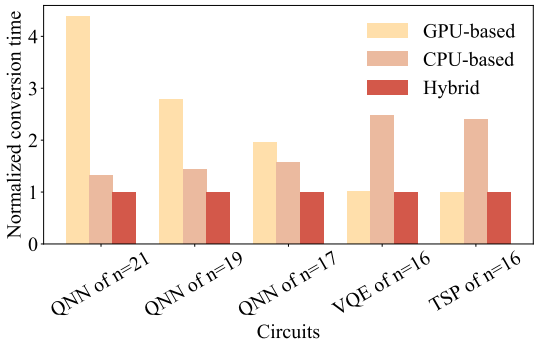
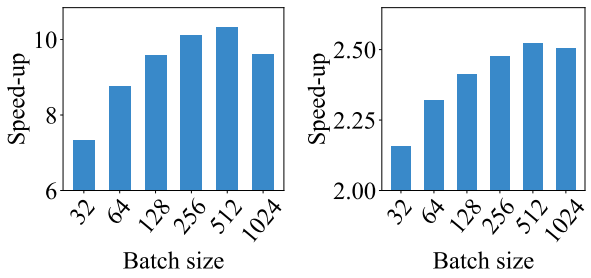


Figure 9. Runtime comparison of GPU-based, CPU-based, and hybrid DD-to-ELL conversion across five circuits. For each circuit, the conversion times are normalized by dividing them by the respective hybrid conversion time.



(a) QNN of $n = 17$. **(b)** VQE of $n = 16$.

Figure 10. Speed-up of BQSim compared with cuQuantum on two quantum circuits (QNN of $n = 17$ and VQE of $n = 16$) with increasing batch sizes.

4.5 Evaluation of Task Graph-based Execution

To evaluate our task graph-based execution strategy, we compare the BQCS runtimes between BQSim and cuQuantum. Since cuQuantum does not have gate fusion, we run cuQuantum with both our BQCS-aware gate-fusion (i.e., cuQuantum+B) and Qiskit Aer's gate-fusion (i.e., cuQuantum+Q) algorithms, to ensure a fair comparison. Table 4 compares the BQCS runtimes among BQSim, cuQuantum+B and cuQuantum+Q. On average, BQSim is 407.42× faster than cuQuantum+B and 3.62× faster than cuQuantum+Q. We attribute this result to our task graph-based model and scheduling using CUDA Graph, which reduce repetitive BQCS kernel call overhead and enhance asynchrony by overlapping BQCS kernel tasks with data movement operations. Another contributing factor is that the gate-level BQCS API in cuQuantum only supports quantum gates in dense format, incurring significant computation overhead.

4.6 Scalability of BQSim

We evaluate the scalability of BQSim with increasing batch sizes (B). Figure 10 illustrates the speed-up of BQSim compared with cuQuantum on two quantum circuits, QNN and VQE. When B increases, BQSim's speed-up over cuQuantum also increases. For instance, for VQE with $n = 16$, the speed-up is 2.16× at $B = 32$ and 2.52× at $B = 512$. The result eventually saturates at $B = 1024$. We attribute this to the use of memory buffers (Section 3.3.2) to store multiple batches at the same time. When the batch size is large, the data movement reaches memory bandwidth limit. Nevertheless, BQSim remains significantly faster than cuQuantum at $B = 1024$ (e.g., 9.59× faster on QNN of $n = 17$). This result highlights the scalability of BQSim at different batch sizes.

4.7 Power Consumption of BQSim

Figure 11 compares the average power among BQSim, cuQuantum, Qiskit Aer, and FlatDD on CPU and GPU for simulating three circuits (QNN, VQE, and TSP) using ten batches. We measure GPU power consumption using nvidia-smi [49] and CPU power consumption using powerstat [35]. Since FlatDD does not utilize GPU, we only measure its CPU power consumption.

BQSim consumes less GPU power than cuQuantum, reducing power consumption by 27.17%—52.76%. We attribute this reduction to BQSim's BQCS-aware gate fusion and task graph-based execution, which minimize redundant operations on the GPU. Additionally, BQSim outperforms Qiskit Aer and FlatDD in terms of CPU power consumption, reducing it by 43.20%—46.59% over Qiskit Aer and by 41.19%—42.54% over FlatDD. The higher power consumption of Qiskit Aer and FlatDD can be attributed to their high utilization of multi-threading and multi-processing. BQSim also outperforms cuQuantum and Qiskit Aer in overall (i.e., CPU+GPU) power consumption, reducing it by 15.29%–41.93% over cuQuantum and by 15.91%–36.48% over Qiskit Aer. Although FlatDD consumes less power than BQSim, its simulation time is considerably longer, resulting in a much higher total energy cost than BQSim.

4.8 Runtime Breakdown of BQSim

Figure 12 shows the runtime breakdown of BQSim on three circuits (Routing, Portfolio optimization, and QNN) at different numbers of batches (N). While BQCS-aware gate fusion and DD-to-ELL conversion can effectively reduce the amount of BQCS computation and enable efficient GPU execution, they also incur some runtime overhead. For example, when simulating the 21-qubit QNN with ten batches, such overhead takes about 16.20% and 41.31% of the total runtime. However, gate fusion and conversion are performed only once for any given circuit, and the result will remain unchanged throughout the simulation. As N increases (typically hundreds to thousands of batches), such overhead can

Table 4. Comparison of BQCS runtimes among BQSim, cuQuantum with BQSim's BQCS-aware gate fusion (cuQuantum+B), and with Qiskit Aer's gate fusion (cuQuantum+Q), on 16 circuits representative of different quantum applications. Some cuQuantum+B runs fail because the gate-level BQCS API in cuQuantum only supports gate matrices in dense format, causing the fused gates to exceed memory capacity.

Circuits			Runtime (ms)			Speed-up vs.	
Name	#Qubits (n)	#Gates	cuQuantum+Q	cuQuantum+B	BQSim (ours)	cuQuantum+Q	cuQuantum+B
QNN	17	934	367121	-	**22605**	16.24×	-
	19	1158	1828465	-	**105745**	17.29×	-
	21	1406	9054894	-	**481913**	18.79×	-
VQE	12	58	1192	24334	**854**	1.40×	28.49×
	14	68	4820	69319242	**2439**	1.98×	28421.17×
	16	78	19655	1266788	**9809**	2.00×	129.15×
Portfolio opt.	16	424	77945	-	**10786**	7.23×	-
	17	476	175373	-	**23790**	7.37×	-
	18	531	386924	-	**49981**	7.74×	-
Graph state	16	32	17253	3053229	**9736**	1.77×	313.60×
	18	36	70727	43888468	**39215**	1.80×	1119.18×
	20	40	286244	-	**155771**	1.84×	-
TSP	9	94	224	46769	**111**	2.02×	421.34×
	16	171	28093	238363	**15919**	1.76×	14.97×
Routing	6	39	36	2889	**22**	1.64×	131.32×
	12	81	1320	6479010	**637**	2.07×	10171.13×

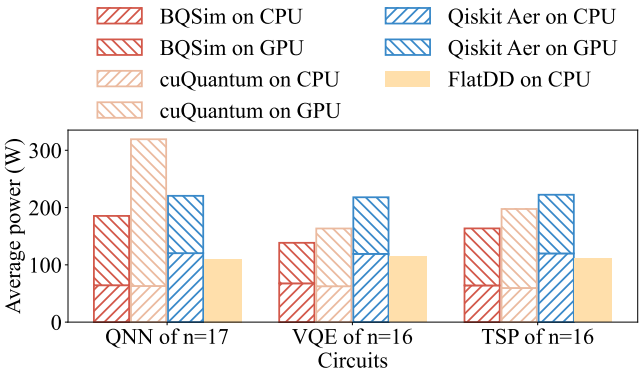


Figure 11. Comparison of average power among BQSim, cuQuantum, Qiskit Aer, and FlatDD on CPU and GPU for simulating three circuits (QNN of $n = 17$, VQE of $n = 16$, and TSP of $n = 16$) with ten batches.

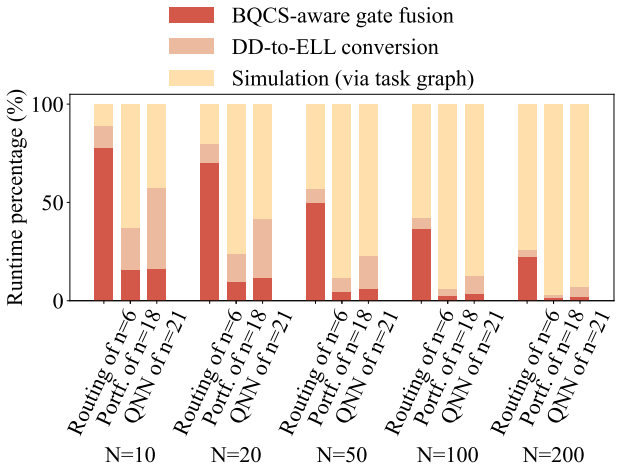


Figure 12. Runtime breakdown of BQSim on three circuits (Routing of $n = 6$, Portfolio optimization of $n = 18$, and QNN of $n = 21$) at different numbers of batches (N).

be amortized by the simulation time which had significantly benefited from fused gates and converted ELL formats. For example, when N increases to 200, the simulation time takes 93.04%, while BQCS-aware gate fusion and DD-to-ELL conversion are amortized to 1.94% and 5.02%.

4.9 Ablation Study of BQSim

We conduct an ablation study to evaluate the contribution of each stage of BQSim to its overall performance on four circuits (QNN, VQE, Portfolio optimization, and TSP) with ten batches. As shown in Figure 13, we compare the runtimes of (1) the original BQSim, (2) BQSim without BQCS-aware gate fusion, (3) BQSim without DD-to-ELL conversion (where BQCS is conducted using DDs stored on GPU, as shown in Section 3.2.1), and (4) BQSim without the task graph-based execution. For each circuit, runtimes are normalized by dividing them by the respective runtime of the original BQSim. The contribution of each stage is quantified as the normalized runtime of BQSim without that stage.

BQCS-aware gate fusion accelerates BQSim by 1.39×–6.73×, with the speed-up largely dependent on the circuit structure. We attribute this speed-up to the effectiveness of our gate-fusion algorithm, which significantly reduces the BQCS cost. DD-to-ELL conversion accelerates BQSim by 5.55×–35.08×, as the ELL format lowers thread divergence and enables efficient memory access patterns on GPU. Additionally, ELL allows direct access to gate matrix values, avoiding the DFS overhead required when accessing DD-based gate matrix values (see Algorithm 1). Task graph-based execution improves BQSim's performance by 1.46×–1.73×. We attribute this speed-up to our task graph-based model and CUDA Graph scheduling, which reduce the overhead of repetitive BQCS kernel calls and enable efficient overlap of kernel execution with data movement.

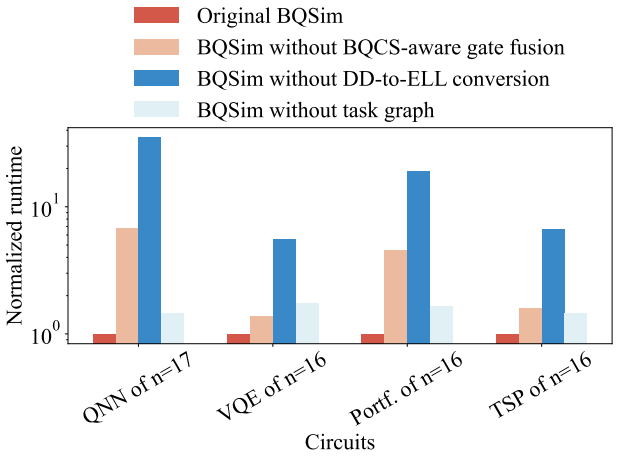


Figure 13. Ablation study of the three stages in BQSim on four circuits (QNN of $n = 17$, VQE of $n = 16$, Portfolio optimization of $n = 16$, and TSP of $n = 16$) with ten batches. For each circuit, the runtimes are normalized by dividing them by the respective runtime of BQSim.

5 Related Work

Existing QCS research has introduced various strategies to improve simulation performance by optimizing both space and time complexity. To name a few recent works, [67] leveraged lossy data compression to simulate more qubits. [32, 48, 72] used DD to explore sparsity and regularity in quantum circuits. [26] leveraged task-parallel decomposition to explore both inter- and intra-gate parallelisms. [34, 37, 68–70] leveraged GPU to further accelerate QCS; in particular, [34, 37, 70] optimized CPU-GPU data communications, and [68, 69] improved GPU memory locality to reduce latency. Additionally, [13, 16, 22, 30, 32, 47, 60, 68] optimized the quantum circuit using gate fusion and pattern matching. While these strategies can improve the performance of QCS, they are limited to the strong scaling within a single simulation input.

On an orthogonal route, several works have explored efficient QCS of multiple quantum circuits. [23, 40, 58] simulated multiple noisy quantum circuits, each representing a different noise condition. [29] simulated a variational quantum algorithm under different circuit configurations. To accelerate this type of QCS, [23, 29, 40] avoided redundant computation shared across consecutive simulation runs on different circuits, whereas [23, 58] leveraged SIMD and GPU parallelism to simulate multiple circuits simultaneously. Although these QCS works share a similar inspiration with BQCS, they target a completely different problem formulation.

6 Conclusions

In this paper, we have presented BQSim, a GPU-accelerated batch quantum circuit simulator, inspired by the state-of-the-art DD but overcomes its limitation of CPU-centric simulation. BQSim uses DD to optimize the quantum circuit for reduced BQCS computation, and converts DD to a GPU-efficient data structure. Additionally, BQSim introduces a task graph-based execution strategy using CUDA Graph to minimize repetitive kernel call overhead and efficiently overlap kernel execution with data movement. Compared with three state-of-the-art quantum circuit simulators, cuQuantum, Qiskit Aer, and FlatDD, BQSim is respectively 3.25×, 159.06×, and 311.42× faster on average.

Our future work will focus on designing efficient accelerators for BQSim by leveraging specialized DD properties, such as their ability to compactly represent gate matrices and state vectors through regularity and sparsity. For example, the structure of DD enables efficient parallel processing, as each DD node represents a unique sub-matrix or sub-vector. After each parallel computation round, identical sub-structures within DDs can be merged, thereby reducing memory overhead and computational redundancy.

Acknowledgement

The research work described in this paper was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust. This work is jointly supported by the Research Grants Council of Hong Kong SAR (No. CUHK14207523). This project is also supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

A Artifact Appendix

A.1 Abstract

This artifact appendix describes the steps to compile and run BQSim on various quantum circuits and provides a performance comparison of BQSim against cuQuantum, Qiskit Aer, and FlatDD, reproducing the runtime results from Section 4.2 and Section 4.5.

The artifact includes the source code for BQSim, cuQuantum, Qiskit Aer, and FlatDD, along with 16 quantum circuits and their corresponding inputs with varying numbers of qubits for evaluation. Running the artifact requires a CUDA-enabled GPU with at least 48 GB of memory, 20 GB of system RAM, and 20 GB of free disk space.

A.2 Artifact check-list (meta-information)

- **Compilation:** GCC 12.3.0, NVCC 12.6, CMake 3.22.1. We provide an automated compilation script and a Dockerfile.
- **Datasets:** 16 quantum circuits, and the corresponding random circuit inputs.
- **Hardware:** CUDA-enabled GPU with 48 GB of memory and 20 GB of system RAM.
- **Metrics:** Simulation runtime.
- **Output:** Metric data in the console log.
- **Experiments:** We provide automated scripts for running the experiments.
- **How much disk space required (approximately)?:** 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes.
- **How much time is needed to complete experiments (approximately)?:** For Section 4.2, all experiments take approximately four days to complete. To reduce runtime, we can evaluate each simulator separately. The fastest simulator, BQSim, takes less than 20 minutes, whereas the slowest simulator, FlatDD, takes more than two days. For Section 4.5, the experiments take approximately 10 minutes to complete.
- **Publicly available?:** Yes.
- **Code licenses?:** MIT License.
- **Archived:** 10.5281/zenodo.14775677

A.3 Description

A.3.1 How to access. The code repository for BQSim can be downloaded from https://github.com/IDEA-CUHK/BQSim.

A.3.2 Hardware dependencies. An x86 host with a CUDA-enabled GPU (at least 48 GB of memory), 20 GB of system RAM, and 20 GB of free disk space.

A.3.3 Software dependencies. Our experiments are conducted on a Ubuntu 22.04.3 LTS machine, which can run either with or without a Docker container. The software dependencies for both cases are listed below:

Without Docker:

- CUDA 12.6 with cuQuantum SDK.
- GCC 12.3.0, NVCC 12.6, CMake 3.22.1.
- libeigen3-dev.
- OpenMP.
- Python 3.10.12 with NumPy 1.26.4, Qiskit Aer GPU 0.15.0, and Qiskit 1.2.0.

With Docker:

- Docker 26.1.3.
- NVIDIA Container Toolkit.

A.3.4 Datasets. The artifact includes 16 MQT-Bench quantum circuits along with their corresponding randomly generated inputs.

A.4 Installation

A.4.1 Without Docker. Clone the repository to a local machine and jump into BQSim folder:

```
~$ git clone https://github.com/IDEA-CUHK/BQSim
~$ cd BQSim
```

Then, run the compilation script compile.sh, which will automatically generate the executables for BQSim and the baseline simulators.

```
~/BQSim$ ./compile.sh
```

A.4.2 With Docker. Build a docker image bqsim_image using Dockerfile. The compilation script docker_compile.sh is included in the Dockerfile, so if the Docker image builds successfully, the program should compile without issues.

```
~/BQSim$ sudo docker build --no-cache -t
bqsim_image .
```

Run a docker container bqsim_container using the image.

```
~/BQSim$ sudo docker run -it --rm --gpus all
--name bqsim_container bqsim_image:latest
```

A.5 Experiment workflow

We provide automated scripts to run BQSim, cuQuantum, Qiskit Aer, and FlatDD on the quantum circuits. The scripts report the runtime of each simulator for each circuit.

A.6 Evaluation and expected results

Both inside and outside the Docker container, the same general steps apply here.

A.6.1 Experiments in Section 4.2. To run simulators BQSim, cuQuantum, Qiskit Aer, and FlatDD on 16 quantum circuits, we run the overall.sh script. However, this process takes approximately four days. To reduce the runtime, we can evaluate each simulator separately by running individual scripts (e.g., bqsim.sh). The fastest simulator, BQSim, takes less than 20 minutes, whereas the slowest simulator, FlatDD, takes more than two days. The simulation runtimes should match those in Section 4.2. For more detailed instructions, please refer to the README.md file in the repository.

A.6.2 Experiments in Section 4.5. First, we export the fused gates obtained from BQCS-aware and Qiskit Aer's gate fusion by running script export_fused_gates.sh. This script will export the fused gates to log/fused_gates/,

where some pre-exported fused gates are already provided. Due to memory and time constraints, we did not export all the fused gates analyzed in Section 4.5. You may modify `export_fused_gates.sh` to export additional gates as needed.

Next, we run `cuQuantum` with both our BQCS-aware gate-fusion (i.e., `cuQuantum+B`) and `Qiskit Aer`'s gate-fusion (i.e., `cuQuantum+Q`) algorithms by executing script `cuquantum_plus_bq.sh`. The simulation runtimes should match those in Section 4.5.

References

[1] 2024. *cuQuantum*. https://github.com/NVIDIA/cuQuantum

[2] 2024. *FlatDD*. https://github.com/IDEA-CUHK/FlatDD

[3] 2024. *HIP Graph*. https://rocm.docs.amd.com/projects/HIP/en/docs-develop/how-to/hipgraph.html

[4] 2024. *Qiskit-Aer*. https://github.com/Qiskit/qiskit-aer

[5] Hervé Abdi. 2010. Coefficient of variation. *Encyclopedia of research design* 1, 5 (2010), 169–171.

[6] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[7] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, et al. 2023. cuQuantum SDK: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 1050–1061.

[8] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[9] Lukas Burgholzer and Robert Wille. 2020. The power of simulation for equivalence checking in quantum computing. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[10] Ke Chen, Linbin Chen, Pedro Reviriego, and Fabrizio Lombardi. 2019. Efficient Implementations of Reduced Precision Redundancy (RPR) Multiply and Accumulate (MAC). *IEEE Trans. Comput.* 68, 5 (2019), 784–790. https://doi.org/10.1109/TC.2018.2885044

[11] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *European Conference on Parallel Processing*. Springer, 468–479.

[12] cuStateVec Functions NVIDIA cuQuantum 24.08.0 documentation. [n. d.]. https://docs.nvidia.com/cuda/cuquantum/latest/custatevec/api/functions.html

[13] Aneeqa Fatima and Igor L Markov. 2021. Faster schrödinger-style simulation of quantum circuits. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 194–207.

[14] Tim M Forcer, Anthony JG Hey, DA Ross, and Peter GR Smith. 2002. Superposition, entanglement and quantum computation. *Quantum Information and Computation* 2, 2 (2002), 97–116.

[15] Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. 2002. Quantum cryptography. *Reviews of modern physics* 74, 1 (2002), 145.

[16] Gian Giacomo Guerreschi. 2022. Fast simulation of quantum algorithms using circuit optimization. *Quantum* 6 (2022), 706.

[17] Thomas Häner, Damian S Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 866–874.

[18] Adam Hayes. 2024. Coefficient of Variation: Meaning and How to Use It. https://www.investopedia.com/terms/c/coefficientofvariation.asp

[19] Dylan Herman, Cody Googin, Xiaoyuan Liu, Yue Sun, Alexey Galda, Ilya Safro, Marco Pistoia, and Yuri Alexeev. 2023. Quantum computing for finance. *Nature Reviews Physics* 5, 8 (2023), 450–465.

[20] Stefan Hillmich, Alwin Zulehner, and Robert Wille. 2020. Concurrency in DD-based quantum circuit simulation. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 115–120.

[21] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. 2022. A tensor network based decision diagram for representation of quantum circuits. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 6 (2022), 1–30.

[22] Hiroshi Horii and Jun Doi. 2021. Optimization of Quantum Computing Simulation with Gate Fusion. *IPSJ SIG Technical Report* (2021).

[23] Hiroshi Horii, Christopher Wood, et al. 2023. Efficient techniques to gpu accelerations of multi-shot quantum computing simulations. *arXiv preprint arXiv:2308.03399* (2023).

[24] Ryszard Horodecki, Paweł Horodecki, Michał Horodecki, and Karol Horodecki. 2009. Quantum entanglement. *Reviews of modern physics* 81, 2 (2009), 865–942.

[25] Zhirui Hu, Peiyan Dong, Zhepeng Wang, Youzuo Lin, Yanzhi Wang, and Weiwen Jiang. 2022. Quantum neural network compression. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[26] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 746–756.

[27] T.-W. Huang, C.-X. Lin, Guannan Guo, and Martin D. F. Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. *IEEE IPDPS*, 974–983.

[28] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2021), 1303–1320.

[29] Yipeng Huang, Steven Holtzen, Todd Millstein, Guy Van den Broeck, and Margaret Martonosi. 2021. Logical abstractions for noisy variational quantum algorithm simulation. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*. 456–472.

[30] Raban Iten, Romain Moyard, Tony Metger, David Sutter, and Stefan Woerner. 2022. Exact and practical pattern matching for quantum circuit optimization. *ACM Transactions on Quantum Computing* 3, 1 (2022), 1–41.

[31] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. https://doi.org/10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]

[32] Shui Jiang, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei Huang. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *Proceedings of the 53rd International Conference on Parallel Processing*. 388–399.

[33] Weiwen Jiang, Jinjun Xiong, and Yiyu Shi. 2021. A co-design framework of neural networks and quantum circuits towards quantum advantage. *Nature communications* 12, 1 (2021), 579.

[34] Chenyang Jiao, Weihua Zhang, and Li Shen. 2023. Communication Optimizations for State-vector Quantum Simulator on CPU+ GPU Clusters. In *Proceedings of the 52nd International Conference on Parallel Processing*. 203–212.

[35] Colin King. 2015. powerstat - a tool to measure power consumption. https://manpages.ubuntu.com/manpages/xenial/man8/powerstat.8.html

[36] Los Alamos National Lab. 2014. ELL format. https://www.lanl.gov/Caesar/node223.html

[37] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Heim, Martin Roetteler, and Sriram Krishnamoorthy. 2021. Sv-sim: scalable pgas-based state vector simulation of quantum circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[38] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. Density matrix quantum circuit simulation via the BSP machine on modern GPU clusters. In *Sc20: international conference for high performance computing, networking, storage and analysis*. IEEE, 1–15.

[39] Gushu Li, Yufei Ding, and Yuan Xie. 2019. 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*. 1001–1014.

[40] Gushu Li, Yufei Ding, and Yuan Xie. 2020. Eliminating redundant computation in noisy quantum computing simulation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[41] Zhiding Liang, Zhepeng Wang, Junhuan Yang, Lei Yang, Yiyu Shi, and Weiwen Jiang. 2021. Can noise on qubits be learned in quantum neural network? a case study on quantumflow. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–7.

[42] Dian-Lun Lin and Tsung-Wei Huang. 2020. A novel inference algorithm for large sparse neural network using task graph parallelism. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[43] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU computation using task graph parallelism. In *Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1–3, 2021, Proceedings 27*. Springer, 435–450.

[44] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 3041–3052. https://doi.org/10.1109/TPDS.2021.3138856

[45] Marco Maggioni and Tanya Berger-Wolf. 2013. AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs. In *2013 42nd International Conference on Parallel Processing*. 11–20. https://doi.org/10.1109/ICPP.2013.10

[46] Marco Maggioni and Tanya Berger-Wolf. 2014. CoAdELL: Adaptivity and Compression for Improving Sparse Matrix-Vector Multiplication on GPUs. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 933–940. https://doi.org/10.1109/IPDPSW.2014.106

[47] Stefano Markidis. 2023. Enabling Quantum Computer Simulations on AMD GPUs: a HIP Backend for Google's qsim. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1478–1486.

[48] Philipp Niemann, Robert Wille, David Michael Miller, Mitchell A Thornton, and Rolf Drechsler. 2015. QMDDs: Efficient quantum function representation and manipulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 1 (2015), 86–99.

[49] NVIDIA. [n. d.]. nvidia-smi. https://docs.nvidia.com/deploy/nvidia-smi/index.html

[50] Tom Peham, Lukas Burgholzer, and Robert Wille. 2022. Equivalence checking of quantum circuits with the ZX-calculus. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 3 (2022), 662–675.

[51] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* (2023). MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[52] Gokul Subramanian Ravi, Pranav Gokhale, Yi Ding, William Kirby, Kaitlin Smith, Jonathan M Baker, Peter J Love, Henry Hoffmann, Kenneth R Brown, and Frederic T Chong. 2022. CAFQA: A classical simulation bootstrap for variational quantum algorithms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 15–29.

[53] John R Rice and Ronald F Boisvert. 2012. *Solving elliptic problems using ELLPACK*. Vol. 2. Springer Science & Business Media.

[54] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 73–82.

[55] SixSigma. 2024. Coefficient of Variation: Mastering Relative Variability in Statistics. https://www.6sigma.us/six-sigma-in-focus/coefficient-of-variation/

[56] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. 2016. qHiPSTER: The quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195* (2016).

[57] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, Yuria Hiraga, Masahiro Nakadai, Jiabao Chen, Ken M Nakanishi, Kosuke Mitarai, Ryosuke Imai, Shiro Tamiya, et al. 2021. Qulacs: a fast and versatile quantum circuit simulator for research purpose. *Quantum* 5 (2021), 559.

[58] Keichi Takahashi, Toshio Mori, and Hiroyuki Takizawa. 2023. Prototype of a Batched Quantum Circuit Simulator for the Vector Engine. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1499–1505.

[59] Siwei Tan, Debin Xiang, Liqiang Lu, Junlin Lu, Qiuping Jiang, Mingshuai Chen, and Jianwei Yin. 2024. MorphQPV: Exploiting Isomorphism in Quantum Programs to Facilitate Confident Verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 671–688.

[60] Quantum AI team and collaborators. 2020. *qsim*. https://doi.org/10.5281/zenodo.4023103

[61] CUDA Toolkit v12.6.2. [n. d.]. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html

[62] Jiyuan Wang, Fucheng Ma, and Yu Jiang. 2021. Poster: Fuzz testing of quantum program. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 466–469.

[63] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. Qdiff: Differential testing of quantum software stacks. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 692–704.

[64] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2021. Quito: a coverage-guided test generator for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1237–1241.

[65] Wikipedia. [n. d.]. Coefficient of Variation. https://en.wikipedia.org/wiki/Coefficient_of_variation

[66] Robert Wille, Lukas Burgholzer, Stefan Hillmich, Thomas Grurl, Alexander Ploier, and Tom Peham. 2022. The basis of design tools for quantum computing: arrays, decision diagrams, tensor networks, and ZX-calculus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1367–1370.

[67] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2019. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–24.

[68] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: hybrid partitioner based quantum circuit simulation system on GPU. In *Proceedings of the ACM International Conference on Supercomputing*. 443–454.

[69] Chen Zhang, Haojie Wang, Zixuan Ma, Lei Xie, Zeyu Song, and Jidong Zhai. 2022. UniQ: a unified programming model for efficient quantum circuit simulation. In *SC22: International Conference for High*

Performance Computing, Networking, Storage and Analysis. IEEE, 1–16.

[70] Yilun Zhao, Yanan Guo, Yuan Yao, Amanda Dumi, Devin M Mulvey, Shiv Upadhyay, Youtao Zhang, Kenneth D Jordan, Jun Yang, and Xulong Tang. 2022. Q-gpu: A recipe of optimizations for quantum circuit simulation using gpus. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 726–740.

[71] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to efficiently handle complex values? Implementing decision diagrams for quantum computing. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7.

[72] Alwin Zulehner and Robert Wille. 2018. Advanced simulation of quantum computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 848–859.