HyQuas: Hybrid Partitioner Based Quantum Circuit Simulation System on GPU

Chen Zhang
chen-zha17@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Zeyu Song
songzy18@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Haojie Wang
wang-hj18@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Kaiyuan Rong
rky18@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Jidong Zhai
zhaijidong@tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

Quantum computing has shown its strong potential in solving certain important problems. Due to the intrinsic limitations of current real quantum computers, quantum circuit simulation still plays an important role in both research and development of quantum computing. GPU-based quantum circuit simulation has been explored due to GPU's high computation capability. Despite previous efforts, existing quantum circuit simulation systems usually rely on a single method to improve poor data locality caused by complex quantum entanglement. However, we observe that existing simulation methods show significantly different performance for different circuit patterns. The optimal performance cannot be obtained only with any single method.

To address these challenges, we propose HyQuas, a **Hy**brid partitioner based **Qu**a**n**tum circuit **S**imulation system on GPU, which can automatically select the suitable simulation method for different parts of a given quantum circuit according to its pattern. Moreover, to make better support for HyQuas, we also propose two highly optimized methods, *OShareMem* and *TransMM*, as optional choices of HyQuas. We further propose a GPU-centric communication pipelining approach for effective distributed simulation. Experimental results show that HyQuas can achieve up to 10.71× speedup on a single GPU and 227× speedup on a GPU cluster over state-of-the-art quantum circuit simulation systems.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**; **Simulation tools**.

KEYWORDS

GPU Computing, Quantum Computing, Simulation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '21, June 15–17, 2021, Worldwide Online
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460357

ACM Reference Format:
Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: Hybrid Partitioner Based Quantum Circuit Simulation System on GPU. In *ICS '21: International Conference on Supercomputing, June 15–17, 2021, Worldwide Online.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3447818.3460357

1 INTRODUCTION

The theory of quantum computing [37] has shown its strong potential in solving certain important problems that beyond the computing capability of classical computers, such as cryptography [13], financial modeling [38], machine learning [14, 35, 40], and molecular quantum mechanics [9, 12]. Many quantum computers have been released in recent years [45], e.g., Intel's 49-qubit quantum computer, Google's 53-qubit quantum computer Sycamore [8] and 72-qubit quantum computer Bristlecone [4], IBM's 53-qubit quantum computer [3], and USTC's 76-qubit quantum computer Jiuzhang [48]. However, quantum computers are still precious resources that cannot be used widely and conveniently. Current quantum computers also suffer from very large error rates and cannot be used to verify complex quantum algorithms. Additionally, a quantum state is destroyed after its measurement, so not all intermediate data can be collected on a real quantum system. Therefore, quantum circuit simulation is necessary for advancing the theory of quantum computing.

The most common way of building a quantum circuit simulation system is representing a quantum state as a vector, and treating quantum gates operating on a quantum circuit as small matrix-vector multiplications. For example, applying a single-qubit quantum gate to an n-qubit state can be regarded as 2^{n-1} $1\times2\times2$ matrix multiplications. As these small matrix multiplications have no data dependency and can be parallelized, GPUs can provide much higher throughput than CPUs, which has been explored by many previous works [7, 10, 11, 21–26, 30, 31, 34, 36, 42, 47].

Building a high-performance quantum circuit simulator is challenging due to the complexity of quantum circuits, especially on heterogeneous GPU platforms. A large number of small matrix multiplications used for simulating various quantum gates lead to poor data locality, which further causes high cache miss rate and low computing resource utilization. To improve data locality, previous studies have explored the idea of grouping a set of quantum gates applied to the circuit, so that multiple gates can be processed

together. Among these works, two approaches are mostly used: *ShareMem* [24] and *BatchMV* [42]. The former one caches states in high-speed shared memory to reduce memory access latency, while the latter one merges a group of gates to reduce total gate number.

However, our observation shows that above two approaches represent different performances on different circuit patterns. The *ShareMem* method has better performance on the sparse parts of circuits, while the *BatchMV* method works much better on the dense parts. The performance differences can be up to 4.93× among these two methods on seven typical quantum circuits we have evaluated. Therefore, building an automatically adaptive quantum circuit simulation system through combining the advantages of both *ShareMem* and *BatchMV* is critical for high performance simulation, especially with the rapid growth in complexity of quantum circuits. However, it is not trivial to combine both approaches as the performance of these two approaches are highly dependent on both upper quantum circuits and underlying hardware architectures.

In this work, we propose a novel *hybrid* approach, called HyQuas, which is a high-performance GPU-based quantum circuit simulation system to address these challenges. The key of the *Hybrid* approach is how to automatically and near-optimally partition a given quantum circuit into different groups and select a suitable method for each group. To achieve that, we design a circuit-aware partition strategy and a high-accuracy performance model to guide partitioning. Moreover, to further improve the performance, we leverage several GPU hardware features and apply a set of common GPU optimization techniques in HyQuas: *OShareMem* method, which deletes redundant computation, reduces data indexing overhead, and uses a new layout to access the shared memory faster; *TransMM* method, which converts a set of special matrix-vector multiplications in quantum circuit simulation into general matrix multiplications (GEMM) to easily take advantage of highly optimized GEMM libraries and hardware-level GEMM compute units like Tensor Cores. We further present a distributed implementation, which can utilize high-throughput NVLink connections to enable GPU directed communication while still preserving low communication traffic and further accelerate simulation via pipelining. To the best of our knowledge, HyQuas is the most efficient GPU-based quantum circuit simulation system so far.

Our contributions are summarized as follows:

- We propose the *OShareMem* method with new task-thread mapping, faster indexing, and bank conflict mitigation, which achieves up to 2.67× speedup over the existing *ShareMem* methods.
- We propose a new *TransMM* method, which enables the usage of highly optimized library cuBLAS and powerful Tensor Cores, and leads to up to 8.43× speedup over the existing *BatchMV* method.
- We propose a GPU-centric communication pipelining approach for distributed quantum circuit simulation, which provides up to 227× speedup (129× on average) over the state-of-the-art GPU-based distributed quantum circuit simulator [22] and achieves better scalability.
- We propose an adaptive quantum circuit simulation system with a hybrid computing approach, called HyQuas[1], which can automatically analyze a given circuit, and select suitable computing approaches for each sub-circuit. HyQuas can achieve up to 10.71× speedup (2.73× on average) compared with state-of-the-art quantum circuit simulation systems.

[1]HyQuas is now available at https://github.com/thu-pacman/HyQuas.

The paper is organized as follows: we give the basic concepts of quantum computation in Section 2. Section 3 gives our motivation and Section 4 introduces the overview of HyQuas. In Section 5, we provide the detail of HyQuas's design. In Section 6, we present the optimization for distributed simulation. We evaluate HyQuas in Section 7, discuss related works in Section 8, and conclude our work in Section 9.

2 BASIC CONCEPT IN QUANTUM COMPUTATION

Single-qubit system. In a classical computer, a bit is a deterministic state being "0" or "1". However, in a quantum computer, a quantum bit (qubit) is a non-deterministic state represented by two complex numbers $\{\alpha_0, \alpha_1\}$ with:

$$|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} \tag{1}$$

The probabilities of being in 0-state and 1-state are $|\alpha_0|^2$ and $|\alpha_1|^2$, respectively, which satisfy $|\alpha_0|^2 + |\alpha_1|^2 = 1$. The complex numbers α_i are also called "amplitudes" in quantum computing.

An operation to the quantum bits is called a quantum "gate". A gate on one qubit can be represented by a unitary matrix U of dimension 2×2 . Applying a single-qubit gate U on a single-qubit state $|\Psi\rangle$ results in an update of the probabilities α_0 and α_1:

$$\begin{pmatrix}\alpha_0'\\ \alpha_1'\end{pmatrix} = U|\Psi\rangle = \begin{bmatrix}U_{0,0} & U_{0,1}\\ U_{1,0} & U_{1,1}\end{bmatrix}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = \begin{pmatrix}U_{0,0}\alpha_0 + U_{0,1}\alpha_1\\ U_{1,0}\alpha_0 + U_{1,1}\alpha_1\end{pmatrix} \tag{2}$$

For example, the NOT gate $X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$ flips the qubit and results in a new state $|\Psi\rangle' = \alpha_1 |0\rangle + \alpha_0 |1\rangle$, whose probability of being in 0-state and 1-state is swapped.

Two-qubit system. A two-qubit quantum system can be represented by four complex numbers showing the amplitudes of the four basis:

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{pmatrix}\alpha_{00}\\ \alpha_{01}\\ \alpha_{10}\\ \alpha_{11}\end{pmatrix} \tag{3}$$

The α_i also represents the probability distribution of the four states and satisfies $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

A gate on two qubits is a 4×4 unitary matrix. One important type of two-qubit gates is the "controlled gate", denoted as:

$$CU = \begin{bmatrix}1 & & & \\ & 1 & & \\ & & U_{0,0} & U_{0,1}\\ & & U_{1,0} & U_{1,0}\end{bmatrix} \tag{4}$$

The two qubits of a "controlled" gates are called "control qubit" and "target qubit", respectively. The result of applying a "controlled gate" is applying a single-bit gate to the target qubit when the control qubit equals one.

A single qubit gate can also be applied to a two-qubit system. The 4×4 matrix representation of the gate is derived from the tensor product of the 2×2 matrix representation of the single qubit gate and a 2×2 identity matrix. It is equivalent to multiplying U to the groups with only one index differs. The gate on the first qubit is applied to $\{\alpha_{00}, \alpha_{10}\}$ and $\{\alpha_{01}, \alpha_{11}\}$ respectively. The gate on the second qubit is applied to $\{\alpha_{00}, \alpha_{01}\}$ and $\{\alpha_{01}, \alpha_{11}\}$ respectively. For example, applying the NOT gate on the first qubit results in the swap between $\{\alpha_{00}, \alpha_{10}\}$ and between $\{\alpha_{01}, \alpha_{11}\}$:

$$\begin{pmatrix} \alpha'_{00} \\ \alpha'_{01} \\ \alpha'_{10} \\ \alpha'_{11} \end{pmatrix} = \begin{bmatrix} & & 1 & \\ & & & 1 \\ 1 & & & \\ & 1 & & \end{bmatrix} \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix} = \begin{pmatrix} \alpha_{10} \\ \alpha_{11} \\ \alpha_{00} \\ \alpha_{01} \end{pmatrix} \tag{5}$$

***N*-qubit system.** An n-qubit system is represented by 2^n complex numbers $\{\alpha_i\}$:

$$|\Psi\rangle = \alpha_{0...00}\,|0...00\rangle + \alpha_{0...01}\,|0...01\rangle + ... + \alpha_{1...11}\,|1..11\rangle \tag{6}$$

A gate on the n-qubit system is a $2^n \times 2^n$ unitary matrix. However, the $2^n \times 2^n$ matrix representations of single qubit gates and controlled gates are very sparse and there is no need to construct the matrix. Instead, these gates can be implemented by several small matrix multiplications.

Applying a single-qubit gate to an n-qubit system can be achieved by 2^{n-1} matrix multiplication tasks. Each task updates two positions with only the tth index differs:

$$\begin{pmatrix} a'_{b_0,...,b_{t-1},\,\mathbf{0},\,b_{t+1},...,b_{n-1}} \\ a'_{b_0,...,b_{t-1},\,\mathbf{1},\,b_{t+1},...,b_{n-1}} \end{pmatrix} = \begin{bmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{bmatrix} \begin{pmatrix} a_{b_0,...,b_{t-1},\,\mathbf{0},\,b_{t+1},...,b_{n-1}} \\ a_{b_0,...,b_{t-1},\,\mathbf{1},\,b_{t+1},...,b_{n-1}} \end{pmatrix} \tag{7}$$

And applying a two-qubit controlled gate on the n-qubit system is equivalent to 2^{n-2} matrix multiplication tasks. Each task updates two positions with the index of the control qubit c equals 1 and only the target qubit t differs:

$$\begin{pmatrix} a'_{b_0,...,b_{c-1},\,\mathbf{1},\,b_{c+1},...,b_{t-1},\,\mathbf{0},\,b_{t+1},...,b_{n-1}} \\ a'_{b_0,...,b_{c-1},\,\mathbf{1},\,b_{c+1},...,b_{t-1},\,\mathbf{1},\,b_{t+1},...,b_{n-1}} \end{pmatrix} = \begin{bmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{bmatrix} \begin{pmatrix} a_{b_0,...,b_{c-1},\,\mathbf{1},\,b_{c+1},...,b_{t-1},\,\mathbf{0},\,b_{t+1},...,b_{n-1}} \\ a_{b_0,...,b_{c-1},\,\mathbf{1},\,b_{c+1},...,b_{t-1},\,\mathbf{1},\,b_{t+1},...,b_{n-1}} \end{pmatrix} \tag{8}$$

3 OBSERVATION AND MOTIVATION

The goal of quantum circuit simulation is to simulate the process of applying the gates in a quantum circuit to a quantum state. In this section, we introduce the existing quantum circuit simulation technologies and analyze problems that motivate the design of HyQuas.

3.1 Different Simulation Methods

As shown in Equation (7) and Equation (8), the apply of each gate consists of 2^{n-1} (single qubit gate) or 2^{n-2} (controlled gate) tasks, and the computation of each task is a small matrix multiplication. Although these tasks are independent and easily parallelized, they have poor data locality and can lead to high cache miss rate and low computing resource utilization. Two main approaches, *ShareMem* and *BatchMV*, are used to address this problem.

***ShareMem* Method** Equation (7) and Equation (8) show that, after applying a gate on qubit t, the new value only depends on previous values at the same position and that at the position with only the tth index differs. More generally, after applying several gates that only operate on a certain k qubits, the result value only depends on 2^k initial values, whose index only differs on these k positions. Therefore, the circuit can be partitioned into several gate groups. The gates in each group are only applied on a certain k qubits. We call these k qubits *active qubits*. To apply a gate group, the 2^n values are split into several independent fragments of size 2^k, which can be stored in GPU shared memory. Each fragment is mapped to one GPU thread block. A thread block loads the fragment from global memory to shared memory, apply the gates on it, and store the fragment back into global memory.

Figure 1 shows an example of processing a fragment with $k = 3$. The active qubits are {0, 2, 4}. $2^3 = 8$ values with only indexes {0, 2, 4} differ are copied to the shared memory and reindexed as 0-7. Gates on qubits {0, 2, 4} can then be processed locally in the shared memory, similar to applying gates on a 3-qubit system. And finally, the 8 values are copied back to their initial position in the global memory.

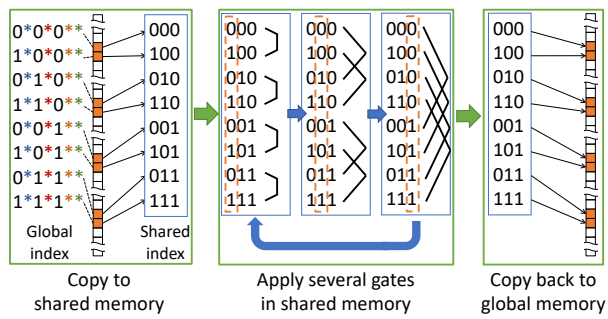


Figure 1: Each GPU block fetches some values from the global memory and applies several gates to them. $*$ with the same color represents the same value.

***BatchMV* Method** If both target qubits and control qubits of a gate group come from certain k qubits, the gates can be further merged into a k-qubit gate. The derived k-qubit gate is represented by a $2^k \times 2^k$ matrix to be applied to these active qubits. The rule of applying the k-qubit gate is similar to applying a single-qubit gate. It can be regarded as 2^{n-k} matrix-vector multiplications tasks. Each task multiplies the $2^k \times 2^k$ gate matrix to a vector of 2^k values whose index only differ on these k positions.

3.2 Observation and Motivation

Due to the different implementations of *ShareMem* and *BatchMV*, they have different time complexity on different circuit patterns, which leads to different performance behaviors. Figure 2 shows the time to apply a group with various numbers of gates on a given circuit.

After loading a subset of values into the shared memory, the *ShareMem* method needs to apply the gates one-by-one, so the time grows linearly to the number of gates, while for the *BatchMV* method, all gates within a group are merged into a single gate, so the time consumption is irrelevant to the gate number in the group.

Therefore, these two methods are suitable for different types of circuits: *ShareMem* works better on sparse circuits that each group contains a small number of gates, while *BatchMV* works better on dense circuits with more gates. To make use of the strengths of the two methods, we propose an adaptive hybrid method, namely HyQuas, which can automatically partition the circuit into different gate groups that are specialized for the two methods.

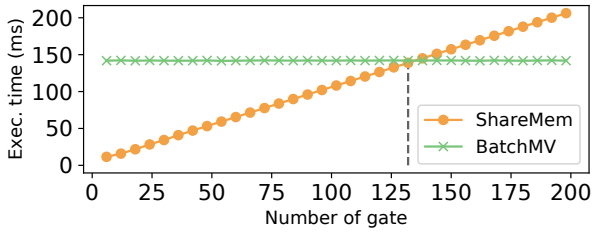


Figure 2: Time for applying a group of gates on 6 active qubits inside a 28-qubit system. Results are measured on an A100-PCIE-32G card.

However, the existing *BatchMV* method cannot take the advantage of HyQuas's design for real world applications. As shown in Figure 2, only when the gate number is larger than 132, the 6-active-qubit *BatchMV* method can outperform the *ShareMem* method, but none of the benchmark circuits in our experiments has a 6-active-qubit gate group of more than 132 gates. The reason for such inefficiency is that matrix-vector multiplications are bounded by the loading of the matrix. We propose *TransMM*, which is 7.65× faster than the *BatchMV* method on such qubit size and can move the intersection in Figure 2 forward to about 15 gates, to make the hybrid method practical. *TransMM* transposes the quantum state to convert the gate applying to standard GEMM operations that can be accelerated by the highly-optimized cuBLAS library, plus the powerful Tensor Cores. We also carefully optimize the *ShareMem* method, called *OShareMem*, by thinking of the gate applying inside the shared memory (step 2 in Figure 1), rather than only considering the reduction and acceleration of global memory access (step 1 & 3).

Furthermore, after HyQuas reduces the computation time, the long communication time across multi-GPU becomes a more emerging problem. To make the simulation more scalable, HyQuas utilizes the high throughput GPU-direct connection while still keeping low communication traffic. Even with our optimized communication strategy, the communication time still takes a large part of the overall running time, so a pipeline implementation is provided by HyQuas to interleave computation with communication.

4 OVERVIEW OF HYQUAS

Figure 3 shows the workflow of HyQuas. HyQuas first analyzes the given circuit, partitions the circuit into stages that GPUs can execute independently and detect the subcircuits to be pipelined. The hybrid partitioner further splits each stage into gate groups partitioned for the *ShareMem* method or the *TransMM* method respectively. A performance-model based time predictor is used to predict the time of different ways of partitioning to guide the selection between the two methods.

The generated schedule is then fed into the executor. For each stage, the executor packs the data with different destination GPU separately, sends the data to the corresponding GPU and executes the pipelined gate groups simultaneously, then executes the remaining gate groups. Each gate group is processed with the highly-optimized *OShareMem* method or *TransMM* method, with the decision from the partitioner.

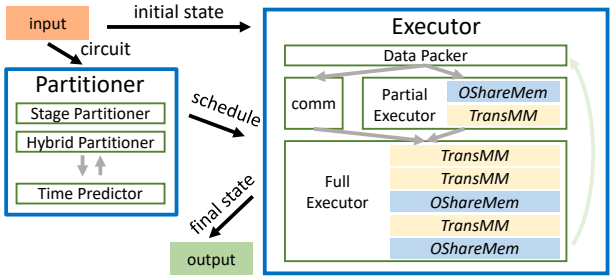


Figure 3: Overview of HyQuas

5 METHODOLOGIES

5.1 Challenges of Hybrid Approach

As mentioned in Section 3.2, the *ShareMem* method is suitable for smaller sparse groups while the *BatchMV* method is more suitable for larger dense groups. To take advantage of both methods, a hybrid method is needed to assign different groups with different methods. Unfortunately, as shown in Figure 2, the current *BatchMV* method is not practical for the *Hybrid* approach due to its low performance which causes it not to outperform *ShareMem* until the gate number in the gate group is larger than 132. The inefficiency of *BatchMV* mainly comes from its large amount of memory access. As for the *ShareMem* method, although it can beat the current *BatchMV* method, careful optimizations are still needed to get better performance, like redundant computation reduction, data indexing optimization, and bank conflict mitigation. Hence, *TransMM* and *OShareMem* are proposed as two key components in HyQuas to provide promising performance for the backend decisions.

Although we have high performance backends, building a hybrid simulator is still challenging. The first problem is how to partition a circuit into different groups. Figure 4 (c) gives an example of the hybrid partition. The dense part in the left bottom corner of the circuit (shown as the yellow part) should use the *TransMM* method while other groups (shown as the blue parts) only contain a small number of gates and should use the *OShareMem* method.

However, the hybrid method is more than partitioning the circuit into groups and find the suitable execution method for each group, because of the different constraints and preferences of these two methods. The *OShareMem* method is more flexible, which does not require the target qubit of the diagonal gates and the control qubit of the controlled gate to be active. While for the *TransMM* method, to make computation more regular so as to be formulated as a GEMM, it requires both control qubits and target qubits to be

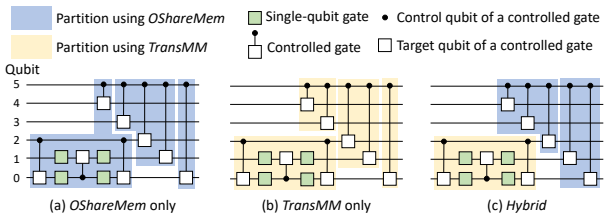


Figure 4: Different ways to partition the circuit into 3-active-qubit groups

active, and the optimization for diagonal gates mentioned above should not be applied. Therefore, in Figure 4 (a) and Figure 4 (b), though both methods partition the circuit into groups with at most 3 active qubits, the two methods result in different partitions, with the *OShareMem* method has fewer groups. Moreover, in the *OShareMem* method, to coalesce the access to global memory, the 3 qubits that act as the least significant bits need to be selected as the active qubit (not shown in Figure 4 (a)).

Another problem is the number of active qubits. More active qubits lead to more gates in each group and fewer groups. For each group, the *OShareMem* method needs to read and write the whole 2^n positions once, so the reduction of group number will result in the reduction of memory traffic and accelerate the simulation. In HyQuas, the maximum active qubit size for the *OShareMem* method is set to 10. In the *TransMM* method, each group is processed with a transpose and a GEMM, so fewer groups will also lead to fewer operations. However, the number of multiplications in each GEMM is 2^{n+k}, where k refers to the number of active qubits. The increase of active qubits will result in exponential growth of multiplication operations, so the active qubit size of the *TransMM* method cannot be too large. The actual active qubit size of the *TransMM* method should be decided with an analysis of the circuit, to balance between the decrease of group number and increase of GEMM size during the growth of active qubit size.

Due to the different partition rules and the different active qubit sizes, the hybrid partitioner should be able to guide the partition rather than only select the backend for each partitioned circuit.

The rest of this section will give the design details of HyQuas to address these challenges: Section 5.2 presents the *Hybrid* approach, including a hybrid partitioner and a time predictor, while Section 5.3 and Section 5.4 show our *TransMM* method and *OShareMem* method respectively.

5.2 *Hybrid* Approach

The key of the *Hybrid* approach is how to automatically and near-optimally partition a given circuit into different groups and decide the backend used for each group. Thus, there are two main components for the *Hybrid* approach: the hybrid partitioner and the time predictor. The hybrid partitioner decides how to partition a circuit into gate groups and the time predictor gives a performance prediction for a given gate group to help the hybrid partitioner to find the partition with the best performance.

Hybrid Partitioner Given a circuit, the hybrid partitioner of HyQuas uses a greedy algorithm to partition the circuit into several groups that map to different methods, as shown in Algorithm 1.

The partitioner continuously splits out groups of gates from the circuit until the circuit becomes empty. In each iteration, the gate group that can be most efficiently processed will be chosen. The candidate group comes from the following sources:

(1) `SHM-GROUPING(circuit)`: Group a sequence of gates with no more than 10 active qubits that can be applied with the *OShareMem* method. The 3 qubits at the lowest positions are required to be counted as the active qubits. Controlled gates with a non-active control qubit and diagonal gates with a non-active target qubit can be included in the group.
(2) `TMM-GROUPING(circuit, k)`: Group a sequence of gates with no more than k active qubits that can be merged into a k-qubit gate and executed by the *TransMM* method. All control qubits and target qubits need to be active qubits. In our implementation, only the medium-sized k, i.e., 4-7, will be tried, since smaller k will result in smaller and more groups and inefficient GEMM, while larger k will result in exponential growth of the matrix size.

The efficiency of processing the group is defined by the number of gates that can be processed within each second:

$$eff = \text{N-GATE}(group)/\text{TIME}(group) \tag{9}$$

N-GATE refers to the number of gates in the group and *TIME* is the predicted execution time of the group.

Algorithm 1 The Hybrid Partitioner with a Greedy Search

```
1:  input: The Input Circuit circuit
2:  output: The Gate Groups groups
3:  groups ← []
4:  while circuit ≠ ϕ do
5:      bestGroup ← SHM-GROUPING(circuit)
6:      bestEff ← N-GATE(bestGroup)/TIME_SHM(bestGroup)
7:      for k ∈ [4, 5, 6, 7] do
8:          group ← TMM-GROUPING(circuit, k)
9:          eff ← N-GATE(group)/TIME_TMM(group)
10:         if eff > bestEff then
11:             bestGroup ← group
12:             bestEff ← eff
13:     groups.append(bestGroup)
14:     circuit.remove(bestGroup)
15: return groups
```

Time Predictor To support the selection of different methods, HyQuas provides an architecture-aware time predictor to predict the time for a given group applied by different methods.

For the *OShareMem* method, the time usage can be predicted by the summation of the time for applying each gate on the n-qubit system plus a constant bias representing the overhead like copying the gate group to the constant memory, transferring data between the global memory and the shared memory, and some scheduling overhead:

$$\text{TIME}_{\text{SHM}}(group) = time_bias[n] + \sum_{g \in group} time_gate[n][g.type] \tag{10}$$

To apply the merged k-qubit gate to the n-qubit system, the *TransMM* method needs two periods - transposing the active qubits to the least significant bits and performing a $2^{n-k} \times 2^k \times 2^k$ GEMM, i.e., multiplying a $2^{n-k} \times 2^k$ state matrix with a $2^k \times 2^k$ gate matrix. Therefore, the time can be predicted by adding the time of the two periods:

$$\text{TIME}_{\text{TMM}}(group) = time_transpose[n] + time_gemm[n][k] \quad (11)$$

The $time_*$ in Equation (10) and Equation (11) are architecture-sensitive parameters that only need to be collected once on each platform. To enable the hybrid method of HyQuas, we need to first run a light-weight preprocessor to collect these values and save them to the database of the time predictor.

5.3 *TransMM* Method

As we mentioned in Section 3.2, to make the *Hybrid* approach practical, the more efficient *TransMM* method has to be proposed. In the *BatchMV* method, applying a k-qubit gate on the k active qubits will be regarded as 2^{n-k} matrix-vector multiplications between the $2^k \times 2^k$ gate matrix and a 2^k fragment of the quantum state. Each matrix-vector multiplication needs to load the whole gate matrix. Such pattern brings very large traffic and bounds the usage of the standard BLAS libraries.

We propose the *TransMM* method to address these problems. *TransMM* regards the n-qubit state as an n-dimensional tensor and the size of each dimension is 2. Then *TransMM* uses the tensor transpose algorithm to reorder its dimensions, so that each fragment can be stored continuously in the memory and thus can be regarded as a row in a $2^{n-k} \times 2^k$ matrix, as shown in Figure 5 (b). In this way, the 2^{n-k} matrix-vector multiplications of size $1 \times 2^k \times 2^k$ are now converted to one $2^{n-k} \times 2^k \times 2^k$ GEMM. This GEMM computation not only offers better data locality, but also enables the usage of the highly-optimized cuBLAS library and can enjoy the huge performance boost provided by Tensor Cores. Moreover, the state does not need to be reordered back to the initial layout, because the qubit id of the following gates can be remapped.

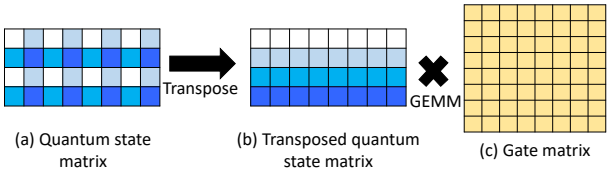
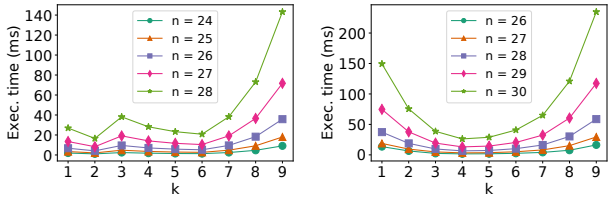


Figure 5: Applying one 3-qubit gate with the *TransMM* method. Blocks with the same color are in the same fragment.

Applying a k-qubit gate to the lowest k qubits of an n-qubit system is identical to a $2^{n-k} \times 2^k \times 2^k$ GEMM. The number of complex number multiplications is 2^{n+k}, which increases with the increment of k. However, Figure 6 shows the time of such GEMM operations can decrease with the growth of k on both platforms, because GEMM on GPU has higher flop/s when all dimensions are large. Therefore, if the applying of k-qubit gates on other qubits can be converted to the $2^{n-k} \times 2^k \times 2^k$ GEMM operation, they can simultaneously benefit from fewer gates offered by gate merging and higher throughput offered by GEMM with GPU-suitable size.



(a) V100-SXM2-16G (b) A100-PCIE-40G

Figure 6: cuBLAS GEMM performance of shape $2^{n-k} \times 2^k \times 2^k$, which is the core operation of applying a k-qubit gate to an n-qubit system

5.4 *OShareMem* Method

The *OShareMem* method is proposed to further improve the performance of HyQuas. In general, three main optimizations are done to reduce the redundant computation, optimize the data indexing, and mitigate the bank conflict.

Reducing redundant computation by task merging In the *ShareMem* method, each thread is assigned with one small matrix multiplication task. However, same operations exist in each task, eg., loading the gate and jumping to the code for the gate. When we merge multiple tasks to one thread, these operations only need to be done once on each thread, and can be reused by multiple tasks. Therefore, the overall operations can be reduced. To ensure that enough threads are active in the GPU streaming multiprocessor, the total thread number cannot be too small. In practice, the thread number is chosen with a balanced usage of shared memory and register within a block.

Lookup table based indexing. The $\overline{i_0, i_1, ..., i_{k-2}}$-th task of applying a gate on qubit t is to update the following two positions:

$$\begin{aligned} \text{lo} &= \overline{b_0, b_1, ..., b_{t-1}, \ \mathbf{0}, b_{t+1}, ..., b_{k-1}} \\ \text{hi} &= \overline{b_0, b_1, ..., b_{t-1}, \ \mathbf{1}, b_{t+1}, ..., b_{k-1}} \end{aligned} \quad (12)$$

, where

$$b_x = \begin{cases} i_x & \text{if} \quad x < t \\ i_{x-1} & \text{if} \quad x > t \end{cases} \quad (13)$$

, i.e., these two positions can be derived from inserting 0 or 1 to the tth position of $\overline{i_0, i_1, ..., i_{k-2}}$. In a general programming language, it can be implemented by the following code, which consists of 8 integer operations ($1 << t$ is only counted once).

```
lo = (pair_id >> t << (t+1)) | (pair_id & ((1<<t)-1));
hi = lo | (1 << t);
```

However, if we use a round-robin schedule to assign the pair to the threads, the increase of index is not related to the thread index within a block, so we can use a lookup table to save the index increment between the tasks. The `num_task` tasks on the target qubit t can be performed with the following code.

```
1 int add[num_task] = get_add(num_task, t);
2 int lo = (thread_id>>t<<(t+1)) | (pair_id&((1<<t)-1));
3 int hi = lo | (1 << t);
4 for (int i = 0; i < num_task; i++) {
5     APPLY_GATE(gate, lo, hi);
6     lo += add[i]; hi += add[i];
7 }
```

As a result, each reindexing only contains two integer operations (as in line 6), and it is also more friendly for compiler optimization.

Mitigate the shared memory bank conflict. In the *ShareMem* method, the data are stored sequentially in the shared memory, as shown in the left figure of Figure 7. Each grid represents a 128-bit double-precision complex type amplitude in the quantum state. For example, grid 0 stores the 0th data (000**00), and grid 1 stores the 1st data (100**00). Be careful that the least significant bit is on the left most of the index. Grids within the same column belong to the same shared memory bank. This layout can cause bank conflicts on gates applied to qubit 0, 1, and 2. As shown in Figure 7, to collect the data for a gate on qubit 0, the 32 threads within a wrap will first visit positions $\{0, 2, 4, 6, ..., 60, 62\}$ (the green grids) respectively, and then visit $\{1, 3, 5, 7, ..., 61, 63\}$ (the white grids). Both of the two visits can only access half of the columns in that matrix, so each visit wastes half of the banks and is serialized to 8 shared memory accesses. However, if the ith data is stored to the $i \oplus \frac{i}{8}$ ($\oplus$ denotes bit-wise exclusive-or) position as the right figure of Figure 7, the data within each visit is evenly divided into the 8 columns, so all banks can be utilized and only 4 shared memory accesses are needed for each visit. This remapping can also solve the bank conflict of single-qubit gates on qubit 1 and qubit 2, and mitigate the bank conflict of controlled gates on qubits 0, 1, and 2.

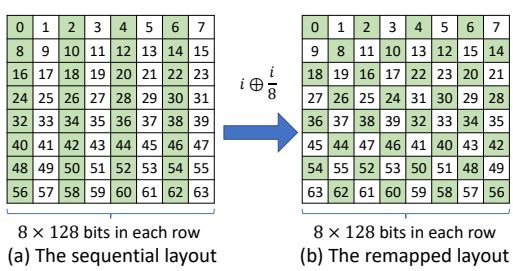


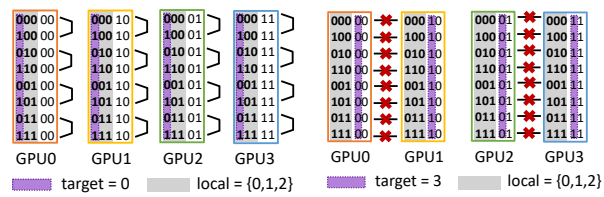
Figure 7: Fix shared memory bank conflicts via remapping

6 DISTRIBUTED SIMULATION

6.1 Global-local Swap Method

HyQuas uses the global-local swap method [27] to collaborate multiple GPUs. When HyQuas is available to 2^g GPUs, it will select g qubits as global qubits and mark the other $n - g$ qubits as local qubits. The amplitudes with the same index on the global qubits are partitioned into the same GPU. After such a partition, the gates operating on the local qubits can be processed within the GPUs while communication is needed for the gates on the global qubits. Figure 8 shows an example of simulating a 5-qubit system with 2^2 GPUs, i.e., $g = 2$. Qubits 3 and 4 are global qubits. Amplitudes within the same GPU share the same index on these two qubits. The applying of gates to qubit 0, 1, and 2 can be processed within each GPU, which is the same as applying gates in a 3-qubit system, as shown in Figure 8(a). However, as shown in Figure 8(b), gates on the global qubits 3 and 4 cannot be applied now, because every pair of data whose 3rd index or 4th index differs is partitioned to different GPUs.

To have the ability of simulating gates on all qubits, we need to partition the circuit into several stages. For example, to simulate the 5-qubit circuit in Figure 9 on 2^2 GPUs, two stages are needed. Stage 1 handles G1, G2, G3, and G4 on qubits 0, 1, and 2. Stage 2 handles G5, G6, and G7 on qubit 1, 3, and 4. G2 is boxed in Figure 9. Its execution can be delayed to the communication stage between stages 1 and 2, which will be discussed in Section 6.2.



(a) Gates can be applied on local qubits (b) Gates cannot be applied on global qubits

Figure 8: Apply different gates to a 5-qubit system simulated by $2^2 = 4$ GPUs

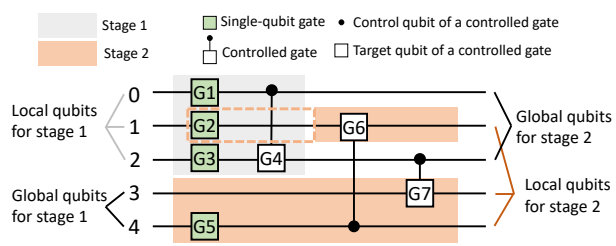


Figure 9: Split a 5-qubit quantum circuit into 2 stages

6.2 Data Redistribution for New Global Qubits

To change the global qubits to enter the next stage, the redistribution of the values is needed. Each value's target GPU is decided by the index of the new global qubits. The data redistribution is processed in the following steps:

(1) Pack the amplitudes that will be sent to the same GPU to a consecutive segment.
(2) Send the segments to the target GPUs and receive the segments from other GPUs simultaneously.

Figure 10 shows the workflow of switching from stage 1 to stage 2 for the circuit in Figure 9. Each GPU needs to send 2 amplitudes to each of the other GPUs. The target GPU is indexed by the new global qubits {0,2}. For example, ***0*** 0 ***0*** 1 0 will be sent to GPU 0, and ***1*** 0 ***1*** 1 0 will be sent to GPU 3. In step 1, a local transpose is issued, making the amplitudes with the same qubit 0 and qubit 2 consecutive. In step 2, the amplitudes are sent to the target GPU.

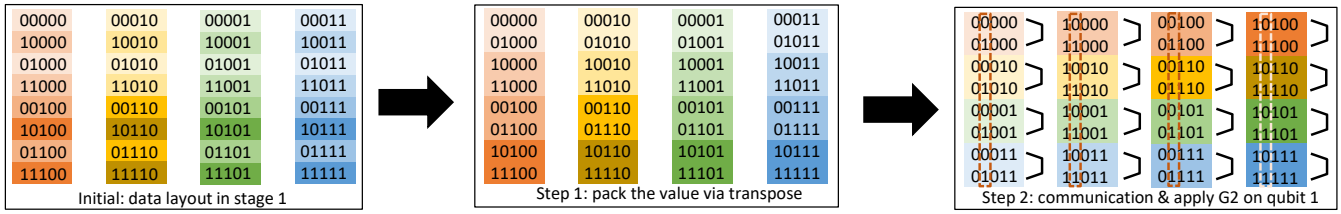


Figure 10: Change the global qubits from {3,4} to {0,2} to go into stage 2. Gates on qubit 2 can be processed simultaneously.

If a qubit is a local qubit in both the current stage and the next stage, gates to be applied to that qubit in the current stage can still be processed when the data arrives at the target GPU, so these gates can be processed in step 2 during the communication. Each data pair with only the index of that qubit differs have the same source GPU and target GPU, so they will be packed to the same segment in step 1 and be sent together in step 2. Once a segment arrives, the gate can be applied on the data pairs in the segment, so we do not need to wait until the end of communication to continue the processing of the gates. For example, in Figure 9, qubit 1 is a common local qubit in the two stages, so G2 can be processed during communication. As shown in Figure 10, data with only the 1st index differs are packed into the same segment (with the same background color) in step 1, and sent to the same target GPU in step 2. Gates on qubit 1 can be processed locally inside each segment. so the applying of gates in each segment can be started once that segment arrives at the target GPU.

7 EVALUATION

7.1 Experiment Setup

We evaluate HyQuas on a V100 platform and an A100 platform. The V100 platform has 4 V100-SXM2-16GB GPUs fully connected by NVLink 2.0. It has two Intel Xeon E5-2620 v4 CPU sockets. The compilers are GCC 8.3.0 and CUDA 10.2.89. The A100 platform has one A100-PCIE-40GB GPU and one AMD EPYC 7282 CPU. The compilers are GCC 8.3.0 and CUDA 11.0.2.

The quantum circuits used in experiments are several 28-qubit circuits collected from Qiskit [23], OpenQASM [1], Cirq [5], and OpenFermion [6], as listed in Table 1. The gate number of these circuits varies from 83 to 4558. In addition, `bc` circuits with 24 to 30 qubits are used to evaluate the performance with different numbers of qubits.

All these circuits are dumped into OpenQASM 2.0 format, with gates defined in "qelib1.inc". Gates not included in this header file are automatically decomposed by Qiskit or Cirq. HyQuas can parse the circuits and execute them automatically. The cuTT library [29] is used for the transpose in HyQuas. cuTT does not support inplace transpose, so an additional buffer is used to save the transpose result. All experiments are done with double precision by default.

7.2 Overall Performance

In this section, we compare HyQuas with state-of-the-art quantum circuit simulators. The versions of the simulators are listed in Table 2.

Table 1: Evaluated quantum circuits

Name	Source	Gates
Basis change (`bc`)	OpenFermion	4558
Bernstein–Vazirani algorithm (`bv`)	OpenQASM	83
Hidden shift (`hs`)	Cirq	144
Quantum approximate optimization algorithm (`qaoa`)	Cirq	1652
Quantum Fourier transform (`qft`)	Qiskit	406
Quantum volume (`qv`)	OpenQASM	1540
Quantum supremacy circuit (`sp`)	Cirq	883

Table 2: The versions of quantum simulators used for comparison

Simulator	Version	GPU package
QCGPU [31]	v0.1.1	N/A
Qibo [22]	v0.1.2	N/A
Qiskit [23]	v0.23.4	qiskit-aer-gpu v0.7.3
QuEST [30]	v3.2.0	N/A
Qulacs [42]	v0.2.0	qulacs-gpu v0.2.0
Yao [36]	v0.6.3	CuYao v0.2.7

For simulators that do not support parsing OpenQASM format, we implemented a parser to parse the code and convert it to their APIs. Some simulators do not support all gates in "qelib1.inc". When testing these simulators, we change the unsupported gates to other gates with less computation, e.g., changing `crx` gates to `cx` gates. QCGPU does not support double precision, so its single-precision performance is reported. Qibo depends on TensorFlow. We use TensorFlow 2.3.1 on the V100 platform as the default configuration of Qibo, and TensorFlow 2.4.1 on the A100 platform to enable the sm_80 support. Some performance-independent modifications are made due to the API changes from TensorFlow 2.3 to TensorFlow 2.4. Qulacs needs to configure an `opt_level` parameter. The best value is different for the benchmark circuits. We set `opt_level` to 4 on both platforms, which minimizes the geometric mean of execution time.

Figure 11 shows that HyQuas leads to 2.73× and 2.20× speedup on average on one V100 GPU and one A100 GPU respectively. The largest speedup comes from `qft`, which is a sparse circuit with a large number of controlled diagonal gates, so it cannot be simulated efficiently with the *BatchMV* method in Qulacs. For this circuit,

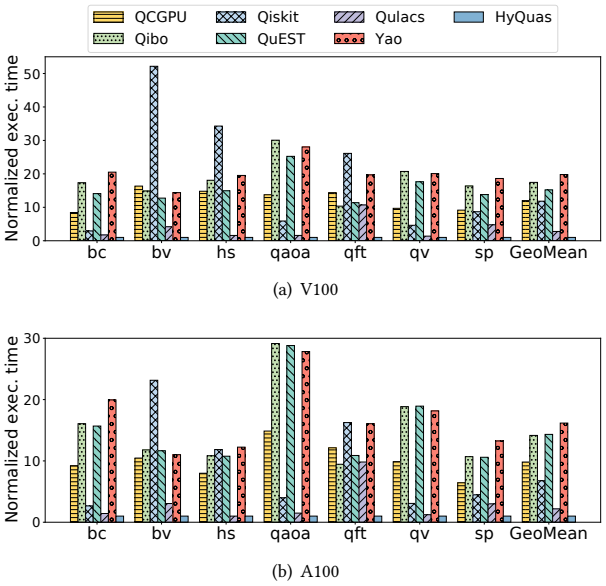


Figure 11: The comparison of single-GPU simulations among different simulators. Execution time is normalized to the execution time of HyQuas.

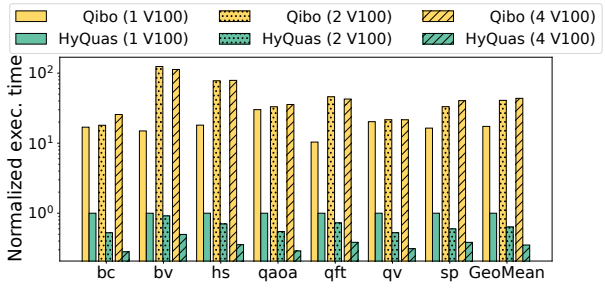


Figure 12: The performance on multi-GPU. Execution time is normalized to the single-GPU time of HyQuas.

HyQuas achieves 10.71× and 9.82× speedup on one V100 GPU and one A100 GPU respectively.

Only Qibo supports multi-GPU simulation with an arbitrary number of GPUs. Figure 12 shows that when 2 V100 cards and 4 V100 cards are used, the time will be 2.35× and 2.52× longer than the single V100 version. However, HyQuas can scale to 2 GPUs and 4 GPUs, with an averaged 1.56× and 2.93× speedup over the single V100 version.

HyQuas can have up to 227× speedup (129× on average) over Qibo on the 4-V100 platform. Such speedup is primarily from reduced communication traffic and improved communication performance of HyQuas. Figure 13 compares the communication traffic of Qibo and HyQuas. On average, HyQuas requires 8.14× and 9.54× less communication traffic than Qibo on 2 V100 cards and 4 V100 cards respectively, because HyQuas partitions the circuits more carefully and results in fewer stages. Moreover, most of HyQuas's

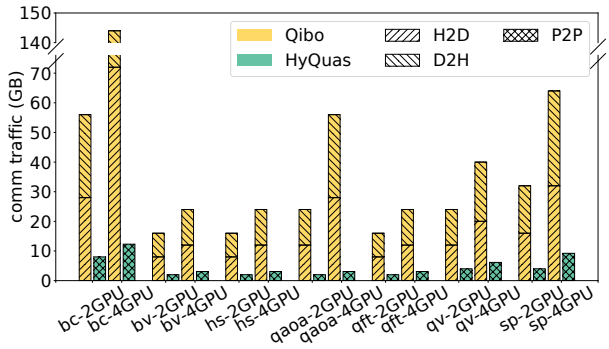


Figure 13: The communication traffic of Qibo and HyQuas. H2D, D2H, and P2P refer to CPU-to-GPU, GPU-to-CPU, and GPU-to-GPU traffic, respectively.

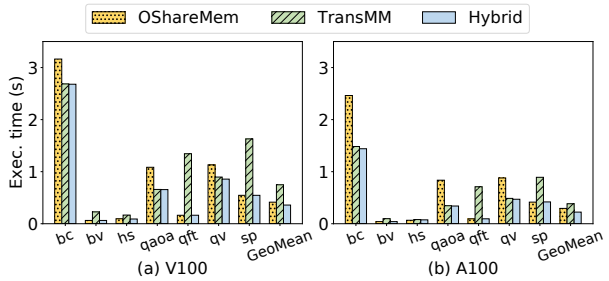


Figure 14: Comparison between the three methods

communication goes through the fast GPU-to-GPU mechanism, which can be accelerated by NVLink. Qibo only uses the slow CPU-to-GPU and GPU-to-CPU communication based on PCIe.

7.3 Single GPU Performace

In this section, we evaluate the performance of HyQuas with a single GPU. All experiments run on one V100-SXM2-16GB GPU (*V100*) or one A100-PCIE-32GB GPU (*A100*).

***Hybrid* Approach** Figure 14 shows the performance comparison of the *OShareMem* method, the *TransMM* method, and the *Hybrid* approach. On *V100*, the *Hybrid* approach has the best performance, with an averaged 1.15× and 2.09× speedup over the *OShareMem* method and the *TransMM* method respectively. And on *A100*, the *Hybrid* approach outperforms *OShareMem* and *TransMM* by 1.33× and 1.73× respectively. It can find a schedule that executes faster than both of the two methods. For example, when executing the qv circuit on *V*100, the *Hybrid* approach is 1.32× and 1.05× faster than the *OShareMem* method and the *TransMM* method respectively.

Figure 15 shows the prediction precision of the time predictor on the two platforms. Each data point represents an *OShareMem* group or a *TransMM* group in the bc circuit. The average relative error on *V*100 and *A*100 are 2.1% and 4.9%, respectively. This precision is enough for us to make a reasonable decision between the *OShareMem* method and the *TransMM* method.

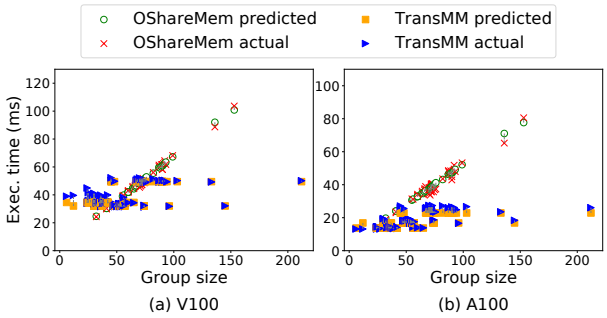


Figure 15: The predicted time and actual execution time of the groups partitioned from the bc circuit

Table 3: The partition time (Part.) and execution time (Exec.). Geo. refers to the geometric mean.

Name	Part. (ms)	Exec. (ms)	Name	Part. (ms)	Exec. (ms)
bc	61.68	2679.9	qft	0.78	162.9
bv	0.51	62.9	qv	34.84	856.9
hs	0.68	90.7	sp	3.40	545.5
qaoa	30.34	655.6	Geo.	4.82	358.7

Table 4: The partition time (Part.) and execution time (Exec.) of bc circuit with different numbers of qubits

Qubits	24	25	26	27	28
Gates	3330	3619	3920	4233	4558
Part. (ms)	55.77	60.53	62.30	56.46	61.68
Exec. (ms)	121.1	257.3	554.7	1188.8	2679.9

In Table 3, we report partition time and execution time of the benchmark circuits on *V100* as well as their geometric mean. The preprocessing stage takes 190 seconds, but it only needs to run once on each platform. In all benchmark circuits, the partition time is less than 5% of the corresponding execution time. Most time of the partition stage is spent on cuTT generating the execution plan of the transposes used in *TransMM* method. Therefore, the dense circuits, i.e., bc, qaoa, and qv spends much longer time to partition the circuit than the sparse circuits bv, hs, qft and sp.

Table 4 shows the partition time and execution time of different scale bc circuits on *V100*. The partition time grows almost linearly with respect to the number of gates in each circuit, while the execution time grows exponentially with respect to the number of qubits. Therefore, the partitioning overhead becomes smaller when circuits with more qubits are simulated.

***TransMM* Method** Figure 16 shows the time for processing a merged gate that targets on different numbers of qubits with the *BatchMV* method in Qulacs [42] and our *TransMM* method. Due to the transpose overhead, the *TransMM* method is slower than the *BatchMV* method when the merged gate size is 3 on the A100 platform and 3 or 4 on the V100 platform. However, a merged 3-qubit gate can only contain a small number of original gates,

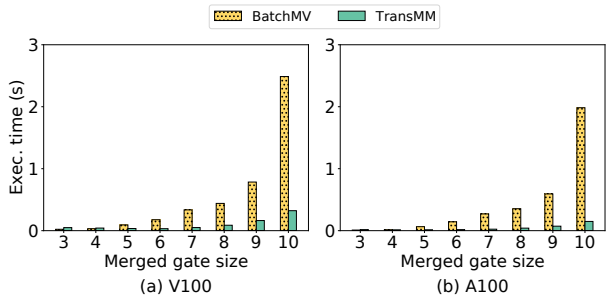


Figure 16: Comparison between the *BatchMV* method and the *TransMM* method

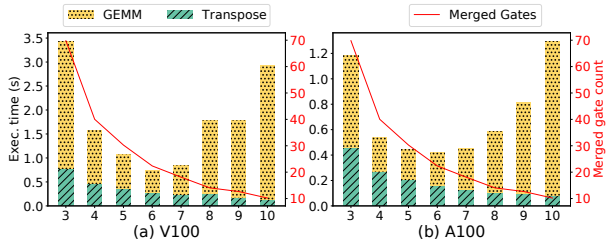


Figure 17: Breakdown of the *TransMM* method on the benchmark circuits

which can be processed efficiently with the *ShareMem* method. And there is no obstacle for us to integrate the *BatchMV* method into HyQuas. With a larger size of merged gates, the *TransMM* method shows significant speedup over the *BatchMV* method. For example, on *V100*, the *TransMM* method outperforms the *BatchMV* method by 5.52× and 7.71× respectively on 6-qubit gates and 10-qubit gates. On *A100*, the speedup is even larger due to the fast GEMM provided by double-precision Tensor Cores. The *TransMM* method can accelerate the applying of 6-qubit gates and 10-qubit gates by 8.43× and 13.32× respectively.

In Figure 17, we summarize the time and number of merged gates with the change of the largest active qubit size. The reported time and gate number are calculated from the geometric mean of the seven 28-qubit benchmark circuits. With the increase of active qubit size, the execution time first decreases due to the reduction of total merged gates, and then increases because of the exponential growth of the GEMM time.

***OShareMem* Method** Figure 18 shows the time for processing one gate by the *ShareMem* method with the three optimizations on *V100*. The "baseline" is the shared memory based method in Section 3.1. None of the existing shared memory based simulators [7, 24, 26, 47] is open sourced, so it is implemented by ourselves. "Multitask", "lookup", and "bank" refer to the three optimizations in Section 5.4. Due to the bank conflict, there is a clear difference in the time for gates on qubit 0-2 and qubit 3-9, but the range of time in each group of qubits is within 3 us. On average, the multitask optimization and the lookup optimization have a speedup of 1.76× and 1.83× over the baseline. The bank optimization has another

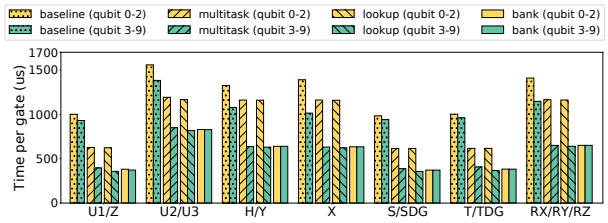


Figure 18: The time for processing a gate with the *ShareMem* method on a 28-qubit system with one V100 card. Gates with similar performance are grouped together using their arithmetic mean, with a variance of no more than 25 us.

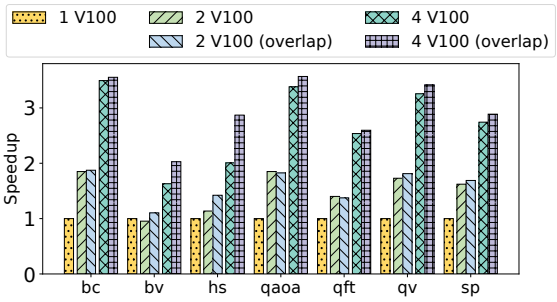


Figure 19: Comparison between the performance w/o overlap execution on the bs circuit

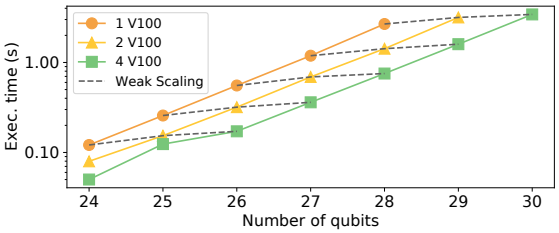


Figure 20: The weak scaling test of the bs circuits

1.67× speedup over the lookup version for gates on qubit 0-2 and keeps the time for gates on qubit 3-9 almost unchanged.

7.4 Multi-GPU Performance

Figure 19 shows the performance gain of overlap execution on different scales. With 2 and 4 V100 cards, the overlap execution can lead to an average speedup of 1.06× and 1.12× over the no overlapped version respectively. Overlap execution accelerates the 4 V100 hs circuit by 1.43×. In this circuit, communication takes 53.2% of the total time and 118 (81.9%) of its gates can be executed in pipeline, so the communication latency can be well hidden.

In Figure 20, we report the result of the weak scaling test on the bs circuits with different qubits. The data points representing the weak scaling tests are connected with dotted lines. The parallel efficiency, defined as $\frac{TimePerGate_{1gpu}}{TimePerGate_{ngpu}}$, varies from 85.6% to 90.8% on 2 V100 cards and from 83.0% to 90.0% on 4 V100 cards.

8 RELATED WORK

There are several existing quantum circuit simulators for different scales and hardware [2].

Some works have directly simulated quantum circuits on supercomputers [19, 27, 32, 41]. Wu et al. [46] and Zulehner et al. [49] save state in a compressed format to save memory, thus being able to simulate more qubits. And many algorithms [16, 18, 28, 34, 39, 43, 44] have been designed to simulate the quantum supremacy circuit [15] with a large number of qubits and a limited depth, to explore the frontier of "quantum supremacy".

There are also simulators designed for small clusters, especially for GPU platforms [7, 10, 11, 21–23, 23–26, 30, 31, 34, 36, 42, 47]. CUDA is widely used for implementing quantum simulators on GPU platform, while TensorFlow [22] and OpenCL [31] are also used in some simulators. Jones et al. [30] have shown that simulation on a single Tesla K40m GPU is 5× faster than on a 24-thread CPU.

To improve the performance of GPU quantum simulation, many works [7, 24–26, 47] choose to copy partial quantum states into shared memory to apply gates with lower latency. Some of these simulators [24, 26, 47] optimize the global memory bandwidth of copying states into shared memory by coalescing the visit of global memory. Qulacs [42] tries to reduce the total gate number by merging quantum gates.

A common way of implementing a multiple-GPU quantum circuit simulator is to distribute 2^n amplitudes to GPUs and exchange intermediate amplitudes with a chunk-based method [21, 23, 47] or with the global-local swap method [22, 34] proposed in distributed CPU quantum circuit simulators [20, 27]. However, none of these simulators would scale well if they adopted the accelerated single GPU implementation in HyQuas.

The need for simulators in quantum area is not limited to the tasks of getting the state after applying a sequence of gates. Sampling [33], density matrix simulation [32], and equivalent checking [17] are also needed by quantum scientists. HyQuas has the potential of being a backend for these tasks and accelerates them.

9 CONCLUSION

We propose HyQuas, a high performance quantum circuit simulation system that can automatically analyze the pattern of a given quantum circuit and choose the proper simulation methods for different parts of it. HyQuas also provides two highly optimized methods, *OShareMem* and *TransMM*, as well as a GPU-centric communication pipelining approach. Experiments show that HyQuas can achieve up to 10.71× speedup on a single GPU and 227× speedup on a GPU cluster compared with state-of-the-art simulators.

10 ACKNOWLEDGMENTS

We would like to show our gratitude to the anonymous reviewers for their valuable comments and suggestions. We would also thank the colleagues from Tsinghua University PACMAN group for the valuable discussions between us. This work is partially supported by National Key R&D Program of China (2018YFA0306702), National Natural Science Foundation of China (U20A20226), Beijing Natural Science Foundation (4202031), and Tsinghua University-Peking Union Medical College Hospital Initiative Scientific Research Program. Jidong Zhai is the corresponding author of this paper.

REFERENCES

[1] [n.d.]. Gate and operation specification for quantum circuits. https://github.com/Qiskit/openqasm/tree/0af8b8489f32d46692b3a3a1421e98c611cd86cc

[2] [n.d.]. List of QC simulators. https://www.quantiki.org/wiki/list-qc-simulators.

[3] [n.d.]. On "Quantum Supremacy" | IBM Research Blog. https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy

[4] [n.d.]. A Preview of Bristlecone, Google's New Quantum Processor. http://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html

[5] [n.d.]. A python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits. https://github.com/quantumlib/Cirq

[6] [n.d.]. Quantum circuits for simulations of quantum chemistry and materials. https://github.com/quantumlib/OpenFermion-Cirq/blob/57558da57569f9979d31cbb895e23128f3773282

[7] Andrei Amariutei and Simona Caraiman. 2011. Parallel quantum computer simulation on the GPU. In *15th International Conference on System Theory, Control and Computing*. IEEE, 1–6.

[8] 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.

[9] Alán Aspuru-Guzik, Anthony D Dutoi, Peter J Love, and Martin Head-Gordon. 2005. Simulated quantum computation of molecular energies. *Science* 309, 5741 (2005), 1704–1707.

[10] Anderson Avila, Adriano Maron, Renata Reiser, Mauricio Pilla, and Adenauer Yamin. 2014. GPU-aware distributed quantum simulation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 860–865.

[11] Anderson Avila, Renata HS Reiser, Mauricio L Pilla, and Adenauer C Yamin. 2016. Optimizing D-GM quantum computing by exploring parallel and distributed quantum simulations under GPUs arquitecture. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 5146–5153.

[12] Ryan Babbush, Jarrod McClean, Dave Wecker, Alán Aspuru-Guzik, and Nathan Wiebe. 2015. Chemical basis of Trotter-Suzuki errors in quantum chemistry simulation. *Physical Review A* 91, 2 (2015), 022311.

[13] Charles H Bennett and Gilles Brassard. 2020. Quantum cryptography: Public key distribution and coin tossing. *arXiv preprint arXiv:2003.06557* (2020).

[14] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549, 7671 (2017), 195–202.

[15] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. 2018. Characterizing quantum supremacy in near-term devices. *Nature Physics* 14, 6 (2018), 595–600.

[16] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, and Hartmut Neven. 2017. Simulation of low-depth quantum circuits as complex undirected graphical models. *arXiv preprint arXiv:1712.05384* (2017).

[17] 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.

[18] Ming-Cheng Chen, Riling Li, Lin Gan, Xiaobo Zhu, Guangwen Yang, Chao-Yang Lu, and Jian-Wei Pan. 2020. Quantum-teleportation-inspired algorithm for sampling large random quantum circuits. *Physical review letters* 124, 8 (2020), 080502.

[19] Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Willsch, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. 2019. Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications* 237 (2019), 47–61.

[20] Koen De Raedt, Kristel Michielsen, Hans De Raedt, Binh Trieu, Guido Arnold, Marcus Richter, Th Lippert, Hiroshi Watanabe, and Nobuyasu Ito. 2007. Massively parallel quantum computer simulator. *Computer Physics Communications* 176, 2 (2007), 121–136.

[21] Jun Doi, Hitomi Takahashi, Rudy Raymond, Takashi Imamichi, and Hiroshi Horii. 2019. Quantum computing simulator on a heterogenous HPC system. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 85–93.

[22] Stavros Efthymiou, Sergi Ramos-Calderer, Carlos Bravo-Prieto, Adrián Pérez-Salinas, Diego García-Martín, Artur Garcia-Saez, José Ignacio Latorre, and Stefano Carrazza. 2020. Qibo: a framework for quantum simulation with hardware acceleration. *arXiv preprint arXiv:2009.01845* (2020).

[23] Héctor Abraham et al. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[24] Eladio Gutierrez, Sergio Romero, Maria A Trenas, and Emilio L Zapata. 2007. Simulation of quantum gates on a novel GPU architecture. In *International Conference on Systems Theory and Scientific Computation*. Citeseer.

[25] Eladio Gutierrez, Sergio Romero, Maria A Trenas, and Emilio L Zapata. 2008. Parallel quantum computer simulation on the CUDA architecture. In *International Conference on Computational Science*. Springer, 700–709.

[26] Eladio Gutiérrez, Sergio Romero, María A Trenas, and Emilio L Zapata. 2010. Quantum computer simulation using the CUDA programming model. *Computer Physics Communications* 181, 2 (2010), 283–300.

[27] Thomas Häner and Damian S Steiger. 2017. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–10.

[28] Cupjin Huang, Fang Zhang, Michael Newman, Junjie Cai, Xun Gao, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, et al. 2020. Classical simulation of quantum supremacy circuits. *arXiv preprint arXiv:2005.06787* (2020).

[29] Antti-Pekka Hynninen and Dmitry I Lyakh. 2017. cutt: A high-performance tensor transpose library for cuda compatible gpus. *arXiv preprint arXiv:1705.01598* (2017).

[30] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific reports* 9, 1 (2019), 1–11.

[31] Adam Kelly. 2018. Simulating quantum computers using OpenCL. *arXiv preprint arXiv:1805.00988* (2018).

[32] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. 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*. 1–15.

[33] 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.

[34] Zhen Li and Jiabin Yuan. 2017. Quantum Computer Simulation on GPU Cluster Incorporating Data Locality. In *International Conference on Cloud Computing and Security*. Springer, 85–97.

[35] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. 2013. Quantum algorithms for supervised and unsupervised machine learning. *arXiv preprint arXiv:1307.0411* (2013).

[36] Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, and Lei Wang. 2020. Yao. jl: Extensible, efficient framework for quantum algorithm design. *Quantum* 4 (2020), 341.

[37] Michael A Nielsen and Isaac Chuang. 2002. Quantum computation and quantum information.

[38] Roman Orus, Samuel Mugel, and Enrique Lizaso. 2019. Quantum computing for finance: overview and prospects. *Reviews in Physics* 4 (2019), 100028.

[39] Edwin Pednault, John A Gunnels, Giacomo Nannicini, Lior Horesh, Thomas Magerlein, Edgar Solomonik, and Robert Wisnieff. 2017. Breaking the 49-qubit barrier in the simulation of quantum circuits. *arXiv preprint arXiv:1710.05867* 15 (2017).

[40] Maria Schuld and Nathan Killoran. 2019. Quantum machine learning in feature Hilbert spaces. *Physical review letters* 122, 4 (2019), 040504.

[41] Mikhail Smelyanskiy, Nicolas P. D. Sawaya, and Alán Aspuru-Guzik. 2016. qHiPSTER: The Quantum High Performance Software Testing Environment. arXiv:1601.07195 [quant-ph]

[42] Yasunari Suzuki, Yoshiaki Kawase, Yuya Masumura, Yuria Hiraga, Masahiro Nakadai, Jiabao Chen, Ken M. Nakanishi, Kosuke Mitarai, Ryosuke Imai, Shiro Tamiya, Takahiro Yamamoto, Tennin Yan, Toru Kawakubo, Yuya O. Nakagawa, Yohei Ibe, Youyuan Zhang, Hirotsugu Yamashita, Hikaru Yoshimura, Akihiro Hayashi, and Keisuke Fujii. 2020. Qulacs: a fast and versatile quantum circuit simulator for research purpose. arXiv:2011.13524 [quant-ph]

[43] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandra. 2018. A flexible high-performance simulator for the verification and benchmarking of quantum circuits implemented on real hardware. (2018).

[44] Benjamin Villalonga, Dmitry Lyakh, Sergio Boixo, Hartmut Neven, Travis S Humble, Rupak Biswas, Eleanor G Rieffel, Alan Ho, and Salvatore Mandrà. 2020. Establishing the quantum supremacy frontier with a 281 pflop/s simulation. *Quantum Science and Technology* 5, 3 (2020), 034003.

[45] Wikipedia contributors. 2021. List of quantum processors — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=List_of_quantum_processors&oldid=999209651 [Online; accessed 16-January-2021].

[46] 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.

[47] Pei Zhang, Jiabin Yuan, and Xiangwen Lu. 2015. Quantum computer simulation on multi-GPU incorporating data locality. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 241–256.

[48] Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. 2020. Quantum computational advantage using photons. *Science* 370, 6523 (2020), 1460–1463.

[49] 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.