dgQuEST: Accelerating Large Scale Quantum Circuit Simulation through Hybrid CPU-GPU Memory Hierarchies

Tianyu Feng[1,2], Siyan Chen[2], Xin You[2], Shuzhang Zhong[2], Hailong Yang[1,2(✉)], Zhongzhi Luan[2], and Depei Qian[2]

[1] State Key Laboratory of Software Development Environment, Beihang University, Beijing, China
hailong.yang@buaa.edu.cn
[2] School of Computer Science and Engineering, Beihang University, Beijing, China

Abstract. With the advancement of quantum computing, verifying the correctness of the quantum circuits becomes critical while developing new quantum algorithms. Constrained by the obstacles of building practical quantum computers, quantum circuit simulation has become a feasible approach to develop and verify quantum algorithms. Although there are many quantum simulators available, they either achieve low performance on CPUs, or limited simulation scale (e.g., number of qubits) on GPUs due to limited memory capacity. Therefore, we propose *dgQuEST*, a novel acceleration method that utilizes hybrid CPU-GPU memory hierarchies for large-scale quantum circuit simulation across multiple nodes. *dgQuEST* adopts efficient memory management and communication schemes to leverage the distributed CPU and GPU memories for accelerating large-scale quantum simulation. Our evaluation demonstrates that *dgQuEST* achieves an average speedup of 403× compared to *QuEST* on quantum circuit simulation with 32 qubits, and scales to quantum circuit simulation with 35 qubits on two GPU nodes, far beyond the state-of-the-art implementation *HyQuas* can support.

Keywords: Quantum simulation · Distributed GPU acceleration · Memory and communication optimization

1 Introduction

With the development of quantum algorithms, quantum computing has become the most likely means to surpass the performance of traditional computing in the future. However, it is difficult to verify quantum algorithms due to the still early age of building practical quantum computers. Therefore, using traditional computers to simulate the operations of quantum circuits has become one feasible approach to develop and verify new quantum algorithms. In general, quantum simulation methods are mainly divided into two categories, such as state vector

Published by Springer Nature Switzerland AG 2022
C. Cérin et al. (Eds.): NPC 2021, LNCS 13152, pp. 16–27, 2022.
https://doi.org/10.1007/978-3-030-93571-9_2

(full amplitude) method and tensor network method. The scalability of the state vector method is limited by the number of qubits, as each additional qubit will double the memory usage. The tensor network method uses tensor shrinkage and decomposition to avoid the exponential memory bloating, however it suffers poor performance for deep circuits or circuits with higher entanglement. Many quantum circuit simulation software have been developed over years, including *Qibo* [3], *QuEST* [7], *qHipster* [10], *AC-QDP* [6], *QuiMB* [5], *TN QVM* [9], etc. Among them, *QuEST* [7] is one of the most widely used high-performance quantum circuit simulators, and a large amount of research work has been carried out based on *QuEST* [1,4,8,12].

QuEST [7] is designed based on the full-amplitude quantum simulation method and supports single-qubit gates with multiple control qubits. Among them, the application of the qubit gate is carried out by applying corresponding computations on the entire state vector in sequence. The computations on the state vector are paired up and each pair is independent from others. Therefore, the simulation of *QuEST* can be easily parallelized on GPU. However, the parallel simulation of *QuEST* can only be supported on a single GPU, which constrains the size of quantum circuit can be simulated due to the limited GPU memory. The latest improvement on *QuEST* such as *HyQuas* [12] can speedup the simulation on multiple GPUs. However, it fails to exploit the hybrid CPU-GPU memory hierarchies, and thus requires more GPUs to support large-scale quantum circuit simulation. Therefore, we propose *dgQuEST*, a novel acceleration method that exploits hybrid CPU-GPU memory hierarchies with efficient memory management and communication schemes to improve the performance of large-scale quantum circuit simulation.

Specifically, this paper makes the following contributions:

- We propose a CPU-GPU hybrid memory management scheme, which effectively utilizes the large capacity of CPU memory as well as the high performance of GPU memory for large-scale quantum simulation.
- We propose a page-table based memory management scheme to manage the qubit mapping of the entire state vector, which improves the data locality of the state vector access for better performance.
- We propose a pipelined communication scheme to reduce the overhead of distributed memory access, which further improves the performance of *dgQuEST* when scaling to multiple GPU nodes.

2 Background and Motivation

2.1 Full-amplitude Quantum Simulator

A full-amplitude quantum simulator (FAQS) stores all the amplitudes in a state vector SV and a quantum gate can be represented as a matrix transformation. Assume that the quantum register R has n qubits and the SV represents the current quantum state $|\Psi\rangle$, i.e. $|\Psi\rangle$ can be represented as in Eq. (1):

$$|\Psi\rangle = \sum SV_{q_{n-1}q_{n-2}\cdots q_1 q_0} |q_{n-1}q_{n-2}\cdots q_1 q_0\rangle \tag{1}$$

where $q_i \in \{0,1\}, i \in \{0,1,\ldots,n-1\}$. If a quantum NOT gate, which is the same as binary NOT operations in classical computers, is applied on the xth qubit (numbered from 0), the state vector SV will be transformed to SV' as shown in Eq. (2), where $q_i \in \{0,1\}, i \in \{0,1,\ldots,x-1,x+1,\ldots,n-1\}$.

$$\begin{pmatrix} SV'_{q_{n-1}q_{n-2}\cdots q_{x+1}0q_{x-1}\cdots q_1q_0} \\ SV'_{q_{n-1}q_{n-2}\cdots q_{x+1}1q_{x-1}\cdots q_1q_0} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} SV_{q_{n-1}q_{n-2}\cdots q_{x+1}0q_{x-1}\cdots q_1q_0} \\ SV_{q_{n-1}q_{n-2}\cdots q_{x+1}1q_{x-1}\cdots q_1q_0} \end{pmatrix} \quad (2)$$

Therefore, performing a single-qubit gate transformation introduces data dependencies between elements in the state vector in pairs. In detail, the distance of the interdependent elements in a pair is 2^i when the gate performs on the i-th qubit, which leads to poor spatial locality in memory. Besides, when a series of quantum gates perform on different qubits separately, the data-dependent structures changes completely during each computation, leading to higher cache miss within the existing memory hierarchy.

2.2 Optimizing FAQS on Distributed CPU Nodes

For quantum simulations that exceed the available memory, FAQS needs to distribute the simulation to fit in the memory available on each node. In detail, the state vector is split into blocks stored separately in each node and the entire state vector needs to be updated when applying a transformation of a gate to the current state. However, the interdependent data may be stored separately in different nodes. Figure 1(a) demonstrates an example of data dependence across nodes in a two-node distributed FAQS, where each node depends on all the data on the other node to finish the simulation of a quantum gate on the 3rd qubit. Figure 1(b) demonstrates a more general case when two single-qubit gates are performed on the higher two qubits, respectively. The *peer* relationships are established between nodes two by two based on the target qubit of the gate and each node needs to obtain all the data of the *peer* to compute its local result.

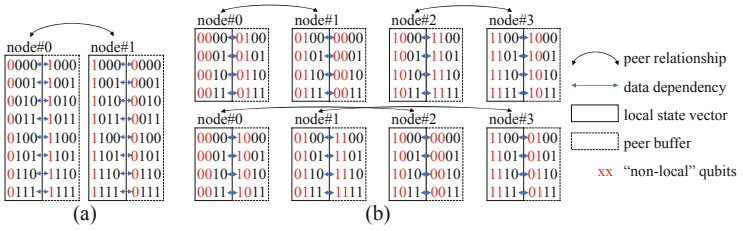

Fig. 1. Peer data interaction in a distributed implementation of FAQS. The simulated quantum register is assumed to contain 4 qubits and distributed to (a) two nodes, each storing 8 state vector elements (3 qubits), and (b) four nodes, each storing 4 state vector elements (2 qubits), with two single-qubit gates acting on the third and fourth qubits shown in the upper and lower halves of the graph, respectively.

2.3 Optimizing FAQS on GPUs

There are several research works to optimize FAQS on GPUs. *HyQuas* combines two existing GPU optimization methods such as *ShareMem* and *BatchMV*, both of which take multiple gates and calculate them together to reduce memory access. The *ShareMem* method loads interdependent data required by the quantum gates into shared memory as a local state vector and sequentially applies quantum gate transformations. Whereas the *BatchMV* method merges the gates into a larger matrix and applies a matrix-vector multiplication between the matrix and the local state vector. Utilizing high speed shared memory, these two methods greatly reduce memory access overhead. *HyQuas* further improves the above two methods and implements simulation on multiple GPUs with optimized data transfer mechanism. However, *HyQuas* requires to store all data into GPU global memory, which strictly limits the number of qubits for simulation if the number of nodes is limited. *Qibo* also supports accelerating quantum simulation on multiple GPUs, but achieves poor performance and high memory usage due to frequent data exchange between CPU and GPUs. On contrast, *dgQuEST* leverages the hybrid CPU-GPU memory hierarchies across multiple nodes efficiently, which accommodates accelerated large-scale quantum circuit simulation.

3 Methodology

3.1 Design Overview

According to the memory hierarchy, *dgQuEST* applies a *three-level memory model* to divide the distribution of state vectors into three top-down levels: CPU main memory, GPU global memory, and shared memory. Figure 2 demonstrates the overall design of *dgQuEST*, including *memory manager*, *gate aggregator*, *task dispatcher*, *executor*, *data reorganizer*, and *communicator*. The *memory manager* maintains the state vector in page tables. The *gate aggregator* is used to reorder the quantum gates when the storage hierarchy is lowered to reduce the number of interactions. Besides, *dgQuEST* also applies *three-level memory model* inside *memory manager* and *executor*, and utilizes *gate aggregator* for *circuit partitioning* along this *three-level memory model*. The *task dispatcher* aims to assign task blocks in main memory to different GPUs on each node. The *executor* performs the actual quantum gate operations with the assistance of *gate aggregator* to decide which data to load into shared memory. After computation, the data is reorganized with *data reorganizer* according to the target qubits of the current circuit and the next circuit to match the new dynamic mapping between qubits and state vector index bits. The reorganized data is sent to the corresponding node via *communicator* or written back into the current node, both with new page numbers calculated from the result of remapping.

3.2 Page-Table Based Memory Management

To achieve efficient data exchange between nodes, we implement page-table based memory management mechanisms and maintain a memory pool for pages.

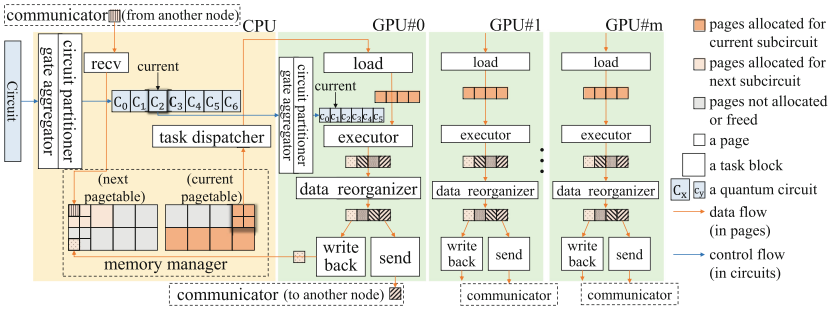

Fig. 2. Overall design of *dgQuEST*. For simplicity, data flows and control flows are omitted except for GPU#0.

Specifically, the structure of the entire m-bits state vector index is divided into *page number* (higher $2k$ bits) and *offset* (lower $m - 2k$ bits), where the number of *page number* bits is always even and is divided into two halves. The lower half and the page offset jointly form the GPU explicit memory index, whose value range is the memory block size of a GPU calculation task. The higher half is further divided into the task number for *task division*. In general, the page is the smallest unit of data migration. Using page tables has the following advantages: ***1)*** The page enables fine-grained buffering and scheduling for communication. During data exchanges, once the node successfully transfers a page in the main memory to the GPU global memory, the page can be released to receive new data. Thus, only a small number of caching pages are needed for data exchange, which effectively reduces the memory overhead for communication. ***2)*** The logical page mapping of the page table enables flexible dynamic qubit mapping for better data locality. Specially, when the qubit mapping change occurs only in the page number part of the state vector index, the dynamic mapping can be completed only by modifying the page table without any data exchanges. ***3)*** The page number provides sufficient information (task number and node index) of the data block for the communicator to confirm the destination and source of the data during data exchange.

3.3 DAG-Based Gate Aggregation

dgQuEST analyzes the dependencies between quantum gates in a quantum circuit to construct a directed acyclic graph (DAG). Then *dgQuEST* applies a heuristic algorithm to derive several aggregated sub-circuits from the DAG. In general, *gate aggregation* reorders the quantum circuit and splits it into as few sub-circuits as possible under the premise that the number of target qubits of each sub-circuit does not exceed a given threshold.

DAG Construction - Each node in the DAG corresponds to a quantum gate in the quantum circuit and each directed edge indicates a directed dependency between the quantum gates. We assume that the control qubit set of the quantum gate corresponding to node P is C_P and the target qubit set is

T_P. For all nodes A and B such that A appears earlier than B in the original execution order, there is a directed edge from A to B, iff there exists a qubit $x \in (C_A \cup T_A) \cap (C_B \cup T_B)$ such that for all gate C appears between A and B, $x \notin C_C \cup T_C$. Therefore, for all nodes in the constructed DAG, the quantum gate corresponding to one node must be calculated before all the quantum gates corresponding to its directed linked nodes. All computation reorderings that guarantee this principle will result in mathematically equivalent quantum circuit simulations. To construct a DAG, we maintain a vector storing heading gates on each qubit (the last gate whose target or control qubits contain that qubit). When adding a gate G, we add directed edges from the heading gates on qubits in $C_G \cup T_G$ to G and make G as the new heading gate on these qubits.

Greedy Sub-circuit Partitioning - According to the *three-level memory model*, a lower memory level has lower data access latency and less available number of qubits. Thus, a single quantum circuit at a higher level has to be divided into multiple sub-circuits. Due to the high data exchanging overhead, the execution efficiency is severely affected by the number of sub-circuits. Therefore, *dgQuEST* reorders the quantum gates with greedy sub-circuit partitioning based on the constructed DAG to reduce the number of sub-circuits when the memory level is lowered. Specifically, successor nodes has a greater chance of having the same target qubit as its predecessor node. Besides, since the interdependent data introduced by each sub-circuit derived from a DAG must meet the maximum memory capacity, *dgQuEST* also needs to constrain the number of target qubits. Thus, we keep selecting a new gate with no predecessor and applying depth-first search from it to add nodes to the sub-circuit until the threshold of target qubit number is reached or no more nodes can be added.

Figure 3 demonstrates an example of aggregating and partitioning a quantum circuit with 7 qubits in a *three-level memory model*, where the number of qubits within the GPU global memory is 5 and the number of qubits within the shared memory is 2. Different colors indicate the result of partitioning the circuit into sub-circuits at GPU global memory level, and with the same color, different patterns further indicate the result of partitioning at the shared memory level. The actual calculation occurs at the shared memory level (the register level is not considered), before which the data will be migrated from the higher level into the lower level according to the partition result.

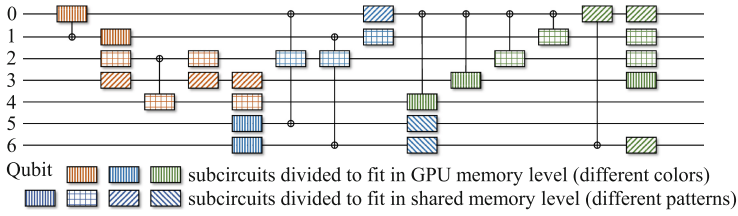

Fig. 3. A quantum circuit aggregated and partitioned during 2 memory level lowering.

Intra-node Gate Aggregation and Fusion - For intra-node optimization, *dgQuEST* utilizes shared memory on GPU and applies *gate aggregators* to reorder the execution of quantum gates in two stages to reduce the amount of both global and shared memory access. In the first stage of gate aggregation, the quantum circuit is divided into several sub-circuits by limiting the maximum target number of qubits of the gate aggregator within the shared memory capacity. The second stage of the gate aggregation (a.k.a., *gate fusion*) gathers the quantum gates of the same target qubit by limiting the number of target qubits to 1, which enables computations with continuous targets. These two stages of gate aggregation further improve the data reuse within shared memory and register during execution.

3.4 Remap-Based Data Reorganization

For better locality, we ensure the data dependency of a global-memory-level sub-circuit is within global memory level by remapping qubits to different state vector index bits and reorganizing data. The remap-based data reorganization involves three stages, including **qubit remapping**, **in-page data reorganization**, and **page remapping**. **Qubit remapping** dynamically remaps the independent qubits into higher memory address bits according to the data dependency obtained by the *gate aggregation*. **In-page data reorganization** rearranges the data within the *offset* bits in the page to adapt the new mapping, which is accelerated by GPU for better performance. **Page remapping** reorganize the data within the *page number* bits in the page by modifying the page table with new mappings for each page. The **qubit remapping** and **in-page data reorganization** are performed after the quantum gate calculation and before sending or writing back the results, while **page remapping** is handled during sending or writing back the results.

Figure 4 demonstrates an example of remap-based data organization with 16 qubits, where the upper 8 bits of state vector index represent page number and the light-colored bits are the qubits without any data dependency. The remapping process can be divided into several steps in 2 scopes: 1) remapping in the scope of the index bits inside a task block within GPU memory, where an in-page data reorganization is needed to keep the consistency between data and the new mapping; 2) remapping in the scope of page number, where page table is changed to match the new mapping. After remapping, qubits involved in the next sub-circuit are all mapped to offset bits inside a GPU task data block and data dependencies are always inside the block.

3.5 Pipelined Communication

In *dgQuEST*, there are the following forms of data migration: 1) Loading a page from the main memory to GPU global memory (*load*); 2) Writing back a page from the GPU global memory to the local node (*write back*); 3) Sending a page from the GPU global memory to a remote node (*send*); 4) A page is received from a remote node and written to the main memory (*receive*).

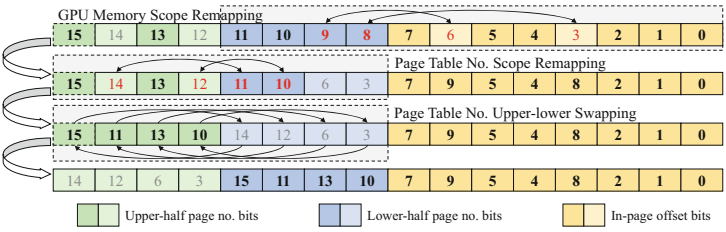

Fig. 4. An example of qubit remapping to state vector index bits. The remapping consists of 3 steps. In-page data is reorganized according to the in-page remapping. Remapping of qubits in page table number scope pages remapping during data exchanging between tasks.

Figure 5 demonstrates the pipelined communication processes. Among them, the *load*, *write back*, and *send* processes of each GPU are controlled by independent threads. The *receive* operation of each node is controlled by a dedicated thread. The *reorganize* operation rearranges the data in memory to match the qubit mapping of the next sub-circuit. Since *write back* occurs more frequently, asynchronous *send* and *write back* operations are adopted to allow overlapping—GPU threads launch asynchronous *send* of a page targeted to other nodes and continue to perform short-running *write back* operations to avoid blocking the execution of the next tasks. Therefore, the followed task execution can overlap with the other three data migration forms of the previous task, which can significantly improve communication efficiency.

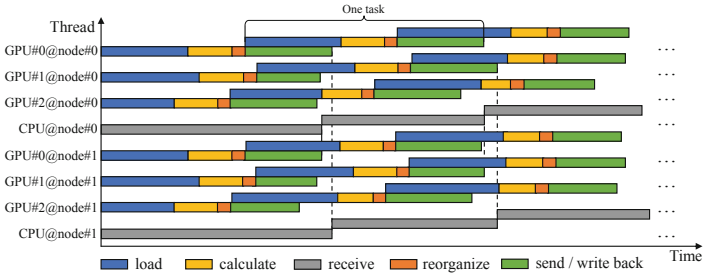

Fig. 5. An execution scenario with data transfer pipelined

4 Evaluation

4.1 Experimental Setup

To evaluate the capability and performance of large-scale quantum circuits simulation by *dgQuEST*, we simulate various circuits with *dgQuEST* on our two-node GPU cluster. Each node consists of two Intel Xeon E5-2680 v4 CPU, two

NVIDIA V100 32 GB GPU and 384 GB DDR4 DRAM. Two nodes are connected with 40Gb/s FDR Infiniband. For comparison, we choose *QuEST* v3.2.1 [7] and *Qibo* v0.1.6.dev2 (with qibotf library v0.0.2) [3]. We did not compare with GPU accelerated *QuEST* and *HyQuas* as they both fail to run the evaluated circuits due to out-of-memory error. For evaluated quantum circuit datasets, we use qubits ranges from 32 to 34 with 1) randomly generated circuits (*RD*), 2) a combination of one GHZ and one QFT circuit (*GQ*), 3) BV circuit (*BV*), and 4) supremacy circuit (*SP*). Circuits 3) and 4) are exported from Cirq [2].

4.2 Overall Performance

We simulated the above-mentioned quantum circuits with 32 to 34 qubits using *QuEST*, *Qibo*, and *dgQuEST* on our GPU cluster. The simulation times and the speedups against *QuEST* are demonstrated in Fig. 6. Note that *Qibo* can only run on one node and it fails to simulate circuits with qubits larger than 32 due to out-of-memory error. Both the execution of *QuEST* and *dgQuEST* are distributed on two nodes. As shown in Fig. 6, *dgQuEST* exhibits 455x (geomean) speedup on 34 qubits compared to the CPU implementation of *QuEST*, and 120x (geomean) speedup on 32 qubits compared to *Qibo* on one node. The significant performance improvement comes from two folds. One is our three-level CPU-GPU hybrid memory model and corresponding page-based memory management approaches can effectively utilize the large memory capacity of DRAM with less swapping overhead, which enables GPU acceleration for larger-scale qubit simulations. The other is our data reorganization and pipelined communication can largely reduce both the memory and time overhead for data exchanges.

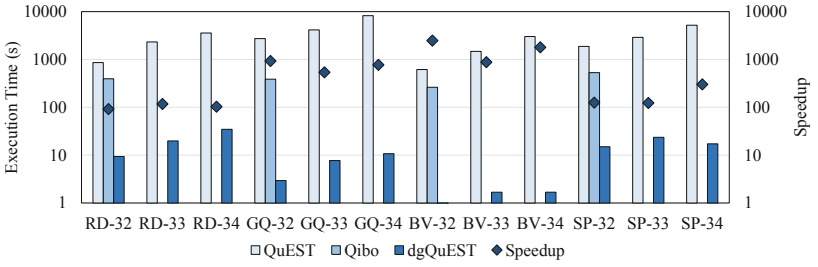

Fig. 6. Overall execution time of *QuEST*, *Qibo*, and *dgQuEST* and the speedup of *dgQuEST* against *QuEST*. Both execution times and speedups are shown in log scale.

4.3 Qubit Scalability

To prove the scalability of the number of simulated qubits within a given memory capacity, we simulate GZ quantum circuits with qubits ranging from 31 to 35 on both one node (*1n2g*) and two nodes (*2n4g*) with *dgQuEST*. The execution times of each evaluated circuit with different configurations are shown in

Fig. 7, where the execution times increase linearly when the simulated number of qubits increases with both one node and two nodes execution. For more detailed comparisons, we analyze the memory usage of the state vector, *QuEST*, and *dgQuEST* during distributed simulations of 31–35 qubits, as shown in Fig. 8. The results demonstrate that *dgQuEST* sharply reduces the memory requirements of simulating distributed on multiple nodes by nearly two folds due to the page-table based memory management. The memory used by *dgQuEST* is nearly equal to the required memory for the state vector. Therefore, the saved large amount of memory further contributes to simulating more qubits within the same memory capability.

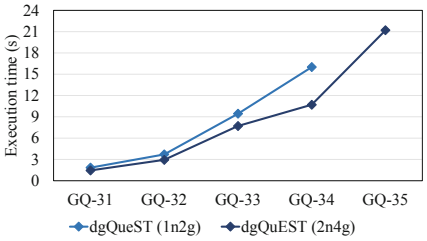

Fig. 7. Execution time and speedup of *dgQuEST* simulating GQ with different number of qubits.

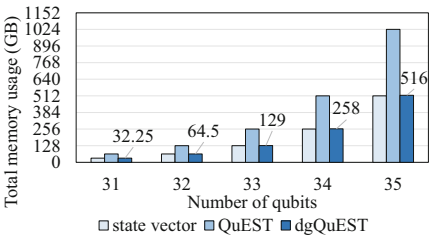

Fig. 8. Evaluated memory usage of state vector, *QuEST*, and *dgQuEST* simulating different numbers of qubits.

4.4 Strong Scalability

To demonstrate the strong scalability of *dgQuEST*, we compared the execution time of each circuit simulation on one node and two nodes. As shown in Fig. 9, most of the evaluated simulations exhibits poor or even negative scalability when it distributes to more nodes. Although the *gate aggregation* can minimize the data exchanges between nodes and *pipelined communication* enables overlapping of computations and communications, the data exchange still becomes a bottleneck when the quantum circuit is not deep enough (e.g., only 79 gates in *BV-32*). Therefore, our approaches to reduce the distributing memory overhead as well as support more qubits regardless of the limited GPU memory capacity can significantly reduce the required nodes for each quantum simulation and thus contribute to overall performance improvement.

4.5 Sensitivity Analysis on Page Size

The page size affects the performance of *dgQuEST*, as shown in Fig. 10. In general, a larger page size leads to a larger task block and potentially fewer data exchanges, while it makes the execution of each task longer and results in less pipeline utilization. In contrast, with a smaller page size, each execution takes less time and the data exchange pipeline is more utilized, but it will result in

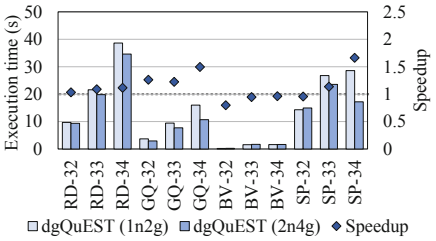

Fig. 9. Execution time and speedup of *dgQuEST* with one node (*1n2g*) and two nodes (*2n4g*).

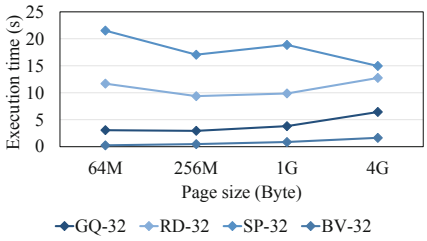

Fig. 10. Execution time of *dgQuEST* simulating 32-qubit circuits on two nodes with different page sizes.

more sub-circuits. Therefore, it is hard to give the best choice for all circuits as it depends on the specific circuit to tune page sizes for best performance. Therefore, we choose the page size with the best performance in our evaluation.

5 Related Work

Several optimizations have been proposed for the full-amplitude quantum simulation. qHipster [10] optimizes the performance with SIMD, thread allocation, communication, multi-threading, and cache blocking. JUQCS [11] provides the GPU accelerated implementation, which reduces communication overhead by relabeling global and local qubits. *Qibo* [3] utilizes TensorFlow and custom operators for its backend and supports acceleration with multiple GPUs. *HyQuas* [12] utilizes both shared memory and batch matrix-vector multiplication to accelerate the quantum simulation, and it implements muiti-GPU simulation and optimizes the communication between GPU. However, *HyQuas* fails to exploit the hybrid CPU-GPU memory hierarchies to support large-scale quantum circuit with more qubits on limited number of nodes. Whereas, *Qibo* cannot utilize distributed GPUs across multiple nodes. Different from above approaches, *dgQuEST* leverages the memory capacity of both CPUs and GPUs across distributed nodes for large-scale quantum circuit simulations with accelerated performance.

6 Conclusion

In this paper, we propose *dgQuEST*, a novel acceleration method that utilizes the CPU-GPU hybrid memory hierarchies for large-scale quantum circuit simulation. *dgQuEST* adopts efficient memory management and communication schemes for better memory access performance and reduced communication overhead across distributed GPU nodes. The experiment results demonstrate that compared to *QuEST*, *dgQuEST* can achieve an average speedup of 403× when simulating quantum circuit with 32 qubits. In addition, *dgQuEST* can easily scale to quantum circuit simulation with 35 qubits on two distributed GPU nodes, far beyond the state-of-the-art GPU implementation such as *HyQuas* can support.

Acknowledgements. This work was supported by National Key Research and Development Program of China (No. 2020YFB1506703), National Natural Science Foundation of China (No. 62072018), and State Key Laboratory of Software Development Environment (No. SKLSDE-2021ZX-06). Hailong Yang is the corresponding author.

References

1. Cai, Z.: Multi-exponential error extrapolation and combining error mitigation techniques for NISQ applications. NPJ Quant. Inf. **7**(1), 80 (2021). https://doi.org/10.1038/s41534-021-00404-3
2. Developers, C.: Cirq (2021),.https://doi.org/10.5281/zenodo.4750446. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors
3. Efthymiou, S., et al.: Qibo: a framework for quantum simulation with hardware acceleration (2020)
4. Endo, S., Benjamin, S.C., Li, Y.: Practical quantum error mitigation for near-future applications. Phys. Rev. X **8**, 031027 (2018). https://doi.org/10.1103/PhysRevX.8.031027
5. Gray, J.: quimb: A python package for quantum information and many-body calculations. J. Open Source Softw. **3**(29), 819 (2018). https://doi.org/10.21105/joss.00819
6. Huang, C., Szegedy, M., Zhang, F., Gao, X., Chen, J., Shi, Y.: Alibaba cloud quantum development platform: applications to quantum algorithm design (2019)
7. Jones, T., Brown, A., Bush, I., Benjamin, S.C.: Quest and high performance simulation of quantum computers. Sci. Rep. **9**(1), 10736 (2019). https://doi.org/10.1038/s41598-019-47174-9
8. McArdle, S., Jones, T., Endo, S., Li, Y., Benjamin, S.C., Yuan, X.: Variational ansatz-based quantum simulation of imaginary time evolution. NPJ Quant. Inf. **5**(1), 75 (2019). https://doi.org/10.1038/s41534-019-0187-2
9. McCaskey, A.J.: Quantum virtual machine (qvm). version 00 (2016). https://www.osti.gov/biblio/1339996
10. Smelyanskiy, M., Sawaya, N.P.D., Aspuru-Guzik, A.: qhipster: The quantum high performance software testing environment (2016)
11. Willsch, D., Willsch, M., Jin, F., Michielsen, K., Raedt, H.D.: Gpu-accelerated simulations of quantum annealing and the quantum approximate optimization algorithm (2021)
12. Zhang, C., Song, Z., Wang, H., Rong, K., Zhai, J.: Hyquas: hybrid partitioner based quantum circuit simulation system on GPU. In: Proceedings of the ACM International Conference on Supercomputing, ICS '21, pp. 443–454. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3447818.3460357