ScaleQsim: Highly Scalable Quantum Circuit Simulation Framework for Exascale HPC Systems

CHANGJONG KIM, Seoul National University of Science and Technology, Republic of Korea
EHAN SOHN, Seoul National University of Science and Technology, Republic of Korea
SEUNGHWAN KIM, Seoul National University of Science and Technology, Republic of Korea
ALEX SIM, Lawrence Berkeley National Laboratory, USA
KESHENG WU, Lawrence Berkeley National Laboratory, USA
HOUJUN TANG, Lawrence Berkeley National Laboratory, USA
YONGSEOK SON, Chung-Ang University, Republic of Korea
SUNGGON KIM*, Seoul National University of Science and Technology, Republic of Korea

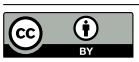
Large-scale quantum circuit simulation on high-performance computing (HPC) systems is crucial for developing and verifying quantum algorithms to overcome the limitations of current noisy quantum computers. However, existing simulators face scalability bottlenecks due to memory limits and communication overhead. Many state-of-the-art approaches rely on static circuit partitioning, which makes it difficult to address the exponential growth in problem size as the number of qubits increases. To overcome these challenges, we present `ScaleQsim`, a highly scalable quantum circuit simulation framework for large-scale HPC systems. `ScaleQsim` focuses on providing a unified representation of the full qubit state, yet provides scalability through efficient synchronization and communication. `ScaleQsim` adopts a novel full state vector partitioning strategy that evenly distributes the full state vector across multiple nodes and GPUs. This distributed structure enables efficient parallel gate execution without costly synchronization, and `ScaleQsim` applies adaptive kernel configuration, which adjusts execution parameters based on GPU resources and task granularity to enhance simulation efficiency. Our evaluation on a leadership-scale supercomputer with up to 512 GPUs demonstrates that `ScaleQsim` simulates quantum circuits with up to 42 qubits and outperforms leading SOTA simulators by up to 77.40× across various quantum circuits.

CCS Concepts: • **Computer systems organization → Quantum computing**; **Distributed architectures**; • **Computing methodologies → Simulation evaluation**.

Additional Key Words and Phrases: High-Performance Computing, Quantum Computing, Quantum Circuit Simulation

*Corresponding author.

Authors' Contact Information: Changjong Kim, Seoul National University of Science and Technology, Seoul, Republic of Korea, changjong5238@seoultech.ac.kr; Ehan Sohn, Seoul National University of Science and Technology, Seoul, Republic of Korea, ehanjsohn@seoultech.ac.kr; Seunghwan Kim, Seoul National University of Science and Technology, Seoul, Republic of Korea, rlatmdghkss@seoultech.ac.kr; Alex Sim, Lawrence Berkeley National Laboratory, Berkeley, USA, asim@lbl.gov; Kesheng Wu, Lawrence Berkeley National Laboratory, Berkeley, USA, asim@lbl.gov; Houjun Tang, Lawrence Berkeley National Laboratory, Berkeley, USA, htang4@lbl.gov; Yongseok Son, Chung-Ang University, Seoul, Republic of Korea, sysganda@cau.ac.kr; Sunggon Kim, Seoul National University of Science and Technology, Seoul, Republic of Korea, sunggonkim@seoultech.ac.kr.



ACM 2476-1249/2025/12-ART62
https://doi.org/10.1145/3771577

ACM Reference Format:
Changjong Kim, Ehan Sohn, Seunghwan Kim, Alex Sim, Kesheng Wu, Houjun Tang, Yongseok Son, and Sunggon Kim. 2025. ScaleQsim: Highly Scalable Quantum Circuit Simulation Framework for Exascale HPC Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 3, Article 62 (December 2025), 28 pages. https://doi.org/10.1145/3771577

1 Introduction

Quantum computers have been advancing rapidly, achieving quantum supremacy by outperforming classical computers on specific tasks. [3, 53]. Unlike classical computers that utilize binary bits defined by 0 and 1, quantum computers operate with qubits. Each qubit represents a quantum state, which is mathematically expressed as a vector called a state vector that describes its probability amplitudes (e.g., $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex amplitudes). Qubits utilize the fundamental principles of superposition and entanglement to adopt a different computational approach from existing methods. Superposition allows qubits to represent both 0 and 1 simultaneously, enabling faster parallel computation [55, 72, 87]. Entanglement enables unique interactions between qubits, allowing efficient information transmission and processing [31, 56]. These properties offer potential advantages in areas such as quantum cryptography [8, 9, 11, 13, 65], quantum simulations [19, 23, 29], and quantum machine learning [12, 17, 45, 69, 76], beyond the capabilities of classical computers.

Despite achieving supremacy in certain limited tasks, quantum computers are still a long way from being adopted as the main computing platform for general or even specific applications. This is because current quantum computers are classified as Noisy Intermediate-Scale Quantum (NISQ) devices that are susceptible to the high error rates inherent in quantum systems. This limitation makes them highly susceptible to accumulated noise and errors, preventing them from producing reliable and consistent output [47, 66, 74]. In addition, NISQ computers require extremely low temperatures to operate, which limits accessibility and increases operational costs [34, 54].

To overcome the limitations of NISQ computers, quantum circuit simulation on High-Performance Computing (HPC) systems has emerged. Simulating a quantum circuit and quantum state throughout the circuit using HPC systems produces accurate and deterministic results, free from the noise that affects physical quantum computers. This accurate simulation is critical for developing quantum hardware and algorithms, as well as supporting hybrid quantum-classical computing systems [27, 49, 50]. HPC systems are equipped with massive computational resources, including powerful CPUs, GPUs, and large memory pools, making them suitable for efficiently simulating complex quantum circuits [25, 35, 48, 52, 82]. The most widely adopted simulation approaches are amplitude sampling and full state vector simulation. Amplitude sampling approximates the amplitudes of the output qubits. By evaluating the entire quantum circuit while retaining only the amplitudes of the final output qubits it reduces memory usage at the cost of accuracy. In contrast, full state vector simulation explicitly simulates the amplitudes of all qubits, ensuring an exact simulation with no loss of accuracy. Accurately simulating the entire qubit state enables the development of correct quantum computers and algorithms [23, 39]. Our work focuses on full state vector simulation, as it is the most accurate and widely adopted quantum simulation method.

To support quantum circuit simulation in HPC, high-performance quantum simulation libraries such as Google Qsim, NVIDIA cuQuantum, Pennylane, and IBM Qiskit are widely used to maximize performance on HPC systems [4, 10, 21, 30]. These frameworks utilize GPUs, which offer higher performance than CPUs, and parallelize qubit simulations to efficiently simulate exponentially growing qubits [2, 42]. However, despite efforts, the existing frameworks show limited performance and scalability, which becomes especially critical as the complexity grows exponentially with the increasing number of qubits. Figure 1 shows the simulation time comparison of the proposed `ScaleQsim`, SOTA (*cusvaer* [4], *HyQuas* [89], *Atlas* [86]), and *Qsim* [30] framework, running the

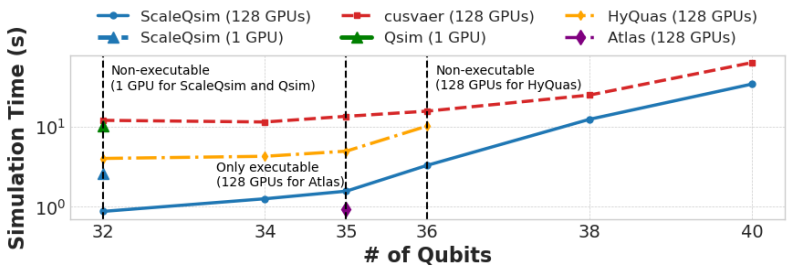


Fig. 1. Simulation time of `ScaleQsim` (Proposed), SOTA (*cusvaer*, *HyQuas*, *Atlas*), and *Qsim* (Existing), executed on 128 GPUs using the QFT circuits (log-scale).

Table 1. Categories and comparison with previous studies (SSP: State Space Partitioning, DTS: Dynamic Task Scheduling, EMN: Enable Multi Node).

Study	Method	SSP	DTS	EMN
Park *et al.* [62]	Full state vector			
Xu *et al.* [86]	Full state vector	✓		✓
Burgholzer *et al.* [15]	Hybrid			✓
Fu *et al.* [28]	Amplitude sampling			✓
Zhang *et al.* [89]	Full state vector	✓		✓
Lykov *et al.* [51]	Amplitude sampling			✓
Westrick *et al.* [83]	Hybrid			✓
Nguyen *et al.* [58]	Amplitude sampling			✓
Häner *et al.* [37]	Full state vector	✓		✓
`ScaleQsim`	Full state vector	✓	✓	✓

QFT circuits on an HPC system equipped with 128 GPUs. As shown in the figure, at 32 qubits, *Qsim* only supports single-GPU and cannot perform the simulation beyond 32 qubits due to the limited GPU memory. *cusvaer* shows limited scalability, as simulations with more qubits require multiple nodes and GPUs, which increases communication and data management overhead. In addition, both *HyQuas* and *Atlas* exhibit limited scalability due to their static designs. *HyQuas* shows lower performance compared to `ScaleQsim` and fails to execute beyond 36 qubits, even with an identical number of GPUs. The failure is due to its reliance on precompiled kernels, which cannot fit within an individual GPU as the problem size grows. In contrast, *Atlas* shows better performance than `ScaleQsim` at 35 qubits but fails to execute simulation at other qubit counts because its configuration is strictly bound to a specific combination of the number of qubits and GPUs (35 qubits and 128 GPUs). To summarize, addressing the poor performance (*cusvaer*, *Qsim*, *HyQuas*) and memory underutilization (*HyQuas*, *Atlas*) of SOTA simulators, our work (`ScaleQsim`) aims to provide fast, scalable performance for more qubits while fully utilizing available resources.

Many previous studies, as shown in Table 1, have focused on optimizing quantum circuit simulation from different architecture and algorithm perspectives. For example, recent studies [37, 62, 86, 89] based on full state vector simulation address memory scalability by introducing storage-based partitioning, hierarchical memory allocation, or multi-node execution to simulate circuits beyond the limits of GPU memory. Other studies mitigate the computational bottleneck by adopting fine-grained task scheduling, GPU-aware kernel tuning, or hierarchical execution frameworks that divide and conquer large circuits through distributed architectures [86, 89]. In addition, some studies adopt hybrid amplitude sampling and full state simulation to reduce simulation complexity by limiting the number of computational paths, particularly in sparse quantum

circuits [15, 83]. Other works use amplitude sampling-based approaches to improve scalability by approximating qubit states through structured tensor contractions, which can reduce both memory and computation requirements [28, 51, 58].

ScaleQsim distinguishes itself from previous studies by implementing a scalable full state vector simulation framework for a leadership-scale HPC system. Previous studies, such as *Atlas* [86] and *HyQuas* [89], divide quantum circuits into sub-circuits and rely on statically generated simulation plans with fixed gate-to-qubit mappings, kernel configurations, and hardware-dependent schedules, limiting scalability and adaptability. In contrast, ScaleQsim performs dynamic execution planning during runtime. It evenly distributes the full state vector across multiple nodes and GPUs, computes the position of the affected state vector in runtime, schedules gate operations, and configures kernel parameters based on the workload and available resources. When accessing a distributed specific state vector, it utilizes distributed metadata for remote data access and a scalable communication protocol (i.e., MPI and CUDA P2P). Our design prevents workload skewness and eliminates the need for precompiled kernels or precomputed simulation plans, enabling scalable performance across diverse hardware configurations and qubits.

In this paper, we present ScaleQsim, a high-performance quantum circuit simulator designed for scalability. ScaleQsim adopts a full state vector simulation and integrates three key techniques to achieve performance and scalability. The goal of ScaleQsim is to 1) provide a unified representation of the full state vector with efficient distribution, 2) support inter-GPU and inter-node communication for distributing the full state vector without synchronization overhead, and 3) reduce the communication overhead arising from the distribution. To achieve these goals, ScaleQsim 1) performs a two-phase partitioning, dividing the state vector of all qubits across nodes and then among the GPUs within each node, 2) precomputes the mapping between local GPU memory and the full state vector across the nodes and 3) distributes the precomputed mapping required for state transitions and utilizes scalable interconnect protocols that are widely adopted in HPC. Our evaluation on a leadership-scale production HPC system shows that ScaleQsim outperforms existing SOTA frameworks by 1.11× to 77.40× and supports simulations with a higher number of qubits using identical resources. We open-source the code of ScaleQsim in the following link: https://github.com/ScaleQsim/ScaleQsim.git

2 Background

2.1 Quantum Circuit Simulation

Several approaches have been designed to simulate quantum circuits on classical hardware, each offering different trade-offs between scalability, accuracy, and computational complexity [1, 57, 62, 81, 86, 89, 90]. These simulation approaches are generally categorized into two types: amplitude sampling and full state vector simulation.

Amplitude sampling simulation. Amplitude sampling simulation represents entangled qubits as tensors and simulates their interactions using tensor contraction. This process contracts multiple input qubit states into a single output qubit amplitude. While the simulation performs well for circuits with low or structured entanglement, its performance decreases significantly for those with high or complex entanglement. A key limitation is that it only simulates the final output amplitude, causing most of the information between the input and output qubits to be lost [28, 60, 64].

Full state vector simulation. Full state vector simulation expresses the full state vector as a complex vector of size 2^n, where n is the number of qubits. It captures all possible superpositions and entanglements, providing high simulation accuracy without approximation [62, 86, 89]. As the most widely popular approach, our work focuses on full state vector simulation.

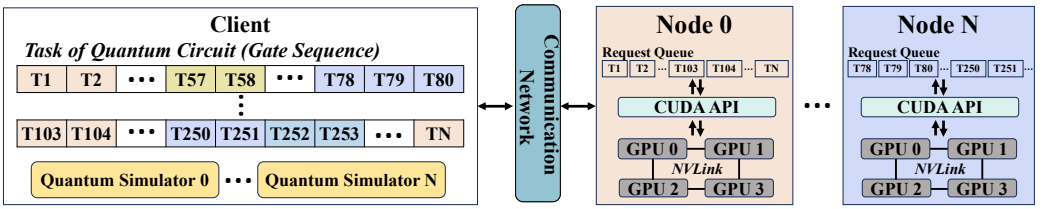


Fig. 2. Distributed architecture in quantum circuit simulation.

Since full state vector simulation is the most accurate and general-purpose approach, it is crucial for validating quantum algorithms, verifying hardware behavior, and benchmarking performance. Preserving the complete quantum state ensures a reliable baseline for correctness and precision. However, it suffers from exponential increases in computational and memory requirements as the number of qubits increases. This is because each additional qubit doubles the state vector size. The memory complexity is $O(2^n)$, with each complex amplitude requiring 16 bytes (e.g., 8 bytes each for the real and imaginary parts). For example, a 40-qubit state requires approximately 17 TB of memory, while a 50-qubit state exceeds 16 PB, far beyond the capability of most classical systems. Additionally, gate operations become equally high computational cost, as they involve matrix-vector multiplications over the full state vector. To address this, memory pooling, distributed execution, and kernel-level optimization are essential. `ScaleQsim` is designed to meet these challenges with a scalable and accurate full state vector simulation on leadership-scale HPC systems.

2.2 Distributed Architecture in Quantum Circuit Simulation

To effectively handle quantum simulations with a large number of qubits, it is essential to use a distributed architecture (scale-out) [2, 5, 24, 46, 61]. Early quantum simulations were limited by the hardware resources of a single server. However, as the number of qubits increases, the size of the state vector grows exponentially, leading to scalability issues with memory and computation resources. This makes it challenging to achieve the performance needed to process large numbers of qubits. In terms of computation devices, efforts have been made to simulate quantum circuits using CPUs (e.g., Intel quantum simulator [35]). However, GPUs offer significantly higher performance due to their superior parallelism. Additionally, multiple GPUs within a node can be directly connected using NVLink [59], reducing communication bottlenecks. In this paper, we focus on distributed quantum simulation using GPUs as accelerators, aiming to achieve both memory scalability and computational efficiency, similar to existing parallelization efforts [2, 61, 77].

Figure 2 shows a distributed architecture for quantum simulation. As depicted, it consists of two components: the client and multiple simulation nodes. The central client first decomposes the entire quantum circuit into a discrete sequence of gate operations called tasks. Then, it partitions and distributes the tasks to multiple nodes. This process divides the circuit into manageable subsets for efficient processing. For example, tasks (e.g., T1-2,...T103-104...TN) are assigned to Node 0, while tasks (e.g., T78-80,...T250-251...) are assigned to Node N. Each node maintains a task queue and sequentially processes the received subsets. Since each node consists of multiple GPUs, the tasks are further distributed and processed across available GPUs. This architecture is commonly adopted by existing quantum simulators, including *cusvaer* [4], to scale across multiple GPUs and nodes. However, *cusvaer* suffers from coordination overheads as simulation complexity increases. This is because each GPU's memory space is managed independently, and GPUs cannot interact directly with one another. Consequently, in *cusvaer*, when communication between GPUs or nodes is required, such as at the end of a gate execution, a global synchronization of all GPUs is necessary.

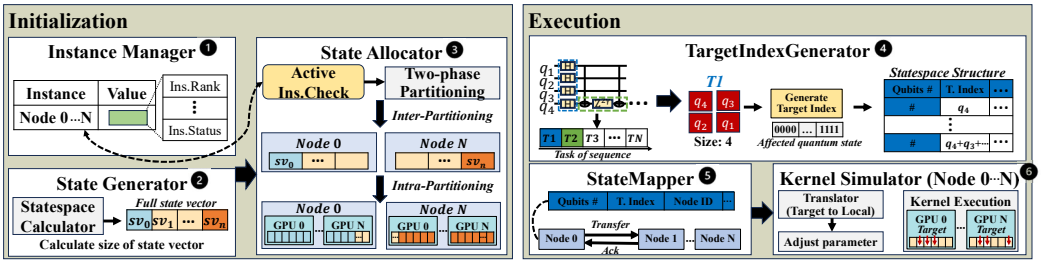


Fig. 3. Overall procedure of ScaleQsim.

To address these problems, SOTA simulators such as *HyQuas* [89] and *Atlas* [86] use more strict yet efficient scheduling and partitioning. Both simulators implement a two-tier partitioning model that divides a large global quantum circuit into local sub-circuits to reduce cross-node communication. For example, *HyQuas* prepares multiple precompiled kernels that are optimized for smaller circuits. It then divides a large circuit into multiple smaller sub-circuits and simulates them independently across multiple GPUs. However, because the precompiled kernels use fixed configurations, it can be difficult to adapt to different problem sizes and hardware, leading to performance variance based on the shape of each sub-circuit. In addition, this approach can suffer from workload skewness when the sub-circuits are imbalanced and may fail to simulate circuits that exceed the memory constraints of a single GPU. Similarly, *Atlas* statically maps each gate from a large quantum circuit into a fixed local qubit and generates a predefined simulation plan at compile time. This approach not only requires a fully specified gate-to-qubit mapping in advance but also suffers from similar workload imbalance and memory constraints. Although partitioning and divide-and-conquer approaches can be suitable for small, restricted environments, extending the full state vector to leadership-scale HPC systems requires a clear scalability model.

3 ScaleQsim Design

In this section, we present the design of ScaleQsim, a scalable quantum circuit simulator for HPC systems. ScaleQsim does not partition a large quantum circuit, but instead allocates the full state vector across multiple nodes and GPUs. This allows for dynamic circuit simulation without precompiled kernels or predefined simulation plans. To overcome the communication overhead arising from this distributed state space, it distributes necessary metadata for accessing remote state vector space and utilizes a scalable interface protocol for inter- and intra-node communication.

3.1 Overall Procedure

Figure 3 shows the overall procedure of ScaleQsim. ScaleQsim provides two main phases to support distributed quantum circuit simulation: *Initialization* and *Execution* phase.

Initialization. When quantum circuit simulation starts, *Instance Manager* checks the available nodes and GPUs in the distributed architecture. Then, it reads the metadata such as node rank, number of GPUs, memory size, and active or inactive status for each node (❶). Once the metadata collection is completed, *State Generator* calculates the total required memory size (i.e., the size of the full state vector) based on the number of input qubits (e.g., $n = 36$, which corresponds to 68.7 billion amplitudes and a total memory size of $2^{36} \times 8$ bytes, resulting in approximately 512 GB). This calculation ensures that 1) the total size of the full state vector does not exceed the total

available GPU memory capacity and 2) the full state vector can be evenly partitioned across all active multiple nodes and GPUs (❷).

After completing the calculation of the full state vector size, *State Allocator* checks the available node instances and performs two-phase partitioning to allocate the full state vector across the distributed architectures. This is to allocate memory regions in advance before the kernel execution and minimize synchronization between the distributed state vectors. First, the full state vector is divided among the nodes (Inter-partitioning). Then, within each node, the assigned subset of the state vector is further divided among the available GPUs (Intra-partitioning) (❸).

Execution. After *Initialization* phase is completed, `ScaleQsim` enters *Execution* phase. First, `ScaleQsim` constructs a sequence of executable tasks, where each task contains one or more gates along with the affected qubits. For each task, *TargetIndexGenerator* uses the included qubits to generate all possible binary combinations affected by the gate operation. These combinations are then converted into indices within the full state vector, which are called `Target Indices` and are used to create *Statespace Structure* (❹). *Statespace Structure* contains the mapping information from a global index in the full state vector to a distributed location, such as a node ID, GPU ID, and offset. This structure is then broadcast to all nodes (❺). By precomputing all possible combinations from every gate operation and their metadata, and by replicating the entire metadata across the nodes, `ScaleQsim` avoids synchronization overhead and the need to iterate over the full state vector to determine which target state locations are affected by each task. Finally, each node kernel simulator performs computation based on the received `Target Index` and metadata. Each `Target Index` is translated into a local index according to the subset of the state vector in each node and passed to the GPU, where the gate operation is applied in parallel. Each GPU accesses only the assigned subset of the state vector, enabling independent execution and minimizing communication overhead (❻).

3.2 Two-phase State Space Partitioning

Before launching the simulation kernel, `ScaleQsim` first allocates the distributed and limited memory within each node and GPU to efficiently utilize the distributed computing resources. Figure 4 shows the structure of the partitioned full state vector space across nodes and GPUs. As shown in the figure, the full state vector is represented as a linear array, ranging from SV0 to SVN. These SVx are contiguous factors of the full state vector, each consisting of multiple amplitudes. For example, as shown in the figure, SV0 contains a sequence of amplitude vectors such as Amp0 (e.g., a + bi), Amp1, and so on. Each amplitude is a complex value corresponding to a specific quantum basis state, which is expressed as a binary string such as $|000\rangle$ or $|101\rangle$, where each bit in the string represents the value of a qubit in that state. To allocate and distribute this massive full state vector, `ScaleQsim` adopts a two-phase partitioning strategy: node-level partitioning (Inter-partitioning) and GPU-level partitioning (Intra-partitioning).

Inter-partitioning. First, `ScaleQsim` performs inter-partitioning, which allocates the full state vector evenly across the N nodes. As shown in the figure, the full state vector is divided and distributed across multiple nodes. To evenly distribute the full state vector, `ScaleQsim` defines two key parameters: N and X. N denotes the total number of compute nodes used in the simulation. X represents the total number of amplitudes in the full state vector. For an n-qubit quantum system, there are 2^n basis states, so the full state vector contains $X = 2^n$ amplitudes. During `ScaleQsim` simulation, each node is assigned a contiguous subset of size X/N, where X is the total number of amplitudes. The assignment is performed sequentially based on amplitude indices. For example, Node 0 is assigned amplitudes in the range $[0, X/N - 1]$. Node 1 receives the next range, $[X/N, 2X/N - 1]$, and so on. Node $N - 1$ is assigned the final subset, from $[(N - 1)X/N]$ to $X - 1$.

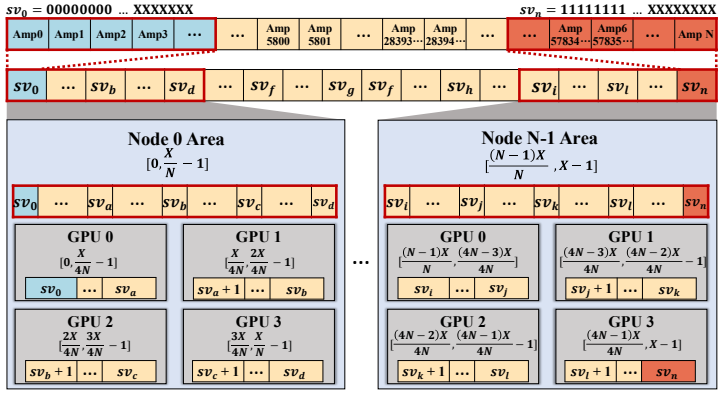


Fig. 4. The structure of the partitioned full state vector across multiple nodes and GPUs.

Intra-partitioning. After node-level partitioning, `ScaleQsim` performs intra-partitioning, which allocates the divided subset of the node's state vector across multiple GPUs within the node. Existing full state vector simulation approaches require all operations to access a single large memory, which is often allocated in a shared CPU DRAM [79] or a single GPU vRAM [4]. However, this quickly becomes a major performance bottleneck due to memory contention and limited bandwidth. In contrast, `ScaleQsim` reduces this problem through direct allocation, where each GPU independently manages a subset of the state vector partition from inter-partitioning. As shown in the figure, each node holds a subset of the full state vector after inter-partitioning. This subset is further split into four contiguous subsets (subrange), each assigned to one of the available GPUs within the node. Starting from Node 0, GPU 0 is assigned the first quarter of the subset (e.g., subrange 0 to $X/(4N) - 1$). GPU 1 receives the next quarter, followed by GPU 2, and GPU 3 takes the final quarter, ending at subrange $X/N - 1$. Intra-partitioning continues sequentially across all nodes. For example, in Node $N - 1$, GPU 0 begins at subrange $(N - 1)X/N$, and the following GPUs are assigned the next quarters in order. GPU 3 ends at the final index $X - 1$. This layout ensures that the full state vector is divided into disjoint and contiguous subsets without any overlap between GPUs. Each subrange is aligned in a globally consistent order and assigned to each GPU.

3.3 Task-based Qubit State Management

Task decomposition. After the full state vectors are allocated in the memory of distributed nodes and GPUs, `ScaleQsim` performs `Target Index` generation. Figure 5 shows the overall procedure of `Target Index` generation. As shown on the left of the figure, when a quantum circuit for an n-qubit simulation is provided as input, it consists of qubit variables (e.g., `q0-q35`) and quantum gates (e.g., CNOT) that operate on subsets of these qubits. `ScaleQsim` utilizes the gate partitioning mechanism from *Qsim* [30], which divides consecutive gates into smaller sub-circuits. Thus, each sub-circuit is treated as an independent task (e.g., `T1`, `T2`) in `ScaleQsim`, with each task containing the qubit operands required for gate execution (e.g., `q1`, `q2`). For example, Task `T1` operates on qubits 34 and 35, while Task `T2` operates on qubits 2, 17, 34, and 35.

Generate `Target Index`. As the full state vector is distributed among multiple nodes and GPUs, not in a single large memory space, `ScaleQsim` should identify the part of the state vector affected by each gate operation. To do this, `ScaleQsim` defines these locations as `Target Index`, which represent specific positions in the full state vector that should be updated by a given task. For

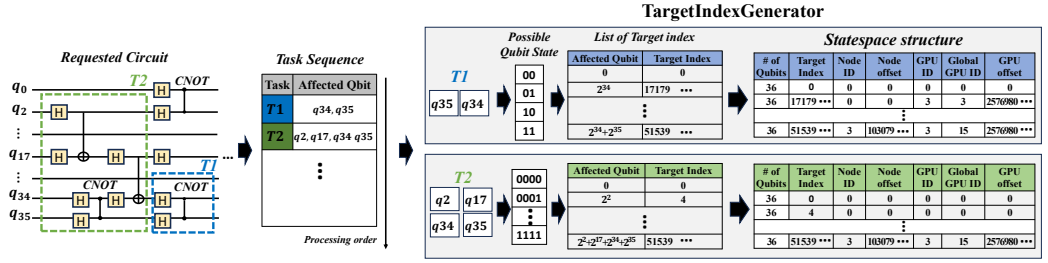


Fig. 5. The procedure of `Target Index` generation performed by *TargetIndexGenerator*.

each task, `ScaleQsim` generates `Target Index` for all possible outcomes from the gate operation included in each task. This is done for two reasons: 1) By generating `Target Index` for all possible outcomes in advance, it reduces additional coordination and communication overhead during the simulation, as `Target Index` specifies where the update will occur. 2) At the time of execution, it is impossible to predict the result of the task. Therefore, we generate `Target Index` for all possibilities. Note that even though `ScaleQsim` precomputes and tracks all possible `Target Indices` in advance, this approach introduces minimal performance and memory overhead, with 0.58 seconds and 54.81 KB for 36 qubits. To minimize communication overhead, `ScaleQsim` performs this operation in Node 0.

As shown on the right of Figure 5, *TargetIndexGenerator* reads the input qubits from each task, generates all possible binary combinations of the selected qubits, and converts them into corresponding `Target Indices` within the state vector. To do this, *TargetIndexGenerator* employs a bitmask-based enumeration method in which only the state vector position of the selected qubit is modified. This is to efficiently generate the full set of indices while leaving unaffected state vectors unchanged, reducing the potential communication overhead. For example, `T1` is a task that operates on qubits 34 and 35. When performing the task, the possible 2-bit combinations are `00`, `01`, `10`, and `11`, where the left bit corresponds to qubit 35, and the right bit corresponds to qubit 34. If the original state of the qubit is `00` and the result of the gate simulation is `01`, the modification of the state vector occurs at the position of qubit 34 (i.e., 2^{34} = 17,179,869,184). Thus, `Target Index` is set to 17,179,869,184. Similarly, `Target Index` for other possible qubit states `10`, and `11` are set to 2^{35} and $2^{35} + 2^{34}$. These combinations are generated by varying only the selected bit positions, while the remaining 34 out of 36 bits remain fixed. Once all `Target Indices` generated for each task are stored in a structured list called *List of Target Indices*, each entry in this list consists of three fields: a bit pattern, a bitmask, and the computed `Target Index`. `Bit` field represents the binary combination assigned to the selected qubits. `Bitmask` specifies which positions in the state vector correspond to each bit in the pattern. `Target Index` indicates the exact location in the state vector where the gate operation should be applied, which is computed by mapping the bit pattern onto the positions specified by the bitmask using bitwise operations.

Manage statespace structure. While `Target Index` represents a single logical location, the full state vector is physically distributed across multiple nodes and GPUs. Consequently, accessing this `Target Index` requires communication between devices, which can cause significant synchronization and communication overhead. While previous works [86, 89] aim to overcome this by optimizing gate partitioning and executing smaller gates within a single GPU using precompiled kernels, this approach can suffer from limited flexibility, workload skewness, and execution failures due to memory overflow. Instead of partitioning gates, `ScaleQsim` addresses this limitation

by generating metadata (called *Statespace Structure*) for inter-GPU and inter-node access and by utilizing a scalable communication protocol.

Statespace Structure stores essential mapping information, including `Target Indices` that will be affected by future gate operations and their corresponding physical locations (i.e., node, GPU, and local memory address). This information is then replicated to all nodes, allowing each to use a local copy during the simulation. This information is critical for two reasons. First, it allows local gate operations to happen entirely within a single GPU, providing memory locations based on the memory of each GPU rather than the full state vector. Second, it enables direct access to remote GPU and node memory when necessary, eliminating the calculation and communication overhead required to find a target location.

Thus, as shown on the right of Figure 5, once the list of `Target Indices` is generated by *TargetIndexGenerator* on Node 0, the corresponding metadata is constructed and stored in *Statespace Structure*, which is then broadcast to all nodes. Each entry in *Statespace Structure* contains detailed information required for gate execution, including node and GPU placement and memory offsets, as described below:

- **# of Qubits**: Number of qubits, which determines the state vector size.
- **`Target Index`**: Index in the full 2^n-dimensional state vector affected by the gate operation.
- **Node ID**: Identifier (e.g., MPI rank) of the compute node responsible for the given `Target Index`.
- **Node Offset**: Start offset (in the full state vector) of the state vector range assigned to the node.
- **GPU ID**: Identifier of the GPU (within the node) that manages the given `Target Index`.
- **Global GPU ID**: Globally unique GPU identifier across all nodes, assigned in node order (e.g., GPU 0 of Node 0 – Global GPU ID 0, GPU 0 of Node 1 – Global GPU ID 4).
- **GPU Offset**: Start offset of the GPU's assigned range within the node-level state vector partition.

3.4 Two-phase Mapping and Kernel Execution

Although `Target Index` is a logical representation of a global position, `ScaleQsim` distributes the state vector across multiple nodes and GPUs. Therefore, `Target Index` must first be mapped to the specific node and GPU where it physically resides. To achieve this, `ScaleQsim` references *Statespace Structure* to translate each `Target Index` into a local index, which corresponds to a memory location based on the memory boundaries of each GPU. Then, with the translated local index, `ScaleQsim` performs the simulation independently and in parallel on each GPU through kernel execution.

Figure 6 shows the process of mapping `Target Index` to the local index, which requires a subset of the state vector space distributed across multiple GPUs within a node. As shown in the figure, `ScaleQsim` consists of two main components: *StateMapper* and *Kernel Simulator*.

Mapping target index to local index. To start kernel execution, `ScaleQsim` converts each `Target Index` defined over the full state vector into a local index within the memory space of its assigned GPU. This mapping is performed hierarchically across nodes and GPUs. As shown in Figure 6, all nodes receive *Statespace Structure* generated by *TargetIndexGenerator*, which includes metadata such as `Target Indices`, node IDs, and partition offsets for all possible resulting qubit states. The generated metadata is broadcast from Node 0 to all nodes using MPI communication. Each node (e.g., Node 1 to N) receives the metadata and sends an acknowledgment to Node 0. After all acknowledgments are received, an MPI synchronization is performed.

After MPI synchronization is completed, first, *StateMapper* assigns each `Target Index` to the corresponding compute node using a static range-based partitioning rule. Each node X is responsible for indices in the range $[X \cdot 2^k, (X+1) \cdot 2^k)$, where k denotes the number of qubits represented by the node. Each node independently filters and selects only `Target Indices` within its assigned

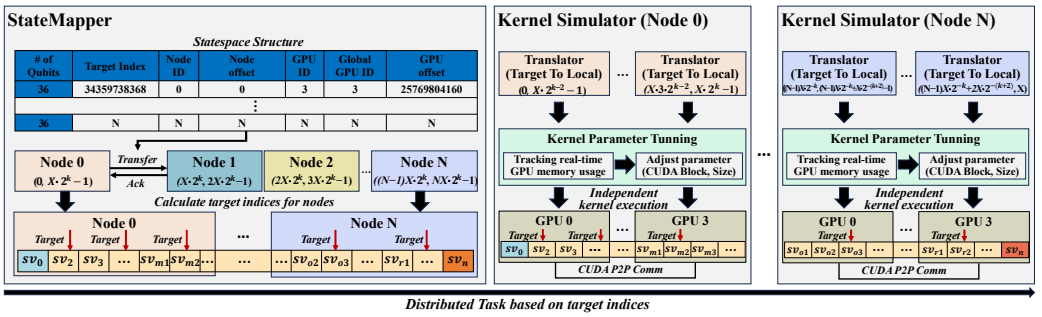


Fig. 6. The procedure for translating a `Target Index` into a local index and executing quantum circuit simulations by *StateMapper* and *Kernel Simulator*.

range. Then, within each node, *Translator* in *Kernel Simulator* maps the filtered `Target Indices` to local indices in each GPU. For example, if Node 0 - GPU 1 is responsible for the range, *Translator* identifies all `Target Indices` that fall into this range and subtracts the GPU's base index (2^{k-2}) to convert them into local indices starting from zero within GPU 1's memory space.

Kernel execution. Once all `Target Indices` have been translated into local indices within the GPU, `ScaleQsim` starts the kernel execution, where gate operations are applied to the state vector. Before launching the kernel, *Kernel Parameter Tuning* is performed to determine appropriate execution parameters for each GPU. This tuning process considers both the size of the local index set and the current GPU memory usage. More details of this tuning mechanism are provided in Section 3.5. Once the kernel parameters are configured, each GPU launches a CUDA kernel based on its assigned set of local indices. Each CUDA thread processes one or more indices and updates the corresponding amplitudes in the state vector. Since these indices have already been translated according to the GPU's memory layout, they can be directly accessed during kernel execution without further transformation.

However, when a gate operation spans multiple GPUs or nodes, communication is required to update the remote state vector. This problem is unique to `ScaleQsim`, as existing frameworks [30, 86, 89] partition a circuit into sub-circuits, process them independently, and synchronize only after all processing is complete. To overcome this, `ScaleQsim` avoids unnecessary synchronization by leveraging the precomputed *Statespace Structure*, which is computed by Node 0 and broadcast to all nodes. In this structure, the location of each affected `Target Index` (i.e., node offset and GPU offset) is already determined per task. This enables `ScaleQsim` to detect whether a task spans multiple nodes or GPUs and apply predefined communication without requiring synchronization during execution. A fixed mapping strategy deterministically assigns each `Target Index` to a specific node and GPU before the kernel executes. Based on this assignment, each node then computes only the amplitudes corresponding to its assigned `Target Indices`, which ensures exclusive access and prevents conflicts.

Our kernel execution strategy applies consistently across both inter- and intra-node cases. Specifically, when a task spans multiple nodes, each node is responsible for processing only its assigned portion of the state vector. For example, if a 2-qubit gate targets amplitudes $[2^{30}, 2^{30} + 8]$ spanning Node 0 and Node 1, each node processes only its assigned part. When the task spans multiple GPUs within a single node, `ScaleQsim` leverages CUDA Peer-to-Peer (P2P) memory access to enable direct reads across GPUs. Since CUDA P2P access does not enforce execution ordering, `ScaleQsim` ensures correctness by maintaining a globally defined task order and assigning exclusive update responsibility to each GPU. For example, if GPU 0 holds $[0, 2^{28}]$ and GPU 1 holds $[2^{28}, 2^{29}]$,

Algorithm 1 Adaptive kernel parameter adjustment.

Function parameterConf(*num_qubits*, *sv*, *gpuID*)
$G \leftarrow sv$.size(), $k \leftarrow 5 + G$, $n \leftarrow (num\_qubits > k)\ ?\ num\_qubits - k : 0$
QUBIT_THRESHOLD $\leftarrow 40$, GATE_THRESHOLD $\leftarrow 5$, $size \leftarrow 2^n$, $threads \leftarrow$ # of threads, $max\_blocks \leftarrow 2^{30}$
▷ ***Enable memory-safe mode under high qubit pressure***
if $num\_qubits \geq$ QUBIT_THRESHOLD **or** $G \geq$ GATE_THRESHOLD **then** $safe\_mode \leftarrow$ true
end if
▷ ***Block size calculation***
if $safe\_mode$ = false **then** $blocks \leftarrow \min(max\_blocks, \max(1, size/threads))$
else
  $(free, total) \leftarrow$ cudaMemGetInfo($gpuID$)
  $mem\_per\_thread \leftarrow 2 \times$ sizeof(fp_type(float))
  $mem\_per\_block \leftarrow threads \times mem\_per\_thread$
  $mem\_blocks \leftarrow free/mem\_per\_block$
  $size\_blocks \leftarrow \max(1, size/threads)$
  $blocks \leftarrow \min(max\_blocks, mem\_blocks, size\_blocks)$
end if
LaunchKernel<<< *blocks*, *threads* >>> (...)

and the gate spans both, CUDA P2P allows GPU 0 to read from GPU 1's memory without host involvement. This is achievable because applying a quantum gate is equivalent to a matrix-vector multiplication. Each element of the resulting state vector is calculated from a linear combination of the input elements and does not depend on other output amplitudes being calculated simultaneously. Thus, this process avoids race conditions or overlapping writes during a single gate application, thereby eliminating the need for synchronization within the gate.

By structurally preventing concurrent updates to identical amplitudes, `ScaleQsim` eliminates the need for fine-grained locking or synchronization during kernel execution. Each node proceeds independently with its assigned portion of the task, and MPI coordination is invoked only after kernel execution of each task to preserve global task ordering.

3.5 Adaptive Kernel Parameter Adjustment

When executing the simulation kernel, a fixed CUDA block size can hinder performance by mismatching GPU resource allocation with the actual workload. When applying small gates that affect a limited number of qubits, using a large number of blocks leads to underutilized threads and increased scheduling overhead. Conversely, for large gates, a fixed block size can cause memory pressure or out-of-memory (OOM) errors during execution. To address these issues, `ScaleQsim` dynamically adjusts the number of CUDA blocks based on the gate size and the portion of the state vector assigned to each GPU. This adaptive tuning maximizes parallelism while ensuring memory safety across various gate sizes.

Algorithm 1 describes the adaptive kernel parameter tuning process used by `ScaleQsim` to execute gate operations on each GPU efficiently. The input parameters include: `num_qubits` (the total number of qubits), `sv` (indices of qubits affected by the current task), and `gpuID` (target GPU identifier). From `sv`, the function derives G = `sv.size()`, which represents the number of affected qubits. This G value is a crucial parameter, as it determines the total number of affected amplitudes in the full state vector. Specifically, operations on G qubits influence a set of 2^G `Target Indices`, which are used to identify the affected amplitudes that must be updated during kernel execution.

Based on G, `ScaleQsim` sets an internal threshold $k = 5 + G$, following *Qsim*'s policy [30]. If `num_qubits` exceeds this, it sets n = `num_qubits` $- k$ and divides the work into 2^n tasks. Each task updates amplitudes defined by `Target Indices`. To execute these tasks on the GPU, `ScaleQsim` uses the CUDA model, organizing execution using threads and blocks. Each block is set to a fixed number of threads, typically 32 or 64, following the kernel parameter configuration of

Qsim [30]. This configuration corresponds to one or two warps. A warp is the fundamental unit of hardware granularity in CUDA, where one warp includes 32 threads. Thus, this approach allows for alignment with the warp level scheduling granularity in CUDA, thereby improving kernel execution efficiency. `ScaleQsim` also applies static thresholds, namely `QUBIT_THRESHOLD` and `GATE_THRESHOLD`, to decide whether to enable `safe_mode` **(Line: 1–3)**. Once enabled, `safe_mode` dynamically checks the available GPU memory (e.g., 80 GB for A100) via `cudaMemGetInfo()` and constrains the number of blocks accordingly to prevent memory pressure during kernel execution.

If the requested task size fits within the available GPU memory, `safe_mode` remains disabled and the original kernel parameter is used. However, a complex circuit with a large number of qubits or intricate gate interactions can cause the number of affected state vector indices to exceed the available GPU memory. In such cases, `ScaleQsim` activates `safe_mode` **(Line: 4–5)**. This dynamically constrains the number of blocks launched on each GPU to prevent memory overuse from excessive parallelism. It first invokes `cudaMemGetInfo()` to check available GPU memory and calculates a safe upper bound for the block count (e.g., 2^{27} blocks for an 80 GB GPU). `ScaleQsim` then compares this memory-constrained limit to the total number of blocks required by the task and launches the kernel with the smaller of the two values to ensure safe execution (**Line: 7–13**).

For example, each thread requires 8 bytes (e.g., for single precision) and 64 threads are assigned per block, each block consumes 512 Bytes. In a task involving 2^{32} blocks (e.g., a 42-qubit with $k = 10$), the total memory demand reaches 2 TB, which is beyond the memory capacity of a single GPU. Assuming that if 80 GB is available on each GPU (e.g., A100 80GB), at most $80 \times 2^{30}/2^9 = 2^{27}$ blocks (134 million) can be safely launched. Thus, `ScaleQsim` only executes $80 \times 2^{30}/2^9 = 2^{27}$ blocks (GPU memory constraints), while the remaining computation of the identical task is handled by subsequent kernel executions. Since each GPU operates exclusively on its statically partitioned region of the state vector, no inter-GPU task coordination is required during this process. This allows each GPU to launch only the number of blocks it can execute efficiently, avoiding memory pressure and task scheduling overhead that may arise from excessive parallelism.

3.6 `ScaleQsim` Implementation

We implemented `ScaleQsim` by extending the Google *Qsim* framework [30]. While *Qsim* is open-source for single GPU simulation, it utilizes cuQuantum for multi-node/GPU distribution, which is closed-source. To overcome this, we used the identical simulation algorithm for each GPU but redesigned the distribution and parallelization for production HPC systems. Our implementation modifies approximately 600 lines of code across three core modules: `simulator_cuda.h`, `vectorspace_cuda.h`, and `simulator_cuda_kernel.h`. 1) We designed a two-phase partitioning scheme to evenly divide the full state vector across nodes and GPUs. 2) We added a new *TargetIndexGenerator* and *StateMapper* module to track and map `Target Index` to the local index within each GPU. In addition, we support correct coordination between nodes and GPUs using MPI and CUDA P2P. 3) We extended the CUDA kernel interface to support adaptive kernel configuration based on resource availability and workload size. We opensource the code of `ScaleQsim` in the following link: https://github.com/ScaleQsim/ScaleQsim.git

4 Evaluation

4.1 Evaluation Setup

We evaluate `ScaleQsim` on a leadership-scale HPC system ranked within the top 20 on the TOP 500 supercomputers list. The target HPC system consists of 3,072 CPU nodes and 1,792 GPU-accelerated nodes. For the evaluation, only the GPU nodes are used. Each GPU node has 256GB of DDR4 DRAM as main memory and a single AMD EPYC 7764 (Milan) CPU, along with four NVIDIA

Table 2. Our benchmark circuits and their size (number of gates).

Circuit	Description	Complexity	Number of qubits										
			32	33	34	35	36	37	38	39	40	41	42
qft	Quantum Fourier Transform	High	522	551	580	609	638	667	696	725	754	783	812
qv	Quantum Volume	Medium	384	396	408	420	432	444	456	468	480	492	504
vqc	Variational Quantum Classifier	High	1044	1068	1092	1116	1140	1164	1188	1212	1236	1260	1284
qsvm	Quantum Support Vector Machine	Medium	63	65	67	69	71	73	75	77	79	81	83
random	Random Parameters	Medium	480	490	510	520	540	550	570	580	600	610	630
ghz	Ghz State	Low	32	33	34	35	36	37	38	39	40	41	42
vqe	Variational Quantum Eigensolver	High	3800	3920	4040	4160	4280	4400	4520	4640	4760	4880	5000

A100 (Ampere) GPUs connected via PCIe 4.0. Each GPU features 80GB of HBM2e memory with 2,039 GB/s bandwidth. The four GPUs are interconnected through third-generation NVLink, each providing 25GB/s per direction.

We evaluate `ScaleQsim` primarily using the Quantum Fourier Transform (qft) circuit, which is known for its dense entanglement and high gate complexity. In addition to qft, we evaluate six other representative quantum circuits. Note that although the Variational Quantum Eigensolver (vqe) circuit is typically a dynamic algorithm with iterative parameter updates [32], its circuit is treated as static in our evaluation. The rotational gate parameters are generated once with a fixed random seed and remain unchanged throughout the simulation to measure the simulator's performance on deep and complex workloads. The total of seven circuit types used is summarized in Table 2. We compare `ScaleQsim` with the original *Qsim* [30] (only supports single GPU natively), *cusvaer* [4] (IBM Qiskit [21] using the cuStateVec backend, which is a distributed multi-node/GPU simulator from NVIDIA's cuQuantum), *HyQuas* [89], and *Atlas* [86].

4.2 Performance in Single-Node with SOTA

Figure 7 shows the performance comparison of *Qsim* (1 GPU), `ScaleQsim` (1, 4 GPUs), *cusvaer* (1, 4 GPUs), *HyQuas* (1, 4 GPUs), and *Atlas* (1, 4 GPUs) using qft circuit. The X-axis represents the number of qubits, while the Y-axis shows the simulation time in seconds (log-scale).

Comparison with *Qsim* and *cusvaer*. As shown in Figure 7, `ScaleQsim` consistently outperforms SOTA simulators across all qubit ranges. At 28 and 30 qubits, `ScaleQsim` (4 GPUs) finished in 0.13s and 0.22s, respectively. This resulted in speedups of 8.12× and 11.50× over *Qsim* (1.08s, 2.57s), and 33.69× and 20.09× over *cusvaer* (4.38s, 4.42s). At 33 qubits, `ScaleQsim` takes 5.89 seconds on 1 GPU, achieving 3.45× and 3.03× speedup over *Qsim* (20.34s) and *cusvaer* (17.84s). On 4 GPUs, `ScaleQsim` reduces the time to 4.01 seconds, achieving a 3.35× speedup over *cusvaer* (13.44s). At 35 qubits, only `ScaleQsim` and *cusvaer* complete execution on 4 GPUs due to memory limitations. `ScaleQsim` finishes in 15.61 seconds, achieving a 2.21× speedup over *cusvaer* (34.61s).

These results show the structural efficiency of `ScaleQsim`. This efficiency is enabled by adaptive kernel parameter tuning, which adjusts thread count and block size based on the number of `Target Indices` and available GPU memory. In contrast, *Qsim* uses fixed kernel parameters regardless of problem size or memory availability, leading to suboptimal performance in both small and large circuits. Similarly, *cusvaer*, despite being built on the cuStateVec backend, suffers from significant overhead because cuStateVec is inherently designed for single-GPU simulation and has an independent memory model. If a gate requires inter-GPU or inter-node communication, memory coordination and synchronization are triggered per every gate operation. Thus, as the number of qubits and circuit complexity increase, it incurs more frequent synchronization, ultimately degrading performance. However, `ScaleQsim` mitigates these issues by using a fixed memory layout that is distributed across all GPUs and precomputing `Target Indices` before the simulation.

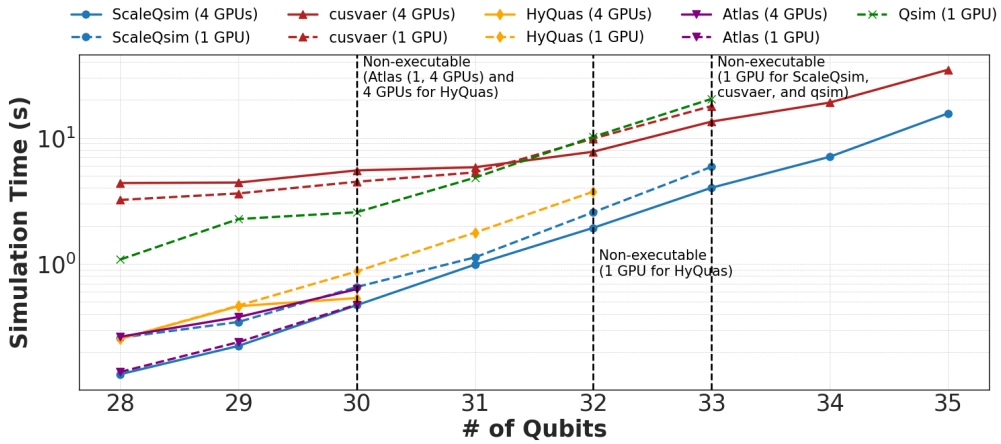


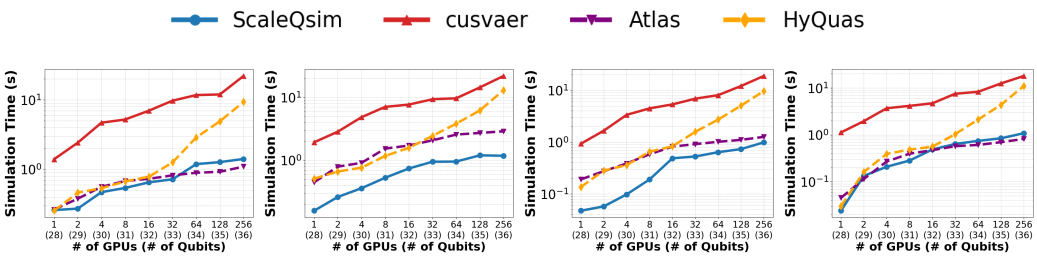
Fig. 7. Scale-in: Multi-GPU support enables simulation of a higher number of qubits (log-scale).

This supports global remote memory access without repeated synchronization. While this fixed memory layout can incur additional communication, as required communication between fixed memory layout is precomputed, this allows reduced communication overhead with optimized interconnect protocol (i.e., MPI and CUDA P2P), which improves both performance and scalability.

Comparison with *HyQuas* and *Atlas*. At 30 qubits on 4 GPUs, `ScaleQsim` achieves 1.32× and 1.73× speedup over *HyQuas* and *Atlas*, respectively. However, as the number of qubits increases, both SOTA simulators fail to scale: *HyQuas* cannot simulate beyond 32 qubits on 1 GPU or 30 qubits on 4 GPUs, while *Atlas* fails to execute above 30 qubits in all configurations. In contrast, `ScaleQsim` successfully completes simulation up to 32 qubits on 1 GPU in 2.57 seconds and on 4 GPUs in 0.99 seconds, which demonstrates both its successful execution at larger scales and its superior scalability. The limitation of *HyQuas* and *Atlas* is caused not by insufficient memory, but by the underlying design of each simulator. A100 GPU (80 GB) is equipped with sufficient memory for up to 33-qubit simulations (e.g., 16–64 GB for 31–33 qubits in single precision, 8 bytes per complex amplitude). Thus, the non-executable of *HyQuas* beyond 30 (4 GPUs), 32 qubits (1 GPU), and *Atlas* beyond 30 qubits (1, 4 GPUs) is not caused by memory limitations, but by structural inflexibility in statically defined kernel structures and fixed simulation plans.

Specifically, *HyQuas* divides the circuit into sub-circuits, each composed of a partitioned group of gates, and simulates them using precompiled kernels with a fixed number of local qubits (e.g., 28) and statically allocated memory layouts. However, as the number of total qubits increases, some sub-circuits require more local qubits than the kernel can handle, and the corresponding sub-circuit exceeds the available memory on a single GPU. When a sub-circuit accesses more qubits than the kernel is designed to process, execution fails due to internal overflow in offset tables or shared memory allocation.

Similarly, *Atlas* adopts a strict gate-to-qubit mapping scheme and constructs a fixed simulation plan based on a predefined GPU configuration. To overcome the fixed plan limitation, we also attempted to generate a simulation plan on a single A100 (80 GB) using the identical setup, but *Atlas* failed to produce a valid simulation plan and timed out. This is further supported by our evaluation results that only the specific qubit-GPU combinations described in the paper [86] executed successfully in our evaluation setup, while other combinations failed to generate a simulation plan, even when using the identical computing setup. These findings reveal the inflexibility of static simulation plans, which fail to accommodate varying circuit sizes or hardware configurations. As the number



(a) qft. (b) vqc. (c) qsvm. (d) ghz.

Fig. 8. Weak scalability of ScaleQsim and SOTA across diverse circuits and GPU scales (log-scale).

of qubits increases, the mismatch between the required state vector size and available GPU memory, combined with the lack of dynamic scheduling, leads to memory conflicts and simulation failure.

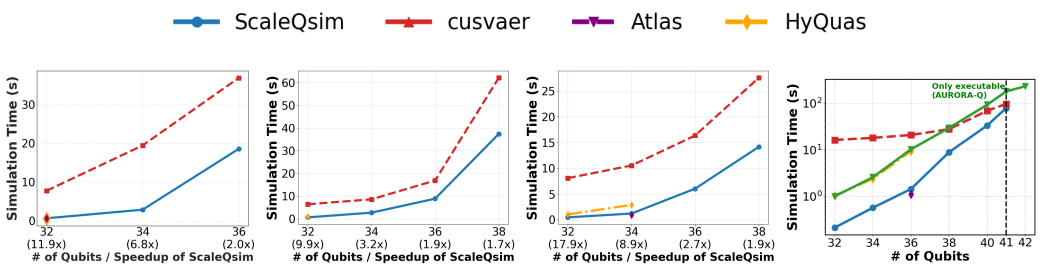
In contrast, ScaleQsim supports dynamic and scalable execution by computing affected Target Indices at runtime and adaptively configuring kernel parameters, including CUDA thread-block structures. Based on the requested number of qubits, ScaleQsim dynamically calculates the required memory size and evenly distributes the state vector, prior to the simulation execution. This dynamic state vector management allows ScaleQsim to evenly distribute gate execution without skewness.

4.3 Multi-Node Scalability

Weak scalability with various circuits. Figure 8 compares the weak scalability of ScaleQsim with SOTA simulators (*cusvaer*, *HyQuas*, *Atlas*) across four circuit types: qft, vqc, qsvm, and ghz. qft and vqc are high-complexity circuits, characterized by strong inter-qubit interactions and large gate counts, while qsvm and ghz represent lower-complexity circuits with simpler execution flows. We varied both the number of qubits and GPUs simultaneously (e.g., from 28 qubits on 1 GPU to 36 qubits on 256 GPUs) to evaluate how each simulator maintains performance under increasing problem size and resource scale. The comparison focuses on how the scalability trend varies depending on circuit characteristics.

As shown in the figure, for high-complexity circuits, in qft, ScaleQsim takes 0.26s at 28 qubits and 1.41s at 36 qubits. At 36 qubits, ScaleQsim achieves 15.63× and 6.62× speedup over *cusvaer* (22.05s) and *HyQuas* (9.34s), respectively. However, *Atlas* achieves the lowest simulation time of 1.10s at 36 qubits, slightly outperforming ScaleQsim in this specific case. In vqc, ScaleQsim maintains stable simulation time, from 0.16s (28 qubits) to 1.18s (36 qubits). At 36 qubits, it achieves 18.30×, 11.00×, and 2.40× speedup over *cusvaer* (21.67s), *HyQuas* (12.95s), and *Atlas* (2.88s), respectively. For low-complexity circuits, in qsvm, ScaleQsim takes 0.05s at 28 qubits and 0.99s at 36 qubits, achieving up to 19.00×, 9.70×, and 1.30× speedup over *cusvaer* (18.83s), *HyQuas* (9.63s), and *Atlas* (1.26s), respectively. In ghz, ScaleQsim completes the 36-qubit simulation in 1.08s, achieving 16.60× and 10.20× speedup over *cusvaer* (17.94s) and *HyQuas* (11.04s), while *Atlas* achieves the lowest simulation time (0.82s) in this specific case. *Atlas* achieves the best performance in specific configurations that utilize a precompiled simulation plan. However, since *Atlas* has to first produce the plan before the execution in a realistic scenario, its total runtime would increase due to the newly included plan generation time.

These results show how circuit characteristics influence simulator scalability. Specifically, in qft, strong inter-qubit interactions arise due to controlled-phase gates, where each qubit interacts with all lower qubits [41, 67]. This structure favors *Atlas*, which statically defines execution paths within



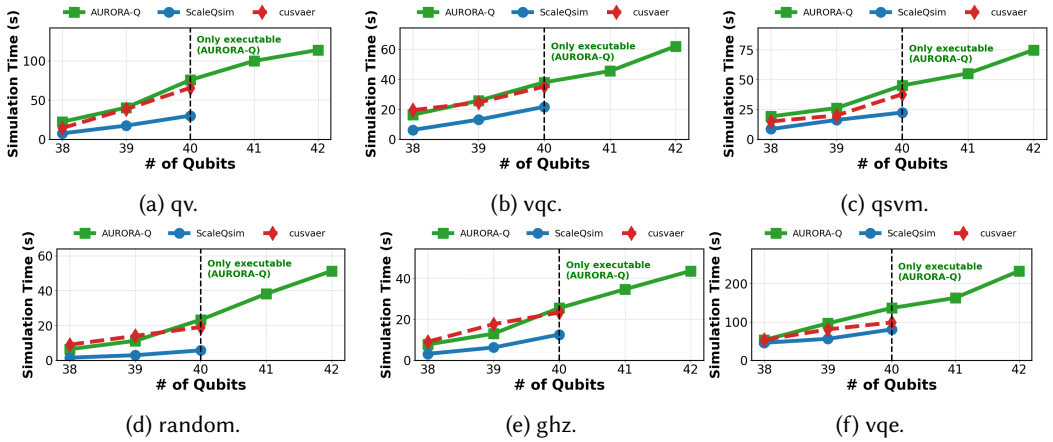
(a) 16 GPUs. (b) 32 GPUs. (c) 64 GPUs. (d) 256 GPUs.

Fig. 9. Strong scalability comparison of ScaleQsim and SOTA on qft circuits with the number of GPUs.

local qubit partitions to minimize communication overhead. Although *Atlas* was slightly faster, the performance gap with ScaleQsim remained small (1.10s vs. 1.41s at 36 qubits), demonstrating that static and even distribution of memory layout across GPUs does not incur too much overhead due to replicated mapping between local and global and efficient protocol usage. In contrast, vqc contains many gate operations but exhibits low inter-qubit interactions, as most gates are single-qubit or local entangling gates confined to adjacent qubits [43, 93]. This structure favors ScaleQsim, which executes operations independently based on affected `Target Indices`, enabling many gate operations to process within each GPU and reducing synchronization and cross-node communication. In addition, qsvm and ghz are relatively lower-complexity as simple entanglement structures. qsvm include many gates, but most are local operations with weak inter-qubit interactions, while ghz contain shallow entanglement that spans all qubits, but a low number of gates. These characteristics allow ScaleQsim to maintain scalability by minimizing cross-node communication and ensuring efficient task execution across GPUs. However, *cusvaer* and *HyQuas* show consistent performance degradation regardless of circuit characteristics. *cusvaer* suffers from overhead caused by memory pooling and centralized gate scheduling, while *HyQuas* causes overhead from global-local swap operations that trigger frequent cross-node shuffling as the number of nodes increases.

Strong scalability with multiple GPUs. Figure 9 compares the strong scalability of ScaleQsim, *cusvaer*, *HyQuas*, and *Atlas* with an increasing number of qubits across varying numbers of GPUs from 16 to 256 GPUs. As shown in the figure, ScaleQsim consistently outperforms *cusvaer* across all configurations. At 16 GPUs, ScaleQsim takes 18.58 seconds at 36 qubits, compared to 36.62 seconds by *cusvaer*, showing a 1.97× speedup. At 32 and 64 GPUs, the simulation time at 38 qubits is 37.43 and 14.16 seconds for ScaleQsim, and 62.14 and 27.61 seconds for *cusvaer*, resulting in 1.70× and 1.95× speedups, respectively. With 256 GPUs, ScaleQsim completes the 40-qubit simulation in 33.01 seconds, while *cusvaer* takes 68.46 seconds, achieving a 2.10× speedup. These results demonstrate that ScaleQsim provides lower simulation time and better scalability across a wide range of configurations, particularly as the number of qubits increases.

In contrast, *HyQuas* and *Atlas* exhibit severe limitations in strong scalability due to their static kernel configurations and memory layouts. *HyQuas* fails to execute beyond 36-qubit simulation in most configurations and only completes a 36-qubit simulation on 256 GPUs with a simulation time of 9.34 seconds. In contrast, ScaleQsim performs the same simulation in 1.41 seconds, achieving a 6.6× speedup. Notably, all 38- and 40-qubit simulations fail on *HyQuas*, regardless of the number of GPUs. These failures stem from its use of precompiled CUDA kernels with fixed local qubit (e.g., 28 qubits), which are statically embedded in both kernel design and scheduling logic, restricting scalability to larger or irregular circuits. *Atlas* also demonstrates limited scalability. Executable



(a) qv. (b) vqc. (c) qsvm.

(d) random. (e) ghz. (f) vqe.

Fig. 10. Performance comparison between ScaleQsim and *cusvaer* across diverse circuits using 128 GPUs.

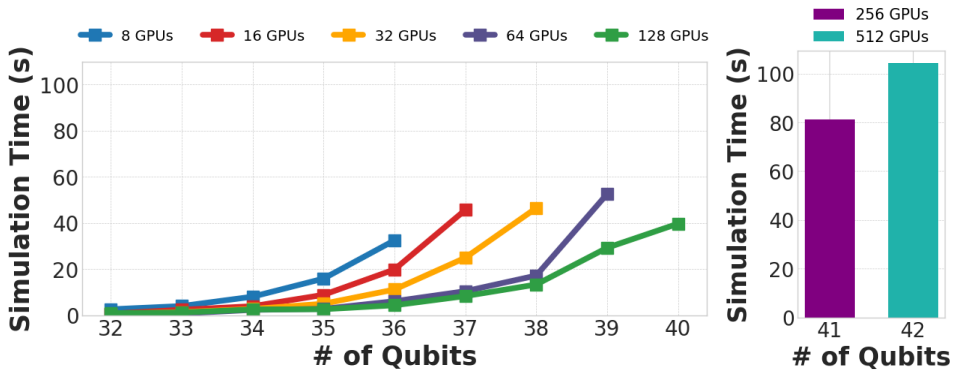
configurations are limited to 32-qubit on 16 GPUs (0.63s), 34-qubit on 64 GPUs (0.89s), and 36-qubit on 256 GPUs (1.09s). All other configurations of qubits and GPUs fail to execute due to the architectural constraints discussed in Section 4.2.

Despite its current inability to execute, we believe the divide and conquer approach of *Atlas* will face scalability issues similar to those of *HyQuas*, even if executed correctly at a higher number of qubits. As the circuit size grows, we anticipate it will become difficult to generate efficient sub-circuits, leading to three problems. 1) The divided sub-circuits will become imbalanced, causing workload skewness. 2) As the number of sub-circuits grows, distributed execution will increase communication and synchronization overhead. 3) Individual sub-circuits will be too large to fit into a GPU memory. This will cause simulation failures at a lower qubit count compared to ScaleQsim, even when using an identical number of GPUs. Thus, these results highlight that ScaleQsim is the only simulator in the evaluation that supports scalable and high-performance execution up to 40-qubit simulation with 256 GPUs, while maintaining performance advantages over all baselines.

4.4 Performance and Scalability in Diverse Circuit Comparison with SOTA (*cusvaer*)

Figure 10 compares the simulation performance of ScaleQsim and *cusvaer* across six representative quantum circuits: qv, vqc, qsvm, random, ghz, and vqe. Note that *HyQuas* and *Atlas* are excluded from this comparison, as both simulators fail to execute circuits with 38 qubits or more in most configurations due to their architectural constraints. All experiments were conducted on the identical 128 GPUs (32 nodes). As shown in the figure, for the 38–40 qubit range, ScaleQsim consistently outperformed *cusvaer* across all circuits. At 40 qubits, ScaleQsim and *cusvaer* take 29.92 and 65.39 seconds for qv circuit, achieving a 2.19× speedup. For vqc and qsvm, they take 21.55 and 22.37 seconds, while *cusvaer* takes 34.97 and 37.69 seconds, resulting in 1.62× and 1.69× speedups, respectively. Random circuit takes 5.66 and 19.02 seconds, yielding a 3.36× speedup, and ghz circuit takes 12.52 and 23.24 seconds, achieving a 1.86× speedup. For vqe, ScaleQsim takes 80.57 seconds, while *cusvaer* takes 98.29 seconds, resulting in a 1.21× speedup. Although performance varies across circuit types, ScaleQsim demonstrated consistent performance improvements up to 6.15× over *cusvaer*.

Notably, ScaleQsim showed significant performance improvements over *cusvaer* for ghz and random. This is because ghz has low depth and a simple linear structure based on CNOT gates, resulting in low computational complexity and minimal memory access. In random circuits, although



(a) 32–40 Qubits. (b) 41–42 Qubits.

Fig. 11. Scale-out: Multi-node support enables simulation of a higher number of qubits.

gate-to-qubit mappings are randomized, many operations remain localized to a small subset of qubits, reducing the need for inter-communication. This locality aligns well with `Target Index`-based execution scheme, enabling efficient parallelization with minimal data movement. In contrast, qv, vqc, qsvm, and vqe have deeper structures and broader qubit interactions, often involving multiple controlled gates. These characteristics increase both computational load and inter-communication. However, `ScaleQsim` maintains consistent performance by precomputing `Target Indices` to access only necessary amplitudes and applying adaptive kernel tuning to maximize thread-level parallelism in regions with heavier workloads.

Additionally, `ScaleQsim` shows reduced speedup on vqe circuits compared to less gate-intensive circuits. This is because, although the circuit primarily consists of communication-efficient local gates (e.g., single-qubit or adjacent entangling operations), its performance is limited by the heavy computational load from its thousands of gate operations. Despite this, `ScaleQsim` still outperforms *cusvaer* on vqe circuit. `ScaleQsim` effectively exploits circuit locality through its `Target Index`-based execution, enabling local operations to run independently within each GPU partition and reducing coordination and cross-node communication. In contrast, *cusvaer* incurs overhead from memory pooling and centralized gate scheduling, which limits its ability to leverage communication-efficient circuits and causes lower performance.

4.5 Extreme Scale-out: Multiple-Nodes

Figure 11 shows the simulation time performance of `ScaleQsim` as the number of qubits increases from 32 to 42 using qft circuits, evaluated across 8 to 512 GPUs. `ScaleQsim` scales from 2 nodes (8 GPUs) up to 32 nodes (128 GPUs) for up to 40 qubits and further extends to 256 and 512 GPUs for 41 and 42 qubits, respectively, demonstrating its ability to support large-scale simulations beyond 40 qubits. As shown in Figure 11a, `ScaleQsim` scales efficiently as the number of GPUs increases, both in terms of the number of supported qubits and simulation time. It supports up to 36 qubits with 8 GPUs and up to 40 qubits with 128 GPUs. This scale-out execution overcomes single-node memory limitations and enables full state simulations at larger scales. In terms of simulation time scalability, `ScaleQsim` achieves lower simulation times as the number of GPUs increases.

For the 36-qubit simulation, a 16-fold increase in GPUs (from 8 to 128) reduces the simulation time from 32.42 to 4.28 seconds, achieving a 7.57× speedup. The scalability is less than the linear ideal due to the initial GPU memory allocation and coordination overhead becoming a bottleneck. However,

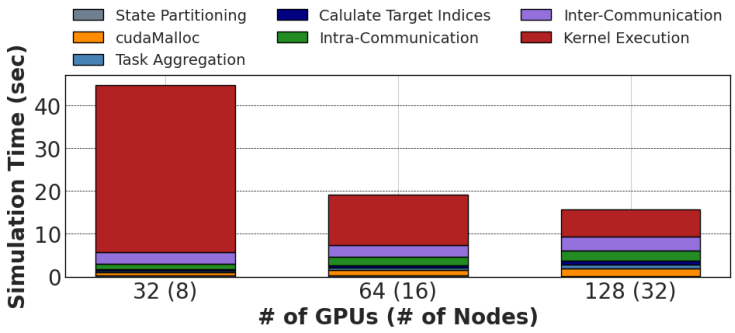


Fig. 12. Time analysis using 38 qubits.

the scalability increases as the number of qubits increases. At 39 qubits, `ScaleQsim` achieves a 1.80× speedup, reducing simulation time from 52.54s (64 GPUs) to 29.22s (128 GPUs). This shows that `ScaleQsim` can effectively utilize an increased number of GPUs to reduce simulation time.

Additionally, as shown in Figure 11b, `ScaleQsim` achieves scalability beyond 40 qubits by leveraging additional GPUs. When doubling the resources from 256 to 512 GPUs and the problem size from a 41 qubit (16 TB) to a 42 qubit (32 TB), the simulation time for `ScaleQsim` increased by only 28% (from 81.25s to 104.31s). This efficiency is due to two key designs. First, a *Two-phase Partitioning* strategy evenly distributes the full state vector across all GPUs and maintains a constant 64 GB workload per GPU for both evaluations. Second, by precomputing and broadcasting all necessary metadata using *Statespace Structure*, `ScaleQsim` eliminates the need for computation path exploration or global synchronization. However, the increase in simulation time is caused by unavoidable communication overheads, such as MPI broadcast latency that grows with the number of nodes and inter-node communication from larger gates that span multiple nodes.

4.6 Time Analysis

Since `ScaleQsim` does not partition the circuit for independent local execution within a single GPU, communication between nodes and GPUs can be frequent. Thus, it is critical to minimize the communication overhead from the globally distributed full state vector. To analyze the execution time of each subcomponent in `ScaleQsim`, Figure 12 presents the simulation time breakdown for simulating a 38-qubit qft circuit across 32, 64, and 128 GPUs. We present the time breakdown by grouping the major components of the simulation into three distinct categories: Initialization (*State Partitioning*, *cudaMalloc*), Execution Planning (*Task Aggregation*, *Calculate Target Indices*), and Computation (*Kernel Execution*, *Intra-Communication*, *Inter-Communication*).

Initialization. *State Partitioning* time decreases as the number of GPUs increases, taking 0.34 seconds with 32 GPUs, 0.21 seconds (38.2%) with 64 GPUs, and 0.17 seconds (50.0%) with 128 GPUs. This reduction occurs because the task is parallelized, and a larger number of GPUs reduces the portion of the state vector handled by each GPU. In contrast, *cudaMalloc* time increases from 0.76 seconds with 32 GPUs to 1.38 seconds (81.6%) with 64 GPUs and 1.79 seconds (135.5%) with 128 GPUs due to the additional overhead of managing memory allocation across more GPUs. Both component times account for only a small portion of the total simulation time. Additionally, this initialization process differs from *Atlas* [86], which relies on an offline process to generate a fixed simulation plan before execution. As the number of qubits and circuit complexity increase, this offline process grows significantly in duration, eventually taking longer than the execution time itself. In contrast, `ScaleQsim` employs a lightweight initialization phase that simply partitions the

full state vector based on the number of qubits and available GPUs, avoiding any time-consuming offline process before proceeding directly to the execution phase.

Execution Planning. *Task Aggregation* time increases as the number of GPUs increases, taking 0.24, 0.48, and 0.88 seconds with 32, 64, and 128 GPUs, respectively. Similarly, *Calculate Target Indices* time increases from 0.43 to 0.61 and 0.94 seconds with 32, 64, and 128 GPUs, respectively. This reflects the increased cost of calculating each `Target Index` to its physical location (i.e., a specific node and GPU) as the state vector is partitioned across more GPUs. However, the increases in planning time are not a significant bottleneck due to lightweight bitmask operations with minimal overhead, and the combined overhead from both components remains a small fraction of the total simulation time.

Computation. *Kernel Execution* occupies the largest portion of the total simulation time. With 32 GPUs, it takes 39.13 seconds out of 44.67 seconds, accounting for approximately 87.58% of the total time. As the number of GPUs increases, *Kernel Execution* time decreases to 10.77 seconds (60.06%) with 64 GPUs and 6.43 seconds (41.39%) with 128 GPUs. This shows that `ScaleQsim` achieves high parallelism by distributing tasks across multiple nodes and GPUs, which enables faster execution through efficient task partitioning and concurrent processing. In contrast, communication overhead gradually increases as the number of nodes and GPUs increases. *Inter-Communication* measures the execution time for three combined operations: broadcasting the metadata (*Statespace Structure*), performing a subsequent MPI coordination, and accessing remote memory during gate operations that span multiple nodes. As the time required for all three operations increases with the number of nodes, the total *Inter-communication* increases from 2.59 seconds (5.79%) with 32 GPUs to 2.84 seconds (15.83%) with 64 GPUs and 3.25 seconds (20.92%) with 128 GPUs, as more nodes require data transfer and MPI coordination. On the other hand, *Intra-Communication* measures communication between GPUs within the same node, a process that utilizes CUDA peer-to-peer (P2P). As the number of qubits increases by utilizing more GPUs, the size of the vector that represents each possible qubit state increases. Thus, the state vector is distributed across multiple devices within a node, leading to more frequent CUDA P2P access. This results in an increase in *Intra-Communication* time from 1.32 seconds (2.95%) with 32 GPUs to 1.94 seconds (10.80%) with 64 GPUs and 2.26 seconds (14.55%) with 128 GPUs.

4.7 Fidelity Analysis

`ScaleQsim` adopts an architecture that evenly distributes the full state vector across multiple nodes and GPUs, overcoming the scalability limits of circuit partitioning. This design achieves high scalability but also introduces frequent communication and complex parallel execution, requiring verification of numerical accuracy. Table 3 evaluates how closely the final state vector generated by `ScaleQsim` (16 GPUs) matches that of the baseline simulator *Qsim* (single GPU). The table is divided into two parts: the 28–32 qubit range shows the measured fidelity, a direct comparison between the two simulators, while the 33–36 qubit range shows the predicted fidelity, used to assess scalability when *Qsim* execution is infeasible. The predicted fidelity is derived from the 32-qubit *Qsim* result using two models: (1) a mathematical model following the observed trend for structured circuits (e.g., qft, ghz), and (2) a probability-fixed model for variational circuits (e.g., qv, vqc, qsvm, random, vqe), assuming that $p = |\text{amplitude}|^2$ remains constant beyond 32 qubits.

Measured Fidelity. As shown in the table, `ScaleQsim` demonstrates high consistency in the measured fidelity range of 28–32 qubits. For qft and ghz, the fidelity reaches a perfect 1.0, while complex variational circuits maintain values above $1 - 10^{-8}$. These results show that `ScaleQsim`'s distributed architecture preserves the precision of a single-GPU setup, with the negligible numerical difference resulting in a mismatch (1 − Fidelity) on the order of 10^{-8}.

Table 3. Fidelity between *Qsim* (single-GPU) and `ScaleQsim` (16 GPUs) across various quantum circuits (Measured fidelity: 28–32 qubit range, comparison with *Qsim*, Predicted fidelity: 33–36 qubit range, estimated from observed trends).

Circuit	Measured Fidelity (*Qsim* Vs. `ScaleQsim`)					Predicted Fidelity			
	28	29	30	31	32	33	34	35	36
qft	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000
qv	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$
vqc	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$
qsvm	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000
random	$1-10^{-8}$	1.00000000	1.00000000	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$
ghz	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000	1.00000000
vqe	$1-10^{-8}$	$1-4\times10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-10^{-8}$	$1-1.2\times10^{-7}$	$1-10^{-8}$	$1-3.6\times10^{-7}$	$1-10^{-8}$

A slight fidelity drop is observed only once, at 29 qubits in vqe, where the fidelity is measured at $1-4\times10^{-8}$. This minor and localized deviation can be attributed to the extreme structural complexity of vqe, which amplifies small floating-point errors. However, the deviation remains at the 10^{-8} level and does not compromise overall accuracy.

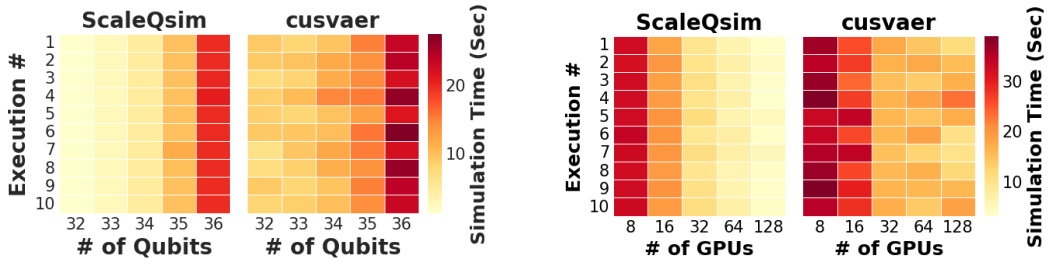
Predicted Fidelity. For the 33–36 qubit range, where *Qsim* cannot run due to memory limitations, we assess the numerical stability of `ScaleQsim` by comparing its measured results with a predicted baseline. The baseline is defined using two models according to circuit characteristics. First, for qft and ghz, the model follows the theoretical trends observed in the 28–32 qubit range, where qft amplitudes decrease by a factor of $1/\sqrt{2}$ with each additional qubit, and ghz amplitudes remain constant at the theoretical value of $1/\sqrt{2}$. Second, for complex variational circuits without a clear pattern, a probability-fixed model assumes that the measured probability $p = |\text{amplitude}|^2$ remains unchanged at 32 qubits for larger circuits. This conservative assumption provides a reasonable reference under the expectation that *Qsim* behavior remains stable. As depicted, `ScaleQsim` achieves high fidelity above $1-10^{-8}$ for most circuits in the predicted range without unexpected deviations from the baseline. In contrast, vqe circuit shows a slightly higher sensitivity to error accumulation at larger scales, with its fidelity decreasing to $1-1.2\times10^{-7}$ and $1-3.6\times10^{-7}$ at 33 and 35 qubits, respectively. However, the deviation remains at the 10^{-7} level and does not compromise overall accuracy.

In summary, these results demonstrate that the scalability of `ScaleQsim` maintains high numerical precision, and the simulator ensures stable and consistent behavior at larger scales.

4.8 Performance Variability and Stability

Figure 13 shows heatmaps of simulation time measurements for `ScaleQsim` and *cusvaer* under two conditions. In Figure 13a, qft circuit is executed 10 times for 32 to 36 qubits on 16 GPUs. In Figure 13b, qft circuit is executed 10 times for 36 qubits with the number of GPUs varied from 8 to 128. Each cell represents the simulation time of a single execution, with darker colors indicating longer simulation times. The stability of the color distribution reflects the consistency of the simulation time for each simulator.

Impact of # Qubits. As shown in Figure 13a, `ScaleQsim` exhibits highly stable simulation times across all qubit ranges. For example, at 36 qubits, the 10 runs have an average simulation time of 19.91 seconds, with the minimum and maximum simulation times being 19.77 and 20.79 seconds, respectively. Similarly, the differences between the minimum and maximum for 32 to 35 qubits are 0.035, 0.023, 0.024, and 2.01 seconds, respectively. In contrast, *cusvaer* demonstrates greater simulation time variability across repeated executions. At 36 qubits, its simulation times span from 23.24 to 27.28 seconds, with average simulation times of 25.63 seconds. Similar variations



(a) Number of Qubits.

(b) Number of GPUs.

Fig. 13. Performance variability of `ScaleQsim` and *cusvaer*.

are observed for lower numbers of qubits, with differences of 2.78, 2.40, 5.10, and 3.02 seconds, respectively.

This consistency in `ScaleQsim` is attributed to its two-phase partitioning, which evenly divides the full state vector across GPUs for independent processing. Additionally, precomputed `Target Indices` enable localized execution and minimal communication. In contrast, *cusvaer*'s dynamic memory pooling causes irregular memory access and higher synchronization and communication overhead, reducing performance consistency.

Impact of # GPUs. Additionally, `ScaleQsim` maintains stable simulation times as the number of GPUs increases. For example, with 128 GPUs, the 10 runs have an average simulation time of 3.92 seconds, with the minimum and maximum times being 3.12 and 5.12 seconds, respectively. Similarly, the difference between the minimum and maximum times for 8, 16, 32, and 64 GPUs is 0.96, 2.04, 1.96, and 1.36 seconds, respectively. In contrast, *cusvaer* exhibits high variability as the number of GPUs grows. At 128 GPUs, simulation times range from 9.85 to 22.94 seconds, with an average of 15.65 seconds. The variability is also notable at smaller GPU setups, with differences of 5.99, 10.33, 2.66, and 6.06 seconds for 8, 16, 32, and 64 GPUs, respectively. This indicates that `ScaleQsim` consistently maintains stable performance across GPU configurations, whereas *cusvaer* becomes increasingly unstable at large scales.

5 Related Work

5.1 Optimizing Quantum Circuit Simulation

There have been many studies that optimize quantum circuit simulation to enhance performance. Previous studies [18, 62, 86, 89, 90] focused on partitioning quantum circuits into sub-circuits for full state simulation. These approaches accelerate execution and extend scalability by parallelizing sub-circuit computations and offloading data to host memory or high-performance storage. Other studies [78, 84, 85, 88] have proposed lossless compression and adaptive error-bounded encoding techniques to reduce memory consumption, enabling quantum circuit simulations with a higher number of qubits. In addition, hybrid (full state vector and amplitude sampling) approaches have been proposed [15, 52, 83]. These methods simulate each sub-circuit independently using amplitude sampling and synchronize the results using full state vectors, aiming to reduce computational complexity and alleviate memory bottlenecks. Some studies [16, 38, 51, 58, 63] focused on amplitude sampling simulation frameworks, where contraction paths are sliced to accelerate computation. These methods also explore optimal contraction orders to improve kernel-level efficiency.

Our study aligns with these prior efforts in improving the scalability and efficiency of quantum circuit simulation. However, `ScaleQsim` aims to provide a unified representation of the full state

vector rather than partitioning, as it can limit scalability in terms of both performance and the number of qubits that can be simulated. Through two-phase partitioning, `ScaleQsim` partitions the state vector itself rather than the quantum circuit, and evenly distributes the full state vector across multiple nodes and GPUs, enabling fine-grained task allocation and balanced memory usage. Additionally, it minimizes communication overhead and improves parallel efficiency without structural modifications to the circuit. This allows `ScaleQsim` to enhance runtime and support more qubits than previous full state simulation frameworks.

5.2 Parallel Optimization Strategies in Exascale HPC Applications

To maximize performance, several simulation applications and frameworks, such as AMReX [91], FLASH-X [26], VASP [36], and LAMMPS [80] have been optimized with various parallelization schemes for extreme-scale HPC systems. Previous studies [44, 71, 75] have focused on accelerating large-scale scientific simulations through domain decomposition, task-based parallelism, and adaptive load balancing. Other works [6, 7, 20, 92] improve memory resource utilization by optimizing I/O parallelism, garbage collection, and overlapping computation with communication. In addition, several studies [40, 68, 70, 73] employ hierarchical parallel models such as MPICH [33], OpenMP [22], and CUDA [14], applying locality-aware scheduling and cache optimization to reduce bottlenecks and improve performance.

These approaches highlight key techniques for improving scalability and efficiency in large-scale simulations. Similarly, `ScaleQsim` faces comparable challenges in quantum circuit simulation, where full state simulation requires retaining all amplitudes in memory and makes domain decomposition ineffective. To address this, `ScaleQsim` employs task-level parallelism by evenly partitioning the full state vector across nodes and GPUs using a two-phase partitioning scheme. This enables fine-grained task allocation and balanced memory usage. Combined with adaptive kernel configuration and efficient communication mechanisms, `ScaleQsim` improves parallel efficiency while minimizing overhead in distributed execution.

6 Conclusion

In this paper, we propose `ScaleQsim`, a highly scalable simulation framework that applies two-phase partitioning for balanced state vector distribution and integrates adaptive kernel tuning with precomputed index management for performance optimization. Our evaluations across various quantum circuits and scales show that `ScaleQsim` achieves up to 77.40× speedup over the state-of-the-art simulators *cusvaer*, *HyQuas*, and *Atlas*, supports full state simulations up to 42 qubits on 512 GPUs (128 nodes), and maintains consistent performance with low variance. These results demonstrate that `ScaleQsim` effectively addresses the limitations of prior simulators, offering a scalable and practical solution for large-scale quantum circuit simulation.

7 Acknowledgment

This research was supported by Seoul National University of Science & Technology. This work was supported by the National Research Foundation of Korea (NRF) under Grant Nos. RS-2025-16070038 and RS-2025-00554650. This work was also supported by the Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and used resources of the National Energy Research Scientific Computing Center (NERSC). We acknowledge the use of LLM to refine the academic language.

References

[1] Armin Ahmadzadeh and Hamid Sarbazi-Azad. 2023. Fast and scalable quantum computing simulation on multi-core and many-core platforms. *Quantum Information Processing* 22, 5 (2023), 215.

[2] Armin Ahmadzadeh and Hamid Sarbazi-Azad. 2024. Performance analysis and modeling for quantum computing simulation on distributed GPU platforms. *Quantum Information Processing* 23, 11 (2024), 1–40.

[3] 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.

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

[5] Robert Beals, Stephen Brierley, Oliver Gray, Aram W Harrow, Samuel Kutin, Noah Linden, Dan Shepherd, and Mark Stather. 2013. Efficient distributed quantum computing. *Proc. of the Royal Society A: Mathematical, Physical and Engineering Sciences* 469, 2153 (2013), 20120686.

[6] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2015. Pattern-driven parallel I/O tuning. In *Proc. of the 10th Parallel Data Storage Workshop*. 43–48.

[7] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing i/o performance of hpc applications with autotuning. *ACM Trans on Parallel Computing (TOPC)* 5, 4 (2019), 1–27.

[8] Charles H Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John Smolin. 1992. Experimental quantum cryptography. *Journal of cryptology* 5 (1992), 3–28.

[9] Charles H Bennett and Gilles Brassard. 2014. Quantum cryptography: Public key distribution and coin tossing. *Theoretical computer science* 560 (2014), 7–11.

[10] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. 2018. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968* (2018).

[11] Daniel J Bernstein and Tanja Lange. 2017. Post-quantum cryptography. *Nature* 549, 7671 (2017), 188–194.

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

[13] Anne Broadbent and Christian Schaffner. 2016. Quantum cryptography beyond quantum key distribution. *Designs, Codes and Cryptography* 78 (2016), 351–382.

[14] Ian Buck. 2007. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*. 6–es.

[15] Lukas Burgholzer, Hartwig Bauer, and Robert Wille. 2021. Hybrid Schrödinger-Feynman simulation of quantum circuits with decision diagrams. In *2021 IEEE Int'l Conf on Quantum Computing and Engineering (QCE)*. IEEE, 199–206.

[16] Lukas Burgholzer, Alexander Ploier, and Robert Wille. 2022. Simulation paths for quantum circuit simulation with decision diagrams what to learn from tensor networks, and what not. *IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems* 42, 4 (2022), 1113–1122.

[17] Marco Cerezo, Guillaume Verdon, Hsin-Yuan Huang, Lukasz Cincio, and Patrick J Coles. 2022. Challenges and opportunities in quantum machine learning. *Nature computational science* 2, 9 (2022), 567–576.

[18] Zhao-Yun Chen, Qi Zhou, Cheng Xue, Xia Yang, Guang-Can Guo, and Guo-Ping Guo. 2018. 64-qubit quantum circuit simulation. *Science Bulletin* 63, 15 (2018), 964–971.

[19] Andrew M Childs, Dmitri Maslov, Yunseong Nam, Neil J Ross, and Yuan Su. 2018. Toward the first quantum simulation with quantum speedup. *Proc. of the National Academy of Sciences* 115, 38 (2018), 9456–9461.

[20] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. 2018. Parallelizing garbage collection with I/O to improve flash resource utilization. In *Proc. of the 27th International Symp on High-Performance Parallel and Distributed Computing*. 243–254.

[21] Andrew Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. In *APS March meeting abstracts*, Vol. 2018. L58–003.

[22] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[23] Andrew J Daley, Immanuel Bloch, Christian Kokail, Stuart Flannigan, Natalie Pearson, Matthias Troyer, and Peter Zoller. 2022. Practical quantum advantage in quantum simulation. *Nature* 607, 7920 (2022), 667–676.

[24] Zohreh Davarzani, Mariam Zomorodi, and Mahboobeh Houshmand. 2022. A hierarchical approach for building distributed quantum systems. *Scientific Reports* 12, 1 (2022), 15421.

[25] Jun Doi, Hitomi Takahashi, Rudy Raymond, Takashi Imamichi, and Hiroshi Horii. 2019. Quantum computing simulator on a heterogenous HPC system. In *Proc. of the 16th ACM Int'l Conf on Computing Frontiers*. 85–93.

[26] Anshu Dubey, Klaus Weide, Jared O'Neal, Akash Dhruv, Sean Couch, J Austin Harris, Tom Klosterman, Rajeev Jain, Johann Rudi, Bronson Messer, et al. 2022. Flash-X: A multiphysics simulation software instrument. *SoftwareX* 19 (2022), 101168.

[27] Lei Fan and Zhu Han. 2022. Hybrid quantum-classical computing for future network optimization. *IEEE Network* 36, 5 (2022), 72–76.

[28] Rong Fu, Zhongling Su, Han-Sen Zhong, Xiti Zhao, Jianyang Zhang, Feng Pan, Pan Zhang, Xianhe Zhao, Ming-Cheng Chen, Chao-Yang Lu, et al. 2024. Achieving energetic superiority through system-level quantum circuit simulation. *arXiv preprint arXiv:2407.00769* (2024).
[29] Iulia M Georgescu, Sahel Ashhab, and Franco Nori. 2014. Quantum simulation. *Reviews of Modern Physics* 86, 1 (2014), 153–185.
[30] Google. 2025. *Qsim*. https://quantumai.google/qsim Accessed: 2025-04-07.
[31] TM Graham, Y Song, J Scott, C Poole, L Phuttitarn, K Jooya, P Eichler, X Jiang, A Marra, B Grinkemeyer, et al. 2022. Multi-qubit entanglement and algorithms on a neutral-atom quantum computer. *Nature* 604, 7906 (2022), 457–462.
[32] Harper R Grimsley, Daniel Claudino, Sophia E Economou, Edwin Barnes, and Nicholas J Mayhall. 2019. Is the trotterized uccsd ansatz chemically well-defined? *Journal of chemical theory and computation* 16, 1 (2019), 1–6.
[33] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.
[34] Thomas Grurl, Jürgen Fuß, and Robert Wille. 2020. Considering decoherence errors in the simulation of quantum circuits using decision diagrams. In *Proc. of the 39th Int'l Conf on Computer-Aided Design*. 1–7.
[35] Gian Giacomo Guerreschi, Justin Hogaboam, Fabio Baruffa, and Nicolas PD Sawaya. 2020. Intel Quantum Simulator: A cloud-ready high-performance simulator of quantum circuits. *Quantum Science and Technology* 5, 3 (2020), 034007.
[36] Jürgen Hafner. 2008. Ab-initio simulations of materials using VASP: Density-functional theory and beyond. *Journal of computational chemistry* 29, 13 (2008), 2044–2078.
[37] Thomas Häner and Damian S Steiger. 2017. 5 petabyte simulation of a 45-qubit quantum circuit. In *Proc. of the Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. 1–10.
[38] Cupjin Huang, Fang Zhang, Michael Newman, Xiaotong Ni, Dawei Ding, Junjie Cai, Xun Gao, Tenghui Wang, Feng Wu, Gengyan Zhang, et al. 2021. Efficient parallelization of tensor network contraction for simulating quantum computation. *Nature Computational Science* 1, 9 (2021), 578–587.
[39] Yipeng Huang and Margaret Martonosi. 2019. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proc. of the 46th International Symp on Computer Architecture*. 541–553.
[40] Surabhi Jain, Rashid Kaleem, Marc Gamell Balmana, Akhil Langer, Dmitry Durnov, Alexander Sannikov, and Maria Garzaran. 2018. Framework for scalable intra-node collective operations using shared memory. In *SC18: Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. IEEE, 374–385.
[41] Yuwei Jin, Xiangyu Gao, Minghao Guo, Henry Chen, Fei Hua, Chi Zhang, and Eddy Z Zhang. 2023. Quantum Fourier Transformation Circuits Compilation (2023). *arXiv preprint arXiv:2312.16114* (2023).
[42] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific reports* 9, 1 (2019), 10736.
[43] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *nature* 549, 7671 (2017), 242–246.
[44] Hamidreza Khaleghzadeh, Ravindranath Reddy Manumachu, and Alexey Lastovetsky. 2018. A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms. *IEEE Trans on Parallel and Distributed Systems* 29, 10 (2018), 2176–2190.
[45] Tariq M Khan and Antonio Robles-Kelly. 2020. Machine learning: Quantum vs classical. *IEEE Access* 8 (2020), 219275–219294.
[46] Ryan LaRose. 2018. Distributed memory techniques for classical simulation of quantum circuits. *arXiv preprint arXiv:1801.01037* (2018).
[47] Jonathan Wei Zhong Lau, Kian Hwee Lim, Harshank Shrotriya, and Leong Chuan Kwek. 2022. NISQ computing: where are we and where do we go? *AAPPS bulletin* 32, 1 (2022), 27.
[48] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Heim, Martin Roetteler, and Sriram Krishnamoorthy. 2021. Sv-sim: scalable pgas-based state vector simulation of quantum circuits. In *Proc. of the Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[49] Jun Li, Xiaodong Yang, Xinhua Peng, and Chang-Pu Sun. 2017. Hybrid quantum-classical approach to quantum optimal control. *Physical review letters* 118, 15 (2017), 150503.
[50] Thomas Lubinski, Cassandra Granade, Amos Anderson, Alan Geller, Martin Roetteler, Andrei Petrenko, and Bettina Heim. 2022. Advancing hybrid quantum–classical computation with real-time execution. *Frontiers in Physics* 10 (2022), 940293.
[51] Danylo Lykov, Roman Schutski, Alexey Galda, Valeri Vinokur, and Yuri Alexeev. 2022. Tensor network quantum simulator with step-dependent parallelization. In *2022 IEEE Int'l Conf on Quantum Computing and Engineering (QCE)*. IEEE, 582–593.
[52] Salvatore Mandrà, Jeffrey Marshall, Eleanor G Rieffel, and Rupak Biswas. 2021. HybridQ: A hybrid simulator for quantum circuits. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. IEEE,

99–109.

[53] Igor L Markov, Aneeqa Fatima, Sergei V Isakov, and Sergio Boixo. 2018. Quantum supremacy is both closer and farther than it appears. *arXiv preprint arXiv:1807.10749* (2018).

[54] Michael James Martin, Caroline Hughes, Gilberto Moreno, Eric B Jones, David Sickinger, Sreekant Narumanchi, and Ray Grout. 2022. Energy use in quantum data centers: Scaling the impact of computer architecture, qubit performance, size, and thermal parameters. *IEEE Trans on Sustainable Computing* 7, 4 (2022), 864–874.

[55] Jorge Miguel-Ramiro, Zheng Shi, Luca Dellantonio, Albie Chan, Christine A Muschik, and Wolfgang Dür. 2023. Enhancing quantum computation via superposition of quantum gates. *Physical Review A* 108, 6 (2023), 062604.

[56] Gary J Mooney, Charles D Hill, and Lloyd CL Hollenberg. 2019. Entanglement in a 20-qubit superconducting quantum computer. *Scientific reports* 9, 1 (2019), 13465.

[57] Mikio Morita, Yoshinori Tomita, Junpei Koyama, and Koichi Kimura. 2024. Simulator demonstration of large scale variational quantum algorithm on HPC cluster. *IEEE Access* (2024).

[58] Thien Nguyen, Dmitry Lyakh, Eugene Dumitrescu, David Clark, Jeff Larkin, and Alexander McCaskey. 2022. Tensor network quantum virtual machine for simulating quantum circuits at exascale. *ACM Trans on Quantum Computing* 4, 1 (2022), 1–21.

[59] NVDIA. 2025. *NVlink*. https://www.nvidia.com/en-us/data-center/nvlink/ Accessed: 2025-04-07.

[60] Feng Pan and Pan Zhang. 2022. Simulation of quantum circuits using the big-batch tensor network method. *Physical Review Letters* 128, 3 (2022), 030501.

[61] Rhea Parekh, Andrea Ricciardi, Ahmed Darwish, and Stephen DiAdamo. 2021. Quantum algorithms and simulation for parallel and distributed quantum computing. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. IEEE, 9–19.

[62] Daeyoung Park, Heehoon Kim, Jinpyo Kim, Taehyun Kim, and Jaejin Lee. 2022. SnuQS: scaling quantum circuit simulation using storage devices. In *Proc. of the 36th ACM Int'l Conf on Supercomputing*. 1–13.

[63] Alfred M Pastor, Jose M Badia, and Maribel Castillo. 2025. A community detection-based parallel algorithm for quantum circuit simulation using tensor networks. *The Journal of Supercomputing* 81, 3 (2025), 450.

[64] Siddhartha Patra, Saeed S Jahromi, Sukhbinder Singh, and Román Orús. 2024. Efficient tensor network simulation of IBM's largest quantum processors. *Physical Review Research* 6, 1 (2024), 013326.

[65] Stefano Pirandola, Ulrik L Andersen, Leonardo Banchi, Mario Berta, Darius Bunandar, Roger Colbeck, Dirk Englund, Tobias Gehring, Cosmo Lupo, Carlo Ottaviani, et al. 2020. Advances in quantum cryptography. *Advances in optics and photonics* 12, 4 (2020), 1012–1236.

[66] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[67] Sahar Ben Rached, Isaac Lopez Agudo, Santiago Rodrigo, Medina Bandic, Artur Garcia-Saez, Sebastian Feld, Hans Van Someren, Eduard Alarcón, Carmen G Almudéver, and Sergi Abadal. 2025. Characterizing the inter-core qubit traffic in large-scale quantum modular architectures. *IEEE Access* (2025).

[68] Bharath Ramesh, Jahanzeb Maqbool Hashmi, Shulei Xu, Aamir Shafi, Mahdieh Ghazimirsaeed, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K Panda. 2021. Towards architecture-aware hierarchical communication trees on modern hpc systems. In *2021 IEEE 28th Int'l Conf on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 272–281.

[69] Somayeh Bakhtiari Ramezani, Alexander Sommers, Harish Kumar Manchukonda, Shahram Rahimi, and Amin Amirlatifi. 2020. Machine learning algorithms in quantum computing: A survey. In *2020 International joint Conf on neural networks (IJCNN)*. IEEE, 1–8.

[70] Carlos Reano, Federico Silla, Dimitrios S Nikolopoulos, and Blesson Varghese. 2017. Intra-node memory safe gpu co-scheduling. *IEEE Trans on Parallel and Distributed Systems* 29, 5 (2017), 1089–1102.

[71] Anne Reinarz, Dominic E Charrier, Michael Bader, Luke Bovard, Michael Dumbser, Kenneth Duru, Francesco Fambri, Alice-Agnes Gabriel, Jean-Matthieu Gallard, Sven Köppel, et al. 2020. ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems. *Computer Physics Communications* 254 (2020), 107251.

[72] Martin J Renner and Časlav Brukner. 2022. Computational advantage from a quantum superposition of qubit gate orders. *Physical Review Letters* 128, 23 (2022), 230503.

[73] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. 2018. Scalable system scheduling for HPC and big data. *J. Parallel and Distrib. Comput.* 111 (2018), 76–92.

[74] Abdullah Ash Saki, Mahabubul Alam, and Swaroop Ghosh. 2019. Study of decoherence in quantum computers: A circuit-design perspective. *arXiv preprint arXiv:1904.04323* (2019).

[75] Matthieu Schaller, Pedro Gonnet, Aidan BG Chalk, and Peter W Draper. 2016. SWIFT: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100,000 cores. In *Proc. of the platform for advanced scientific computing Conf*. 1–10.

[76] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. 2015. An introduction to quantum machine learning. *Contemporary Physics* 56, 2 (2015), 172–185.

[77] Raffaele Solcà, Anton Kozhevnikov, Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Thomas C Schulthess. 2015. Efficient implementation of quantum materials simulations on distributed CPU-GPU systems. In *Proc. of the Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[78] Yuhong Song, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Rui Xu, and Han Wang. 2023. Efficient algorithm for full-state quantum circuit simulation with DD compression while maintaining accuracy. *Quantum Information Processing* 22, 11 (2023), 413.

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

[80] Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J In't Veld, Axel Kohlmeyer, Stan G Moore, Trung Dac Nguyen, et al. 2022. LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer physics communications* 271 (2022), 108171.

[81] Meng Wang, Fei Hua, Chenxu Liu, Nicholas Bauman, Karol Kowalski, Daniel Claudino, Travis Humble, Prashant Nair, and Ang Li. 2023. Enabling scalable vqe simulation on leading hpc systems. In *Proc. of the SC'23 Workshops of the Int'l Conf on High Performance Computing, Network, Storage, and Analysis*. 1460–1467.

[82] Zhimin Wang, Zhaoyun Chen, Shengbin Wang, Wendong Li, Yongjian Gu, Guoping Guo, and Zhiqiang Wei. 2021. A quantum circuit simulator and its applications on Sunway TaihuLight supercomputer. *Scientific reports* 11, 1 (2021), 355.

[83] Sam Westrick, Pengyu Liu, Byeongjee Kang, Colin McDonald, Mike Rainey, Mingkuan Xu, Jatin Arora, Yongshan Ding, and Umut A Acar. 2024. Grafeyn: Efficient parallel sparse simulation of quantum circuits. In *2024 IEEE Int'l Conf on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 1132–1142.

[84] Xin-Chuan Wu, Sheng Di, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2018. Memory-efficient quantum circuit simulation by using lossy data compression. *arXiv preprint arXiv:1811.05630* (2018).

[85] 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 *Proc. of the Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. 1–24.

[86] Mingkuan Xu, Shiyi Cao, Xupeng Miao, Umut A Acar, and Zhihao Jia. 2024. Atlas: Hierarchical partitioning for quantum circuit simulation on gpus. In *SC24: Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–17.

[87] Charles Yuan and Michael Carbin. 2022. Tower: data structures in Quantum superposition. *Proc. of the ACM on Programming Languages* 6, OOPSLA2 (2022), 259–288.

[88] Boyuan Zhang, Bo Fang, Fanjiang Ye, Yida Gu, Nathan Tallent, Guangming Tan, and Dingwen Tao. 2024. Overcoming memory constraints in quantum circuit simulation with a high-fidelity compression framework. *arXiv preprint arXiv:2410.14088* (2024).

[89] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. 2021. HyQuas: hybrid partitioner based quantum circuit simulation system on GPU. In *Proc. of the 35th ACM Int'l Conf on Supercomputing*. 443–454.

[90] Chen Zhang, Haojie Wang, Zixuan Ma, Lei Xie, Zeyu Song, and Jidong Zhai. 2022. UniQ: A unified programming model for efficient quantum circuit simulation. In *SC22: Int'l Conf for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[91] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software* 4, 37 (2019), 1370.

[92] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: Exploring value patterns in GPU-Accelerated applications. In *Proc. of the 27th ACM Int'l Conf on Architectural Support for Programming Languages and Operating Systems*. 171–185.

[93] Juan C Zuñiga Castro, Jeffrey Larson, Sri Hari Krishna Narayanan, Victor E Colussi, Michael A Perlin, and Robert J Lewis-Swan. 2024. Variational quantum state preparation for quantum-enhanced metrology in noisy systems. *Physical Review A* 110, 5 (2024), 052615.

Received July 2025; revised September 2025; accepted October 2025