Atlas: Hierarchical Partitioning for Quantum Circuit Simulation on GPUs

Mingkuan Xu
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
mingkuan@cmu.edu

Shiyi Cao
Department of EECS
UC Berkeley
Berkeley, CA, USA
shicao@berkeley.edu

Xupeng Miao
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
xupeng@cmu.edu

Umut A. Acar
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
umut@cmu.edu

Zhihao Jia
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
zhihao@cmu.edu

***Abstract*—This paper presents techniques for theoretically and practically efficient and scalable Schrödinger-style quantum circuit simulation. Our approach partitions a quantum circuit into a hierarchy of subcircuits and simulates the subcircuits on multi-node GPUs, exploiting available data parallelism while minimizing communication costs. To minimize communication costs, we formulate an Integer Linear Program that rewards simulation of "nearby" gates on "nearby" GPUs. To maximize throughput, we use a dynamic programming algorithm to compute the subcircuit simulated by each kernel at a GPU. We realize these techniques in Atlas, a distributed, multi-GPU quantum circuit simulator. Our evaluation on a variety of quantum circuits shows that Atlas outperforms state-of-the-art GPU-based simulators by more than 2× on average and is able to run larger circuits via offloading to DRAM, outperforming other large-circuit simulators by two orders of magnitude.**

***Index Terms*—Parallel programming, quantum simulation.**

I. Introduction

Quantum computing has established an advantage over classical computing, especially in areas such as cryptography, machine learning, and physical sciences [1]–[8]. Several quantum computers have been built recently, including Sycamore [9], Bristlecone [10], Jiuzhang [11], Osprey [12], and Condor [13]. However, the robustness demands of many quantum applications exceed those of modern *noisy intermediate-scale quantum* (NISQ) computers, which suffer from decoherence and lack of error-correction [14], [15]. Furthermore, NISQ computers are expensive resources—many remain inaccessible beyond a small group. Therefore, there is significant interest in quantum circuit simulation, which enables performing robust quantum computation on classical parallel machine.

Several approaches to simulating quantum circuits have been proposed, including Schrödinger- and Feynman-style simulation. Even though Feynman-style simulation can require a small amount of space, its time requirement appears very high. Therefore, most modern simulators use Schrödinger-style quantum circuit simulation, which maintains an entire quantum state in a *state vector* of size 2^n for n qubits, and applies each gate of the circuit to it.

To tackle the scalability challenges of state-vector simulation, researchers exploit the plethora of parallelism available in this task by using CPUs, GPUs, and parallel machines [16]–[26]. To cope with the increasing demand for simulating larger circuits, recent work has proposed techniques for storing state vectors on DRAM and secondary storage (e.g., disks) [23], [27]. Even after over a decade of research, quantum circuit simulation continues to remain challenging for performance and scalability.

At a high level of abstraction, there are three key challenges to (Schrödinger style) state-vector simulation. The first challenge is the space requirements of storing the state vector with 2^n amplitudes, each of which is a complex number. Overcoming this challenge requires distributing the state vector in a parallel machine with heterogeneous memory. Second, simulating the application of a quantum gate to one or multiple qubits involves strided accesses to the state vector, which requires *fine-grained* inter-node and/or inter-device communications. The cost of these communications becomes the performance bottleneck for large-scale circuit simulation. Third, because quantum gates typically operate on a small number of qubits, simulating a quantum gate involves multiplying a sparse matrix with the state vector, resulting in low arithmetic intensity, e.g., as little as 0.5 on real-world quantum circuits [27]. This harms performance on modern accelerators (e.g., GPUs) designed for compute-intensive workload.

In this paper, we propose techniques for performant and scalable quantum circuit simulation on modern GPU systems. Specifically, we consider a multi-node GPU architecture, where each node may host a number of GPUs. We then perform on this architecture quantum circuit simulation in the (Schrödinger) state-vector style by using the available memory across all nodes to store the state. To minimize the communication costs, we partition the quantum circuit into a number of *stages*, each of which consists of a (contiguous) subcircuit of the input circuit that can be simulated on a single GPU without requiring non-local accesses outside the memory of the GPU. To maximize the benefits from parallelism, we further partition the subcircuit of each stage into *kernels* or groups of gates that are large enough to benefit from parallelism but also small enough to prevent exponential blowups in cost.

Given such a partitioned circuit consisting of stages and kernels, we present a simulation algorithm that performs the simulation in stages and that restricts much of the expensive communication to take place only between the stages. To maximize throughput and parallelism within each stage, the algorithm *shards* the state vector into contiguous pieces such that each shard may be used to simulate the subcircuit of the stage on a single GPU. The approach allows multiple shards (of a stage) to be executed in parallel or sequentially depending on the availability of resources.

We formulate the partitioning problem consisting of staging and kernelization and present provably effective algorithms for solving them. For the staging problem, we provide a solution based on Integer Linear Programming, which can be solved by an off-the-shelf solver. For the kernelization problem, we present a dynamic programming solution that can ensure optimality under some assumptions.

We present an implementation, called Atlas, that realizes the proposed approach. Our evaluation on a variety of quantum circuits on up to 256 GPUs (on 64 nodes) shows that Atlas is up to 20.2× faster than state-of-the-art GPU-based simulators and can scale to large quantum circuits that go beyond GPU memory capacity (up to 178× faster than state-of-the-art simulators that also go beyond GPU memory capacity).

In summary, this paper makes the following contributions.

- A hierarchical partitioning approach to scaling performant quantum circuit simulation based on staging and kernelization.
- An ILP algorithm to stage a circuit for simulation that can minimize the number of stages.
- A dynamic programming algorithm for kernelizing each stage to ensure efficient parallelism.
- An implementation that realizes hierarchical partitioning and significantly outperforms existing simulators.

II. Background

Architectural Model. We assume a multi-node GPU architecture with 2^G nodes. Each node contains multiple GPUs and a single CPU with an attached DRAM module. Each GPU can store in its *local* memory 2^L amplitudes (complex numbers). Each node can store in its *regional* memory 2^{L+R} amplitudes, where 2^R can be the number of GPUs per node or can be set such that 2^{L+R} equals the number of amplitudes that can be stored in the DRAM memory of the node.

Quantum Information Fundamentals. A quantum circuit consists of a number of qubits (quantum bits) and gates that operate on them. The state $|\psi\rangle$ of a n-qubit quantum circuit is a superposition of its *basis states* denoted $|0\rangle$, $|1\rangle$, ..., $|2^n-1\rangle$ and is typically written as $|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle$, where α_i is a complex coefficient (also called amplitude) of the basis state $|i\rangle$. When measuring the state of the system, the probability of observing the state $|i\rangle$ as the output is $|\alpha_i|^2$; therefore, $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$. When simulating a quantum circuit with n qubits, we can represent its state with a vector of 2^n complex values $\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_{2^n-1})^\top$.

The semantics of a k-qubit gate is specified by a $2^k \times 2^k$ unitary complex matrix U and applying the gate to a quantum circuit with state $|\psi\rangle$ results in a new state: $|\psi\rangle \to U|\psi\rangle$. For example, applying a 1-qubit gate on the q-th qubit of a quantum system updates its state vector as follows:

$$\begin{bmatrix} \alpha_{f(i)} \\ \alpha_{f(i)+2^q} \end{bmatrix} \to \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix} \times \begin{bmatrix} \alpha_{f(i)} \\ \alpha_{f(i)+2^q} \end{bmatrix}, \qquad (1)$$

where $\begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix}$ is the 2×2 unitary complex matrix that represents the 1-qubit gate, and $f(i) = 2^{q+1}\lfloor \frac{i}{2^q} \rfloor + (i \bmod 2^q)$ for all integers between 0 and $2^{n-1}-1$.

III. Hierarchical Partitioning for Simulation

We present our quantum simulation algorithm with hierarchical circuit partitioning.

Algorithm 1 Hierarchical partitioning of a quantum circuit $\mathcal{C}$ and the simulation algorithm for simulating the circuit starting with the input state *state*. The parameters L, R, G describe the distributed execution model with L local qubits, R regional qubits, and G global qubits.

```
1: function PARTITION($\mathcal{C}, L, R, G$)
2:     $stages$ = STAGE($\mathcal{C}, L, R, G$)
3:     $stagedKernels$ = []
4:     for $i = 0 \ldots |stages| - 1$ do
5:         $(\mathcal{C}_i, \mathcal{Q}) = stages[i]$
6:         $kernels_i$ = KERNELIZE($\mathcal{C}_i$)
7:         $stagedKernels[i] = (kernels_i, \mathcal{Q})$
8:     return $stagedKernels$
9: function EXECUTE($stagedKernels, state$)
10:    $\mathcal{P}$ = identity permutation
11:    $shards = [state]$
12:    for $i = 0 \ldots |stagedKernels| - 1$ do
13:        $(kernels_i, \mathcal{Q}) = stagedKernels[i]$
14:        $(shards, \mathcal{P})$ = SHARD($shards, \mathcal{Q}, \mathcal{P}$)
15:        parfor $shard$ in $shards$ do
16:            for $k$ in $kernels_i$ do
17:                LAUNCHKERNEL($k, shard$)
18: function SIMULATE($\mathcal{C}, state, L, R, G$)
19:    $stagedKernels$ = PARTITION($\mathcal{C}, L, R, G$)
20:    EXECUTE($stagedKernels, state$)
```

Algorithm 1 shows the pseudocode for our algorithms for quantum circuit simulation with hierarchical partitioning. The algorithm PARTITION takes as arguments the input circuit $\mathcal{C}$ and the architecture parameters consisting of the number of local, regional, and global qubits, L, R, G respectively. It partitions the input circuit $\mathcal{C}$ into stages using the STAGE function. Each

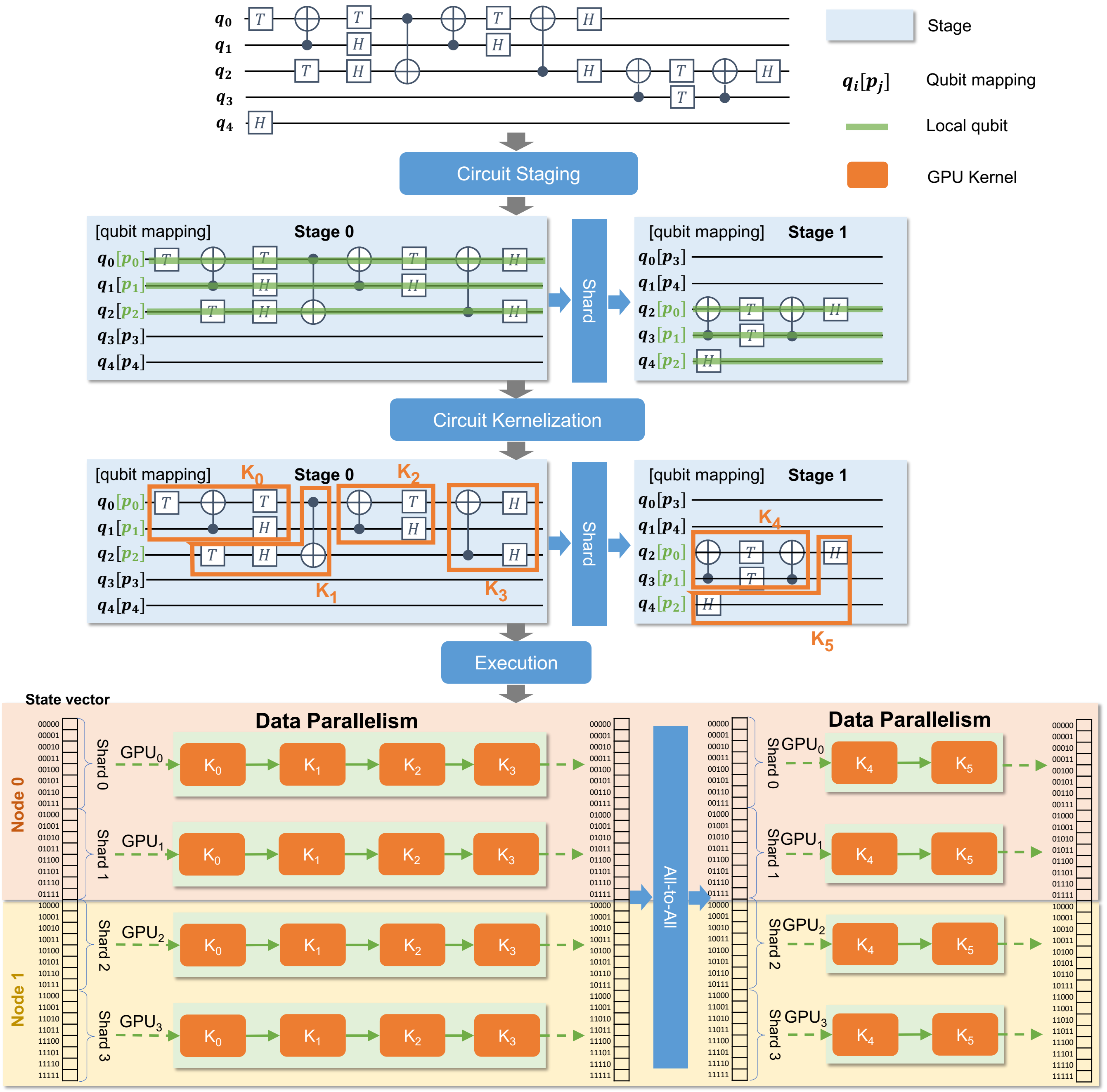


Fig. 1: An example application of circuit partitioning and execution. We stage the circuit so that qubits of each gate map to local qubits (i.e., green lines in each stage). The notation $q_i[p_j]$ indicates that the i-th logical qubit maps to the j-th physical qubit. The KERNELIZE algorithm then partitions the gates of each stage into kernels that provide for data parallelism.

stage consists of a subcircuit and a partition of the logical qubits of the (entire) circuit into sets of local, regional, and global qubits, such that the subcircuit can be simulated entirely locally by a single GPU, performing local memory accesses only. We describe the staging algorithm STAGE in Section IV.

After staging the circuit, the algorithm PARTITION, proceeds to kernelize each circuit by calling KERNELIZE on the subcircuit of each stage. The algorithm KERNELIZE further partitions the subcircuit for the stage into subcircuits each of which may be efficiently executed on a single GPU by taking advantage of data parallelism. The basic idea behind the algorithm is to group gates to amortize the cost of data parallelism on modern GPUs (e.g., due to cost of launching kernels) and do so without leading to an exponential blow-up of costs, because grouping gates can increase the cost exponentially in the number of qubits involved. We describe the kernelization algorithm KERNELIZE in Section V. The algorithm PARTITION completes by returning a list of stages, each of which consists of a list of kernels.

The algorithm EXECUTE takes a list of kernelized stages and input state consisting of a vector of amplitudes, and executes each stage in order. For each stage, the algorithm permutes the state vector and shards it into contiguous sections as demanded by the qubit partition $\mathcal{Q}$ for that stage. To minimize communication costs during this permutation and sharding process, the algorithm keeps track of the current permutation $\mathcal{P}$. The algorithm then considers each shard in parallel and applies the kernels of the stage to the shard sequentially. Because our algorithm staged and kernelized the circuit to maximize throughput for a distributed GPU architecture, each kernel can run efficiently on a single GPU, by applying each kernel to all the amplitudes in a shard of the state vector in a data parallel fashion. If sufficiently many GPUs are available, then each shard may be assigned to a GPU for parallel application. If, however, fewer GPUs are available, the shards may be stored in the shared DRAM at each node, and swapped in and out of the GPUs for execution.

Our simulation algorithm builds on top of our PARTITION and EXECUTE algorithms. Given the input circuit $\mathcal{C}$ and an input state *state*, along with architecture parameters L, R, G, our simulation algorithm SIMULATE starts by partitioning the input circuit into a list of kernelized stages. It then calls the algorithm EXECUTE to execute the stages.

Figure 1 shows an example application of the algorithm. We note that PARTITION does not depend on *state*. Both STAGE and KERNELIZE work for arbitrary input states.

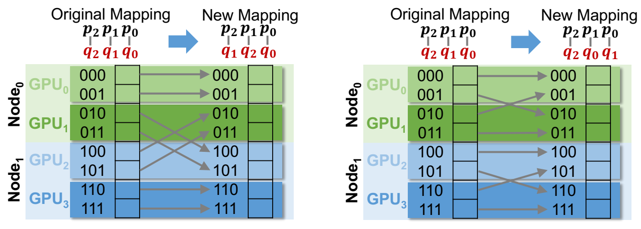
IV. CIRCUIT STAGING

We present an algorithm for (circuit) staging that partitions the circuit into *stages*, each of which is a contiguous subcircuit of the input circuit, along with a partition of the qubits into local, regional, and global sets such that each gate in the subcircuit of the stage operates on the local qubits.

Definition 1 (Local, regional, and global qubits). *For a quantum circuit simulation with n qubits, let q_i $(0 \le i \le n-1)$ be the i-th physical qubit. Let each shard include 2^L states, and let the DRAM of each node save 2^{L+R} states.*

- *The first L physical qubits (i.e., $q_0, ..., q_{L-1}$) are* local *qubits.*
- *The next R physical qubits (i.e., $q_L, ..., q_{L+R-1}$) are* regional *qubits.*
- *The final $G = n - L - R$ physical qubits (i.e., $q_{L+R}, ..., q_{n-1}$) are* global *qubits.*

We categorize physical qubits into local, regional, and global subsets based on the communications required to simulate the application of a general quantum gate to these qubits. First, applying a gate to a local qubit only requires accessing states within the same shard, and therefore avoids communications. Second, applying a gate to a regional qubit requires accessing states in different shards stored on the same compute node, which only requires inter-device (intra-node) communications. Finally, applying a gate to a global qubit requires states



(a) Sharding with inter-node communication.

(b) Sharding with intra-node (inter-GPU) communication.

Fig. 2: Sharding with different types of communication. The simulation has 1 local, 1 regional, and 1 global qubit. $p_i - q_j$ indicates that the i-th physical qubit is mapped to the j-th logical qubit. Inter-node communication is triggered if we update any global qubits (Figure 2a), and only intra-node communication is triggered otherwise (Figure 2b).

stored on different compute nodes and thus involves inter-node commutations.

Given this partition of local, regional, and global qubits, we aim to map logical qubits of a circuit to physical qubits in a way that avoids excessive communication. To minimize communication cost, we leverage a specific type of qubits in certain types of gates, called *insular qubits*.

Definition 2 (Insular Qubit). *For a single-qubit gate, the qubit is insular if the unitary matrix of the gate is diagonal or anti-diagonal, i.e., the non-zero entries are along the (anti) diagonal. For multi-qubit controlled-U gates*[1]*, all control qubits are insular*[2]*. All other qubits are non-insular.*

Our idea of insular qubits was inspired by *global gate specialization* introduced by Häner *et al.* [27]. Intuitively, for a single-qubit gate with an insular qubit, computing each output state only depends on one input state (since the gate's unitary matrix is diagonal or anti-diagonal), which allows Atlas to map insular qubits to regional and/or global physical qubits without introducing any extra communication. This property allows Atlas to only consider the non-insular qubits of quantum gates when mapping qubits.

Atlas staging algorithm (Algorithm 2) splits a quantum circuit's simulation into multiple stages, each of which includes a subcircuit and uses a different mapping from logical to physical qubits. Within each subcircuit, all non-insular qubits of all gates can only operate on *local* physical qubits. This approach avoids any inter-device or inter-node communications within a stage and only requires all-to-all communications between stages to perform qubit remapping.

We formalize circuit staging as a constrained optimization problem and design a binary integer linear programming (ILP)

[1]A *controlled-U* gate has some *control* qubits controlling a U gate (can be any unitary gate) on some *target* qubits. If at least one of the control qubits is $|0\rangle$, then the target remains unchanged. If all of the control qubits are $|1\rangle$, then the gate U is applied to the target qubits.

[2]In some controlled-U gates, any qubit can be chosen as the control qubit without changing the output. In such gates, all qubits are insular.

algorithm to discover a staging strategy that minimizes the total communication cost.

a) Circuit staging problem: Given an inter-node communication cost factor c, for a given input circuit $\mathcal{C}$ with n qubits and an integer s, Atlas circuit staging algorithm splits $\mathcal{C}$ into at most s stages $\mathcal{C}_0, ..., \mathcal{C}_{s-1}$ and determines a qubit partition $\mathcal{Q}$ of local/regional/global qubits for each stage, minimizing the total communication cost:

$$\sum_{i=1}^{s-1} \left(\left| \mathcal{Q}_i^{local} \setminus \mathcal{Q}_{i-1}^{local} \right| + c \cdot \left| \mathcal{Q}_i^{global} \setminus \mathcal{Q}_{i-1}^{global} \right| \right). \qquad (2)$$

where $\mathcal{Q}_i^{local}$ is the local qubit set of stage i and $\mathcal{Q}_i^{global}$ is the global qubit set of stage i. $\left| \mathcal{Q}_i^{local} \setminus \mathcal{Q}_{i-1}^{local} \right|$ is the number of local qubits that need to be updated, approximating the inter-GPU communication cost; $\left| \mathcal{Q}_i^{global} \setminus \mathcal{Q}_{i-1}^{global} \right|$ is the number of global qubits that need to be updated, approximating the extra inter-node communication cost. Although regional qubits do not appear directly in (2), they are critical for allowing the number of global qubits that need to be updated to be smaller than the local one. Without regional qubits, any inter-GPU communication will also be inter-node.

Let $\mathcal{G}$ denote the gates of the circuit $\mathcal{C}$, and let $\mathcal{E}$ denote their dependencies (adjacent gate pairs on the same qubit). Let $A_{q,k}$ and $B_{q,k}$ ($0 \le q < n$, $0 \le k < s$) denote whether the q-th logical qubit is mapped to a local/global physical qubit at the k-th stage, and let $F_{g,k}$ ($g \in \mathcal{G}$, $0 \le k < s$) denote if the gate g is finished by the end of the k-th stage. For each $A_{q,k}$ and $B_{q,k}$ ($k < s-1$), we also introduce two ancillary variables $S_{q,k}$ and $T_{q,k}$ that indicate whether the q-th logical qubit is updated from local to non-local or from global to non-global from the k-th to the $(k+1)$-th stage (i.e., $S_{q,k} = 1$ if and only if $A_{q,k} = 0$ and $A_{q,k+1} = 1$; $T_{q,k} = 1$ if and only if $B_{q,k} = 0$ and $B_{q,k+1} = 1$). Note that $A_{q,k}$, $B_{q,k}$, $F_{g,k}$, $S_{q,k}$, and $T_{q,k}$ are all binary variables. Minimizing the total cost of local-to-non-local and global-to-non-global updates with constraints yields the following objective:

$$\min \sum_{k=0}^{s-2} \sum_{q=0}^{n-1} (S_{q,k} + c \cdot T_{q,k}) \qquad (3)$$

subject to

$$A_{q,k+1} \le A_{q,k} + S_{q,k} \quad \forall q \in [0, n) \quad \forall k \in [0, s-1) \qquad (4)$$
$$B_{q,k+1} \le B_{q,k} + T_{q,k} \quad \forall q \in [0, n) \quad \forall k \in [0, s-1) \qquad (5)$$
$$F_{g,k} \le F_{g,k+1} \quad \forall g \in \mathcal{G} \quad \forall k \in [0, s-1) \qquad (6)$$
$$F_{g,k} \le F_{g,k-1} + A_{q,k} \quad q \text{ is a non-insular qubit of } g \qquad (7)$$
$$F_{g_1,k} \ge F_{g_2,k} \quad \forall (g_1, g_2) \in \mathcal{E} \quad \forall k \in [0, s) \qquad (8)$$
$$F_{g,s-1} = 1 \quad \forall g \in \mathcal{G} \qquad (9)$$
$$A_{q,k} + B_{q,k} \le 1 \quad \forall q \in [0, n) \quad \forall k \in [0, s) \qquad (10)$$
$$\sum_{q=0}^{n-1} A_{q,k} = L \quad \sum_{q=0}^{n-1} B_{q,k} = G \quad \forall k \in [0, s) \qquad (11)$$

We now describe these constraints. First, constraints 4, 5 and 6 can be derived from the definitions of $S_{q,k}$, $T_{q,k}$ and $F_{g,k}$. Second, constraint 7 is a locality constraint that a gate g can be executed at the k-th stage only if g's non-insular qubits q are all mapped to local physical qubits (i.e., $A_{q,k} = 1$). Third, constraint 8 represents a dependency constraint that all gates must be executed by following a topological order with respect to dependencies between them. Fourth, constraint 9 is derived from the completion constraint that all gates must be executed in the s stages. Next, constraint 10 declares that any qubit cannot be both local and global at the same time. Finally, constraint 11 specifies the hardware constraint that each stage has L local physical qubits and G global qubits.

Algorithm 2 Atlas circuit staging algorithm.

```
1: function STAGE($\mathcal{C}, L, R, G$)
2:     for $s = 1, 2, \dots$ do
3:         $status, A, G, F$ = SOLVEILP($\mathcal{C}, L, R, G, s$)
4:         if $status = feasible$ then
5:             Find $(\mathcal{C}_0, \mathcal{Q}_0), \dots, (\mathcal{C}_{s-1}, \mathcal{Q}_{s-1})$ based on $A$, $B$, $F$
6:             return $(\mathcal{C}_0, \mathcal{Q}_0), \dots, (\mathcal{C}_{s-1}, \mathcal{Q}_{s-1})$
```

b) ILP-based circuit staging.: For a given input circuit $\mathcal{C}$ and an integer s that specifies the maximum number of stages $\mathcal{C}$ can be partitioned into, Atlas sends the objective (i.e., Equation (3)) and all constraints (i.e., Equations (4) to (11)) to an off-the-shelf integer-linear-programming (ILP) solver, which returns an assignment for matrices A, B, F, S, T that minimizes the objective while satisfying all constraints. Atlas then stages the circuit $\mathcal{C}$ into $(\mathcal{C}_0, \mathcal{Q}_0), \dots, (\mathcal{C}_{s-1}, \mathcal{Q}_{s-1})$ based on A, B and F, where $\mathcal{C}_i$ is a subcircuit and $\mathcal{Q}_i$ is a qubit partition into local/regional/global qubit sets. Specifically, for a quantum gate $g \in \mathcal{G}$, g will be executed by the stage with index: $\min\{k | F_{g,k} = 1\}$. Atlas maps the q-th logical qubit to a local (physical) qubit at the k-th stage if $A_{q,k} = 1$, to a global qubit at the k-th stage if $B_{q,k} = 1$, and to a regional qubit if $A_{q,k} = B_{q,k} = 0$.

Algorithm 2 shows Atlas staging algorithm. It uses the ILP solver as a subroutine and returns the first feasible result.

Theorem 1 (Optimality of STAGE). *Algorithm 2 returns the minimum feasible number of stages.*

Proof. Algorithm 2 returns the first feasible result of the ILP solver, and the loop in line 2 ensures that the number of stages is minimized. The ILP constraints are the feasible constraints for stages, so Algorithm 2 returns the minimum feasible number of stages. □

On top of the minimum number of stages, the ILP minimizes the total communication cost (the ILP objective).

V. Circuit Kernelization

After partitioning an input circuit into multiple stages, Atlas must efficiently execute the gates of each stage on GPUs. GPU computations are organized as *kernels*, each of which is a function simultaneously executed by multiple hardware threads in a single-program-multiple-data (SPMD) fashion [28]. A straightforward approach to executing gates is to launch a kernel for each gate, which yields suboptimal performance due

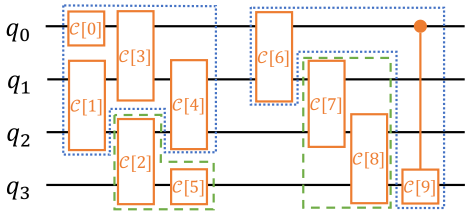


Fig. 3: Kernel examples. The two green dashed kernels satisfy Constraint 1. The two blue dotted kernels do not satisfy Constraint 1 and thus are not considered by the KERNELIZE algorithm.

to the low arithmetic intensity of each gate. Another extreme is to fuse all gates into a single kernel, which would exponentially increase the amount of computation.

To improve simulation performance on GPUs, Atlas uses a dynamic programming algorithm (Algorithm 3) to partition the gates of a stage into kernels. We formulate circuit kernelization as an optimization problem as follows.

Problem 1 (Optimal circuit sequence kernelization)**.** *Given a cost function* COST$(\cdot)$ *that maps one or multiple kernels (a sequence of gates) to their cost (i.e., a real number), the optimal circuit sequence kernelization problem takes as input a quantum circuit $\mathcal{C}$ represented as a sequence of gates, and returns a kernel sequence $\mathcal{K}_0, \ldots, \mathcal{K}_{b-1}$ that minimizes the total execution cost:*

$$\sum_{i=0}^{b-1} \text{COST}(\mathcal{K}_i) \tag{12}$$

such that each kernel consists of a contiguous segment of the gate sequence $\mathcal{C}$.

We note that summing up the cost of individual kernels (Equation (12)) is a faithful representation of the total cost, because on a GPU, each kernel may execute in parallel, but multiple kernels are executed sequentially.

Also, note that each kernel consisting of a contiguous segment of the gate sequence $\mathcal{C}$ is a conservative requirement. It is feasible to execute any kernel sequence that forms a topologically equivalent sequence to $\mathcal{C}$. However, there are exponentially many ways to consider kernels with non-contiguous segments. To consider them efficiently, we present a key constraint on the kernels of the KERNELIZE algorithm in Algorithm 3. We include a study of an algorithm that only considers contiguous kernels in the extended version of this paper [29].

Constraint 1 (Constraints on kernels)**.** *Each kernel $\mathcal{K}$ considered by the* KERNELIZE *algorithm satisfies the following constraints:*

1) *Weak convexity: $\forall j_1 < j_2 < j_3$, if $\mathcal{C}[j_1] \in \mathcal{K}$, $\mathcal{C}[j_2] \notin \mathcal{K}, \mathcal{C}[j_3] \in \mathcal{K}$, then* QUBITS$(\mathcal{C}[j_1]) \cap$ QUBITS$(\mathcal{C}[j_2]) \cap$ QUBITS$(\mathcal{C}[j_3]) = \emptyset$. *Informally, weak convexity requires that for any three gates $\mathcal{C}[j_1], \mathcal{C}[j_2], \mathcal{C}[j_3]$, if only the middle gate is not in the kernel, then they cannot share a qubit.*

Algorithm 3 The KERNELIZE algorithm. The operators "+" on lines 13 and 14 mean adding the cost and appending the kernel set to the sequence. We maintain the order in place in line 11 (replacing $\mathcal{K} \setminus \{\mathcal{C}[i]\}$ with $\mathcal{K}$) if the monotonicity constraint in Constraint 1 applies to $\mathcal{K}$, or move $\mathcal{K}$ to the end of the kernel set otherwise. We require $\tilde{\kappa}$ to be a suffix of κ' in line 13.

```
1:  function KERNELIZE($\mathcal{C}$)
2:      Input: A quantum circuit $\mathcal{C}$ represented as a sequence of gates.
3:      Output: A sequence of kernels.
4:      // Suppose that we have an ordered kernel set $\kappa$ in the first $i$ gates, $DP[i,\kappa]$ stores the minimum cost to kernelize the first $i$ gates except for the gates in $\kappa$ and the corresponding kernel sequence.
5:      $DP[i,\kappa] = (\infty, [])$ for all $i, \kappa$
6:      $DP[0,\emptyset] = (0, [])$
7:      for $i = 0$ to $|\mathcal{C}| - 1$ do
8:          for each $\kappa$ such that $\exists \mathcal{K}$, $\mathcal{C}[i] \in \mathcal{K} \in \kappa$ and $\mathcal{K}$ satisfies Constraint 1 do
9:              $\tilde{\kappa} = \kappa \setminus \{\mathcal{K}\}$
10:             if $|\mathcal{K}| > 1$ then
11:                 $DP[i+1,\kappa] = DP[i, \tilde{\kappa} \cup \{\mathcal{K} \setminus \{\mathcal{C}[i]\}\}]$
12:             else // $\mathcal{K} = \{\mathcal{C}[i]\}$
13:                 $DP[i+1,\kappa] = \min_{\kappa' \supseteq \tilde{\kappa}} \{DP[i,\kappa'] + \text{COST}(\kappa' \setminus \tilde{\kappa})\}$
14:     $DP_{best} = \min_\kappa \{DP[|\mathcal{C}|,\kappa] + \text{COST}(\kappa)\}$
15:     return $DP_{best}.kernels$
```

2) *Monotonicity: for any gate $\mathcal{C}[j] \notin \mathcal{K}$, if $\mathcal{C}[j]$ shares a qubit with $\mathcal{K} \downarrow j$ then* QUBITS$(\mathcal{K}) =$ QUBITS$(\mathcal{K} \downarrow j)$ *where $\mathcal{K} \downarrow j = \mathcal{K} \cap \{\mathcal{C}[0], \mathcal{C}[1], \ldots, \mathcal{C}[j-1]\}$. Informally, if we decide to exclude a gate from $\mathcal{K}$ while it shares a qubit with $\mathcal{K}$, then we fix the qubit set of $\mathcal{K}$ from that gate on.*

In Figure 3, the blue dotted kernel on the left violates weak convexity because $\mathcal{C}[1]$, $\mathcal{C}[2]$ and $\mathcal{C}[4]$ share q_2, and only $\mathcal{C}[2]$ is not in the kernel. If we allow this kernel to be considered, it will be mutually dependent with the kernel containing $\mathcal{C}[2]$, yielding no feasible results. The blue dotted kernel on the right does not violate weak convexity, but it violates monotonicity. $\mathcal{C}[7]$ is excluded from the kernel while sharing the qubit q_1 with the kernel, so the qubit set of the kernel should be fixed to $\{q_0, q_1\}$. So it cannot include $\mathcal{C}[9]$. In fact, including $\mathcal{C}[9]$ in this kernel causes mutual dependency with the kernel $\{\mathcal{C}[7], \mathcal{C}[8]\}$.

Constraint 1 ensures the kernels to form a valid sequence in Theorem 2.

Theorem 2 (Correctness of the KERNELIZE algorithm)**.** *Algorithm 3 returns a kernel sequence such that concatenating the kernels forms a sequence topologically equivalent to $\mathcal{C}$.*

Proof. Proof by induction on i that for each $DP[i,\kappa].cost < \infty$, appending κ to $DP[i,\kappa].kernels$ results in the sequence $\mathcal{C}_\kappa \downarrow i$ which is topologically equivalent to a the sequence $\mathcal{C}[0], \mathcal{C}[1], \ldots, \mathcal{C}[i-1]$ (denote this sequence $\mathcal{C} \downarrow i$). This holds for $i = 0$ where $DP[i,\emptyset].kernels$ and $\mathcal{C} \downarrow 0$ are both empty.

Suppose the induction hypothesis holds for i. For $i+1$, for any $\mathcal{K}$ satisfying Constraint 1, if $|\mathcal{K}| = 1$, line 13 in the algorithm makes the order of appending κ to $DP[i+1,\kappa].kernels$ the same as the order of appending κ' to $DP[i,\kappa'].kernels$ for some κ' when $\tilde{\kappa}$ is a suffix of κ', so we can directly apply the induction hypothesis to prove it for $i+1$.

Suppose $|\mathcal{K}| > 1$. If the monotonicity constraint in Constraint 1 applies to $\mathcal{K}$, line 11 just inserts $\mathcal{C}[i]$ to the end of $\mathcal{K}$ in $\mathcal{C}_\kappa \downarrow i$. So it suffices to prove that $\mathcal{C}[i]$ does not depend on any gate after the end of $\mathcal{K}$ in $\mathcal{C}_\kappa \downarrow i$, i.e., $\mathcal{C}[i]$ does not share any qubit with these gates.

By monotonicity,

$$\textsc{Qubits}(\mathcal{K}) = \textsc{Qubits}(\mathcal{K} \downarrow i). \tag{13}$$

For any gate $\mathcal{C}[j_2]$ after the last gate of $\mathcal{K}$ in the sequence $\mathcal{C}_\kappa \downarrow i$ before the insertion, by weak convexity, $\forall j_1 < j_2 < i$ such that $\mathcal{C}[j_1] \in \mathcal{K}$ (we know that $\mathcal{C}[j_2] \notin \mathcal{K}, \mathcal{C}[i] \in \mathcal{K}$), $\textsc{Qubits}(\mathcal{C}[j_1]) \cap \textsc{Qubits}(\mathcal{C}[j_2]) \cap \textsc{Qubits}(\mathcal{C}[i]) = \emptyset$. Because all gates in $\mathcal{K}$ are before $\mathcal{C}[j_2]$ in the sequence $\mathcal{C}_\kappa \downarrow i$ before the insertion,

$$\textsc{Qubits}(\mathcal{K} \downarrow i) \cap \textsc{Qubits}(\mathcal{C}[j_2]) \cap \textsc{Qubits}(\mathcal{C}[i]) = \emptyset \tag{14}$$

where $\mathcal{K} \downarrow i$ is the kernel $\mathcal{K}$ before the insertion of $\mathcal{C}[i]$.

By Equation (13),

$$\textsc{Qubits}(\mathcal{C}[i]) \subseteq \textsc{Qubits}(\mathcal{K} \downarrow i). \tag{15}$$

Plugging this into Equation (14),

$$\textsc{Qubits}(\mathcal{C}[j_2]) \cap \textsc{Qubits}(\mathcal{C}[i]) = \emptyset \tag{16}$$

for any gate $\mathcal{C}[j_2]$ after the last gate of $\mathcal{K}$ in the sequence $\mathcal{C}_\kappa \downarrow i$ before the insertion.

So we can safely insert $\mathcal{C}[i]$ to the end of $\mathcal{K}$ in $\mathcal{C}_\kappa \downarrow i$ without breaking the topological equivalence.

If the monotonicity constraint in Constraint 1 does not apply to $\mathcal{K}$, line 11 moves $\mathcal{K}$ to the end of $\mathcal{C}_\kappa \downarrow i$. Because the monotonicity constraint does not apply to $\mathcal{K}$, for any gate $\mathcal{C}[j_0] \in \mathcal{K}$, for any gate $\mathcal{C}[j] \notin \mathcal{K}$ after $\mathcal{C}[j_0]$ in the original sequence $\mathcal{C}_\kappa \downarrow i$, $\mathcal{C}[j]$ and $\mathcal{C}[j_0]$ do not share any qubits. So we can safely move the entire kernel $\mathcal{K}$ to the end without breaking the topological equivalence.

In all cases, we have proved that $\mathcal{C}_\kappa \downarrow (i+1)$ is topologically equivalent to $\mathcal{C} \downarrow (i+1)$. So $\mathcal{C}_\kappa \downarrow |\mathcal{C}|$ is topologically equivalent to $\mathcal{C} \downarrow |\mathcal{C}|$, which proves that any state in line 14 in the algorithm returns a kernel sequence such that concatenating the kernels forms a sequence topologically equivalent to $\mathcal{C}$. □

Theorem 3 (Constraint 1 allows all contiguous kernels)**.** *Any kernel of a contiguous segment of the gate sequence $\mathcal{C}$ satisfies Constraint 1.*

Proof. For any kernel $\mathcal{K}$ of a contiguous segment of gates, for any $j_1 < j_2 < j_3$ such that $\mathcal{C}[j_1] \in \mathcal{K}, \mathcal{C}[j_2] \notin \mathcal{K}$, we know that $\mathcal{C}[j_3] \notin \mathcal{K}$ by contiguity. So weak convexity holds, and monotonicity holds because $\mathcal{K} \downarrow j_2 = \mathcal{K}$. □

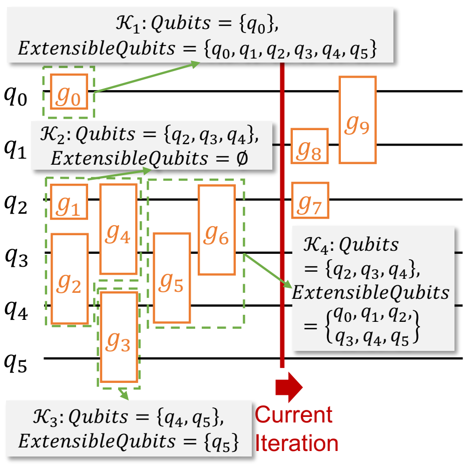

$\mathcal{K}_1$: $Qubits = \{q_0\}$,
$ExtensibleQubits = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
q_0
q_1
q_2
q_3
q_4
q_5
g_0
$\mathcal{K}_2$: $Qubits = \{q_2, q_3, q_4\}$,
$ExtensibleQubits = \emptyset$
g_1
g_2
g_3
g_4
g_5
g_6
g_7
g_8
g_9
$\mathcal{K}_4$: $Qubits = \{q_2, q_3, q_4\}$,
$ExtensibleQubits = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
$\mathcal{K}_3$: $Qubits = \{q_4, q_5\}$,
$ExtensibleQubits = \{q_5\}$
Current Iteration

Fig. 4: An example DP state in the implementation. The circuit sequence is $\mathcal{C} = [g_0, g_1, \ldots, g_9]$.

Algorithm 4 Maintaining the extensible qubit set $\textsc{ExtQ}$ for each kernel after each iteration of the for loop in line 8 of Algorithm 3.

```
 1: // $\mathcal{C}[i] \in \mathcal{K} \in \kappa$
 2: if $\mathcal{K} = \{\mathcal{C}[i]\}$ then // a new kernel
 3:     $\textsc{ExtQ}(\mathcal{K}, i+1) = \textsc{AllQubits}$
 4: else
 5:     $\textsc{ExtQ}(\mathcal{K}, i+1) = \textsc{ExtQ}(\mathcal{K}, i)$
 6: for $\mathcal{K}' \in (\kappa \setminus \mathcal{K})$ do
 7:     if $\textsc{ExtQ}(\mathcal{K}', i) = \textsc{AllQubits}$ then
 8:         if $\textsc{Qubits}(\mathcal{C}[i]) \cap \textsc{Qubits}(\mathcal{K}') \neq \emptyset$ then
 9:             $\textsc{ExtQ}(\mathcal{K}', i+1) = \textsc{Qubits}(\mathcal{K}') \setminus \textsc{Qubits}(\mathcal{C}[i])$
10:         else
11:             $\textsc{ExtQ}(\mathcal{K}', i+1) = \textsc{AllQubits}$
12:     else
13:         $\textsc{ExtQ}(\mathcal{K}', i+1) = \textsc{ExtQ}(\mathcal{K}', i) \setminus \textsc{Qubits}(\mathcal{C}[i])$
```

VI. Implementation

A. Reducing Size of DP State in Kernelize

In a quantum circuit, the number of gates is typically orders of magnitude greater than the number of qubits. Therefore, instead of tracking the gates for each kernel in Algorithm 3, we maintain the qubit set as well as an *extensible qubit set* for each kernel.

Definition 3 (Extensible qubit for a kernel)**.** *For a kernel $\mathcal{K}$ at position i, i.e., $\mathcal{K} \downarrow i$, a qubit q is extensible if and only if adding a gate operating on qubit q to $\mathcal{K} \downarrow i$ satisfies Constraint 1. Formally, a qubit q is extensible for $\mathcal{K} \downarrow i$ if and only if both of the following hold:*

1) *Weak convexity: $\forall j_1 < j_2 < i$, if $\mathcal{C}[j_1] \in \mathcal{K} \downarrow i$, $\mathcal{C}[j_2] \notin \mathcal{K} \downarrow i$, then $q \notin \textsc{Qubits}(\mathcal{C}[j_1]) \cap \textsc{Qubits}(\mathcal{C}[j_2])$.*
2) *Monotonicity: for any gate $\mathcal{C}[j] \notin \mathcal{K} \downarrow i$, if $j < i$ and $\mathcal{C}[j]$ shares a qubit with $\mathcal{K} \downarrow j$, then $q \in \textsc{Qubits}(\mathcal{K} \downarrow j)$.*

This definition is a direct characterization of Constraint 1: adding the gate $\mathcal{C}[i]$ to a kernel $\mathcal{K}$ satisfies Constraint 1 if and only if all qubits of $\mathcal{C}[i]$ are extensible for $\mathcal{K} \downarrow i$. For example, the extensible qubit set is $\{q_5\}$ for $\mathcal{K}_3$ in Figure 4 – we can only add gates operating only on this qubit to $\mathcal{K}_3$ to satisfy Constraint 1.

We append the code fragment of Algorithm 4 to the end of each iteration of the for loop in line 8 of Algorithm 3 to maintain the extensible qubit set for each kernel.

Theorem 4. *Algorithm 4 correctly computes the extensible qubit set for each kernel.*

Proof. Proof by induction on the position i. For the kernel $\mathcal{K}$ such that $\mathcal{C}[i] \in \mathcal{K}$, if $\mathcal{K} = \{\mathcal{C}[i]\}$, all qubits are extensible by Definition 3. Otherwise, $\textsc{ExtQ}(\mathcal{K}, i)$ is computed in the previous iteration.

$\forall \mathcal{K}' \in (\kappa \setminus \{\{\mathcal{C}[i]\}\})$, suppose $\textsc{ExtQ}(\mathcal{K}', i)$ is computed correctly.

Case 0. Consider the kernel $\mathcal{K}$ such that $\mathcal{C}[i] \in \mathcal{K}$ while $|\mathcal{K}| > 1$. In the condition for weak convexity, $\mathcal{C}[j_2]$ cannot be $\mathcal{C}[i]$ because it requires $\mathcal{C}[j_2] \notin \mathcal{K} \downarrow (i+1)$, so this constraint is exactly the same as weak convexity for $\mathcal{K} \downarrow i$.

If monotonicity applies to $\mathcal{K} \downarrow i$, it still applies to $\mathcal{K} \downarrow (i+1)$, and $\textsc{Qubits}(\mathcal{K} \downarrow (i+1)) = \textsc{Qubits}(\mathcal{K} \downarrow i)$; if monotonicity does not apply to $\mathcal{K} \downarrow i$, $\mathcal{C}[j]$ cannot be $\mathcal{C}[i]$ in the condition because it requires $\mathcal{C}[j] \notin \mathcal{K} \downarrow (i+1)$, so it still does not apply to $\mathcal{K} \downarrow (i+1)$. So $\textsc{ExtQ}(\mathcal{K}, i+1) = \textsc{ExtQ}(\mathcal{K}, i)$.

Now we case over each kernel $\mathcal{K}' \in (\kappa \setminus \mathcal{K})$. If monotonicity applies to $\mathcal{K}' \downarrow i$, there will be a gate $\mathcal{C}[j] \notin \mathcal{K}' \downarrow i$ satisfying $j < i$ and $\textsc{Qubits}(\mathcal{C}[j]) \cup \textsc{Qubits}(\mathcal{K}' \downarrow j) \neq \emptyset$, causing all qubits in this intersection to be not extensible for $\mathcal{K} \downarrow i$ by weak convexity (picking $j_2 = j$). So if the condition in line 7 of Algorithm 4 holds, i.e., all qubits are extensible for $\mathcal{K}' \downarrow i$, then monotonicity does not apply to $\mathcal{K}' \downarrow i$.

Case 1. If the condition in line 8 of Algorithm 4 does not hold, i.e., $\textsc{Qubits}(\mathcal{C}[i]) \cap \textsc{Qubits}(\mathcal{K}' \downarrow (i+1)) = \emptyset$, then monotonicity does not apply to $\mathcal{K}' \downarrow (i+1)$. Consider the additional constraint of weak convexity for $\mathcal{K}' \downarrow (i+1)$ than weak convexity for $\mathcal{K}' \downarrow i$, that is, $\forall j_1 < j_2 = i$, if $\mathcal{C}[j_1] \in \mathcal{K}' \downarrow (i+1), \mathcal{C}[i] \notin \mathcal{K}' \downarrow (i+1)$, then $q \notin \textsc{Qubits}(\mathcal{C}[j_1]) \cap \textsc{Qubits}(\mathcal{C}[i])$. However, $\textsc{Qubits}(\mathcal{C}[j_1]) \cap \textsc{Qubits}(\mathcal{C}[i]) = \emptyset$, so $\textsc{ExtQ}(\mathcal{K}', i+1) = \textsc{AllQubits}$.

Case 2. If the condition in line 8 of Algorithm 4 holds, i.e., $\textsc{Qubits}(\mathcal{C}[i]) \cap \textsc{Qubits}(\mathcal{K}' \downarrow (i+1)) \neq \emptyset$, then monotonicity applies to $\mathcal{K}' \downarrow (i+1)$ because $\mathcal{C}[i] \notin \mathcal{K}' \downarrow (i+1)$ and $\mathcal{C}[i]$ shares a qubit with $\mathcal{K}' \downarrow (i+1)$. So only qubits in $\textsc{Qubits}(\mathcal{K}' \downarrow (i+1))$ can possibly be extensible.

$\forall q \in \textsc{Qubits}(\mathcal{K}' \downarrow (i+1))$, q is extensible if and only if it satisfies weak convexity. Because all qubits were extensible for $\mathcal{K}' \downarrow i$, we only need to consider the constraint on $\forall j_1 < j_2 = i$. Because $q \in \textsc{Qubits}(\mathcal{K}' \downarrow (i+1))$, we can always find some $j_1 < i$ satisfying $q \in \textsc{Qubits}(\mathcal{C}[j_1])$ and $\mathcal{C}[j_1] \in \mathcal{K}' \downarrow (i+1)$, so q is extensible if and only if $q \notin \textsc{Qubits}(\mathcal{C}[i])$. Therefore, $\textsc{ExtQ}(\mathcal{K}', i+1) = \textsc{Qubits}(\mathcal{K}' \downarrow (i+1)) \setminus \textsc{Qubits}(\mathcal{C}[i])$.

Case 3. If the condition in line 7 of Algorithm 4 does not hold, i.e., not all qubits are extensible for $\mathcal{K}' \downarrow i$, we know that line 9 has been executed for some $i' < i$ (so monotonicity applies to $\mathcal{K}' \downarrow (i'+1)$), so $\textsc{ExtQ}(\mathcal{K}', i'+1) \subseteq \textsc{Qubits}(\mathcal{K}' \downarrow i')$. Because line 13 of Algorithm 4 ensures $\textsc{ExtQ}(\mathcal{K}', i''+1) \subseteq \textsc{ExtQ}(\mathcal{K}', i'')$ for all $i' < i'' < i$, we have $\textsc{ExtQ}(\mathcal{K}', i) \subseteq \textsc{Qubits}(\mathcal{K}' \downarrow i') \subseteq \textsc{Qubits}(\mathcal{K}' \downarrow i)$.

For each qubit q, if it is not in $\textsc{ExtQ}(\mathcal{K}', i)$, it cannot be in $\textsc{ExtQ}(\mathcal{K}', i+1)$ because Definition 3 is more restrictive when i increases; for each $q \in \textsc{ExtQ}(\mathcal{K}', i)$, $q \in \textsc{ExtQ}(\mathcal{K}', i+1)$ if and only if it satisfies the additional weak convexity constraint $\forall j_1 < j_2 = i$. Because $\textsc{ExtQ}(\mathcal{K}', i) \subseteq \textsc{Qubits}(\mathcal{K}' \downarrow i)$, we can always find $j_1 < i$ such that $q \in \textsc{Qubits}(\mathcal{C}[j_1])$ and $\mathcal{C}[j_1] \in \mathcal{K} \downarrow i$, so q is extensible if and only if $q \notin \textsc{Qubits}(\mathcal{C}[i])$. Therefore, we conclude that $\textsc{ExtQ}(\mathcal{K}', i+1) = \textsc{ExtQ}(\mathcal{K}', i) \setminus \textsc{Qubits}(\mathcal{C}[i])$.

In all of these cases, we have proved the correctness of $\textsc{ExtQ}(\mathcal{K}', i+1)$ for any $\mathcal{K}'$. So Algorithm 4 correctly computes the extensible qubit set for each kernel. □

Because we cannot add any gates to kernels with an empty extensible qubit set such as $\mathcal{K}_2$, we compute their cost and remove them from κ in line 13 of Algorithm 3. Because line 9 of Algorithm 4 restricts the extensible qubit set to be a subset of the qubit set of $\mathcal{K}'$, we no longer need to keep track of the qubit set because it will no longer change.

Theorem 5 (Time complexity of the $\textsc{Kernelize}$ algorithm)**.** *With the extensible qubit sets maintained in Algorithm 4, the* $\textsc{Kernelize}$ *algorithm runs in* $O\left(\left(\frac{3.2n}{\ln n}\right)^n \cdot |\mathcal{C}|\right)$*, where n is the number of qubits.*

Proof. For each iteration i in Algorithm 3, let us compute the maximum number of kernel sets κ that Atlas must consider.

We observe that if we consider the qubit set for each kernel with the extensible qubit set being all qubits, and the extensible qubit set for other kernels, any two of these sets do not intersect. This can be proven by induction on i and the fact that whenever we add $\textsc{Qubits}(\mathcal{C}[i])$ to $\mathcal{K}$, we effectively remove these qubits from all other sets.

Because kernels with empty extensible qubit sets are removed from κ, there are at most n kernels in κ and the number of ways to partition all qubits into disjoint qubit sets is at most the Bell number B_{n+1}, which is less than $\left(\frac{0.792(n+1)}{\ln(n+2)}\right)^{n+1}$ [30]. Each qubit set can be either the qubit set or the extensible qubit set of a kernel, so we need to multiply this number by at most 2^n; each kernel can be either fusion or shared-memory, so we need to multiply this number by another 2^n. With the extensible qubit sets maintained, the order of kernels does not affect the DP transitions, so the number of different kernel sets is at most $4^n B_{n+1}$. Each iteration of the for loop in line 8 of Algorithm 3 takes polynomial time, so the time per iteration i is at most $O\left(\left(\frac{3.2n}{\ln n}\right)^n\right)$. There are $|\mathcal{C}|$ iterations, so the total time complexity is $O\left(\left(\frac{3.2n}{\ln n}\right)^n \cdot |\mathcal{C}|\right)$. □

We also implemented other optimizations in the $\textsc{Kernelize}$ algorithm. See the extended version [29] for details.

B. Cost Function in $\textsc{Kernelize}$

The cost function used in Equation (12) should faithfully represent the running time of each kernel. Inspired by HyQuas [26], we use two approaches to execute each kernel:

TABLE I: Our benchmark circuits and their size (number of gates).

Circuit Name	Description	Number of qubits 28	29	30	31	32	33	34	35	36
ae	amplitude estimation	514	547	581	616	652	689	727	766	806
dj	Deutsch–Jozsa algorithm	82	85	88	91	94	97	100	103	106
ghz	GHZ state	28	29	30	31	32	33	34	35	36
graphstate	graph state	56	58	60	62	64	66	68	70	72
ising	Ising model	302	313	324	335	346	357	368	379	390
qft	quantum Fourier transform	406	435	465	496	528	561	595	630	666
qpeexact	exact quantum phase estimation	432	463	493	524	559	593	628	664	701
qsvm	quantum support vector machine	274	284	294	304	314	324	334	344	354
su2random	SU2 ansatz with random parameters	1246	1334	1425	1519	1616	1716	1819	1925	2034
vqc	variational quantum classifier	1873	1998	2127	2260	2397	2538	2683	2832	2985
wstate	W state	109	113	117	121	125	129	133	137	141

1) *Fusion*: Fuse all gates in a kernel into a single gate by pre-computing the product of the gate matrices and executing that single gate using cuQuantum [31]. The cost function maps the number of qubits in the kernel to a constant, reflecting the running time of a matrix multiplication of that size.
2) *Shared-memory*: Load the state vector into GPU shared memory in batches and execute the gates one by one instead of fusing them to a single matrix. [3] The cost function is $\alpha + \sum_{g \in \mathcal{K}} \textsc{Cost}(g)$, where α is a constant that corresponds to the time to load a micro-batch of state coefficients into GPU shared memory, and $\textsc{Cost}(g)$ is the time to simulate the application of the gate g in the GPU shared memory.

The way to determine these constant values is described in Section VII-A.

In addition to the qubit set and the extensible qubit set, we add a Boolean tag for the type of each kernel (fusion or shared-memory) in KERNELIZE, and maintain the cost of the kernel during the DP algorithm according to the type. Whenever a new kernel is created, we generate two copies of DP states with the new kernel's type being fusion or shared-memory correspondingly.

C. Atlas

We built our system Atlas on top of FlexFlow [32], a distributed multi-GPU system for DNN training. Atlas uses the mapper interface of FlexFlow to distribute the state vector across the DRAM and GPU device memory of a GPU cluster, and uses Legion [33], FlexFlow's underlying task-based runtime, to launch simulation tasks on CPUs and GPUs and automatically overlap communications with computations. Furthermore, Atlas uses the NCCL library [34] to perform inter-GPU communication for sharding. We set the cost factor of the inter-node communication in the circuit staging algorithm $c = 3$ in Equation (2).

[3]Shared-memory kernels correspond to `SHM-GROUPING` in HyQuas. Similar to HyQuas, to improve the I/O efficiency of shared-memory kernels, we require the three least significant qubits of the state vector to be in all shared-memory kernels. As a result, each state vector load consists of at least $2^3 = 8$ complex numbers (i.e., 128 bytes as each complex number consists of 2 double-precision floating-point numbers).

Atlas is publicly available as an open-source project [35] and also in the artifact supporting this paper [36].

VII. Evaluation

A. Experimental Setup

We use the Perlmutter supercomputer [37] to evaluate Atlas. Each compute node is equipped with an AMD EPYC 7763 64-core 128-thread processor, 256 GB DRAM, and four NVIDIA A100-SXM4-40GB GPUs. The nodes are connected with HPE Slingshot 200 Gb/s interconnects. The programs are compiled using GCC 12.3.0, CUDA 12.2, and NCCL 2.19.4.

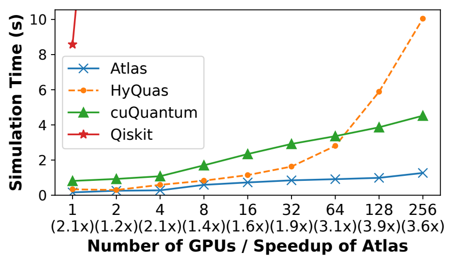
The cost function used in KERNELIZE has some constants that require benchmarking on the GPU. For fusion kernels, we measure their execution time with different numbers of qubits. For shared-memory kernels, we measure the run time of an empty shared-memory kernel to estimate the cost of loading a state vector to GPU shared memory, and profile the run times for different types of gates using the GPU shared memory.

a) Benchmarks: We collect 11 types of scalable quantum circuits from the MQT Bench [38] and NWQBench [39], where we can select the desired number of qubits for each circuit type. This allows us to conduct a weak-scaling evaluation, where we can compare the performance of Atlas and existing GPU-based quantum circuit simulators using different numbers of GPUs. Table I summarizes the circuits used in our evaluation.

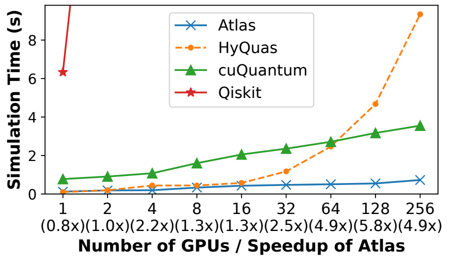
b) Preprocessing circuits: To speedup simulation, we partition each circuit by (1) splitting it into stages using the STAGE algorithm, and (2) kernelizing each stage using the KERNELIZE algorithm. This preprocessing needs to be done once per circuit and finishes in 7.2 seconds on average for each circuit in a single thread on an Intel Xeon W-1350 @ 3.30GHz CPU. In our evaluation, we use the PuLP library [40] with the HiGHS solver [41] for ILP. The ILP-based STAGE algorithm takes 3.3 seconds on average, and KERNELIZE takes 3.9 seconds on average. A detailed analysis of the running time of KERNELIZE is included in the extended version [29].

B. End-to-end Performance

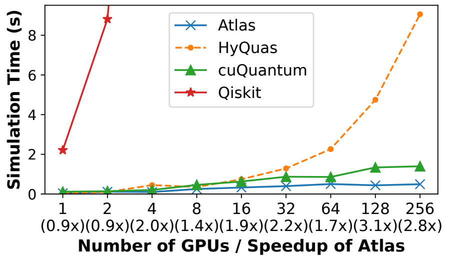
We first compare Atlas against existing distributed GPU-based quantum circuit simulators, including HyQuas [26], cuQuantum [31], and Qiskit [19]. UniQ [42] is extremely



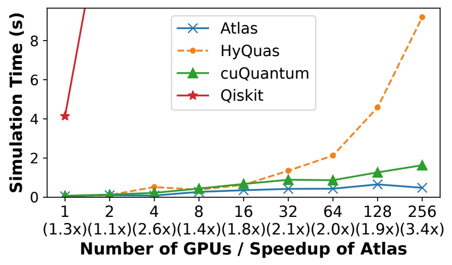
(a) ae



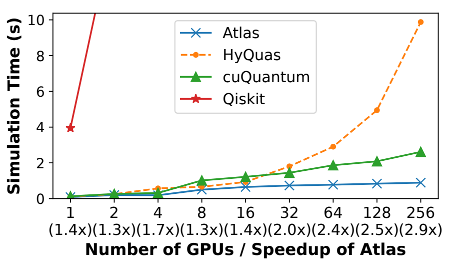
(b) qft



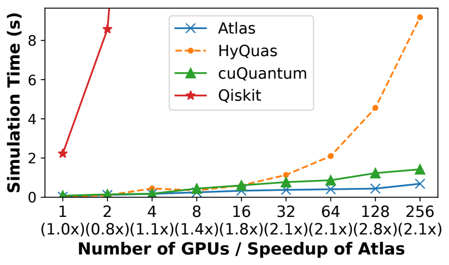
(c) graphstate



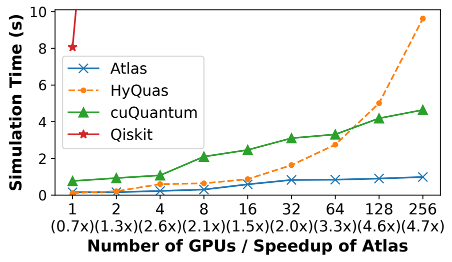
(d) wstate



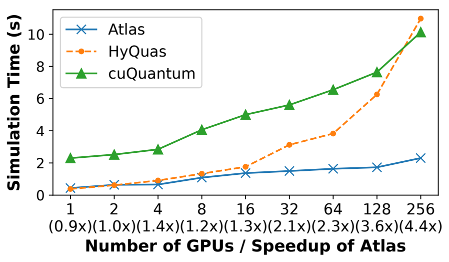
(e) qsvm



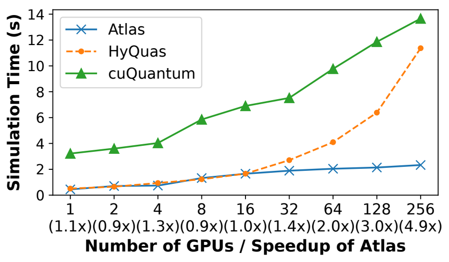
(f) ghz



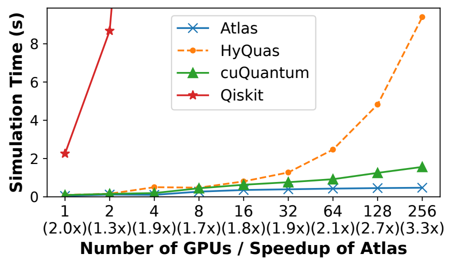
(g) qpeexact



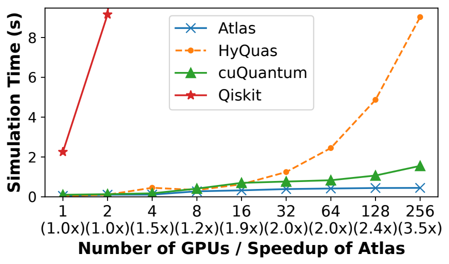
(h) su2random



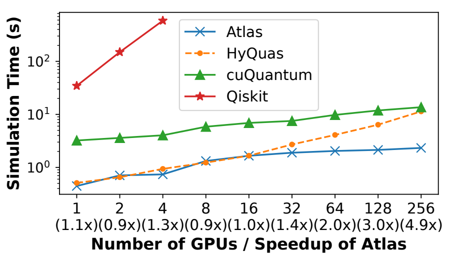
(i) vqc



(j) ising



(k) dj



(l) vqc (log scale)

Fig. 5: Weak scaling of Atlas, HyQuas, cuQuantum, and Qiskit with 28 local qubits as the number of global qubits increases from 0 (on 1 GPU) to 8 (on 256 GPUs). Qiskit is slow and usually does not fit into our charts.

similar to HyQuas in state-vector-based GPU-based quantum circuit simulation, so we do not compare with UniQ. Both Qiskit and cuQuantum use the Qiskit Python interface as the frontend. cuQuantum uses Aer's cuStateVec integration (cusvaer) to simulate quantum circuits, and Qiskit directly uses its native Aer GPU backend. All baselines perform state-vector-based simulation and directly store the entire state vector on GPUs. While Atlas also supports more scalable quantum circuit simulation by offloading the state vector to a much larger CPU DRAM, to conduct a fair comparison, in this experiment, we disable Atlas' DRAM offloading and directly store the entire state vector on GPUs. We further evaluate Atlas' DRAM offloading performance in Section VII-C.

Figure 5 shows the end-to-end simulation performance of the 11 circuit families on up to 256 GPUs (on 64 nodes). We use 28 local qubits and increase the number of non-local qubits from 0 (on 1 GPU) to 8 (on 256 GPUs). The number of regional qubits is at most 2 (there are 4 GPUs in each node). Qiskit is slow, so we only evaluate it on up to 4 GPUs and present a log-scale plot in Figure 5l.

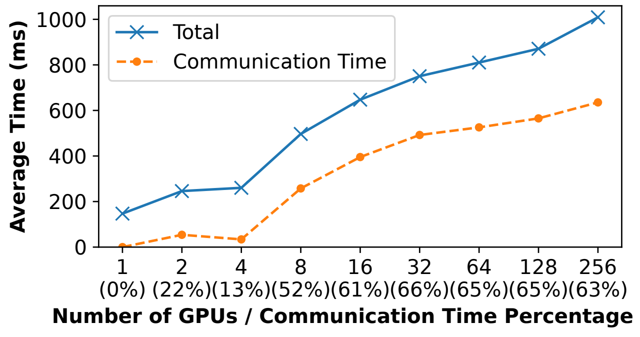


Fig. 6: Simulation time breakdown: The average communication time and its percentage in the average total simulation time of Atlas.

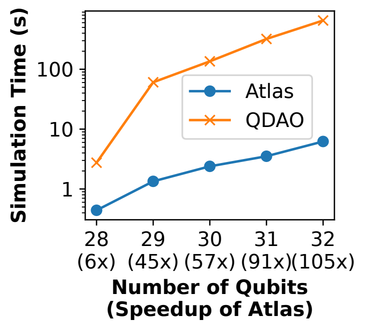


Fig. 7: Atlas outperforms QDAO by 61× on average. Log-scale simulation time (single GPU) with DRAM offloading for `qft` circuits.

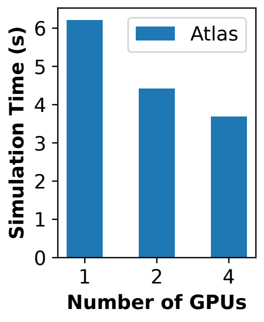


Fig. 8: DRAM offloading scales. Atlas simulation time of a 32-qubit `qft` circuit with 1, 2, and 4 GPUs.

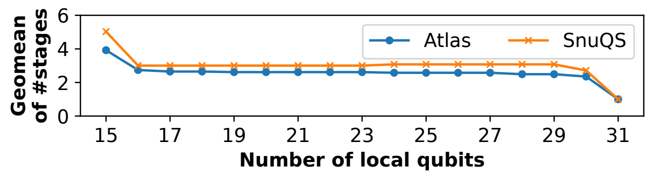


Fig. 9: Number of stages, Atlas versus SnuQS: The geometric mean over all our benchmark circuits with 31 qubits.

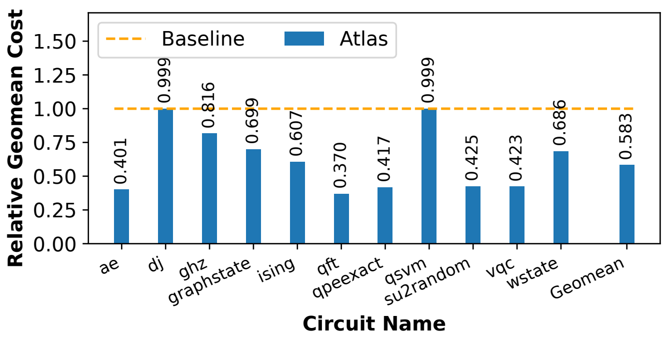


Fig. 10: Kernelization effectiveness: The relative geometric mean cost of KERNELIZE compared to greedy packing up to 5 qubits.

Atlas is up to 20.2× (4.0× on average) faster than HyQuas, 7.2× (3.2× on average) faster than cuQuantum, and 2,126× (286× on average) faster than Qiskit across all types of circuits and possible numbers of GPUs supported by the baselines. The speedups over existing systems are achieved by two important optimizations. First, Atlas' ILP-based circuit staging algorithm can discover a staging strategy that minimizes expensive inter-node communications, allowing Atlas to scale extremely well as the number of GPUs increases. For example, to scale `graphstate` from 28 qubits (on 1 GPU) to 36 qubits (on 256 GPUs), HyQuas' simulation time increases by 267.9×, while the simulation time of Atlas only increases by 13.7×. Second, the DP-based KERNELIZE algorithm enables Atlas to achieve high-performance circuit performance on each GPU.

Figure 6 breaks down the total simulation time into communication time (including intra-node and inter-node communications) and computation time. We take the average of the 11 benchmark circuits for each number of GPUs. Inter-node communication is much more expensive than intra-node communication, so the computation is dominant when there is only one node (at most 4 GPUs). When the number of nodes increases, inter-node communication becomes dominant. We note that the communication time takes a greater portion with 2 GPUs than with 4 GPUs, because each pair of the 4 GPUs in the node is connected. The total bandwidth of each GPU increases by 3× when the number of GPUs available increases from 2 to 4, so intra-node communication is faster with 4 GPUs than with 2 GPUs.

C. DRAM Offloading

In addition to improving quantum circuit simulation performance, another key advantage of Atlas over existing systems is its ability to scale to larger circuits that go beyond the GPU memory capacity. As described in Section III, Atlas does not require that the entire state vector of a quantum circuit be stored in GPU memory and uses DRAM offloading to support larger circuits. QDAO [43] also uses DRAM offloading to support larger circuits and scales to 32 qubits. We compare Atlas with QDAO with the Qiskit backend. We use 28 local qubits, and all remaining qubits are regional. For QDAO, we set $m = 28$, and $t = 19$ which runs the fastest.

Figure 7 shows the simulation performance of `qft` circuits with different numbers of qubits on a single GPU. For a 28-qubit circuit, the GPU memory is sufficient, so both Atlas and QDAO are fast; for circuits with 29 to 32 qubits, Atlas runs 74× faster than QDAO on average, and also scales better. The speedups are also achieved by both the STAGE and the KERNELIZE algorithms.

Figure 8 shows the simulation performance of the 32-qubit `qft` circuit on 1, 2, and 4 GPUs. Atlas scales across multiple GPUs. The simulation time of QDAO stays the same when the number of GPUs increases.

D. Circuit Staging

We further compare the ILP-based circuit staging algorithm with heuristic-based approaches in existing simulators. In particular, we use the heuristics used in SnuQS as a baseline, which greedily selects the qubits with more gates operating on non-local gates to form a stage and uses the number of total gates as a tiebreaker [25]. Other works did not describe how the heuristics were implemented.

Similar to Section VII-B, we set at most 2 non-local qubits to be regional, and all other non-local qubits to be global. Figure 9 shows the geometric mean number of stages for all 11 circuits with 31 qubits in Table I. Our circuit staging algorithm is guaranteed to return the minimum number of stages by Theorem 1, and it always outperforms SnuQS' approach. Note that SnuQS may give a worse circuit partition for the same circuit when the number of local qubits increases (from 23 to 24 in Figure 9), but this is guaranteed not to happen in our ILP-based approach.

E. Circuit Kernelization

We evaluate our KERNELIZE algorithm and compare it with a baseline that greedily packs gates into fusion kernels of up to 5 qubits, the most cost-efficient kernel size in the cost function used in Section VII-A.

Each circuit family exhibits a pattern, so we take the geometric mean of the cost for 9 circuits with the number of qubits from 28 to 36 for each circuit family. Figure 10 shows that the greedy baseline performs well in `dj` and `qsvm` circuits, but it does not generalize to other circuits. The KERNELIZE algorithm is able to exploit the pattern for each circuit and find a low-cost kernel sequence accordingly.

VIII. Related Work

a) Distributed quantum circuit simulators: Recent work has introduced a number of state-vector-based quantum circuit simulators that avail parallelism processing elements on the state vector in a data-parallel fashion [23], [25], [26], [42]–[46]. Even though the state vector can be trivially partitioned, applying quantum gates on it can easily result in a large amount of communication (e.g., between memory and processors and between different nodes). Researchers have therefore proposed circuit partitioning techniques that coalesce computations that operate on a smaller subset of the state space as a single partition so as to reduce communication costs [24]–[26], [47]. An important limitation of all this prior work is that they use heuristic techniques to partition the circuits, leading to suboptimal behavior. We formulate this circuit partitioning problem as an integer linear programming (ILP) problem, which can be solved by applying existing solvers optimally.

b) Quantum gate fusion: Prior work has developed gate fusion techniques that use various heuristics to fuse multiple gates that operate on nearby qubits into a single larger gate which can then be applied at once [19], [24], [26], [48]. For example, Qulacs either leaves it to the user or applies the "light" or "heavy" approaches to merging gates [24]. In this paper, we formulate the circuit kernelization problem of grouping gates into kernels and propose a dynamic programming algorithm to systematically solve this problem.

c) Domain-specific quantum circuit simulators: This paper focuses on the general quantum circuit simulation problem and makes no assumption about input circuits. There has been significant research on developing simulators for specific classes of quantum circuits [42], [49]–[55].

In addition, our work considers an idealized quantum computing environment and does not attempt to simulate errors experienced by modern, NISQ-era quantum computers. There has also been work on developing error-aware quantum circuit simulators [56], [57]. Error simulation is more expensive, because of the need to track errors.

IX. Limitation

Although our techniques can improve run time, the running time of algorithms STAGE and KERNELIZE may depend on the circuit structure, the circuit size, and the ILP solver used. For example, Theorem 5 establishes an exponential upper bound in the number of qubits for the running time of KERNELIZE. But we do not know if this upper bound is tight: our algorithms only take a few seconds on average to preprocess each circuit in our evaluation. We believe that improving the scalability of algorithms STAGE and KERNELIZE is an important problem and will need to be revisited when quantum circuit simulators are able to handle larger circuits. Future work could narrow the gap between theoretical time complexity and practice and explore the trade-off between preprocessing and simulation.

X. Conclusion

In this paper, we propose an ILP-based algorithm STAGE that minimizes the communication cost between GPUs, and a dynamic programming algorithm, KERNELIZE, that ensures efficient kernelization of the GPU work. We then present an implementation of a distributed, multi-GPU quantum circuit simulator, called Atlas, that realizes the proposed algorithms. Previous work also partitioned the circuit for improved communication and kernelization, but relied on heuristics to determine the partitioning. We show that it is possible to achieve substantial improvements by using provable algorithms that can partition the circuit hierarchically.

Acknowledgement

We thank the anonymous SC reviewers for their feedback on this work. This research is partially supported by NSF awards CNS-2147909, CNS-2211882, CNS-2239351, CCF-1901381, CCF-2115104, CCF-2119352, and CCF-2107241 and research awards from Amazon, Cisco, Google, Meta, Oracle, Qualcomm, and Samsung. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award DDR-ERCAP0023403.

References

[1] M. A. Nielsen and I. L. Chuang, “Quantum information and quantum computation,” *Cambridge: Cambridge University Press*, vol. 2, no. 8, p. 23, 2000.

[2] C. H. Bennett and G. Brassard, “Quantum cryptography: Public key distribution and coin tossing,” *arXiv preprint arXiv:2003.06557*, 2020.

[3] R. Orús, S. Mugel, and E. Lizaso, “Quantum computing for finance: Overview and prospects,” *Reviews in Physics*, vol. 4, p. 100028, 2019.

[4] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, “Quantum machine learning,” *Nature*, vol. 549, no. 7671, pp. 195–202, 2017.

[5] S. Lloyd, M. Mohseni, and P. Rebentrost, “Quantum algorithms for supervised and unsupervised machine learning,” *arXiv preprint arXiv:1307.0411*, 2013.

[6] M. Schuld and N. Killoran, “Quantum machine learning in feature hilbert spaces,” *Physical review letters*, vol. 122, no. 4, p. 040504, 2019.

[7] A. Aspuru-Guzik, A. D. Dutoi, P. J. Love, and M. Head-Gordon, “Simulated quantum computation of molecular energies,” *Science*, vol. 309, no. 5741, pp. 1704–1707, 2005.

[8] R. Babbush, J. McClean, D. Wecker, A. Aspuru-Guzik, and N. Wiebe, “Chemical basis of trotter-suzuki errors in quantum chemistry simulation,” *Physical Review A*, vol. 91, no. 2, p. 022311, 2015.

[9] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell *et al.*, “Quantum supremacy using a programmable superconducting processor,” *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[10] G. Q. A. lab, “A preview of bristlecone, google’s new quantum processor,” https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html, 2018.

[11] H.-S. Zhong, H. Wang, Y.-H. Deng, M.-C. Chen, L.-C. Peng, Y.-H. Luo, J. Qin, D. Wu, X. Ding, Y. Hu *et al.*, “Quantum computational advantage using photons,” *Science*, vol. 370, no. 6523, pp. 1460–1463, 2020.

[12] C. Q. Choi, “Ibm unveils 433-qubit osprey chip,” Mar 2023. [Online]. Available: https://spectrum.ieee.org/ibm-quantum-computer-osprey

[13] D. Castelvecchi, “Ibm releases first-ever 1,000-qubit quantum chip,” Dec 2023. [Online]. Available: https://www.nature.com/articles/d41586-023-03854-1

[14] J. Preskill, “Quantum computing in the NISQ era and beyond,” *Quantum*, vol. 2, p. 79, aug 2018. [Online]. Available: https://doi.org/10.22331%2Fq-2018-08-06-79

[15] A. D. Corcoles, A. Kandala, A. Javadi-Abhari, D. T. McClure, A. W. Cross, K. Temme, P. D. Nation, M. Steffen, and J. M. Gambetta, “Challenges and opportunities of near-term quantum computing systems,” *Proceedings of the IEEE*, vol. 108, no. 8, pp. 1338–1352, aug 2020. [Online]. Available: https://doi.org/10.1109%2Fjproc.2019.2954005

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

[17] A. Avila, A. Maron, R. Reiser, M. Pilla, and A. Yamin, “Gpu-aware distributed quantum simulation,” in *Proceedings of the 29th Annual ACM symposium on applied computing*, 2014, pp. 860–865.

[18] A. Avila, R. H. Reiser, M. L. Pilla, and A. C. Yamin, “Optimizing d-gm quantum computing by exploring parallel and distributed quantum simulations under gpus arquitecture,” in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 5146–5153.

[19] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen *et al.*, “Qiskit: An open-source framework for quantum computing,” *Accessed on: Mar*, vol. 16, 2019.

[20] E. Gutiérrez, S. Romero, M. A. Trenas, and E. L. Zapata, “Quantum computer simulation using the cuda programming model,” *Computer Physics Communications*, vol. 181, no. 2, pp. 283–300, 2010.

[21] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, “Quest and high performance simulation of quantum computers,” *Scientific reports*, vol. 9, no. 1, pp. 1–11, 2019.

[22] P. Zhang, J. Yuan, and X. Lu, “Quantum computer simulation on multi-gpu incorporating data locality,” in *Algorithms and Architectures for Parallel Processing: 15th International Conference, ICA3PP 2015, Zhangjiajie, China, November 18-20, 2015, Proceedings, Part I 15*. Springer, 2015, pp. 241–256.

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

[24] Y. Suzuki, Y. Kawase, Y. Masumura, Y. Hiraga, M. Nakadai, J. Chen, K. M. Nakanishi, K. Mitarai, R. Imai, S. Tamiya *et al.*, “Qulacs: a fast and versatile quantum circuit simulator for research purpose,” *Quantum*, vol. 5, p. 559, 2021.

[25] D. Park, H. Kim, J. Kim, T. Kim, and J. Lee, “SnuQS: scaling quantum circuit simulation using storage devices,” in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS ’22. New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3524059.3532375

[26] C. Zhang, Z. Song, H. Wang, K. Rong, and J. Zhai, “Hyquas: Hybrid partitioner based quantum circuit simulation system on gpu,” in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 443–454. [Online]. Available: https://doi.org/10.1145/3447818.3460357

[27] T. Häner and D. S. Steiger, “0.5 petabyte simulation of a 45-qubit quantum circuit,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3126908.3126947

[28] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, “A single-program-multiple-data computational model for epex/fortran,” *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.

[29] M. Xu, S. Cao, X. Miao, U. A. Acar, and Z. Jia, “Atlas: Hierarchical partitioning for quantum circuit simulation on gpus (extended version),” 2024. [Online]. Available: https://arxiv.org/abs/2408.09055

[30] D. Berend and T. Tassa, “Improved bounds on bell numbers and on moments of sums of random variables,” vol. 30, no. 2, pp. 185–205. [Online]. Available: https://www.math.uni.wroc.pl/~pms/files/30.2/Article/30.2.1.pdf

[31] H. Bayraktar, A. Charara, D. Clark, S. Cohen, T. Costa, Y.-L. L. Fang, Y. Gao, J. Guan, J. Gunnels, A. Haidar, A. Hehn, M. Hohnerbach, M. Jones, T. Lubowe, D. Lyakh, S. Morino, P. Springer, S. Stanwyck, I. Terentyev, S. Varadhan, J. Wong, and T. Yamaguchi, “cuQuantum SDK: A high-performance library for accelerating quantum science.” [Online]. Available: http://arxiv.org/abs/2308.01999

[32] Z. Jia, M. Zaharia, and A. Aiken, “Beyond data and model parallelism for deep neural networks,” in *Proceedings of the 2nd Conference on Systems and Machine Learning*, ser. SysML’19, 2019.

[33] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[34] “Nvidia nccl,” https://developer.nvidia.com/nccl, 2021.

[35] “Atlas: High-performance gpu-based quantum circuit simulator,” https://github.com/quantum-compiler/atlas, 2024.

[36] M. Xu and S. Cao, “Artifact for sc24 paper: Atlas: Hierarchical partitioning for quantum circuit simulation on gpus,” Jun. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.12588145

[37] “The perlmutter supercomputer,” https://docs.nersc.gov/systems/perlmutter/, 2023.

[38] N. Quetschlich, L. Burgholzer, and R. Wille, “MQT Bench: Benchmarking software and design automation tools for quantum computing,” 2022, MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[39] A. Li, S. Stein, S. Krishnamoorthy, and J. Ang, “Qasmbench: A low-level qasm benchmark suite for nisq evaluation and simulation,” *arXiv preprint arXiv:2005.13018*, 2021.

[40] I. Dunning, S. Mitchell, and M. O’Sullivan, “PuLP: A linear programming toolkit for python.” [Online]. Available: https://optimization-online.org/?p=11731

[41] Q. Huangfu and J. A. J. Hall, “Parallelizing the dual revised simplex method,” vol. 10, no. 1, pp. 119–142. [Online]. Available: https://doi.org/10.1007/s12532-017-0130-5

[42] C. Zhang, H. Wang, Z. Ma, L. Xie, Z. Song, and J. Zhai, “UniQ: A unified programming model for efficient quantum circuit simulation,” in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–16, ISSN: 2167-4337.

[43] Y. Zhao, Y. Chen, H. Li, Y. Wang, K. Chang, B. Wang, B. Li, and Y. Han, “Full state quantum circuit simulation beyond memory limit,” in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2023, pp. 1–9.

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

[45] Z.-Y. Chen, Q. Zhou, C. Xue, X. Yang, G.-C. Guo, and G.-P. Guo, "64-qubit quantum circuit simulation," *Science Bulletin*, vol. 63, no. 15, pp. 964–971, 2018.

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

[47] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. Garcia-Saez, J. I. Latorre, and S. Carrazza. Qibo: a framework for quantum simulation with hardware acceleration. [Online]. Available: https://arxiv.org/abs/2009.01845v2

[48] S. Westrick, P. Liu, B. Kang, C. McDonald, M. Rainey, M. Xu, J. Arora, Y. Ding, and U. A. Acar, "Grafeyn: Efficient parallel sparse simulation of quantum circuits," in *Proceedings of the IEEE International Conference on Quantum Computing and Engineering*, 2024.

[49] R. Jozsa and N. Linden, "On the role of entanglement in quantum-computational speed-up," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 459, no. 2036, pp. 2011–2032, aug 2003.

[50] D. Aharonov, Z. Landau, and J. Makowsky, "The quantum fft can be classically simulated," *arXiv preprint quant-ph/0611156*, 2006.

[51] D. Gottesman, "The heisenberg representation of quantum computers," *arXiv preprint quant-ph/9807006*, 1998.

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

[53] Y. Zhao, Y. Guo, Y. Yao, A. Dumi, D. M. Mulvey, S. Upadhyay, Y. Zhang, K. D. Jordan, J. Yang, and X. Tang, "Q-gpu: A recipe of optimizations for quantum circuit simulation using gpus," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 726–740.

[54] D. Lykov, R. Shaydulin, Y. Sun, Y. Alexeev, and M. Pistoia, "Fast simulation of high-depth QAOA circuits," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. Association for Computing Machinery, pp. 1443–1451. [Online]. Available: https://doi.org/10.1145/3624062.3624216

[55] D. Lykov, R. Schutski, A. Galda, V. Vinokur, and Y. Alexeev, "Tensor network quantum simulator with step-dependent parallelization," in *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 582–593. [Online]. Available: https://ieeexplore.ieee.org/document/9951269

[56] D. B. Trieu, *Large-scale simulations of error prone quantum computation devices*. Forschungszentrum Jülich, 2010, vol. 2.

[57] A. Li, O. Subasi, X. Yang, and S. Krishnamoorthy, "Density matrix quantum circuit simulation via the bsp machine on modern gpu clusters," in *Sc20: international conference for high performance computing, networking, storage and analysis*. IEEE, 2020, pp. 1–15.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1	A hierarchical partitioning approach to scaling performant quantum circuit simulation based on staging and kernelization.
C_2	An ILP algorithm to stage a circuit for simulation that can minimize the number of stages.
C_3	A dynamic programming algorithm for kernelizing each stage to ensure efficient parallelism.
C_4	An implementation that realizes hierarchical partitioning and significantly outperforms existing simulators.

B. Computational Artifacts

A_1 https://github.com/quantum-compiler/atlas-artifact or https://doi.org/10.5281/zenodo.13334618

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_4	Figures 5-8
	C_2	Figure 9
	C_3	Figure 10

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The artifact includes the implementation of Atlas, benchmark circuits, instructions to compare with other quantum circuit simulators, and the evaluation of the algorithms Stage and Kernelize.

Expected Results

Stage should always outperform SnuQS' approach.

Kernelize should achieve a lower cost on average than greedily packing up to 5 qubits. The cost reduction should be similar within each circuit family (the same circuit name with different numbers of qubits).

In end-to-end performance without DRAM offloading, Atlas should outperform HyQuas, cuQuantum, and Qiskit by more than 2×; with DRAM offloading, Atlas should outperform QDAO by two orders of magnitude. In all of these experiments, Atlas should scale better than other quantum circuit simulators.

Expected Reproduction Time (in Minutes)

The expected computational time of this artifact is 153 minutes on a single node of four NVIDIA A100-SXM4-40GB GPUs plus 25 minutes on up to 16 such GPU nodes for end-to-end performance with and without DRAM offloading (excluding the waiting time to allocate the nodes), plus 27 hours on a single-core CPU for the evaluation of Stage and Kernelize. Additional experiments on up to 256 GPUs (64 GPU nodes) require an extra 25 minutes (excluding the waiting time to allocate the nodes).

We expect around 60 minutes for the Artifact Setup and around 30 minutes for the Artifact Analysis.

Artifact Setup (incl. Inputs)

Hardware: We use the Perlmutter supercomputer to evaluate Atlas. We use up to 16 compute nodes. Each compute node is equipped with an AMD EPYC 7763 64-core 128-thread processor, 256 GB DRAM, and four NVIDIA A100-SXM4-40GB GPUs. The nodes are connected with HPE Slingshot 200 Gb/s interconnects. For the evaluation of Stage and Kernelize, we use a single thread on an Intel Xeon W-1350 @ 3.30GHz CPU (other CPUs are also supported).

Software: Atlas (included in the artifact A_1), HyQuas (modified from https://github.com/thu-pacman/HyQuas/tree/f976382512a74958ec7f76e74e02765e182dc8cb), cuQuantum Appliance 23.03 (https://catalog.ngc.nvidia.com/orgs/nvidia/containers/cuquantum-appliance), Qiskit 1.0.2 (https://pypi.org/project/qiskit/), QDAO 0.1.0 (modified from https://github.com/Zhaoyilunnn/qdao/tree/v0.1.0), and Quartz (https://github.com/quantum-compiler/quartz/tree/atlas-artifact).

Datasets / Inputs: The benchmark circuits are generated from MQT Bench (https://www.cda.cit.tum.de/mqtbench/) and NWQBench (https://github.com/pnnl/nwqbench). We replace the SWAP gates with logical qubit swaps because some previous work does not support SWAP gates, and this replacement does not affect the result. We include the modified benchmark circuits in the artifact along with detailed instructions for replacing the SWAP gates.

Installation and Deployment: We require GCC 12.3.0, GCC 11.2.0, CUDA 12.2, CUDA 11.7, NCCL 2.19.4, NCCL 2.15.5, CMake 3.18+, Conda 23.3+, Python 3.8, Python 3.9, and Python 3.11. We tested only on Linux-like operating systems.

Artifact Execution

The main tasks $T_7, T_8, T_9, T_{10}, T_{12}, T_{13}$ collect performance data for each quantum circuit simulator. They depend on T_2 which copies the benchmark circuits for each task. The tasks to plot the figures T_4, T_5, T_{11}, T_{14} can also be run without running the main tasks to plot the existing results included in the artifact.

T_1	Setup, Part 1: Create a Python environment for Quartz and build Quartz.
T_2	($T_1 \rightarrow T_2$) Setup, Part 2: Install the HiGHS solver in Quartz.
T_3	Setup, Part 3: Replace the account name in the scripts.
T_4	($T_1, T_2 \rightarrow T_4$) Circuit Staging: Evaluate Stage and plot Figure 9.
T_5	($T_1, T_2 \rightarrow T_5$) Circuit Kernelization: Evaluate Kernelize and plot Figure 10.

T_6	End-to-end experiments, Atlas, Part 1: Create a Python environment for Atlas.
T_7	($T_2, T_3, T_6 \rightarrow T_7$) End-to-end experiments, Atlas, Part 2: Build and run Atlas without DRAM offloading.
T_8	($T_2, T_3 \rightarrow T_7$) End-to-end experiments, HyQuas: Build and run HyQuas.
T_9	($T_1, T_2, T_3 \rightarrow T_8$) End-to-end experiments, cuQuantum: Build and run cuQuantum.
T_{10}	($T_1, T_2, T_3 \rightarrow T_9$) End-to-end experiments, Qiskit: Build and run Qiskit.
T_{11}	($T_7, T_8, T_9, T_{10} \rightarrow T_{11}$) End-to-end experiments: Plot Figures 5-6.
T_{12}	($T_2, T_6 \rightarrow T_{12}$) DRAM Offloading, Atlas: Build and run Atlas with DRAM offloading.
T_{13}	($T_2 \rightarrow T_{13}$) DRAM Offloading, QDAO: Create a Python environment for QDAO, and build and run QDAO.
T_{14}	($T_{12}, T_{13} \rightarrow T_{14}$) DRAM Offloading: Plot Figures 7-8.

We run all experiments with CUDA 12.2 except for HyQuas where we use CUDA 11.7 because the cuTT library it uses does not support CUDA 12.

For the experiments on QDAO where it improves the performance by warming it up beforehand, we run the first experiment on QDAO twice to warm it up and only record the time of the second run. For all other experiments, we run them for one time.

Artifact Analysis (incl. Outputs)

Executing the artifact produces plots that should be similar to Figures 5-10 in the paper, as well as an additional plot with 42 qubits for circuit staging, additional individual circuit plots for circuit kernelization, and additional logarithmic-scale plots for end-to-end experiments. Please see the detailed aspects to compare in the Artifact Analysis Section of the Artifact Evaluation Appendix.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

We use the Perlmutter supercomputer to evaluate the performance of Atlas, and use a single-core CPU to evaluate the algorithms STAGE and KERNELIZE.

To run the artifact, please follow the instructions about the prerequisites on https://github.com/quantum-compiler/atlas-artifact. This needs to be done on both the single-core CPU and Perlmutter. This corresponds to the tasks T_1, T_2, T_3 in the Artifact Description.

Artifact Execution

Please refer to the instructions on the GitHub repository https://github.com/quantum-compiler/atlas-artifact. You can find the instructions for each task in the corresponding sections in the `README.md` file in the repository.

Artifact Analysis (incl. Outputs)

Circuit Staging. Task T_4 plots Figure 9 and stores it at `staging_bench/ilp_plot_31.pdf`. Both Atlas and SnuQS should have exactly one stage at 31 local qubits. Atlas should always outperform SnuQS from 15 to 30 local qubits. The number of stages should be non-ascending from 15 to 31 for Atlas, but this generally does not hold for SnuQS. This task also plots a figure for 42 total qubits at `staging_bench/ilp_plot_42.pdf` which shows the same pattern: both Atlas and SnuQS have exactly one stage at 42 local qubits, Atlas always outperform SnuQS from 18 to 41 local qubits. The number of stages should be non-ascending from 18 to 42 for Atlas, but this does not hold for SnuQS.

Circuit Kernelization. Task T_5 plots Figure 10 and stores it at `kernelization_bench/dp_circuit_geomean_relative.pdf`. It also plots figures for individual circuits and stores them at `kernelization_bench/dp_plot_[circuit name].pdf`. The cost reduction should be similar within each circuit family in each individual circuit plot. The greedy baseline performs well in `dj` and `qsvm` circuits, but KERNELIZE achieves a much lower cost on other circuits. The relative geomean cost of KERNELIZE should be similar to Figure 10 in the paper ($0.583\times$).

End-to-end experiments. Task T_{11} plots Figure 5 and stores it at `perlmutter/e2e/logs/figures/[circuit name]_perf.pdf`. Atlas should scale the best on each of the individual circuit families. On a single GPU, Atlas should achieve similar or better performance than cuQuantum and HyQuas. On 256 GPUs, Atlas should be much better than both cuQuantum and HyQuas. On average, Atlas should outperform HyQuas or cuQuantum by more than $2\times$. The logarithmic versions at `perlmutter/e2e/logs/figures/[circuit name]_perf_log.pdf` should show that Atlas outperforms Qiskit by orders of magnitude.

Task T_{11} also plots Figure 6 and stores it at `perlmutter/e2e/logs/figures-comm/comm_mean.pdf`. The percentage of communication time should increase from 0% to around 65% when the number of GPUs increases from 1 to 256. It may decrease when the number of GPUs increases from 2 to 4 if all pairs of GPUs are connected in the GPU node.

DRAM Offloading. Task T_{14} plots Figures 7-8 and stores them at `perlmutter/offload/scalability-qdao.pdf` and `perlmutter/offload/scalability-atlas.pdf`. Figure 7 should show that Atlas outperforms QDAO by two orders of magnitude and also scales better. Figure 8 should show that the simulation time of Atlas decreases when the number of GPUs increases. It shows that the simulation time of QDAO stays the same when the number of GPUs increases by comparing `perlmutter/offload/logs/qdao-qiskit/qdao_1_30_19.log` (the content is the simulation time) with `perlmutter/offload/logs/qdao-qiskit/qdao_2_30_19.log` and `perlmutter/offload/logs/qdao-qiskit/qdao_4_30_19.log`.