2023 IEEE International Conference on Quantum Computing and Engineering (QCE)

cuQuantum SDK: A High-Performance Library for Accelerating Quantum Science

Harun Bayraktar*, Ali Charara*, David Clark*, Saul Cohen*, Timothy Costa*, Yao-Lung L. Fang*,
Yang Gao*, Jack Guan*, John Gunnels*, Azzam Haidar*, Andreas Hehn*, Markus Hohnerbach*,
Matthew Jones*, Tom Lubowe*, Dmitry Lyakh*, Shinya Morino*, Paul Springer*, Sam Stanwyck*,
Igor Terentyev*, Satya Varadhan*, Jonathan Wong*, and Takuma Yamaguchi*
**NVIDIA Corp*, Santa Clara, CA, USA

2023 IEEE International Conference on Quantum Computing and Engineering (QCE) | 979-8-3503-4323-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/QCE57702.2023.00119

***Abstract*—We present the NVIDIA *cuQuantum SDK* [1], a state-of-the-art library of composable primitives for GPU-accelerated quantum circuit simulations. As the size of quantum devices continues to increase, making their classical simulation progressively more difficult, the availability of fast and scalable quantum circuit simulators becomes vital for quantum algorithm developers, as well as quantum hardware engineers focused on the validation and optimization of quantum devices. The *cuQuantum SDK* was created to accelerate and scale up quantum circuit simulators developed by the quantum information science community by enabling them to utilize efficient scalable software building blocks optimized for NVIDIA GPU-based platforms. The functional building blocks provided cover the needs of both state vector- and tensor network- based simulators, including approximate tensor network simulation methods based on matrix product state, projected entangled pair state, and other factorized tensor representations. By leveraging the enormous computing power of the latest NVIDIA GPU architectures, quantum circuit simulators that have adopted the *cuQuantum SDK* demonstrate significant acceleration, compared to CPU-only execution, for both the state vector and tensor network simulation methods. Furthermore, by utilizing the parallel primitives available in the *cuQuantum SDK*, one can easily transition to distributed GPU-accelerated platforms, including those furnished by cloud service providers and high-performance computing systems deployed by supercomputing centers, extending the scale of possible quantum circuit simulations. The rich capabilities provided by the *cuQuantum SDK* are conveniently made available via both Python and C application programming interfaces, where the former is directly targeting a broad Python quantum community and the latter allows tight integration with simulators written in any programming language.**

***Index Terms*—quantum circuit simulation, GPU computing, state vector, tensor network**

I. Introduction

Quantum circuit simulators are a critical part of quantum algorithm and application development workflows. Today's quantum computers are prohibitively small, error-prone, hard to access, capacity-constrained, and, at times, expensive. Even as they scale, this will not likely change until Fault-Tolerant Quantum Computing (FTQC) systems are broadly deployed and no longer capacity-constrained. Therefore, researchers and developers rely on quantum circuit and analog simulators as a critical tool in their toolbox. Many of these simulators are based on state vector (SV) and tensor network (TN) simulation methods, both of which rely heavily on linear algebra and matrix/tensor multiplications. Graphics Processing Units (GPUs) have traditionally been great computational engines for these types of problems, given their ability to utilize thousands of threads to efficiently parallelize computations. For these reasons, NVIDIA introduced the *cuQuantum SDK* with its two main component libraries, *cuStateVec* and *cuTensorNet* (Fig. I). Our strategy is focused on accelerating and scaling up all quantum circuit simulators on GPUs. By working to improve GPU kernels and provide other performance enhancements, in addition to enabling advanced simulation techniques, we have provided simulator developers around the world with the ability to perform quantum circuit simulations at scales and speeds previously not available to them.

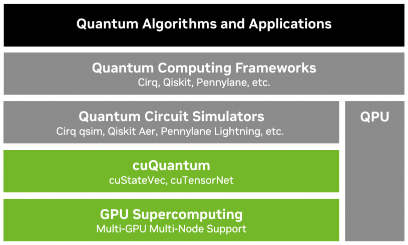

Quantum Algorithms and Applications
Quantum Computing Frameworks
Cirq, Qiskit, Pennylane, etc.
Quantum Circuit Simulators
Cirq qsim, Qiskit Aer, Pennylane Lightning, etc.
QPU
cuQuantum
cuStateVec, cuTensorNet
GPU Supercomputing
Multi-GPU Multi-Node Support

Fig. 1. Overview of the NVIDIA *cuQuantum SDK*. QPU stands for quantum processing units.

Before *cuQuantum*, quantum circuit simulator developers predominantly used basic linear and tensor algebra libraries in their computational backends. The *cuQuantum* model has raised the level of abstraction by providing more convenient and flexible building blocks targeting quantum circuit simulator developers. At the same time, the provided data structures and computational primitives are restricted to the core subset of features typically exposed by higher-level libraries, for example existing TN libraries like iTensor [2], Quimb [3], ExaTN [4], Cyclops Tensor Framework [5], TensorNetwork [6], or TensorTrace [7], to name a few. This uniquely positions *cuQuantum* as a library created to broadly benefit the existing quantum information science (QIS) software ecosystem rather than compete with some of its components. That is, while the primary adoption target is quantum circuit simulators, *cuQuantum* can also boost the performance of more generic software libraries used in quantum sciences.

cuQuantum was introduced to the public in September 2021, with three main components: *cuStateVec*, *cuTensorNet*,

979-8-3503-4323-6/23/$31.00 ©2023 IEEE
DOI 10.1109/QCE57702.2023.00119

and *cuQuantum Python*. At the time we introduced support for NVIDIA A100 and V100 devices, followed by a quick expansion of supported systems to include all devices with CUDA compute capability 7.0+. We demonstrated the largest TN based simulation of the QAOA MaxCut problem, achieving a 93% accurate solution to a 10,000 vertex MaxCut graph (5000 qubits). In March 2022, we introduced the first *cuQuantum Appliance* as a Docker container with a Cirq frontend and *cuQuantum* backend. The 22.07 release added major performance enhancements for multi-GPU simulations, enabling noisy quantum circuit simulations at scale in the *cuQuantum Appliance*. We also introduced APIs for TN slicing and enabling multi-GPU multi-node execution in *cuTensorNet*. In September 2022, we demonstrated the fastest multi-node SV simulations in the world, exceeding the previous state-of-the-art by over $3x$, and scaling much further. The 22.11 release included this functionality in the *cuQuantum Appliance*, introducing IBM Qiskit as a frontend. This release of *cuTensorNet* featured support for the NVIDIA Hopper GPU, user-friendly multi-node APIs, and approximate TN contraction primitives supporting the matrix product state (MPS), matrix product operator (MPO), and projected entabled pair state (PEPS) based simulation algorithms. The 23.03 release added multi-node APIs in *cuStateVec* and intermediate caching/reuse in *cuTensorNet*. Additionally, we introduced the *cuQuantum Appliance* VMI on several major cloud service providers.

The overarching goal of *cuQuantum* is to fulfill a pressing need from community QIS codes to accelerate and scale up quantum simulations to help drive quantum applications towards new discoveries. Our effort builds upon, focuses on, and unifies decades of code development for NVIDIA GPU architectures. By adopting *cuQuantum*, quantum circuit simulators receive the unique ability to study extremely large quantum simulations using NVIDIA DGX and HGX systems, cloud based systems, and some of the largest parallel supercomputers. The *cuQuantum SDK* encapsulates SV, TN, and approximate TN simulation methods which allow studying and accelerating a wide range of quantum simulations at unprecedented scale. Our software is the result of the combined efforts of theoretical physicists, computational scientists, and applied mathematicians who have formulated effective methods and algorithms, allowing users to explore quantum algorithms for a wide range of qubit counts and circuit depths.

Despite its relatively short history, *cuQuantum* has already demonstrated significant impact by accelerating the QIS research community worldwide, from industry to academia. Supporting results include the world record for supercomputing scale simulation performance [8] and a wide range of novel quantum research, such as optimization [9–12], simulating quantum chemistry and other quantum systems [13, 14], security and privacy [15, 16], and much more.

II. The *cuStateVec* Library

We developed the *cuStateVec* library to accelerate and scale up SV simulation, a brute-force, exact method of quantum circuit simulation. An SV simulator represents the quantum state as a complex-valued 1D vector. Thus, it behaves as if it were an ideal quantum computer, as the entire quantum state is preserved in the simulator. Users can successively apply quantum logic gates as needed in their quantum algorithms. Qubits can be measured without collapsing the quantum state by numerical computation of the probability distribution.

In order to hold the entire quantum state, the size of a state vector grows exponentially as 2^n, where n is the number of qubits. When the size of the state vector exceeds the capacity of a single device, it needs to be distributed across multiple compute devices and nodes connected by high-speed interconnects. Hence, large-scale SV simulation is considered a high-performance computing (HPC) challenge.

In this section, we first describe several key features of *cuStateVec*, followed by an introduction to the SV simulators provided in the NVIDIA *cuQuantum Appliance* container.

A. *cuStateVec API Design*

The *cuStateVec* library was developed to provide a set of common primitives used in SV simulators. The latest version of the *cuStateVec* library, v23.03, provides the features shown in Tab. I. The design of the *cuStateVec* library is based on the following considerations:

Provide a set of SV primitives with a relaxed memory model: The *cuStateVec* library provides the basic GPU-accelerated APIs such as gate application, measurement, expectation value computation, and sampling. The state vector should be allocated in GPU memory, however, there is no other specific requirement for the memory model. This facilitates the adoption of the *cuStateVec* library in existing SV simulators.

Small memory footprint: As SV simulation can require a huge amount of memory to store the SV, most APIs are designed to execute in-place operations to eliminate the use of both source and destination buffers. Most operations are executed by using a small amount of the library-internal temporary buffer attached to the library handle, which therefore does not increase the amount of memory required for simulations.

Extensible to multi-GPU and multi-node simulations: Most *cuStateVec* APIs work as single GPU primitives. However, these APIs are designed to work as a part of multi-GPU and multi-node simulations. The library provides a set of APIs to swap the index bits of a SV. These are used to manage the index bit ordering of the state vector. There are multi-GPU and multi-node versions to perform qubit reordering used for distributed SV simulations.

B. *Gate Application Performance and Gate Fusion*

In typical SV simulations, gate application – applying the quantum logic gates from a quantum circuit to a state vector – is the most time-consuming operation. While most gates in quantum circuits have just one or two target qubits, *cuStateVec* provides optimized gate application APIs for larger numbers of target qubits. Fig. 2 shows the performance of gate applications

TABLE I
KEY FEATURES OF CUSTATEVEC

Feature	Description
Resource management	The library handle holds a few tens of MB of GPU memory and other resources.
Gate application	Gate application with dense matrix and generalized permutation matrix.
Pauli rotation	Rotation of state vector by Pauli string.
Measurement	Measurement on Z-product basis and batched single-qubit measurement. Probability computation and collapse function are separately provided.
Expectation	Expectation value computation with dense matrix observable or Pauli strings.
Sampler	Sampling bit-strings as measurement outcomes without collapsing state vector.
Accessor	Copy state vector between CPU and GPU while manipulating the qubit ordering.
Index bit swap	Swap index bit pairs in the state vector. Single-GPU, multi-GPU, and multi-node versions are available.

on both an NVIDIA H100 80GB SXM GPU and an NVIDIA A100 80 GB SXM GPU, with peak memory bandwidths of 3.35 TB/s and 2.04 TB/s, respectively. On the H100, the utilized memory bandwidth generally surpasses 2.35 TB/s and reaches 3.0 TB/s for the best cases, corresponding to 70% and 90%, respectively, while on the A100, a sustained bandwidth of 1.6 TB/s and a peak of 1.7 TB/s, 78% and 83%, respectively, was measured. For up to 5 qubits for Complex 64 and up to 6 qubits for Complex 128 on an NVIDIA H100 (4 and 5 qubits, respectively on the NVIDIA A100), gate application is a memory-bound operation. The total simulation time increases in proportion to the number of these gate application API calls as long as the gate matrices are sufficiently small.

To reduce the computation cost, we can introduce gate fusion [17]. By fusing numerous small gate matrices into a single, multi-qubit gate matrix, one can apply the fused gate matrix in one shot instead of repeating the application of small gate matrices. Thus, the number of fused gates directly contributes to the acceleration of the simulations. Gate fusion can yield drastic performance acceleration in cases where both the high memory bandwidth and high compute performance of the GPU are utilized.

C. Distributed State Vector Simulation

In SV simulation, each bit in the state vector index corresponds to one qubit in a quantum circuit. During simulations, qubits are mapped to the index bits of the state vector. For distributed SV simulations, the state vector is equally sliced and distributed to multiple computing devices as shown in Fig. 3. In this configuration, the index bits in a slice of the state vector are referred to as local index bits; similarly, the index bits to identify the state vector ordinal are referred to as global index bits. When applying a gate onto a local index bit, the gate matrix can be applied independently and concurrently to each slice. However, applying a gate onto a global index bit requires accessing multiple slices of the state vector.

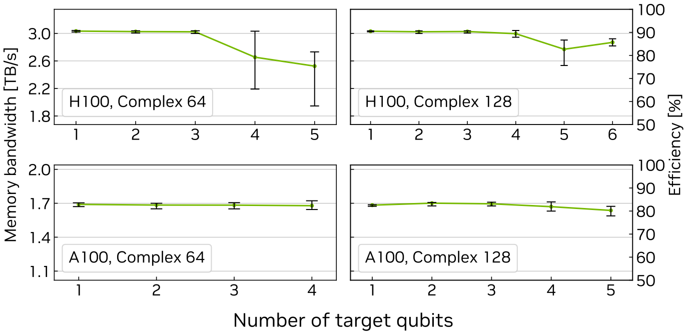


Fig. 2. Gate application performance on the NVIDIA H100 80GB SXM and NVIDIA A100 80 GB SXM GPUs. The performance was measured with 33- and 32-qubit state vectors for Complex 64 and Complex 128, respectively. The bandwidth was calculated from 100 measurement results with randomly-selected sets of target qubits.

Qubit reordering is a known technique to resolve this limitation, which has been reported in [18]. As shown in Fig. 4, one can move a global index bit into a local index bit position, and update the qubit-to-index-bit mapping to move a qubit to the local index bits. Therefore, a gate that acts on a remapped qubit is applied to a local slice of the state vector. For this purpose, the *cuStateVec* library provides distributed index bit swap APIs. One can develop a distributed SV simulator by keeping target qubits local to the state vector slices, then single-GPU APIs can be applied to update the state vector. In the *cuQuantum Appliance* container, both multi-GPU and multi-node quantum circuit simulations are built with index bit swap APIs and single-GPU *cuStateVec* APIs.

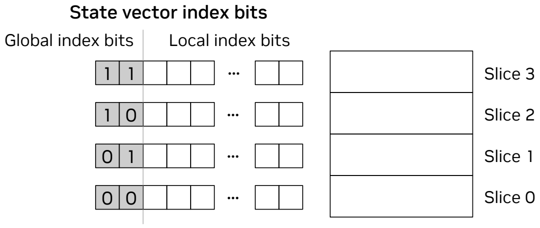
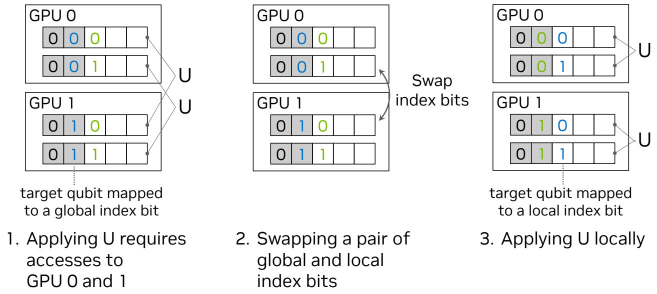


Fig. 3. State vector distribution. The state vector is sliced into four slices and distributed to four computing devices.



GPU 0
0 0 0
0 0 1
GPU 1
0 1 0
0 1 1
U
U
target qubit mapped to a global index bit
1. Applying U requires accesses to GPU 0 and 1
Swap index bits
2. Swapping a pair of global and local index bits
target qubit mapped to a local index bit
3. Applying U locally

Fig. 4. Qubit reordering by swapping a pair of global and local index bits. Two state vector slices are distributed to GPU 0 and GPU 1.

D. Simulators in the Appliance Container

The NVIDIA *cuQuantum Appliance* container provides our multi-GPU-optimized qsim backend for the *Cirq* frontend, and a *cusvaer* backend that supports distributed SV simulations for the *Qiskit Aer* [19] frontend. In this section, we provide an overview of each backend.

1) Multi-GPU qsim backend: *qsim* [20, 21] is a Schrödinger full SV simulator written in C++ and has been integrated with Cirq [22], a Python framework for quantum circuit simulation. Originally, its backends were designed for computations with either multithreading CPUs or single GPUs. We provide a new backend, *qsim-mgpu*, with multi-GPU functionalities. We introduce qubit reordering in our backend so that there is no device-to-device communication within each gate application kernel call. When the reordering is required, we call `custatevecMultiDeviceIndexBitSwaps` to swap the sub state vector components on multiple devices.

a) Introduction of Dense/Diagonal Gate Fusion: Some typical gate matrices, e.g., a controlled-Z gate, can be represented as diagonal matrices. Gate applications for diagonal matrices have lower arithmetic intensity than those for dense matrices due to their sparsity. Therefore, for diagonal matrices, gate application kernels can exhibit nearly identical performance across a wide range of matrix sizes.

Considering these characteristics, we extend gate fusion in Sec. II-B to accept diagonal matrices and generate fused diagonal matrices. In our backend, we provide two options for gate fusion, `max_fused_gate_size` and `max_fused_diagonal_gate_size`, which set the maximum number of qubits per fused dense and diagonal gate, respectively. The gate applications for dense and diagonal matrices are executed using the `custatevecApplyMatrix` and `custatevecApplyGeneralizedPermutationMatrix` APIs.

b) Performance Evaluations: In this section, we report the performance of our qsim backend. We use a single DGX A100 node that consists of eight NVIDIA A100 80GB SXM GPUs and two AMD EPYC 7742 64-core CPUs, a single DGX H100 node that consists of eight NVIDIA H100 80GB SXM GPUs and two Intel Xeon Platinum 8480C CPUs, and version 23.03 of the *cuQuantum Appliance* container. We target Quantum Fourier Transform (QFT) [23], Quantum Approximate Optimization Algorithm (QAOA) [24] with the parameter $p = 2$, and Quantum Volume [25] (depth=30) circuits. Quantum Volume is measured for 10 different circuits and their average elapsed time is evaluated. The maximum gate fusion size is set to 4 on the DGX A100 and 5 on the DGX H100 by the `max_fused_gate_size` option. We focus on the 33-qubit Complex 64 problems and measure strong scaling using up to 8 GPUs, using the timer in the qsim backends.

Tab. II summarizes the elapsed time in each circuit. Compared to the 1-GPU cases, the 8-GPU cases on the DGX H100 attained 4.6-, 7.0-, and 6.5-times speedups in QFT, QAOA, and Quantum Volume circuits, respectively. Additionally, we computed the same QFT simulation on CPUs with the `qsim_cpu` backend. The elapsed time was 78.6 s; therefore, our backend with 8 GPUs achieved a 297-fold speedup over the CPU backend.

TABLE II
qsim-mgpu BACKEND PERFORMANCE ON DGX A100 AND DGX H100

Circuit	# of GPUs	# of gates	DGX A100		DGX H100	
			# of fused gates	*Time (s)*	*# of fused gates*	*Time (s)*
QFT	1	577	18	2.30	18	1.21
	2	577	30	1.59	27	0.911
	4	577	32	0.895	29	0.523
	8	577	33	0.474	27	0.265
QAOA	1	1650	131	10.7	90	4.85
	2	1650	132	5.74	91	2.71
	4	1650	131	2.90	88	1.34
	8	1650	132	1.48	89	0.692
Quantum Volume	1	480	154	12.6	114	6.92
	2	480	158	6.96	122	3.91
	4	480	158	3.67	120	2.04
	8	480	160	1.93	120	1.07

2) Qiskit/Qiskit Aer Multi-node Simulator: Starting with *cuQuantum Appliance* v22.11, *cusvaer* is provided for multi-node distributed SV simulations. *Qiskit Aer* [26] provides a full-fledged multi-node simulator, enabled at compile time, to *Qiskit* users, and *cusvaer* is an extension to *Qiskit Aer* that enables a new multi-node simulator, optimized to extract the best performance from HPC clusters built with NVIDIA GPUs.

The performance limitation of distributed SV simulations comes from data transfer that happens among distributed slices of the state vector due to necessary qubit reordering. Using the DGX A100 as a representative GPU server, NVLink and NVSwitch enable high-speed communication between eight GPUs in one node. NVLink provides 600 GB/s of bidirectional bandwidth for GPU-to-GPU data transfers. NVSwitch connects eight GPUs in a single node with a bisectional bandwidth of 2.4 TB/s. In clusters based on the DGX SuperPOD [27] reference architecture, eight Mellanox ConnectX-6 cards in a DGX A100 are utilized for inter-node communication, which provides 200 GB/s of unidirectional bandwidth to/from one DGX A100 node. The *cusvaer* backend is designed and optimized to achieve nearly-optimal performance, accelerating distributed SV simulations.

Fig. 5 shows the performance results of the multi-node SV simulator for Quantum Volume (depth=30) and quantum phase estimation [28]. The performance has been measured on NVIDIA's Selene supercomputer, which is an NVIDIA-internal DGX A100 cluster based on the DGX SuperPOD architecture. The number of qubits is varied from 32 qubits (1 GPU, 1 node) to 40 qubits (256 GPUs, 32 nodes) for Complex 128 value type. From 32 to 35 qubits, the number of GPUs doubled in a single node, where NVLink and NVSwitch are utilized for the data transfer for qubit reordering. From 35 qubits, the number of nodes is doubled for each addition of one qubit. The slope of the simulation time from 35 to 40 qubits is steeper than that from 32 to 35 qubits, which reflects the difference in data transfer bandwidths between

NVLink/NVSwitch and IB network. These results suggest that it is necessary to implement algorithms to reduce inter-node and inter-device communications such as qubit reordering for better scalability. The simulation times of 32 qubit circuits with two sockets of AMD EPYC 7742 64-core CPU were 178 seconds for quantum volume (depth=30) and 102 seconds for QFT, while the simulation times for 40 qubit circuits of our multi-node simulator are less than 40 seconds. The performance shown in Fig. 5 is one of the best among multi-node SV simulators. The performance comparison with other SV simulators is discussed in the NVIDIA Developer Blog [8].

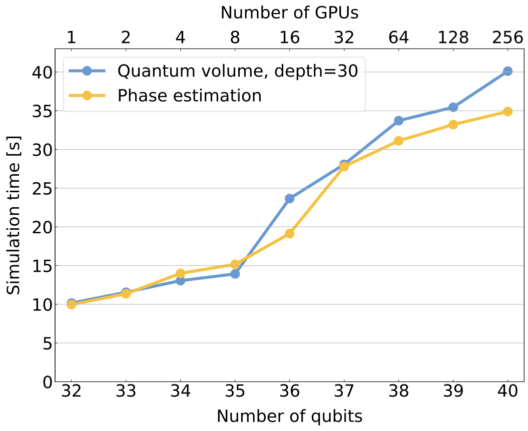


Fig. 5. The simulation time of the extended *Qiskit Aer* multi-node simulator on the Selene supercomputer.

III. The *cuTensorNet* Library

Tensor network theory provides powerful and versatile numerical machinery that has been successfully employed across multiple quantum domains, including condensed matter physics [29, 30], quantum chemistry [31], and quantum information science (QIS) [32–34], as well as applications in other domains such as probabilistic graphical models [35] and machine learning [36, 37]. In all cases, TNs enable the exploration and exploitation of low-rank structure in inherently multi-dimensional problems such as quantum many-body simulations. TN methods have found numerous applications in quantum circuit simulations. The most straightforward approach is based on the direct conversion of a quantum circuit to the corresponding TN, followed by its contraction [32]. While quite powerful for low-depth high-qubit-count quantum circuits [33, 38–40], this approach will suffer from exponential scaling once the circuit depth grows sufficiently large. More sophisticated TN methods tend to introduce controllable approximations with tolerable errors by enforcing an approximate representation of the quantum circuit wavefunction or density matrix. For example, methods based on the MPS [34, 41–43] or PEPS [44] form of compression. These approximate techniques are extremely powerful for simulating quantum circuits with a moderate degree of quantum entanglement [45].

Given that the classical simulation of quantum phenomena is often highly amenable to GPU acceleration, there was a clear need for a library of performant generic building blocks which would make feasible an efficient implementation of the TN methods inside the corresponding domain-specific libraries and simulators. The *cuTensorNet* library consists of two main modules. The network contraction module that performs contraction of a TN and the approximate tensor module that performs exact or approximate tensor(s) decomposition. The *cuTensorNet* library from the *cuQuantum SDK* offers both C and Python APIs (see Sec. IV). The APIs are flexible, exposing most of the features implemented in the library allowing users to control, explore, and investigate the minor details of the algorithmic techniques.

A. Tensor Network Contraction Module

TN contractions are computed as a sequence of pairwise tensor contractions (called a *path*). Defining which pair of tensors go together and the order of the pairwise contractions are the most crucial and complex phase in TN contraction. The ratio between the cost (computational and memory) of an optimal or *close-to-optimal* and a naive path (random, left to right, right to left) can differ by many orders of magnitude emphasizing the importance of the *path*. Once the *path* is defined, there is another phase of optimization related to computing the pairwise contractions using the most efficient GPU kernels. As a result, the TN contraction module of *cuTensorNet* can be described as a combination of two components, a pathfinder component that runs on the CPU and an execution component that computes contractions on the GPU.

1) Path Finding: The role of a pathfinder is to find a contraction path that minimizes the cost of contracting the TN. Finding an optimal contraction path is an NP-hard problem; its cost grows exponentially with the size of the network, making such a technique an impractical or unrealistic solution even for small networks of size dozens of tensors. Most TN simulations, in particular, quantum circuit simulations, consist of networks of hundreds of tensors. Different techniques such as optimal, greedy, or branching [46] have been developed, but they either provide a far from optimal path or require exponential time to find a path. The approach taken in *cuTensorNet* is similar to the one presented in [33], where it was shown to provide superior quality paths (*close-to-optimal*). We developed many algorithmic advancements and optimization techniques to quickly provide such high-quality paths. It starts by simplifying the network. Simplification is a technique that preprocesses the large TN to find all sets of straightforward contractions. It removes them from the network and replaces each set with its final tensor. The result is a smaller tensor network that is easier to process. *cuTensorNet* then uses a hyper-optimization technique [33] that tunes the pathfinder procedure. The core of the pathfinder engine is based on recursive graph partitioning combined with a "bubbling" mechanism. The bubbling method can be described as an iterative process that selects hot spots in the contraction tree (e.g., subtrees that are the most expensive in terms of flops or memory) and adjusts their cost using multiple pathfinder algorithms suitable for small networks. The hyper-optimizer loop creates a distinct set of configurations of parameters for the pathfinder engine and picks the best path found. This hierarchical and recursive design embedded into the hyper-optimizer loop provides superior quality contraction paths (see Fig. 7). Higher quality contraction paths allow larger simulations to be tackled, opening new frontiers in research and discovery.

2) Slicing: Slicing is a technique implemented in *cuTensorNet* to either make a network contraction fit into available device memory or to create parallelism for distributed execution. Large TN contractions need more memory than that available in a single device. Techniques like out-of-device memory might be implemented in such situations but at the cost of a large performance drop due to the significant data traffic between the device and the CPU memory required in this case. Realistic simulations, or the type of simulations needed to develop new horizons in quantum computing, may require more memory than is available on GPU devices or on CPU making it impossible to perform without a different approach. By slicing (also known as variable projection or bond cutting), we split the contraction of the entire TN into several independent smaller contractions, where each contraction considers only one particular position of a certain mode (or combination of modes). The result for full TN contraction can be obtained by summing over the output of each sliced contraction. In Fig. 6, we illustrate an example. If we slice over the mode i, it is no longer implicitly summed over as part of the tensor contraction, but instead explicitly summed in the last step. As a result, slicing can effectively reduce the memory footprint for contracting large TNs, particularly those arising from quantum circuits.

Since each sliced contraction is independent of the others, the computation can be efficiently parallelized in various distributed settings. As a result, the slicing techniques can also be used to generate parallelism and to speed up TN contractions even if memory is not an issue.

Despite all the benefits above, the downside of slicing is that it often increases the total FLOP count of the entire contraction. The overhead of slicing heavily depends on the contraction path and the modes that are sliced. In general, there is no straightforward way to determine the best set of modes to slice. To increase the probability of finding the best contraction path and the best slicing plan, we integrated this phase inside the pathfinder module that is itself encapsulated inside the hyper-optimizer.

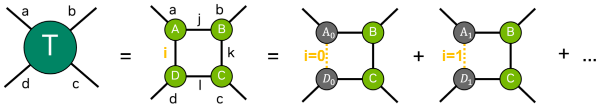


Fig. 6. Overview of the slicing technique.

3) Planning and Workspace: *cuTensorNet* utilizes the NVIDIA *cuTENSOR* library [47] as a backend to perform all the pairwise tensor contractions in the TN, over the existing GPU devices. Once a contraction path has been generated (or received from the user) for the TN, a contraction plan needs to be constructed, This will hold the pairwise contraction plans for *cuTENSOR*. Such a contraction plan could be used to contract the network multiple times, possibly with different data each time. Given the multiple ways a network can be contracted, in terms of the order of contracting the tensor pairs within the same contraction path, the contraction plan is by default configured to use the contraction order that utilizes the minimum workspace possible during network contraction. However, since the provided workspace memory has a direct impact on the choice and performance of the underlying *cuTENSOR* kernels, *cuTensorNet* provides APIs to query the minimum workspace memory size required, the recommended size for good performance, and the maximum size the contraction plan can utilize. The user can then supply any workspace size larger than the minimum required. To get the best performance when contracting the network, *cuTensorNet* offers an API to automatically tune the contraction plan, by executing multiple *cuTENSOR* kernels on each of the pairwise contractions and selecting the most performant one that can operate within the scratch workspace-size constraint, as well as optimizing the intermediate tensor shapes to best utilize device throughput and minimize data transfers.

4) Contraction: *cuTensorNet* facilitates the contraction of the whole TN using the contraction plan, while leveraging the high performance of *cuTENSOR* on GPU devices. The contraction plan serves as the vehicle for repeatedly contracting the TN, each time with possibly different data while using the same contraction plan. The contraction API also facilitates the contraction of the network per slice. It is possible to select the whole set of slices to contract at once, or select a set of slices, picked by indices or by ranges, to be contracted at a contraction call, with the ability to accumulate results on the target buffer or overwrite it. This flexibility allows for integration of the contraction process with user workflows and easy distribution of the contraction computation to many hardware resources, as will be discussed in a subsequent section.

5) Performance: There are two relevant metrics when considering the performance of a pathfinder: the quality of the path found, and the time taken to find that path. The former is plotted in Fig. 7, measured by the cost of the obtained contraction path in FLOPS. The circuits used for benchmarking are the random quantum circuits at depths 12, 14, and 20 described in [33]. *cuTensorNet* performs significantly better than the optimized opt_einsum library [46] in finding a *close-to-optimal* path and slightly better than Cotengra [33] for these circuits.

cuTensorNet also finds a high-quality path quickly. The time in seconds required by *cuTensorNet* compared to Cotengra [33] to run 1000 hyper-optimizer samples is depicted in Tab. III, for Sycamore-53 quantum circuits with different depths. For the most complex problem, with over 3,000 tensors in the network, *cuTensorNet* takes an average of about 8 seconds per path compared to 730 seconds for Cotengra.

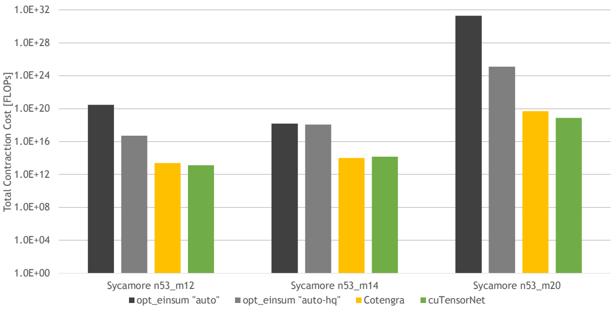


Fig. 7. *cuTensorNet* pathfinder quality of the resulting contraction path compared to similar packages, measured in FLOPs.

TABLE III
PATHFINDING TIME (SEC), FOR THE SYCAMORE RQC PROBLEMS ON INTEL(R) CORE(TM) I9-7900X CPU.

Time in seconds for 1000 hyper-opt samples	**n53_m10** **168 tensors**	**n53_m20** **382 tensors**	**n53_m20** **3316 tensors**
Cotengra [33]	1.5 10^3	8.8 10^4	7.3 10^5
cuTensorNet	2 10^2	1.4 10^3	8 10^3

For realistic quantum simulation, the time to find a path is considered to be negligible when compared to the cost of performing the contraction of the network. *cuTensorNet* performs the contraction on GPUs using the *cuTENSOR* library that provides tuned and optimized kernels for each GPU architecture. *cuTensorNet* also implements internal optimizations to further reduce the memory footprint required to perform all the pairwise contractions, as well as a mechanism to reorder the intermediate tensors' modes to realize the best performance of the GPU kernels.

Fig. 8 depicts the speedup of the network contraction using *cuTensorNet* compared to PyTorch [48] and CuPy [49] running either on a single NVIDIA H100 80GB SXM GPU or a single NVIDIA A100 80GB SXM GPU. Fig. 8 also illustrates the speedup of *cuTensorNet* compared to NumPy [50] running on AMD EPYC 7742 8-core CPUs with multithreaded OpenBLAS, for several key quantum benchmarks ranging from search-based circuit synthesis to quantum approximate optimization algorithm to Sycamore random quantum circuits. Depending on the circuit, *cuTensorNet* offers as much as a 19x speedup for the execution of the network contraction when compared to PyTorch and CuPy. *cuTensorNet* shows impressive speedup when compared to NumPy running multithreaded on the CPU.

B. Distributed Multi-GPU Multi-Node Execution

In order to enable and further accelerate larger quantum circuit simulations, *cuTensorNet* introduced automatic multi-GPU multi-node (distributed) parallelization in the 22.11 release. By activating distributed parallelization, quantum applications which have already adopted *cuTensorNet* for single-GPU acceleration can immediately transition to large-scale GPU-accelerated cloud platforms and HPC systems by leveraging one of the Message Passing Interface (MPI) libraries, thus scaling up the size of possible quantum simulations. The activation of distributed parallelization is performed via a single API call to the `cutensornetDistributedResetConfiguration` function which takes a user-provided MPI communicator. Once activated, both the *pathfinder* and the *execution* procedures will be parallelized across all MPI processes associated with the given MPI communicator. The *pathfinder* will distribute hyper-sampling of contraction path candidates and the execution procedure will distribute

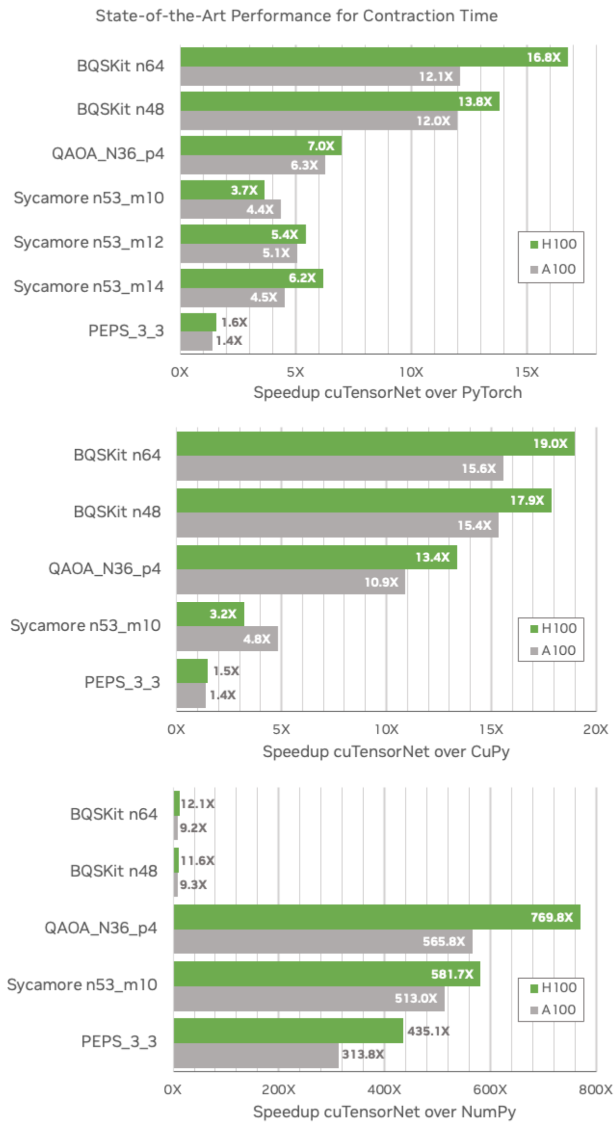


Fig. 8. Contraction speedup of *cuTensorNet* vs PyTorch [48] and CuPy [49] running on a single NVIDIA H100 80GB SXM GPU or NVIDIA A100 80GB SXM GPU, as well as vs NumPy [50] running on an AMD EPYC 7742 8-core CPU for several key quantum benchmarks.

the generated tensor network slices. Since both procedures are embarrassingly parallel, one can expect strong scaling to a very large number of GPUs. Fig. 9 shows a strong scaling plot for a simulation of a single bit string probability amplitude of a random quantum circuit that was part of the validation experiments on Google's Sycamore quantum chip in 2019 [51]. The simulation was run on the Selene supercomputer at NVIDIA, with up to 128 Ampere A100-80GB GPUs. The simulation time scales almost ideally with the number of GPUs.

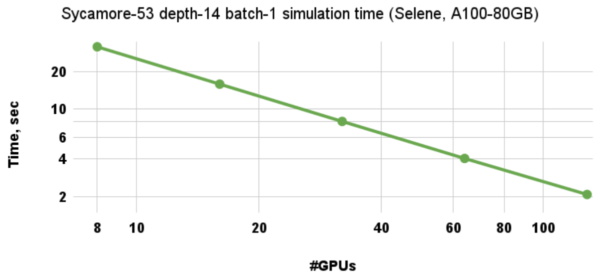


Fig. 9. Strong scaling of the Sycamore-53 random quantum circuit (RQC) simulation (circuit depth = 14) in which a single bit string probability amplitude is computed.

Since different MPI libraries do not necessarily conform to the same ABI specification, special care had to be taken in interfacing *cuTensorNet* with MPI. In particular, the same *cuTensorNet* library will work with any standard-conforming MPI implementation that supports CUDA. This is achieved by dynamically loading a thin shared library (at run-time) built from the provided cuTensorNet-MPI interface C source code and linked to user's MPI library.

C. Intermediate Tensor Caching

1) Intermediate Tensor Reuse: In quantum circuit simulations, some tasks may require multiple evaluations of TNs of the same structure where only a small subset of input tensors undergoes a value update in repeated executions. In particular, validation of quantum processors often requires computing probability amplitudes of individual bit-strings consistently[51]. In this case, the tensor networks for all requested bit-strings have the same structure, only differing in the values of the output bits. Another relevant example is the quantum circuit sampling procedure which requires repetitive computation of projected reduced density matrices (RDMs). In this case, one needs to recompute the same TN (projected RDM) many times where only a small subset of input tensors change their values (projected output bits). The simulation of these and similar cases can be accelerated by storing and reusing the intermediate tensors which stay constant across all repeated TN evaluations. To mark input tensors as constant or mutable, *cuTensorNet* provides the corresponding API. To facilitate reuse of intermediate tensors, *cuTensorNet* accepts two kinds of workspace memory, *scratch* (used to hold temporary data during computations) and *cache* (used to store the constant intermediate tensors upon the first network contraction call to enable their reuse in subsequent TN contraction calls).

2) Performance Impact: Intermediate tensor reuse through caching can bring drastic speed-ups to the contraction of TNs where only a small subset of input tensors (mutable tensors) change their value across many network contractions. Tab. IV shows the huge performance impact of intermediate tensor reuse, on a synthetic network, where the number of input tensors is indicated as network size, while varying the number of constant input tensors, and running 1000 repetitions of TN contraction.

TABLE IV
INTERMEDIATE TENSOR REUSE PERFORMANCE IMPACT ON NVIDIA H100 80GB SXM GPU AND NVIDIA A100 80GB SXM GPU WITH 1000 REPETITIONS OF TN CONTRACTION

Network Size	# slices	Constant tensors	Speedup H100	Speedup A100	Utilized / Recommended Cache (GB)
242	1	80% 85% 90%	4.19 X 4.25 X 4.32 X	4.12 X 4.22 X 4.24 X	1.21 / 1.21 1.10 / 1.10 1.11 / 1.11
327	1024	80% 85% 90%	1.92 X 815 X 838 X	1.71 X 804 X 827 X	55.34 / 1099.51 4.29 / 4.29 4.29 / 4.29

D. Approximate Tensor Network Features

1) Overview: At the core of prevalent approximate TN methods are numerical techniques, such as QR and Singular Value Decomposition (SVD), that can be employed to efficiently represent the quantum state. For instance, QR decomposition is heavily used in MPS canonicalization for orthogonalization, while SVD is employed to truncate the bond dimension, a parameter that determines the accuracy of the MPS [30, 52]. While these two decomposition techniques are well-defined at the matrix level, extending them to the tensor level can be described as a transpose-decompose-transpose-transpose process, similar to the way tensor contraction is generalized from matrix multiplication. In practice, tensor decomposition is frequently used in conjunction with contractions to compress the network graph in a controlled manner.

2) Functionalities: cuTensorNet provides APIs that target different levels of single and compound tensor operations in a hierarchical manner, aiming to support the needs of the numerous approximate TN algorithms. At the tensor level, *cuTensorNet* provides C APIs for both QR and SVD functionalities on a single GPU device, leveraging fast cuSOLVER kernels [53] for transposition and decomposition. For the tensor SVD API, truncation of singular values and the corresponding U and V tensors, as well as post-processing of output tensors, are supported in addition to the standard exact decomposition.

At the TN level, *cuTensorNet* provides a specialized C API named `cutensornetGateSplit` that targets a compound operation frequently used in constructing a TN representation of a quantum circuit, such as MPS. In this context, the gate split process refers to a computational task wherein the gate operand is factorized onto two connecting tensors that it acts upon. Fig. 10 illustrates two algorithms supported in *cuTensorNet* for performing the gate split task. The direct algorithm involves a full contraction followed by decomposition, while the reduced algorithm uses QR decomposition before the full contraction to potentially reduce the intermediate size for the SVD computation.

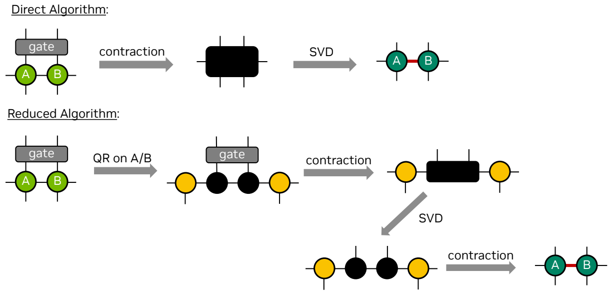


Fig. 10. Two algorithms for the gate split operation. The upper panel shows the direct algorithm, while the lower panel presents the reduced algorithm.

3) Performance: In this subsection we present performance benchmarks for the C APIs, including tensor QR, SVD, and *GateSplit*, on NVIDIA A100 and H100 80GB GPUs. For the QR and SVD benchmarks, we measure the the execution time of decomposing a rank-3 MPS tensor with a shape of $(D, 2, D)$ as a function of D, where D denotes the bond dimension of the MPS. For the gate split benchmarks, the execution time of *GateSplit* was measured for applying a two qubit-gate to an MPS with a bond dimension D. The performance of *cuTensorNet* was benchmarked against an equivalent CPU-based (*NumPy*) implementation using all 64 cores of an AMD EPYC 7742 CPU and the results are summarized in Fig. 11. With tensor QR, *cuTensorNet* exhibits a speedup over the CPU implementation for bond dimensions greater than 32, with a peak of 102x for A100 and 230x for H100 at $D = 4096$. The performance curves of tensor SVD and *GateSplit* are similar to that of tensor QR, but speedup is observed at a larger bond dimension. The peak speedup of tensor SVD at $D = 4096$ reaches 6.4x for A100 and 8.8x for H100. Meanwhile, the speedup of *GateSplit* is observed to be 8.6x and 13.8x for A100 and H100, respectively. The similarity between the peak speedup of tensor SVD and *GateSplit* is consistent with the fact that the cost of SVD is expected to dominate the contraction cost at large scales.

IV. cuQuantum Python

One of the goals of the NVIDIA *cuQuantum SDK* is to allow users to easily access its full functionalities from within Python, enabling interoperability with other Python frameworks and projects. So, in addition to the C APIs provided in the SDK, we also offer *cuQuantum Python* as a natural starting point for those who aim to accelerate their Python workloads using NVIDIA GPUs. To do so, we adopt a two-layer approach:

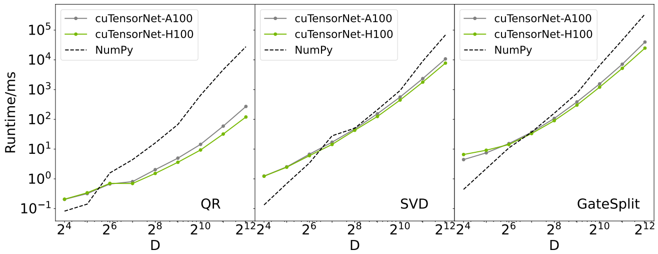


Fig. 11. Performance benchmark of tensor QR, SVD and GateSplit. The execution time for the NumPy implementation is indicated by the dashed black line, while that of the cuTensorNet APIs is shown by the solid green lines.

1) Provide 1:1 Python bindings for the C APIs of both *cuStateVec* and *cuTensorNet* libraries from the NVIDIA *cuQuantum SDK*.
2) Provide high-level, pythonic APIs for easier integration with Python applications in QC or other domains.

Generally, we follow PEP 8 as our style guideline, for both 1:1 bindings and pure Python functions. This allows us to offer APIs that feel natural to Python users. We offer both wheels and conda packages for users to quickly spin up a working Python environment without compiling anything from source. Below we briefly introduce each layer.

A. Low-Level Python bindings

The low-level bindings are 1:1 mappings of the C public APIs to Python. They are exposed under the library modules `cuquantum.custatevec` and `cuquantum.cutensornet`. They currently are written in Cython [54] to support different kinds of user-provided arguments so as to offer flexibility, allowing users to explore the trade-offs between performance and convenience brought by Python's dynamic nature. If a C API requires an array of plain-old-data (POD) structs, for example, we map them to Python using NumPy's structured dtype [55] so that users can allocate such structs using the familiar NumPy APIs, e.g., `arr = np.zeros((10,), dtype=my_struct_dtype)`, and pass the array to the corresponding Python binding. The low-level bindings are then used by high-level APIs when possible, as discussed below.

B. High-Level Pythonic APIs

The high-level pythonic APIs provide encapsulated functionalities that are natural and handy for Python users. With these APIs, just a few lines of Python code can provide functionality that would otherwise require hundreds of lines of C code. Our pythonic APIs keep all the low-level boilerplate code, including memory and resource management, away from the concerns of users. Taking *cuTensorNet* as example, we offer `cuquantum.einsum()` that performs TN contraction following the equivalent `numpy.einsum()` API signature and semantics [56] and, thus, can be used as a drop-in replacement, `cuquantum.contract()` that extends `einsum()` to allow customization and controllability over the tensor network path finding and contraction execution, `cuquantum.cutensornet.tensor.decompose()` that supports single-tensor decomposition with QR and SVD, and `cuquantum.cutensornet.experimental.contract_decompose()` that provides access to additional functionalities beyond gate split operations, enabling it to handle contraction and decomposition operations for arbitrary TNs.

However, if users need finer control over resource management or prefer less automation, they can also use the underlying Python classes and methods. For example, the `cuquantum.Network` class offers the aforementioned encapsulation of a TN. Its constructor parses an einsum expression with tensor operands to create a TN topology and prepare metadata. It also has a `contract_path` method for finding an optimal path, an `autotune` method for auto-selecting the best contraction kernel, and a `contract` method for actual contraction execution. All of the configurability offered at the C API level is exposed through `NetworkOptions`, `OptimizerOptions`, `PathFinderOptions`, `ReconfigOptions`, and `SlicerOptions`.

Moreover, as mentioned above, we can comfortably interact with other Python libraries and frameworks at this level. For example, the *cuTensorNet* distributed contraction support can be easily turned on, by passing the MPI communicator from *mpi4py* [57] (a `mpi4py.MPI.Comm` object) to our helper function and then to `cuquantum.cutensornet.distributed_reset_configuration()`, allowing us to support users of differnt underlying MPI vendors thanks to mpi4py's abstraction. Another example involves the `cuquantum.BaseCUDAMemoryManager` protocol for permitting *cuQuantum Python* to share and use the same memory allocation solution (e.g. CuPy or PyTorch mempool) from other components of the user's workload, so as to avoid encountering the out-of-memory issues commonly seen when multiple Python GPU libraries are involved. Finally, besides using CuPy as the default tensor framework, *cuQuantum Python* also supports NumPy ndarrays and PyTorch tensors as input tensor operands.

The layered hierachy also allows us to build QC capability upon the lower-level, "QC-agnostic" functionalities. For example, `cuquantum.CircuitToEinsum` is a Python class specifically designed to parse a user-provided, fully parameterized quantum circuit (from either the Cirq or Qiskit framework) and turn it into a TN contraction task, by generating einsum-compatible contraction inputs (with the chosen tensor framework). The class supports various computation targets, such as SV coefficients, single or batched bitstring amplitudes, RDMs, and expectation values for Pauli strings. By specifying the computation target, users can obtain the corresponding contraction inputs that can be passed to `einsum()`, `contract()`, or `Network()`, using either default settings for expedited computation or a customized path optimization solution. Users can optionally enable the so-called reverse lightcone simplification technique [33] when computing RDMs and expectation values, so as to reduce the effective TN size and further accelerate the computation.

V. Conclusions

As we have shown, NVIDIA *cuQuantum SDK* provides flexible and highly optimized software building blocks for quantum circuit simulator developers and other domain scientists interested in the efficient GPU implementation of quantum-inspired algorithms. In particular, the quantum circuit simulation primitives provided by the cuStateVec library ensure the optimal memory footprint for state vector simulators. The gate application procedure achieves high memory bandwidth. State vector simulators which adopted the cuStateVec library have demonstrated large GPU speed-ups with respect to CPU-only execution. The distributed execution primitives have been shown to scale well on multi-node multi-GPU platforms, enabling even larger state vector simulations.

The cuTensorNet library from *cuQuantum SDK* has been shown to generate high-quality tensor network contraction paths much faster than the current state-of-the-art software packages targeting tensor network simulator developers. The use of the cuTENSOR library as the computational backend, combined with additional optimizations of tensor network contraction planning, deliver significant GPU speed-ups in the tensor contraction phase. The intermediate caching/reuse offers further speed-ups in the workloads based on repetitive tensor network contractions. The automated distributed multi-node multi-GPU parallelization provided by the cuTensorNet library enables straightforward transition of the tensor network simulators which adopted cuTensorNet to cloud platforms and HPC systems, often showing close-to-ideal scalability. A set of flexible and performant tensor contraction-decomposition primitives facilitates efficient implementation of a wide variety of approximate tensor network contraction schemes.

Finally, to allow easy access to all functionalities of *cuQuantum SDK* from within Python frameworks and projects, we developed *cuQuantum Python* APIs as a natural extension for those who aim to directly accelerate their Python workloads using NVIDIA GPUs.

We envisage to continue our progress towards accelerating and scaling up a more broad circle of quantum applications where we intend to extend our coverage by developing more functionalities and features essential for the ongoing and future quantum research efforts.

Acknowledgment

We gratefully acknowledge Anima Anandkumar, Jean-Marie Eichner, RayLynn Harwood, Jean Kossaifi, Taylor Patti, Jeremy Wang, Demy Xu, Huawei Li, Marvel Zeng, Micky Abir, Marc G. Davis, Omid Khosravani, Fereshte Mozafari, and Shi-Ning Sun for their support and help throughout many phases and releases of the cuQuantum project. We also acknowledge the cuTENSOR [47], cuSOLVER [53], and CuPy [49] teams for fruitful collaboration, and the conda-forge community [58] and the Kitmaker team for packaging support.

References

[1] "cuQuantum SDK: A High-Performance Library for Accelerating Quantum Science." [Online]. Available: https://developer.nvidia.com/cuquantum-sdk

[2] M. Fishman, S. R. White, and E. M. Stoudenmire, "The ITensor Software Library for Tensor Network Calculations," *arXiv:2007.14822*, 2020.

[3] J. Gray, "quimb: A Python package for quantum information and many-body calculations," *Journal of Open Source Software*, vol. 3, no. 29, p. 819, 2018.

[4] D. I. Lyakh, T. Nguyen, D. Claudino, E. Dumitrescu, and A. J. McCaskey, "ExaTN: Scalable GPU-Accelerated High-Performance Processing of General Tensor Networks at Exascale," *Frontiers in Applied Mathematics and Statistics*, vol. 8, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fams.2022.838601

[5] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. USA: IEEE Computer Society, 2013, p. 813–824. [Online]. Available: https://doi.org/10.1109/IPDPS.2013.112

[6] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, "TensorNetwork: A Library for Physics and Machine Learning," *arXiv:1905.01330*, 2019.

[7] G. Evenbly, "TensorTrace: an application to contract tensor networks," *arXiv:1911.02558*, 2019.

[8] "Best-in-Class Quantum Circuit Simulation at Scale with NVIDIA cuQuantum Appliance." [Online]. Available: https://developer.nvidia.com/blog/best-in-class-quantum-circuit-simulation-at-scale-with-nvidia-cuquantum-appliance/

[9] J. R. Finzgar, P. Ross, L. Holscher, J. Klepsch, and A. Luckow, "QUARK: A Framework for Quantum Computing Application Benchmarking," *arXiv:2202.03028v3*, 2022.

[10] M. Cattelan and S. Yarkoni, "Parallel circuit implementation of variational quantum algorithms," *arXiv:2304.03037v1*, 2023.

[11] A. Lowe, M. M. A. Hayes, L. J. O'Riordan, T. R. Bromley, J. M. Arrazola, and N. Killoran, "Fast quantum circuit cutting with randomized measurements," *arXiv:2207.14734v2*, 2023.

[12] H. Makhanov, K. Setia, J. Liu, V. Gomez-Gonzalez, and G. Jenaro-Rabadan, "Quantum Computing Applications for Flight Trajectory Optimization," *arXiv:2304.14445*, 2023.

[13] K. Sugisaki, V. S. Prasannaa, S. Ohshima, T. Katagiri, Y. Mochizuki, B. K. Sahoo, and B. P. Das, "Bayesian phase difference estimation algorithm for direct calculation of fine structure splitting: accelerated simulation of relativistic and quantum many-body effects," *arXiv:2212.02058v2*, 2022.

[14] C. Sims, "Simulation of Higher Dimensional Discrete Time Crystals on a Quantum Computer," *arXiv:2303.02727v2*, 2023.

[15] P. Gokhale, E. R. Anschuetz, C. Campbell, F. T. Chong, E. D. Dahl, P. Frederick, E. B. Jones, B. Hall, P. G. S. Issa, S. Lee, P. Noell, V. Omole, D. Owusu-Antwi, M. A. Perlin, R. Rines, M. Saffman, K. N. Smith, and T. Tomesh, "SupercheQ: Quantum Advantage for Distributed Databases," *arXiv:2212.03850v1*, 2022.

[16] W. Watkins, S. Y.-C. Chen, and S. Yoo, "Quantum machine learning with differential privacy," *Nature Scientific Reports*, vol. 13, no. 2453, 2023.

[17] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik, "qHiPSTER: The quantum high performance software testing environment," *arXiv preprint arXiv:1601.07195*, 2016.

[18] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito, "Massively parallel quantum computer simulator," *Computer Physics Communications*, vol. 176, no. 2, pp. 121–136, 2007.

[19] "Qiskit Aer." [Online]. Available: https://github.com/Qiskit/qiskit-aer

[20] S. V. Isakov, D. Kafri, O. Martin, C. V. Heidweiller, W. Mruczkiewicz, M. P. Harrigan, N. C. Rubin, R. Thomson, M. Broughton, K. Kissell *et al.*, "Simulations of quantum circuits with approximate noise using qsim and cirq," *arXiv preprint arXiv:2111.02396*, 2021.

[21] Quantum AI team and collaborators, "qsim," 2020, accessed: 2023-04-20. [Online]. Available: https://github.com/quantumlib/qsim

[22] Cirq Developers, "Cirq," 2021, accessed: 2023-04-20. [Online]. Available: https://github.com/quantumlib/Cirq

[23] Y. S. Weinstein, M. Pravia, E. Fortunato, S. Lloyd, and D. G. Cory, "Implementation of the quantum Fourier transform," *Physical review letters*, vol. 86, no. 9, p. 1889, 2001.

[24] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[25] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, "Validating quantum computers using randomized model circuits," *Physical Review A*, vol. 100, no. 3, p. 032328, 2019.

[26] Qiskit contributors, "Qiskit: An Open-source Framework for Quantum Computing," 2023.

[27] "NVIDIA DGX SuperPOD." [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-superpod/

[28] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum algorithms revisited," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1969, pp. 339–354, 1998.

[29] R. Orus, "Tensor Networks for Complex Quantum Systems," *Nat. Rev. Phys.*, vol. 1, p. 538, Aug 2019.

[30] U. Schollwöck, "The density-matrix renormalization group in the age of matrix product states," *Annals of Physics*, vol. 326, no. 1, pp. 96–192, 2011, january 2011 Special Issue. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0003491610001752

[31] G. K.-L. Chan and S. Sharma, "The Density Matrix Renormalization Group in Quantum Chemistry," *Annual Review of Physical Chemistry*, vol. 62, no. 1, pp. 465–481, 2011, pMID: 21219144. [Online]. Available: https://doi.org/10.1146/annurev-physchem-032210-103338

[32] 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. [Online]. Available: https://doi.org/10.1137/050644756

[33] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," *Quantum*, vol. 5, p. 410, Mar. 2021. [Online]. Available: https://doi.org/10.22331/q-2021-03-15-410

[34] Y. Zhou, E. M. Stoudenmire, and X. Waintal, "What Limits the Simulation of Quantum Computers?" *Phys. Rev. X*, vol. 10, p. 041038, Nov 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.10.041038

[35] F. Pan, P. Zhou, S. Li, and P. Zhang, "Contracting Arbitrary Tensor Networks: General Approximate Algorithm and Applications in Graphical Models and Quantum Circuit Simulations," *Phys. Rev. Lett.*, vol. 125, p. 060503, Aug 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.125.060503

[36] E. Stoudenmire and D. J. Schwab, "Supervised Learning with Tensor Networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paperfiles/paper/2016/file/5314b9674c86e3f9d1ba25ef9bb32895-Paper.pdf

[37] G. Evenbly, "Number-state preserving tensor networks as classifiers for supervised learning," *Frontiers in Physics*, vol. 10, 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fphy.2022.858388

[38] B. Villalonga, D. Lyakh, S. Boixo, H. Neven, T. S. Humble, R. Biswas, E. G. Rieffel, A. Ho, and S. Mandrà, "Establishing the quantum supremacy frontier with a 281 Pflop/s simulation," *Quantum Science and Technology*, vol. 5, no. 3, p. 034003, apr 2020. [Online]. Available: https://dx.doi.org/10.1088/2058-9565/ab7eeb

[39] C. Huang, F. Zhang, M. Newman, J. Cai, X. Gao, Z. Tian, J. Wu, H. Xu, H. Yu, B. Yuan, M. Szegedy, Y. Shi, and J. Chen, "Classical Simulation of Quantum Supremacy Circuits," *arXiv:2005.06787*, 2020.

[40] T. L. Patti, J. Kossaifi, A. Anandkumar, and S. F. Yelin, "Variational quantum optimization with multibasis encodings," *Phys. Rev. Res.*, vol. 4, p. 033142, Aug 2022. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevResearch.4.033142

[41] J. Chen, E. M. Stoudenmire, and S. R. White, "The Quantum Fourier Transform Has Small Entanglement," *arXiv:2210.08468*, 2022.

[42] E. M. Stoudenmire and X. Waintal, "Grover's Algorithm Offers No Quantum Advantage," *arXiv:2303.11317*, 2023.

[43] A. McCaskey, E. Dumitrescu, M. Chen, D. Lyakh, and T. Humble, "Validating quantum-classical programming models with tensor network simulations," *PLoS ONE*, vol. 13, no. 12, p. e0206704, 2018.

[44] C. Guo, Y. Liu, M. Xiong, S. Xue, X. Fu, A. Huang, X. Qiang, P. Xu, J. Liu, S. Zheng, H.-L. Huang, M. Deng, D. Poletti, W.-S. Bao, and J. Wu, "General-Purpose Quantum Circuit Simulator with Projected Entangled-Pair States and the Quantum Supremacy Frontier," *Phys. Rev. Lett.*, vol. 123, p. 190501, Nov 2019. [Online]. Available:

https://link.aps.org/doi/10.1103/PhysRevLett.123.190501

[45] H. Shang, L. Shen, Y. Fan, Z. Xu, C. Guo, J. Liu, W. Zhou, H. Ma, R. Lin, Y. Yang, F. Li, Z. Wang, Y. Zhang, and Z. Li, "Large-Scale Simulation of Quantum Computational Chemistry on a New Sunway Supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[46] D. Smith and J. Gray, "opt_einsum - A Python package for optimizing contraction order for einsum-like expressions," Jun. 2019. [Online]. Available: https://github.com/dgasmith/opt_einsum

[47] "cuTENSOR: Tensor Linear Algebra on NVIDIA GPUs." [Online]. Available: https://developer.nvidia.com/cutensor

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[49] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[50] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357–362, 2020.

[51] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5

[52] I. P. McCulloch, "From density-matrix renormalization group to matrix product states," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2007, no. 10, p. P10014, oct 2007. [Online]. Available: https://dx.doi.org/10.1088/1742-5468/2007/10/P10014

[53] "cuSOLVER: Direct Linear Solvers on NVIDIA GPUs." [Online]. Available: https://developer.nvidia.com/cusolver

[54] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The Best of Both Worlds," *Computing in Science Engineering*, vol. 13, no. 2, pp. 31–39, 2011. [Online]. Available: https://doi.org/10.1109/MCSE.2010.118

[55] [Online]. Available: https://numpy.org/doc/stable/user/basics.rec.html#structured-datatypes

[56] [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.einsum.html

[57] L. Dalcin and Y.-L. L. Fang, "mpi4py: Status Update After 12 Years of Development," *Computing in Science & Engineering*, vol. 23, no. 4, pp. 47–54, 2021.

[58] conda-forge community, "The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem," Jul. 2015. [Online]. Available: https://conda-forge.org