Tensor Network Quantum Virtual Machine for Simulating Quantum Circuits at Exascale

THIEN NGUYEN, Quantum Computing Institute, Oak Ridge National Laboratory, USA and Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA
DMITRY LYAKH, Quantum Computing Institute, Oak Ridge National Laboratory, USA and National Center for Computational Sciences, Oak Ridge National Laboratory, USA
EUGENE DUMITRESCU, Quantum Computing Institute, Oak Ridge National Laboratory, USA and Computational Sciences and Engineering Division, Oak Ridge National Laboratory, USA
DAVID CLARK, NVIDIA Corp., USA
JEFF LARKIN, NVIDIA Corp., USA
ALEXANDER MCCASKEY, Quantum Computing Institute, Oak Ridge National Laboratory, USA and Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA

The numerical simulation of quantum circuits is an indispensable tool for development, verification and validation of hybrid quantum-classical algorithms intended for near-term quantum co-processors. The emergence of exascale high-performance computing (HPC) platforms presents new opportunities for pushing the boundaries of quantum circuit simulation. We present a modernized version of the Tensor Network Quantum Virtual Machine (TNQVM) which serves as the quantum circuit simulation backend in the eXtreme-scale ACCelerator (XACC) framework. The new version is based on the scalable tensor network processing library ExaTN (Exascale Tensor Networks). It provides multiple configurable quantum circuit simulators which perform either an exact quantum circuit simulation via the full tensor network contraction or an approximate simulation via a suitably chosen tensor factorization scheme. Upon necessity, stochastic noise modeling from real quantum processors is incorporated into the simulations by modeling quantum channels with Kraus tensors. By combining the portable

Authors' addresses: Thien Nguyen, nguyentm@ornl.gov, Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN, USA and Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA; Dmitry Lyakh, Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN, USA and National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA; Eugene Dumitrescu, Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN, USA and Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA; David Clark, NVIDIA Corp., Santa Clara, CA, USA; Jeff Larkin, NVIDIA Corp., Santa Clara, CA, USA; Alexander McCaskey, mccaskeyaj@ornl.gov, Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, TN, USA and Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA.

2643-6817/2022/7-ART $15.00
https://doi.org/10.1145/3547334

XACC quantum programming frontend and the scalable ExaTN numerical processing backend, we introduce an end-to-end virtual quantum development environment which can scale from laptops to future exascale platforms. We report initial benchmarks of our framework which include a demonstration of the distributed execution, incorporation of quantum decoherence models, and simulation of the random quantum circuits used for the certification of quantum supremacy on Google's Sycamore superconducting architecture.

1 INTRODUCTION

Quantum circuit simulation on classical computers is an important tool for development, verification, validation, and analysis of quantum algorithms in the noisy intermediate-scale quantum (NISQ) regime [33]. There exist a wide variety of simulation techniques that have been developed for this purpose, ranging from the state vector [10, 17, 19] or density matrix [23] simulators to Clifford-based [2, 6–8, 13] and tensor-based simulators [16, 20, 24–26, 34, 37, 38]. In particular, the tensor network based techniques have proven their power in constructing effective simulators of rather large quantum circuits with memory requirements that scale in accordance with the quantum state entanglement properties [26, 34]. More generally, tensor processing has been recognized as a computing technique applicable to many scientific and engineering domains [14, 18, 35, 36] that has resulted in highly-optimized software leveraging the state-of-the-art classical hardware capabilities to simulate complex physical phenomena [32].

The tensor network quantum virtual machine (TNQVM) was first introduced in [26] as a tensor-network-based quantum circuit simulation back-end for the XACC framework [27]. The original implementation leveraged the matrix product state (MPS) representation of the quantum circuit wave-function based on the data structures provided by the ITensor library [11] — a popular library that (at the time of implementation) only supported single-core CPU execution. In this work, we present an enhanced TNQVM implementation with a direct focus on the modern HPC deployment via the utilization of the state-of-the-art Exascale Tensor Networks (ExaTN) library [1] as the computational backend. This re-architected TNQVM code runs on both CPU and GPU hardware, and supports multi-node, multi-GPU execution contexts. One of the primary drivers of this work is the need for a flexible high-performance simulator that can (1) extract experimentally verifiable results from large quantum circuits, and (2) take full advantage of computing resources by devising custom strategies for each simulation task and balance the workload (memory and compute) across all available resources.

Tensor network theory is a natural fit for large-scale quantum circuit simulations. Quantum computers encode computation and information in an exponentially large tensor space which is not directly accessible experimentally. One can only collect discrete observable statistics on a given quantum state (qubit measurement bit-strings, expectations values, etc.). The tensor network theory provides the most natural way of dealing with the exact and approximate tensor representations in such exponentially large spaces, thereby enabling an efficient expression of the quantum state observable quantities. Combined with a low-rank compression via low-order tensor factorizations, this approach also becomes highly amenable to memory-bound flop-oriented compute platforms, which most of the current HPC systems are. Thus, the main goal of the TNQVM code is to provide an implementation of a set of tensor network algorithms which are well-suited for quantum circuit simulations in different use case scenarios. All necessary construction, manipulation, and processing of the derived tensor networks is automated via the ExaTN backend. Importantly, the ExaTN backend also provides the foundation for the numerical processing workload optimization. Not only can we

describe quantum circuits in various tensor network forms, such as matrix product state (MPS) [30], tensor tree network (TTN) [36], etc., but we can also delegate the runtime execution optimization to ExaTN where it can decompose and schedule the tensor processing tasks across all available resources, including multi-core CPUs, GPUs, and potentially more specialized accelerators (details of the ExaTN library will be described in an upcoming publication).

In this manuscript, we present implementation details and some preliminary results for the following TNQVM capabilities:

- A generic tensor network contraction based simulator that expresses and evaluates the entire quantum circuit as a tensor network (including expectation values).
- A distributed-memory MPS-factorized state-vector simulator.
- A density matrix (noisy) simulator based on the hierarchical tensor network or locally-purified matrix product operator representations.
- An automatic divide-and-conquer tensor network reconstruction simulator based on arbitrary tensor network factorization schemes with a target fidelity control.

We note that the generic tensor network contraction based simulator supports both noiseless and noisy simulations — we use tensor networks to represent either the state vector or density matrix evolution, respectively. The noise-modelling operations expressed in terms of the channel (Kraus) operators can be incorporated into the latter to mimic the hardware noise models. Similarly, approximate tensor representations of pure or mixed quantum states, based on different tensor factorizations, are provided as alternative approaches geared towards larger-scale simulations. In essence, TNQVM provides a multi-modal simulation platform whereby one can quickly prototype and evaluate accuracy, runtime, parallelism, memory consumption, etc., of varying tensor-based approaches for quantum circuit simulation, as well as execute the actual production runs on workstations, HPC platforms and clouds.

Compared to many other available quantum circuit simulation platforms, the XACC-TNQVM-ExaTN software stack offers unique features in terms of scalability, flexibility, extensibility, performance, and availability. There are not many quantum circuit simulators that have been rigorously tested in a state-of-the-art HPC environment. For example, the Flexible Quantum Circuit (qFlex) Simulator [37, 38] and the QCMPS simulator [9] have demonstrated scalability and accuracy on large supercomputers for the contraction-based and MPS simulation approaches, respectively. However, both of these simulators have been developed for rather specific and narrow use cases, not targeting generic quantum circuit simulation workflows in a complete end-to-end quantum programming stack. Moreover, most tensor-based simulators, including qFlex and QCMPS, tend to associate their internal representation to a particular form of tensor networks as opposed to the multi-modal flexibility of TNQVM. Very recently, classical simulations of the random quantum circuits used in Google's quantum supremacy experiments [5] have been revisited with new simulators leveraging a powerful tensor contraction path optimization algorithm [16]. The QUIMB [15] and ACQDP [20] simulators have positioned themselves as potentially the fastest simulators for the quantum supremacy circuits on distributed HPC platforms and clouds. Although they can be used for simulating other quantum computing circuits as well, their main focus has so far been on the direct tensor network contraction technique and noiseless amplitudes. Similarly, two other recent quantum circuit simulators based on tensor network representations, Jet [39] and QTensor [24], have also focused on the direct tensor network contraction algorithm. In particular, QTensor has demonstrated significant GPU acceleration

for the Quantum Approximate Optimization Algorithm (QAOA) via the new NVIDIA cuPy backend. In contrast, to further boost its GPU performance, TNQVM has been integrated with the cuTensorNet library from the NVIDIA cuQuantum framework [29].

Last but not least, we note that the modular full-stack integration between XACC, TNQVM, and ExaTN allows us to support multiple quantum programming languages and run on different classical compute platforms seamlessly. This full-stack integration proves beneficial, especially for the TNQVM noisy simulators, which can query device noise models directly from the cloud-based hardware providers, e.g., IBM, using the XACC remote connection capabilities. Finally, we also want to stress our commitment to open-source development principles. All of our development activities and implementations are in the public domain under permissive licenses.

To summarize, this work makes the following contributions:

- We present an end-to-end C++ system for quantum circuit expression and simulation with a full support of heterogeneous HPC platforms.
- We introduce multiple levels of parallelization in tensor network processing via our new ExaTN library.
- We design and implement a scheme to parallelize MPS simulations across MPS sites. This scheme differs from the prior work which parallelized each contraction in such simulations.
- We provide the ability to import and incorporate device noise models into quantum circuit simulations.
- We provide the capability of performing locally-purified MPO simulations with explicit noise modeling.
- We introduce a novel general tensor network reconstruction (ExaTN-Gen) simulation method. To the best of our knowledge, we are the first to employ this technique for quantum circuit simulations.

The subsequent sections are organized as follows. Section 2 provides some background information about the XACC programming framework (TNQVM frontend) and the ExaTN library, the scalable numerical backbone behind TNQVM. Section 3 details the implementation of various simulators in TNQVM in terms of the tensor language used by ExaTN. Section 4 provides examples and demonstration results of TNQVM for various tasks ranging from a large-scale quantum circuit simulation to noisy quantum circuit modeling. Conclusions and outlooks are given in Section 5.

2 BACKGROUND

2.1 ExaTN library

The ExaTN library (Exascale Tensor Networks) provides generic capabilities for construction, manipulation and processing of arbitrary tensor networks on single workstations, commodity clusters and leadership supercomputers [1]. On heterogeneous platforms, it can leverage GPU acceleration provided by NVIDIA GPUs (support of AMD GPUs is still experimental). Our TNQVM simulator uses the ExaTN library as a parallel tensor processing backend. The native ExaTN C++ application programming interface (API) consists of declarative and executive API (partial Pybind11 [21] bindings are available for Python users). The declarative API provides functions for constructing arbitrary tensors and tensor networks and performing different formal manipulations on them. The executive API provides functions for allocating tensor storage and parallel processing of tensor networks, for example to perform tensor network contraction. The latter operation is automatically decomposed

by the ExaTN parallel runtime into smaller tasks which are distributed across all MPI processes. The decomposition is performed via the standard technique of intermediate tensor slicing [16, 38]. Individual tensor network slices received by each MPI process are further decomposed into pairwise tensor contractions by a pluggable contraction path finder. The pairwise contractions are then subsequently appended into the dynamic directed acyclic graph (dynamic DAG) executed by the ExaTN parallel runtime in a fully asynchronous fashion on all available compute units (CPUs and GPUs).

ExaTN also provides API for higher-level algorithms, specifically for tensor network reconstruction and tensor network optimization. Tensor network reconstruction allows approximation of a given tensor network by another tensor network, normally with a simpler structure. Tensor network optimization allows finding extrema of a given symmetric tensor network functional (expectation value of some tensor operator). The provided capabilities are sufficient for reformulating common linear algebra procedures on low-rank tensor network manifolds. Such a low-rank compression of linear algebra procedures allows their efficient computation for rather large problems with a tiny fraction of their exact Flop and memory cost.

2.2 XACC quantum programming framework

XACC is a system-level quantum programming framework that enables quantum-language-agnostic programming targeting multiple physical and virtual quantum backends via a novel quantum intermediate representation (IR) [3]. Ultimately, XACC puts forward a service-oriented architecture and defines a number of interfaces or extension points that span the typical quantum-classical programming, compilation, and execution workflow. This platform provides an extensible backend interface for quantum program execution in a retargetable fashion, and this is the interface we target for this work. TNQVM directly extends this layer and enables execution of the XACC IR via multiple tensor-network simulation schemes.

Here, we briefly summarize pertinent XACC interfaces that are relevant to TNQVM and direct interested readers to [27] for a comprehensive introduction. At the high-level, we can classify the framework components into three categories, namely frontend, middle-end, and backend. The frontend exposes a `Compiler` interface which is responsible for converting the input kernel source strings to the XACC `IR`. `IR` is a pertinent data structure of the framework, capturing both the `Instruction` and `CompositeInstruction` service interfaces specialized for concrete quantum gates and collections of those gates, respectively. Using the IR representation of the quantum kernel as its core data structure, the middle-end layer also exposes an `IRTransformation` interface enabling general transformations of quantum circuits (`CompositeInstruction`) for tasks such as circuit optimization and qubit placement. Lastly, XACC provides an `Accelerator` interface enabling integration with physical and virtual (simulator) quantum computing backends. TNQVM implements this `Accelerator` interface, thus providing a universal virtual quantum backend for the framework. In other words, one can use TNQVM interchangeably with other physical QPUs or simulators available in XACC.

Internally, XACC `Accelerator` implementations usually leverage the object-oriented visitor pattern (the XACC `InstructionVisitor`) [12] to walk the `IR` tree representation of the compiled quantum circuits. Each `Accelerator` may opt to perform different actions while walking the `IR` tree. For instance, for physical hardware backends, the `Accelerator` adapter needs to convert XACC `IR` to the native gate set that the platform supports. As we will describe in detail later in the text,

TNQVM makes use of this `InstructionVisitor` interface to construct different tensor network representations of the input circuit depending on the selected mode of simulation, e.g., exact or approximate, ideal or noisy simulation, etc.

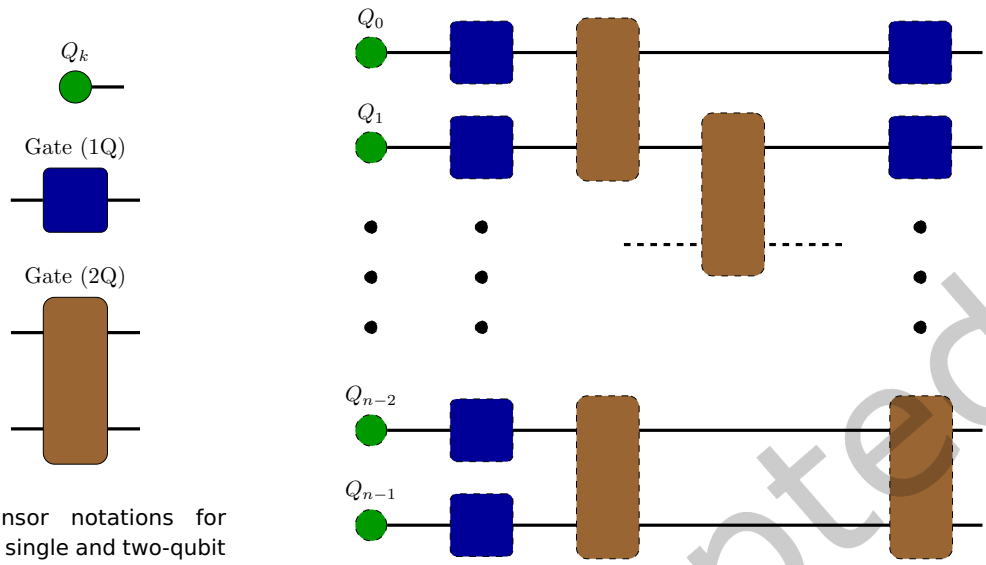
3 EXATN-ENABLED TNQVM IMPLEMENTATION

The ultimate goal of our updated TNQVM implementation targeting heterogeneous HPC systems is to map the input XACC IR to unique tensor data structures provided by ExaTN via the XACC `InstructionVisitor`. TNQVM will walk the IR tree via custom visitors (visitor design pattern [12] as described in 2.2) and visit the IR nodes (quantum gates) and construct, evaluate, and post-process the corresponding ExaTN tensor and tensor network objects. Broadly speaking, TNQVM simulation methods can be categorized as either exact or approximate. The first category comprises backend simulators (visitors) that faithfully translate quantum circuits to equivalent tensor networks and then contract them to evaluate the value of interest. On the other hand, approximate simulation methods rely on factorized forms of the state vector or density matrix where some form of a tensor network compression is used, for example, via the matrix product-state (MPS) tensor network or other tensor network topologies. The factorized representation is maintained throughout the circuit simulation via a suitable decomposition procedure. Thus, we can balance the accuracy and complexity of these approximate representations throughout the simulation process. We note that TNQVM can incorporate stochastic noise into both forms of quantum circuit simulation.

3.1 Direct Tensor Network Contraction

In this mode of execution, we construct a tensor network that embodies the entire quantum circuit before evaluating it numerically. More specifically, qubits, single-qubit gates, and two-qubit gates are represented by rank-1, rank-2, and rank-4 tensors, respectively, as depicted in Fig. 1a. The quantum circuit dictates the connectivity of tensors within the tensor network (see Fig. 1b). Once completed, the tensor network is submitted to the ExaTN numerical server for parallel (GPU-accelerated) evaluation.

During the evaluation phase, ExaTN first analyzes the structure of the tensor network in order to determine the pseudo-optimal tensor contraction sequence (contraction path), that is, it performs minimization of the Flop count (or other relevant metrics) necessary for the evaluation of the tensor network as a sequence of pairwise tensor contractions. The Flop count is minimized by using an algorithm based on recursive graph partitioning (via the METIS library [22]) and some heuristics, following an approach similar to the one presented in Ref. [16]. We should note that our default contraction path finder currently does not include Bayesian optimization. Although such a simplified version of the contraction path finder implemented in ExaTN does not provide the top quality, it prioritizes the speed of the tensor contraction path search, to ensure that the search process does not take more time than the actual evaluation of the tensor network on a highly-parallel HPC platform. Furthermore, a common interface for the tensor contraction path optimization provided by ExaTN allows easy integration with existing high-quality tensor contraction path finders, like CoTenGra [16] or cuQuantum [29] (in fact, ExaTN has been integrated with both in experimental branches). Once a pseudo-optimal tensor contraction path has been identified, the ExaTN numerical server executes the determined tensor contraction sequence across multiple nodes with an optional GPU acceleration capability. Each compute node executes its own subset of tensor sub-networks generated by slicing



(a) Tensor notations for qubits, single and two-qubit quantum gates.

(b) Tensor network representation of a quantum circuit

Fig. 1. ExaTN tensor network representation of a quantum circuit.

some of the tensor network edges, a standard technique used for creating parallel work and reducing the memory footprint of intermediate tensors [16, 37, 38]. It is worth noting that ExaTN supports GPU processing of tensors that are too large to fit into GPU memory. In this case, all cross-device data transfers are orchestrated by the library automatically and transparently to the user. In particular, ExaTN is capable of splitting larger tensors into manageable pieces, automatically transferring tensors to/from GPU memory, prefetching tensors to GPU memory in advance, caching tensors in GPU memory for a subsequent reuse, and pipelining data transfers with computations in order to amortize data transfer overheads.

The direct tensor contraction works best for computing individual amplitudes or their batches. Since the number of open edges in the tensor network is equal to the number of qubits, we cannot obtain the full wave-function for a large number of qubits. Instead, for large-scale circuits, we have implemented a variety of utility functions to extract observable values, as described in Table 1.

3.1.1 Single-state amplitudes. Once the full circuit tensor network has been constructed, we can append appropriate conjugate qubit tensors to each open qubit leg to compute a desired quantum state amplitude (as shown in Fig. 2, red triangles project open qubit legs to a specific bit-string).

Effectively, here we construct the following tensor network to evaluate

$$\langle \Psi_i | U_{circuit} | \Psi_0 \rangle, \tag{1}$$

where $|\Psi_0\rangle$ and $U_{circuit}$ are the initial state and the equivalent unitary matrix of the quantum circuit, respectively. $|\Psi_i\rangle$ is the bit-string state whose amplitude we want to calculate. The result of (1) is just a scalar. This procedure can be repeated for different amplitudes. It is worth noting that despite

Table 1. Tensor Network utility functions for evaluating observables for large-scale quantum circuits.

Mode	Description
Single-state amplitudes	Closing the quantum circuit tensor network with a $\langle\Psi_i|$, where Ψ_i represents a chosen bit-string state, i.
Expectation value by conjugate	Adding a tensor network which represents the observable and then closing with the conjugate quantum circuit (plus light-cone simplification).
Expectation value by state vector slicing	A subset of open tensor legs is projected to a bit-string to keep the number of open legs within the memory constraints. Accumulating the expectation values computed on the partial state vector slices for all possible projected bit-strings to compute the overall expectation value.
Direct unbiased bit-string sampling	Connecting the quantum circuit tensor network with its conjugate while leaving a subset of qubit legs open to compute the reduced density matrices (marginals) for bit-string sampling and measurement projection.

its low rank, the numerical evaluation of a single-state amplitude for large-scale quantum circuits involving many qubits and gates is numerically challenging [38]. Small gate tensors are contracted internally to form larger tensors and any intermediate tensors that require more memory than available will be split into smaller slices and distributed across multiple MPI processes. In principle, this allows simulating output amplitudes of an arbitrarily large quantum circuit in terms of the number of qubits involved, while keeping a bounded memory footprint. The individual or small-batch amplitude evaluation plays a key role in validating near-term quantum hardware via procedures such as the random quantum circuit sampling protocol [5].

3.1.2 Operator expectation values. A ubiquitous use case in quantum circuit simulation is the calculation of the expectation values of Hermitian operators, i.e.,

$$\langle\Psi_f|H|\Psi_f\rangle, \tag{2}$$

where $|\Psi_f\rangle = U_{circuit}|\Psi_0\rangle$ is the final state of the qubit register and H is a general Hermitian operator representing an observable of interest. For instance, H could be a Hermitian sum of products of Pauli operators, $\{\sigma_I, \sigma_X, \sigma_Y, \sigma_Z\}$, on different qubits. We have implemented two different methods to compute Eq. (2) for circuits that have more qubits than a state-vector simulator can efficiently handle: (a) via the use of the conjugate tensor network, and (b) via the wave-function slicing approach, as described in Table 1.

In the first approach, after constructing the tensor network which represents $U_{circuit}|\Psi_0\rangle$, we append the measure operator tensors and then close the obtained tensor network with the Hermitian conjugate of the $U_{circuit}|\Psi_0\rangle$ network. The resulting tensor network evaluates to a scalar and in the worst case consists of approximately twice the number of component tensors, as shown in Fig. 3.

The evaluation of this tensor network yields the expectation value. It is worth mentioning that ExaTN can intelligently collapse a tensor and its conjugate in case they are contracted with each other, which can therefore simplify the tensor network if the measurement operator product is

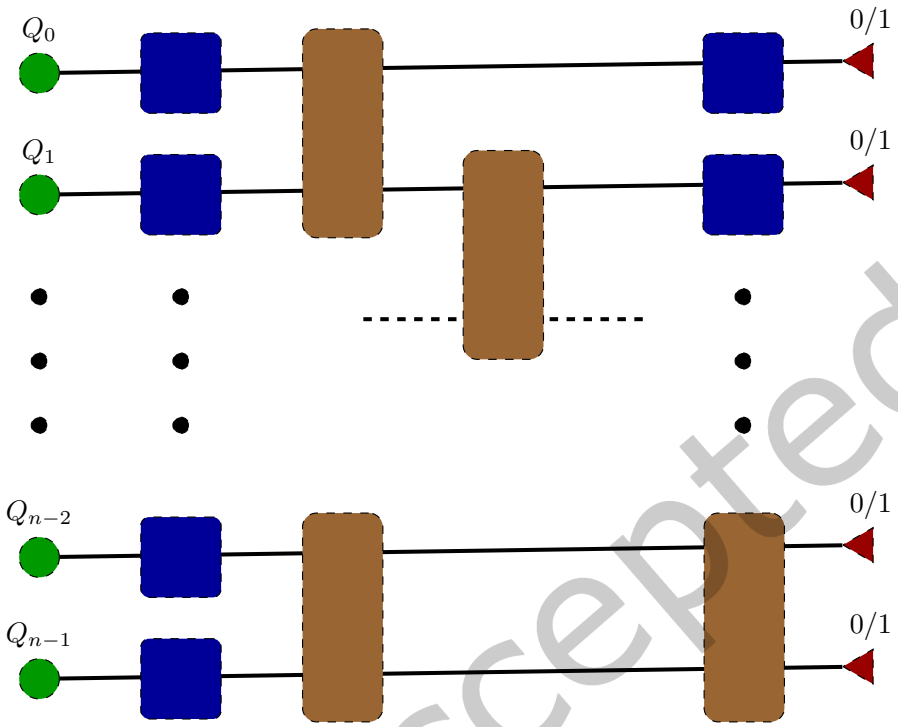


Fig. 2. Single amplitude calculation by tensor network contraction: Open qubit legs are closed with tensors representing the projected 0 or 1 values.

sparse, i.e., it affects only a small fraction of qubits in the circuit. This simplification is commonly called the light-cone simplification.

In the wave-function slicing evaluation method, we slice the output wave-function based on the memory constraint, compute the expectation value for each slice, and recombine them to form the final result at the end. Specifically, the workflow is as follows

(1) Based on the memory constraint, determine the max number of open qubit legs ($rank\_max$) in the output tensor.
(2) Determine the number of wave-function slices (N_{slices}), which is $2^{N_{projected}}$, $N_{projected} = N_{qubits} - rank\_max$.
(3) Distribute the wave-function slice compute tasks (N_{slices}) across all MPI processes.
(4) Compute the expectation value of the measurement operator for each slice.
(5) Sum (reduce) the partial expectation values to compute the final expectation value.

In practice, there are pros and cons in using the above two methods, one based on the quantum circuit conjugation and the other one based on the output wave-function slicing. The first method is more favorable for larger qubit counts (circuit width) and lower circuit depth, in which case the light-cone simplification becomes very effective in reducing the tensor network complexity, provided

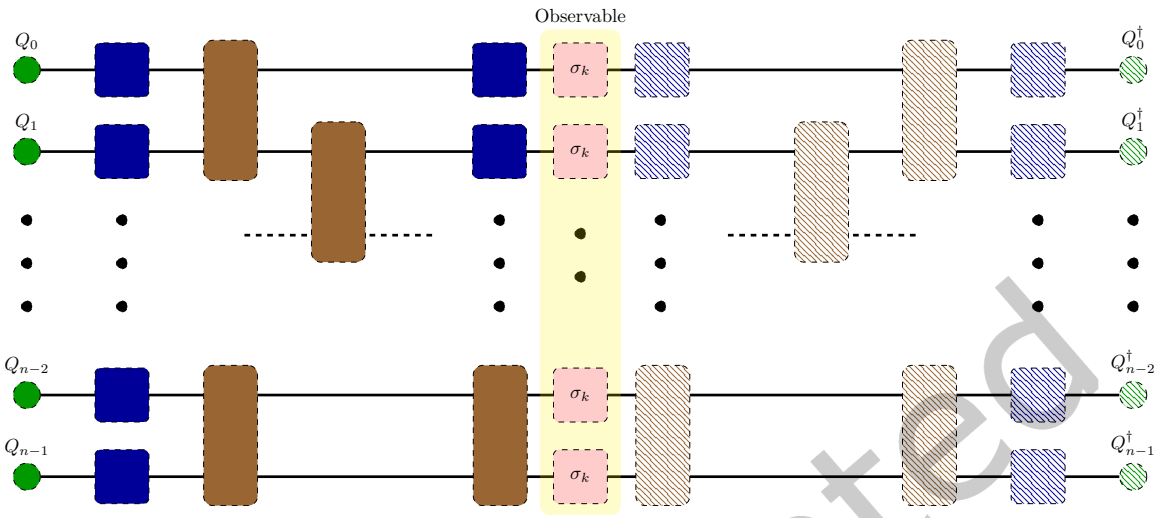


Fig. 3. Expectation value calculation by a double-depth circuit. The observable Pauli tensors $\{\sigma_k\} = \{I, X, Y, Z\}$. Hatched tensors after observable Pauli operators are the conjugates of the ones on the left-hand side.

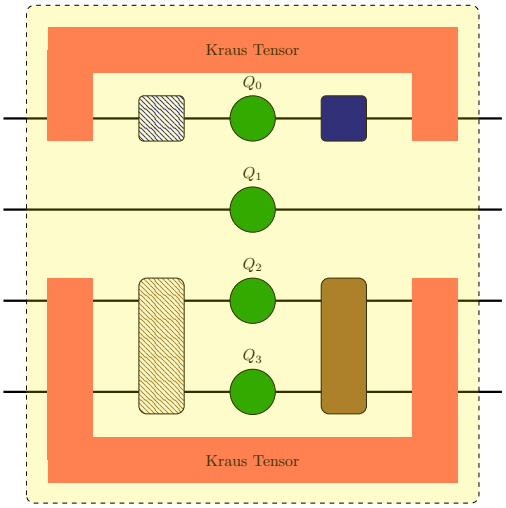
that the observable Pauli products are sufficiently local. For deeper quantum circuits and/or less local observables, the light-cone simplification is unable to reduce the computational complexity significantly enough to justify doubling of the tensor network depth. In this case, the second method will be faster, provided that the number of qubits is not too high. On the other hand, both methods are highly amenable to parallelization, and TN-QVM exploits that.

3.2 Matrix Product State Simulation

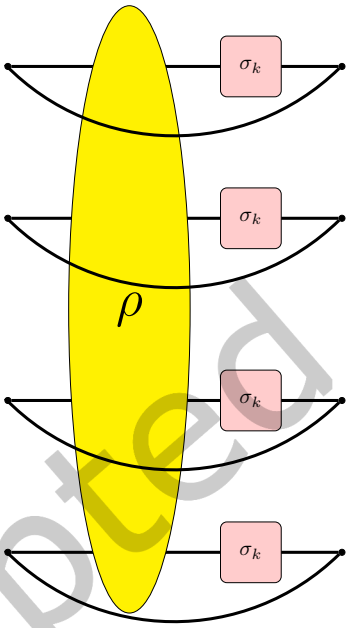
TNQVM also provides an approximate simulator based on the MPS factorization of the circuit wave-function [26], where a user can specify the numerical limit for the singular value truncation as well as the maximum entanglement bond dimension. Built upon the parallel capabilities of ExaTN, we have implemented a distributed MPS tensor processing scheme in which the MPS tensors are distributed evenly across available compute nodes.

A quantum circuit simulation via the MPS simulator backend (named `exatn-mps`) is performed via the standard sequential contraction and decomposition steps [26]. Single-qubit gate tensors can be absorbed into the qubit MPS tensors directly. The application of the two-qubit entangling gates on two neighboring MPS tensors is computed by:

- Merge (contract) the two MPS tensors with the rank-4 gate tensor.
- Decompose the resulting tensor back into two MPS tensors via the singular value decomposition API (`exatn::decomposeTensorSVDLR`) of ExaTN.
- Truncate the dimension of the connecting leg between the two post-SVD MPS tensors, the so-called bond dimension, according to chosen numerical accuracy or memory constraint settings.
- Update the MPS ansatz with the new MPS tensors.



(a) Cartoon of a density matrix simulation. The circles represent a tensor product of unentangled qubit density matrices, the adjoint action of one and two qubit unitaries is indicated by the rounded rectangles, and the local Kraus superoperators are U-shaped tensors.



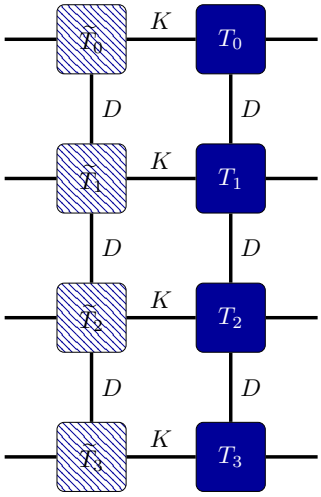
(b) Expectation value by trace

Fig. 4. TNQVM density matrix simulation with noise inclusion

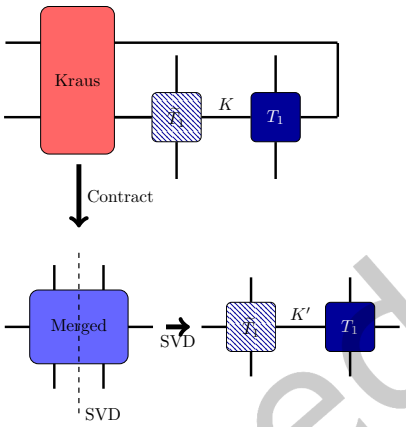
By following this procedure, we can compute the MPS tensor network approximating the full state vector at the end of the quantum circuit. Expectation values or bit-string amplitudes can be computed in the same manner as previously described for the full tensor network contraction strategy. We also want to note that the transformation of quantum circuits into this nearest-neighbor form by injecting the `SWAP` gates is performed automatically by the XACC `IRTransformation` service when the `exatn-mps` backend is selected.

In our implementation of the distributed MPS algorithm each MPI process holds a sub-set of MPS tensors ($N_{qubits}/N_{processes}$). Multiple process groups (`exatn::ProcessGroup`) are then created where each process group consists of a pair of neighboring MPI processes to facilitate local communication. The application of entangling gates between neighboring tensors on different MPI processes is performed by:

- Use `exatn::replicateTensor` API within a pair of neighboring MPI processes to broadcast the MPS tensor right-to-left.
- The left process (smaller MPI rank) performs the contraction and SVD decomposition locally.
- The resulting right tensor will then be forwarded left-to-right using `exatn::replicateTensor`. The two processes now decouple and can continue their independent processing of gates on their subset of managed qubits.



(a) Locally-purified MPS tensor network: K is the Kraus dimension and D is the bond dimension.



(b) Contract a Kraus tensor and decompose (SVD) back to the locally-purified matrix product state form. The Kraus dimension is updated (K to K' after the SVD decomposition.

Fig. 5. Locally-purified matrix product state tensor network representation.

3.3 Density Matrix Simulation

In TNQVM, we can also construct the density matrix by taking the outer product of a state vector with its dual. In this form the density matrix tensor has a rank of 2N (number of dimensions), with N being the number of qubits. Using a density matrix representation of the quantum state, we can thus incorporate (non-unitary) noise processes into the simulation workflow. A convenient representation for noise processes (channels) is the Kraus expansion,

$$\rho \mapsto \sum_k A_k \rho A_k^\dagger, \tag{3}$$

where $\rho = |\Psi\rangle\langle\Psi|$ is the density matrix and $\{A_k\}$ is the set of Kraus operators, satisfying $\sum_k A_k^\dagger A_k = 1$, describing the channel of interest.

To simulate noisy quantum circuits via our `exatn-dm` backend, we append gate tensors to both sides of the tensor network representing the density matrix. Specifically, for each quantum gate, the gate tensor and its conjugate are applied to the ket (right) and bra (left) sides, respectively. Noise operators, on the other hand, are tensors that need to be connected to *both* ket and bra legs as shown in Fig. 4a. We want to note that for illustration purposes, noise tensors are represented as U-shaped tensors in Fig. 4a. We construct them as rank-4 and rank-8 tensors for single- and two-qubit noise processes, respectively, and then append them to our tensor network. In our examples, we have defined depolarizing and dephasing Kraus tensors. To formally construct these tensors, we contract (trace over) an environmental qubit in a dilated unitary formulation [28].

Following this construction procedure, we have a full tensor network capturing the noisy evolution according to the input quantum circuit and a given noise model. At this point, we can evaluate this tensor network to retrieve the density matrix, whose diagonal elements equal the probability of measuring a particular computational basis state. For larger systems, however, full density matrix contraction is not practical due to memory constraints. We can compute specific quantities such as bit-string probabilities or expectation values by adding projection or observable tensors and then contracting the bra and ket qubit legs to form a trace value as depicted in Fig. 4b. The final tensor network is then submitted to ExaTN, which will analyze the structure of the network to determine the tensor contraction sequence and perform the evaluation similar to the simulation algorithm described in Sec. 3.1.

Using the `exatn-dm` backend of TNQVM, users can incorporate quantum noise models into the simulation workflow, such as those that mimic the IBM-Q hardware backends. XACC provides utilities to convert IBM's JSON-based backend configurations into concrete relaxation and depolarization Kraus tensors which are subsequently incorporated into the density matrix tensor network as illustrated in Fig. 4.

```
#include "xacc.hpp"

int main(int argc, char **argv) {
  // Initialize the XACC Framework
  xacc::Initialize(argc, argv);

  // Use ExaTN based TNQVM Accelerator
  auto qpu =
    xacc::getAccelerator("tnqvm:exatn",
                    {{"shots", 1024}});

  // Create a Program
  auto xasmCompiler =
        xacc::getCompiler("xasm");
  std::shared_ptr<IR> ir =
  xasmCompiler->compile(
  R"(__qpu__ void Bell(qbit q) {
    H(q[0]);
    CX(q[0], q[1]);
    Measure(q[0]);
    Measure(q[1]);
  })", qpu);
  std::shared_ptr<CompositeInstruction>
  program = ir->getComposite("Bell");
  // Allocate a register of 2 qubits
  auto qubitReg = xacc::qalloc(2);
  // Execute
  qpu->execute(qubitReg, program);
  // Print the result in the buffer.
  qubitReg->print();

  // Finalize the XACC Framework
  xacc::Finalize();

  return 0;
}
```

Fig. 6. Code snippet demonstrating TNQVM usage with XACC.

3.4 Locally-Purified Matrix Product Operator Simulation

Just as one can factorize a full state vector into an MPS tensor network, one can also factorize the full density matrix tensor (as described in Sec. 3.3) into a similar structure. One method, known as the locally-purified matrix product state (LP-MPS) ansatz [40], is depicted in Fig. 5a. It is implemented in TNQVM via another simulation algorithm, named `exatn-pmps`, that performs approximate density matrix-based simulation following this decomposition procedure (this particular algorithm does not support distributed execution yet).

The application of quantum gates is similar to the MPS algorithm, as described in Sec. 3.2. Two-qubit gates are contracted with the LP-MPS tensors, denoted by T_i, to form a merged tensor which is then decomposed into the canonical tensor product. This procedure only modifies the virtual bond dimension D (see Fig. 5a between the neighboring LP-MPS tensors) which captures the systems entanglement properties. To simulate a non-unitary channel Kraus tensors, such as the ones shown in Fig. 4a, are contracted with the qubit legs of the MPS tensor and its conjugate, see Fig. 5b. After this contraction, we can apply SVD along the Kraus dimension to recover the canonical (locally-purified) form of the LP-MPS factorization. The Kraus dimension (K) between the MPS tensor and its conjugate, encoding statistical mixture in the density operator, is updated after each noise operator iteration.

3.5 Generic Tensor Network Reconstruction Based Simulation

The most recent quantum circuit simulation algorithm implemented in TNQVM is based on the ability to reconstruct an arbitrary tensor network (the whole quantum circuit or any of its parts) as another tensor network with a different, presumably simpler topology. This advanced capability is provided by the ExaTN library (`exatn::TensorNetworkReconstructor` used in the ExaTN-Gen algorithm in TNQVM). In one of the most straightforward settings [26], that is currently implemented in TNQVM, a quantum circuit can be split into chunks of constant depth. Then, by keeping the quantum circuit wave-function (or density matrix, in general) in a factorized form (by some tensor network, e.g., matrix-product state or tensor tree state), each step of this simulation algorithm consists of (A) applying the next quantum circuit chunk to the input tensor network by simply joining both tensor networks, (B) projecting the obtained product on another tensor network of the same or different topology and configuration. The projection step consists of closing the product tensor network with another tensor network and optimizing the tensors in the latter tensor network to maximize the overlap (reconstruction fidelity). At the end of the reconstruction step, the new tensor network approximates the action of a given chunk (slice) of the quantum circuit on the previous wave-function (previous tensor network state). The total reconstruction fidelity for the entire quantum circuit will be a product of reconstruction fidelities for all reconstruction steps, thus providing a user with a quantitative measure of simulation quality (approximation error). In the current implementation, this algorithm in TNQVM has two built-in tensor network topologies, matrix product state and tensor tree state. In general, a user can extend this algorithm to other topologies by providing specialized tensor network builders. Furthermore, one can implement more sophisticated simulation algorithms using this general tensor network reconstruction procedure, in which the quantum circuit is partitioned in a more clever way (the work in this direction is currently in progress).

4 DEMONSTRATIONS

In this section, we seek to demonstrate the utility, flexibility and performance of some of the ExaTN-based backends implemented in TNQVM, while also providing relevant code snippets.

4.1 Quantum circuit simulation

As a first example, Fig. 6 shows a typical usage of TNQVM as a virtual Accelerator in the XACC framework. In particular, after TNQVM is compiled and installed to the XACC plugin directory, users can use `xacc::getAccelerator` API to retrieve an instance of the TNQVM accelerator using the

name key `tnqvm`. In addition, one of the backends described in Section 3 can be specified after the ':' symbol. For example, the code snippet in Fig. 6 calls for the full tensor network contraction simulator (`exatn`).

```
// Query the noise model from an IBM device
auto noiseModel =
    xacc::getService<xacc::NoiseModel>("IBM");
noiseModel->initialize(
           {{"backend", "ibmqx2"}});
auto qpu = xacc::getAccelerator(
    "tnqvm:exatn-dm",
    {{"noise-model", noiseModel}});

auto qubitReg = xacc::qalloc(1);

// Create a test program:
// Apply back-to-back Hadamard gates
// to assess gate noise
auto xasmCompiler = xacc::getCompiler("xasm");
auto ir = xasmCompiler->compile(R"(
__qpu__ void conjugateTest(qbit q) {
    for (int i = 0; i < NB_CYCLES; i++) {
      H(q[0]);
      H(q[0]);
    }
    Measure(q[0]);
  })", qpu);
```

Fig. 7. Noisy quantum circuit simulation with TNQVM. The device noise model (IBMQ Yorktown, `ibmqx2`) is generated from online calibration data and provided to TNQVM as a `noise-model` configuration. In this code snippet, we show the experiment on the first qubit (`q[0]`) of the device. Other qubits can also be experimented with similarly by specifying their indices. Here, we only show a snippet of the code, not including header files and other initialization/finalization steps.

Any specialized configurations are given in terms of a dictionary (key-value pairs) when requesting the accelerator. For example, we can specify the number of simulation runs (shots), as shown in Fig. 6. There are a lot of configurations specific to each simulator backend documented on the XACC documentation website. Simulation results, e.g., shot count distribution, are persisted to the qubit register (`xacc::AcceleratorBuffer`) for later retrieval or post-processing.

The above example demonstrates the seamless integration of TNQVM and all of its backends into the XACC stack. All user codes can use TNQVM as a drop-in replacement for the backend Accelerator. Furthermore, when the simulation demands an HPC platform, users will get instant scalability, i.e., no code changes required, thanks to the TNQVM-ExaTN abstraction layer.

4.2 Noisy simulation

One of the advantages of being part of the XACC framework is that TNQVM can query device characteristics of hardware backends, e.g., IBMQ devices, from XACC to perform hardware emulation. Since TNQVM fully supports noisy quantum circuit simulations, local noise channels can be incorporated into the simulation process. In Fig. 7, we show a simple example how one can construct a noise model from the IBMQ `ibmqx2` (Yorktown) device configuration via the XACC `NoiseModel` utility, followed by the initialization of the density matrix based backend of TNQVM (`exatn-dm`, see Sec. 3.3).

In this demonstration, we simulate a simplified randomized benchmarking procedure whereby the gate set only contains a single gate (Hadamard). By repeating this gate back-to-back over multiple cycles, we can quantify the gate noise in terms of deviation from an ideal identity operation. In other words, if the Hadamard gate is ideal, we would see the qubit state stays at 0 ($\langle Z \rangle = 1$) regardless of

the number of gates. However, due to device noise, we expect a decay of the ground state population as the number of cycles increases, and we see this in the resultant data shown in Fig. 8.

In this experiment, we test the Hadamard gate sequence on both qubit 0 and 1, which have a quite significant single-qubit gate fidelity difference (8.906e-4 for Q0 and 1.935e-3 for Q1). It is worth noting that these calibration parameters are provided by IBM in real-time, which XACC uses to construct the `NoiseModel` object. The simulation results from TNQVM, as shown in Fig. 8, are consistent with the device characteristics. We can clearly see a much faster decay for Q1, whose gate error rate is reported to be more than double that of Q0.

Using the matrix trace bra-ket connection, as depicted in Fig. 4b, we can simulate noisy quantum circuits that contain a large number of qubits as low-rank tensor networks. Intermediate tensor slices appearing in these large-scale tensor network contractions can be effectively distributed across many compute nodes by ExaTN.

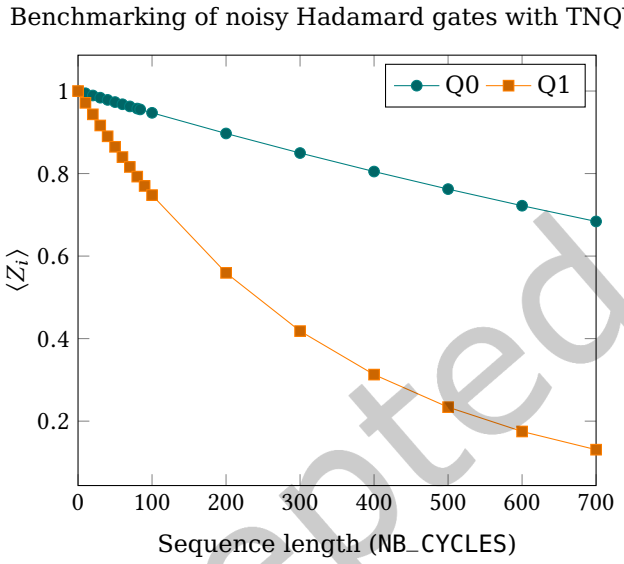


Fig. 8. Plots of expectation values of Pauli-Z operator vs the length of the H-H sequence (`NB_CYCLES` in Fig. 7). Since TNQVM simulates all noise channels according to the device model (`ibmqx2`), the $\langle Z \rangle$ expectation decays as the number of cycles increases. Calibration data: Single qubit Pauli-X error: 8.906e-4 (Q0) and 1.935e-3 (Q1).

4.3 Single amplitude calculation

In order to demonstrate the parallel performance and efficient use of GPU by the TNQVM-ExaTN simulator, here we examine the run time of the direct tensor contraction algorithm when simulating a single output state amplitude (see Table 1) for the Sycamore random quantum circuits involving 53 qubits [4]. The code snippet setting up the simulation experiment is shown in Fig. 9, whereby we can recognize the familiar `Accelerator` initialization, `AcceleratorBuffer` allocation, and execution workflow patterns. The only difference is that we provide a bit-string initialization parameter to request that the amplitude of that specific bit-string be computed.

The Sycamore test circuits involve a large number of qubits (53), thus making the full state-vector calculation unfeasible. Instead, we use the projection, as shown in Fig. 2, to compute the amplitude of a particular bit-string of interest. Also, since we intended to run this test on a cluster, configurations such as the RAM buffer size per MPI process can be customized when initializing the TNQVM accelerator. The random quantum circuit (`program` variable in Fig. 2) is adopted from the Google's quantum supremacy experiment [4] in which the ideal simulation of the depth-14 circuit on Summit was already considered prohibitively expensive at that time. The performance results for

```
// Compute the amplitude of a bit-string:
// BIT_STRING is a vector of length 53, e.g. 000000000...00
// represents the state whose amplitude we want to compute.
auto qpu =
    xacc::getAccelerator("tnqvm:exatn", {{"bitstring", BIT_STRING},
                                         {"exatn-buffer-size-gb", 2}});
// Allocate a register of 53 qubits (Sycamore chip)
auto qubitReg = xacc::qalloc(53);
// Program is the random quantum circuit.
qpu->execute(qubitReg, program);
// Retrieve the amplitude result
const double realAmpl = (*qubitReg)["amplitude-real"].as<double>();
const double imagAmpl = (*qubitReg)["amplitude-imag"].as<double>();
```

Fig. 9. Simulating a Sycamore bit-string amplitude with TNQVM. Variable `program` is an XACC's `CompositeInstruction` instance compiled from the Sycamore test circuits. Here, we only show a snippet of the code, not including header files and other initialization/finalization steps*.

*Code for this example is available at https://github.com/ORNL-QCI/tnqvm/tree/master/examples/sycamore

the depth-14 Sycamore random quantum circuit are listed in Table 2. Our compilation of the circuit comprises 2828 quantum gates, a higher count than originally reported because some 1-body gates had to be additionally decomposed inside XACC, which does not affect computational complexity (the number of two-body gates is the same).

Table 2. Performance comparison of simulating a single amplitude of the depth-14 2D random quantum circuit with 53 qubits.

System	Precision	Time to solution [s]	Avg. Tflop/s/GPU	Flop count per GPU	Bit-string amplitude
DGX-A100, 8 A100 GPU	FP32	2003.23	15.06	3.0160E+16	6.4899E-09
	TF32	868.38	34.73	3.0160E+16	6.4840E-09
DGX-1, 8 V100 GPU	FP32	13028.92	3.05	3.9791E+16	6.4896E-09
OLCF Summit					
16 nodes, 96 V100 GPU	FP32	695.5	4.03	2.7995E+15	6.4899E-09
64 nodes, 384 V100 GPU	FP32	125.52	4.85	6.0856E+14	6.4899E-09
64 nodes, 384 V100 GPU[1]	FP32	60.695	7.99	4.8508E+14	6.4900E-09
Dual AMD Rome CPU					
1 node, 2 x 64-core CPU	FP32	40571.41[2]	2.98	3.0160E+16	6.4899E-09

[1]Faster tensor network contraction path
[2]Extrapolated after 2-hour execution

```
// Create a Program
auto program = xasmCompiler->compile(
R"(__qpu__ void entangle(qbit q) {
    H(q[0]);
    for (int i = 1; i < 60; i++) {
        CX(q[0], q[i]);
    }
})")->getComposite("entangle");
const int NB_QUBITS = 60;
// Measure 2 random qubits & compute marginal
// wavefunction slice of 2 random qubits (Q2&Q47)
BIT_STRING[2] = -1;
BIT_STRING[47] = -1;
// Note: Other bits in the BIT_STRING
// array can be set to 0 or 1
// to denote their projection values.
auto qpu =
    xacc::getAccelerator("tnqvm:exatn",
            {{"bitstring", BIT_STRING}});
// Allocate qubit register and execute
auto qubitReg = xacc::qalloc(NB_QUBITS);
qpu->execute(qubitReg, program);
```

Fig. 10. Compute the marginal wave function slice for a subset of qubits (2) out of a qubit register of size 60. The qubits are initialized to a cat state. Here, we only show a snippet of the code, not including header files and other initialization/finalization steps.

As seen from Table 2, on Summit [3] we observe an excellent strong scaling (from 16 to 64 nodes) as well as a reasonably good absolute efficiency (27 - 53% of the theoretical FP32 peak per GPU). The second 64-node experiment on Summit used a faster tensor contraction path which took longer to find as the price to pay. The 16-node and the first 64-node experiments spent less time in the tensor contraction path search than in its actual execution, whereas the second 64-node experiment spent more time in finding a faster tensor contraction path than in its actual execution (this contraction path also turned out to deliver a better Flop efficiency). In all these experiments we used an out-of-core tensor contraction algorithm implemented in ExaTN, in which the participating tensors may exceed the GPU RAM limit as long as they fit in a normally larger Host RAM. The performance of this algorithm can easily become bound by the Host-to-Device data transfer bandwidth that can be clearly seen from the DGX-1 [4] results where GPUs communicate with the CPU Host via the slower PCIe-3 bus instead of faster NVLink-2. Combined with a lesser amount of Host RAM per MPI process (8 MPI processes on 8 V100 GPUs versus 6 MPI processes on 6 V100 GPUs on Summit), it resulted in a significant performance drop, down to about 20% of the absolute FP32 peak per GPU. On the other hand, the new DGX-A100 box [5] with 2 TB of Host RAM and 80 GB RAM per GPU, as well as with a faster PCIe-4 bus, delivers much better FP32 performance for the out-of-core algorithm. Furthermore, the new A100 tensor cores running with the TF32 precision bump up the performance with an impressive additional 2.3X speed up while keeping the result correct to 3 decimal digits. Despite such a great performance of the out-of-core algorithm on DGX-A100 with a quickly generated, but suboptimal tensor contraction sequence, the simulation of the Sycamore random quantum circuits with a higher depth will result in a computational workload with a lower arithmetic intensity, necessitating the full

[3] Summit node: 2 IBM Power9 CPU with 21 cores each, 6 NVIDIA V100 GPU with 16 GB RAM each, NVLink-2 all-to-all, 512 GB Host RAM

[4] DGX-1 node: 2 Intel Xeon E5-2698 CPU with 20 cores each, 8 NVIDIA V100 GPU with 32 GB RAM each, PCIe-3 bus between CPU and GPU, 512 GB Host RAM

[5] DGX-A100 node: 2 AMD Rome CPU with 64 cores each, 8 NVIDIA A100 GPU with 80 GB RAM each, PCIe-4 bus between CPU and GPU, 2 TB Host RAM

transition of all tensors into the GPU RAM (in-core). For this purpose, we have already integrated TN-QVM with the NVIDIA cuQuantum framework and are looking forward to benchmark it soon.

4.4 Marginal wave-function slice calculation

Similar to the bit-string amplitude calculation (Fig. 9), one can also use TNQVM to compute a marginal wave-function (state-vector) slice for a subset of qubits given other qubits are projected to classical 0 or 1 states. In particular, by keeping a set of n qubits open, the computed vector slice will have a length of 2^n, representing the marginal (conditional) wave function of these qubits given other qubits are fixed. This calculation applies to the chaotic sampling of random quantum circuits or divide-and-conquer parallelization of the state-vector based simulation.

Fig. 10 demonstrates the use of this calculation for a many-qubit cat state, namely, the state $|\Psi\rangle = \frac{1}{\sqrt{2}}(|00..00\rangle + |11..11\rangle)$. This state is generated by the Hadamard gate followed by a sequence of entangling CNOT gates. For demonstration purposes, we randomly selected two open qubits, 2 and 47, to compute the wave-function slice. Thus, the result vector is expected to have a length of 4 and will be returned in the `AcceleratorBuffer`.

In TNQVM, we use a special value of -1 to denote open qubits in the input bit-string. Other qubits are projected to either 0 or 1 values as specified in the bit-string. For instance, given the cat-state, only when other qubits are all 0's or all 1's, then the marginal wave-function result is non-zero. And, we get a marginal state vector of [1,0,0,0] or [0,0,0,1] for the two cases of all others are 0's or all 1's, respectively. This confirms the expected entanglement property of the cat state. As reported in Ref. [31], we can further draw bit-string samples from the resulting marginal wave-function slice to simulate sampling from a subspace of the total Hilbert space determined by those projected qubits.

5 CONCLUSIONS

We have introduced a general tensor network based quantum circuit simulator capable of modeling both ideal and noisy quantum circuits as well as computing various experimentally accessible properties depending on the tensor network formalism used. The versatility and scalability of the ExaTN numerical backend used by this new re-architected version of TNQVM enables simulations of large-scale quantum circuits on leadership HPC platforms. In addition, we have also demonstrated algorithms that incorporate noisy dissipative processes into the simulations.

TNQVM provides a number of capabilities allowing users to efficiently calculate expectation values, exactly or approximately, generate unbiased random measurement bit-strings, or compute state-vector amplitudes. These properties are pertinent to near-term experimental endeavors, such as studying the variational quantum algorithms and validating the new quantum hardware. In this respect, TNQVM presents itself as a valuable tool for analysis and verification of quantum algorithms and devices in pursuit of advancing the progress towards large-scale, fault-tolerant quantum computing. Our continuous goal is to keep extending the functionality of TNQVM and ExaTN by incorporating new and more efficient tensor-based techniques into the simulation workflow in order to enable classical simulation of larger and deeper quantum circuits.

ACKNOWLEDGMENTS

This work has been supported by the US Department of Energy (DOE) Office of Science Advanced Scientific Computing Research (ASCR) Quantum Computing Application Teams (QCAT), Accelerated

Research in Quantum Computing (ARQC), and National Quantum Information Science Research Centers. The development of the core capabilities of the ExaTN library was funded by a laboratory directed research and development (LDRD) project at the Oak Ridge National Laboratory (LDRD award 9463). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Oak Ridge National Laboratory is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725. We would also like to thank Tom Gibbs and NVIDIA for providing access to the DGX-A100 computational resources for performance benchmarking.

REFERENCES

[1] 2020. ExaTN library: Exascale Tensor Networks. https://github.com/ORNL-QCI/exatn

[2] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Phys. Rev. A* 70 (Nov 2004), 052328. Issue 5. https://doi.org/10.1103/PhysRevA.70.052328

[3] QIR Alliance. [n.d.]. Quantum Intermediate Representation. https://github.com/qir-alliance/qir-spec.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.

[5] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, and et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (Oct 2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5

[6] Ryan S. Bennink, Erik M. Ferragut, Travis S. Humble, Jason A. Laska, James J. Nutaro, Mark G. Pleszkoch, and Raphael C. Pooser. 2017. Unbiased simulation of near-Clifford quantum circuits. *Phys. Rev. A* 95 (Jun 2017), 062337. Issue 6. https://doi.org/10.1103/PhysRevA.95.062337

[7] Sergey Bravyi, Dan Browne, Padraic Calpin, Earl Campbell, David Gosset, and Mark Howard. 2019. Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum* 3 (Sept. 2019), 181. https://doi.org/10.22331/q-2019-09-02-181

[8] Sergey Bravyi and David Gosset. 2016. Improved Classical Simulation of Quantum Circuits Dominated by Clifford Gates. *Phys. Rev. Lett.* 116 (Jun 2016), 250501. Issue 25. https://doi.org/10.1103/PhysRevLett.116.250501

[9] Aidan Dang, Charles D. Hill, and Lloyd C. L. Hollenberg. 2019. Optimising Matrix Product State Simulations of Shor's Algorithm. *Quantum* 3 (Jan. 2019), 116. https://doi.org/10.22331/q-2019-01-25-116

[10] Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Willsch, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. 2019. Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications* 237 (2019), 47–61. https://doi.org/10.1016/j.cpc.2018.11.005

[11] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. 2020. The ITensor Software Library for Tensor Network Calculations. arXiv:2007.14822 [cs.MS]

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1 ed.). Addison-Wesley Professional. http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

[13] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. arXiv:2103.02202 [quant-ph]

[14] Ivan Glasser, Nicola Pancotti, and J. Ignacio Cirac. 2019. From probabilistic graphical models to generalized tensor networks for supervised learning. arXiv:1806.05964 [quant-ph]

[15] Johnnie Gray. 2018. quimb: a python library for quantum information and many-body calculations. *Journal of Open Source Software* 3, 29 (2018), 819. https://doi.org/10.21105/joss.00819

[16] Johnnie Gray and Stefanos Kourtis. 2021. Hyper-optimized tensor network contraction. *Quantum* 5 (March 2021), 410. https://doi.org/10.22331/q-2021-03-15-410

[17] Gian Giacomo Guerreschi, Justin Hogaboam, Fabio Baruffa, and Nicolas P D Sawaya. 2020. Intel Quantum Simulator: a cloud-ready high-performance simulator of quantum circuits. *Quantum Science and Technology* 5, 3 (may 2020), 034007. https://doi.org/10.1088/2058-9565/ab8505

[18] Zhao-Yu Han, Jun Wang, Heng Fan, Lei Wang, and Pan Zhang. 2018. Unsupervised Generative Modeling Using Matrix Product States. *Phys. Rev. X* 8 (Jul 2018), 031012. Issue 3. https://doi.org/10.1103/PhysRevX.8.031012

[19] Thomas Häner and Damian S. Steiger. 2017. 0.5 petabyte simulation of a 45-qubit quantum circuit. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Nov 2017). https://doi.org/10.1145/3126908.3126947

[20] Cupjin Huang, Fang Zhang, Michael Newman, Junjie Cai, Xun Gao, Zhengxiong Tian, Junyin Wu, Haihong Xu, Huanjun Yu, Bo Yuan, Mario Szegedy, Yaoyun Shi, and Jianxin Chen. 2020. Classical Simulation of Quantum Supremacy Circuits. arXiv:2005.06787 [quant-ph]

[21] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11 – Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

[22] George Karypis and Vipin Kumar. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1999), 359–392. http://glaros.dtc.umn.edu/gkhome/views/metis

[23] Ang Li, Omer Subasi, Xiu Yang, and Sriram Krishnamoorthy. 2020. Density Matrix Quantum Circuit Simulation via the BSP Machine on Modern GPU Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[24] Danylo Lykov, Angela Chen, Huaxuan Chen, Kristopher Keipert, Zheng Zhang, Tom Gibbs, and Yuri Alexeev. 2021. Performance Evaluation and Acceleration of the QTensor Quantum Circuit Simulator on GPUs. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. 27–34. https://doi.org/10.1109/QCS54837.2021.00007

[25] Igor L. Markov and Yaoyun Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (2008), 963–981. https://doi.org/10.1137/050644756

[26] Alexander McCaskey, Eugene Dumitrescu, Mengsu Chen, Dmitry Lyakh, and Travis Humble. 2018. Validating quantum-classical programming models with tensor network simulations. *PloS one* 13, 12 (2018), e0206704.

[27] Alexander J McCaskey, Dmitry I Lyakh, Eugene F Dumitrescu, Sarah S Powers, and Travis S Humble. 2020. XACC: a system-level software infrastructure for heterogeneous quantum–classical computing. *Quantum Science and Technology* 5, 2 (2020), 024002.

[28] Michael A. Nielsen and Isaac L. Chuang. 2000. *Quantum Computation and Quantum Information*. Cambridge University Press.

[29] NVIDIA. 2021. cuQuantum: Accelerate Quantum Information Science. https://developer.nvidia.com/cuquantum-sdk.

[30] Román Orús. 2014. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics* 349 (2014), 117–158. https://doi.org/10.1016/j.aop.2014.06.013

[31] Feng Pan and Pan Zhang. 2021. Simulating the Sycamore quantum supremacy circuits. arXiv:2103.03074 [quant-ph]

[32] Agustin Di Paolo, Thomas E. Baker, Alexandre Foley, David Sénéchal, and Alexandre Blais. 2021. Efficient modeling of superconducting quantum circuits with tensor networks. *npj Quantum Information* 7, 1 (2021). https://doi.org/10.1038/s41534-020-00352-4

[33] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[34] Justin Reyes and Miles Stoudenmire. 2020. A Multi-Scale Tensor Network Architecture for Classification and Regression. arXiv:2001.08286 [stat.ML]

[35] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. 2019. TensorNetwork: A Library for Physics and Machine Learning. arXiv:1905.01330 [physics.comp-ph]

[36] Y.-Y. Shi, L.-M. Duan, and G. Vidal. 2006. Classical simulation of quantum many-body systems with a tree tensor network. *Phys. Rev. A* 74 (Aug 2006), 022320. Issue 2. https://doi.org/10.1103/PhysRevA.74.022320

[37] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. 2019. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *npj Quantum Information* 5, 1 (2019). https://doi.org/10.1038/s41534-019-0196-1

[38] Benjamin Villalonga, Dmitry Lyakh, Sergio Boixo, Hartmut Neven, Travis S Humble, Rupak Biswas, Eleanor G Rieffel, Alan Ho, and Salvatore Mandrà. 2020. Establishing the quantum supremacy frontier with a 281 Pflop/s simulation. *Quantum Science and Technology* 5, 3 (apr 2020), 034003. https://doi.org/10.1088/2058-9565/ab7eeb

[39] Trevor Vincent, Lee J. O'Riordan, Mikhail Andrenkov, Jack Brown, Nathan Killoran, Haoyu Qi, and Ish Dhand. 2021. Jet: Fast quantum circuit simulations with parallel task-based tensor-network contraction. (2021). arXiv:2107.09793 [quant-ph]

[40] Albert H Werner, Daniel Jaschke, Pietro Silvi, Martin Kliesch, Tommaso Calarco, Jens Eisert, and Simone Montangero. 2016. Positive tensor network approach for simulating open quantum many-body systems. *Physical review letters* 116,

23 (2016), 237201.