PLOS ONE

RESEARCH ARTICLE

Validating quantum-classical programming models with tensor network simulations

Alexander McCaskey[1,2]*, **Eugene Dumitrescu**[1,3], **Mengsu Chen**[4], **Dmitry Lyakh**[1,5], **Travis Humble**[1,3,6]

1 Quantum Computing Institute, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, United States of America, **2** Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, United States of America, **3** Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, United States of America, **4** Department of Physics, Virginia Tech, Blacksburg, Virginia, 24060, United States of America, **5** National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, United States of America, **6** Bredesen Center for Interdisciplinary Research, University of Tennessee, Knoxville, Tennessee 37996, United States of America

* mccaskeyaj@ornl.gov

Citation: McCaskey A, Dumitrescu E, Chen M, Lyakh D, Humble T (2018) Validating quantum-classical programming models with tensor network simulations. PLoS ONE 13(12): e0206704. https://doi.org/10.1371/journal.pone.0206704

Editor: Nicholas Chancellor, Durham University, UNITED KINGDOM

Received: June 29, 2018

Accepted: October 15, 2018

Published: December 10, 2018

Data Availability Statement: Data underlying this study have been uploaded to Github and are available at https://github.com/ornl-qci/tnqvm in the examples/plos_one_experiments folder.

Funding: This work was supported by U.S. Department of Energy ASCR Quantum Algorithms Team (ERKJ332, Mr. Alexander McCaskey), U.S. Department of Energy ASCR Quantum Testbed Pathfinder (ERKJ335, Mr. Alexander McCaskey), Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ID 8297, Mr. Alexander McCaskey), and U.S.

Abstract

The exploration of hybrid quantum-classical algorithms and programming models on noisy near-term quantum hardware has begun. As hybrid programs scale towards classical intractability, validation and benchmarking are critical to understanding the utility of the hybrid computational model. In this paper, we demonstrate a newly developed quantum circuit simulator based on tensor network theory that enables intermediate-scale verification and validation of hybrid quantum-classical computing frameworks and programming models. We present our tensor-network quantum virtual machine (TNQVM) simulator which stores a multi-qubit wavefunction in a compressed (factorized) form as a matrix product state, thus enabling single-node simulations of larger qubit registers, as compared to brute-force state-vector simulators. Our simulator is designed to be extensible in both the tensor network form and the classical hardware used to run the simulation (multicore, GPU, distributed). The extensibility of the TNQVM simulator with respect to the simulation hardware type is achieved via a pluggable interface for different numerical backends (e.g., ITensor and ExaTENSOR numerical libraries). We demonstrate the utility of our TNQVM quantum circuit simulator through the verification of randomized quantum circuits and the variational quantum eigensolver algorithm, both expressed within the eXtreme-scale ACCelerator (XACC) programming model.

1 Introduction

Quantum computing is a computational paradigm that relies on the principles of quantum mechanics in order to process information. Recent advances in both algorithmic research, which has found remarkable speed-ups for a growing number of applications [1–3], and hardware development [4, 5] continue to progress the field of quantum information processing.

Department of Energy Early Career Award (Dr. Travis Humble). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exists.

The near-term state of quantum computing is defined by the noisy intermediate-scale quantum (NISQ) paradigm which involves small-scale noisy quantum processors [6] being used in a hybrid quantum-classical framework. In this context, recent experimental demonstrations [7–11] of hybrid computations have reinforced the need for robust programming models and classical validation frameworks.

The successful integration of quantum processors into conventional computational workloads is a complex task which depends on the programming and execution models that define how quantum resources interact with conventional computing systems [12, 13]. Many different models have been proposed for programming quantum computers and a number of software development efforts have begun focusing on high-level hybrid programming mechanisms capable of integrating both conventional and quantum computing processors together [14–21]. For example, recent efforts have focused on Python-based programming frameworks enabling the high-level expression of quantum programs in a classical context, which may target numerical simulators or a variety of physical quantum processing units (QPUs) [22–24]. The eXtreme-scale ACCelerator programming model (XACC) is a recently-developed quantum-classical programming, compilation, and execution framework that enables programming across multiple languages and QPU targets, including both virtual and physical QPUs [25].

In all cases, the verification of quantum program correctness is a challenging and complex task due to the intrinsically noisy nature of near-term QPUs, and this is additionally complicated by remote hosting. As a remedy, numerical simulation techniques can greatly expedite the analysis of quantum-classical programming efforts by providing direct insight into the prepared quantum states, as well as serving to test a variety of quantum computing hardware models. Modeling and simulation is essential for designing effective program execution mechanisms because it provides a controlled environment for understanding how complex computational systems interact, subsequently generating feedback based on the state machine statistics. For example, the performance of existing QPUs is limited by the hardware connectivity [4] and numerical simulations can draw on a broad range of parameterized models to test new processor layouts and architectures.

In practice, exact brute-force simulations of quantum computing are notoriously inefficient in memory complexity due to the exponential growth in resources with respect to the system size. These brute-force methods explicitly solve the Schrodinger equation, or a mixed-state master equation, using a full representation of the quantum state in its underlying (exponentially large) Hilbert space. Limits on available memory place upper bounds on the size of the vectors or density operators that can physically be stored, severely restricting the size of the simulated quantum circuit. Even with the availability of current large-scale HPC systems, including the state-of-the-art supercomputing systems, recent records for quantum circuit simulations are limited to less than 50 qubits [26, 27]. The performance of the brute-force quantum circuit simulators on current supercomputing architectures is also limited by the inherently low arithmetic intensity (Flop/Byte ratio) of the underlying vector operations (sparse matrix-vector multiplications) required for simulating a discrete sequence of one- and two-qubit gates.

The inherent inefficiency of the brute-force state-vector quantum circuit simulators has motivated a search for approximate numerical simulation techniques increasing the upper bound on the number of simulated qubits. As we are interested in general-purpose (universal) quantum circuit simulators, we will omit efficient specialized simulation algorithms that target certain subclasses of quantum circuits, for example, quantum circuits composed of only Clifford operations [28]. As a general solution, we advocate for the use of tensor network (TN) theory as a tool for constructing factorized approximations to the exact multi-qubit wave-

function tensor. The two main advantages offered by the tensor-network based wave-function factorization are (1) the memory (space) and time complexity of the quantum circuit simulation reflect the level of entanglement in the quantum system, (2) the numerical action of quantum gates on the factorized wave-function representation results in numerical operations (tensor contractions) which become arithmetically intensive for entangled systems, thus potentially delivering close to the peak utilization of modern HPC platforms.

2 Quantum circuit simulation with tensor networks

Tensor network theory [3, 29] provides a versatile and modular approach to dimensionality reduction in high-dimensional tensor spaces. For the following discussion, a *tensor* is a generalization of a vector that is defined in a linear space constructed as the direct (tensor) product of two or more primitive vector spaces. Consequently, the components of a tensor $T_{i_1 \ldots i_n}$ are enumerated by a tuple of indices, instead of by a single index as is the case for vectors. From the numerical perspective, a tensor can be viewed as a multi-dimensional array of objects, which may be real or complex numbers. In this work, following the physics nomenclature, we shall refer to the number of indices in a tensor $T_{i_1 \ldots i_n}$ as its *rank*, which is n in this case (in math nomenclature n is the tensor *order*). Each index represents a distinct vector space contributing to the composite space defined by the tensor product. The extent of the range of each index gives the dimension of the corresponding vector space. In their essence, tensor networks aim at decomposing a higher-rank tensor into a contracted product of lower-rank tensors (tensor factors) such that this factorized product of lower-rank tensors reconstructs the original tensor with sufficient accuracy (i.e. a variant of lossy compression in linear spaces). In principle, any tensor can be approximated by a tensor network with arbitrary precision [30], however the size of the constituent tensor factors may become prohibitively large in worst case examples, showing that the chosen tensor network delivers a poor compression. Tensor factorizations, which we also refer to as decompositions, are not unique in general and the problem of finding the optimal tensor decomposition is a difficult non-convex optimization problem [31].

In practice, tensor network factorization is typically specified by a graph in which the nodes are the tensor factors and the edges represent physical or auxiliary vector spaces which are associated with the indices of the corresponding tensor factors. In this representation, a graph node with n edges is a rank-n tensor with each of its n indices uniquely associated with a specific edge. A closed edge, that is, an edge connecting two nodes, represents a contracted index shared by two tensor factors over which a summation is to be performed. In a standalone tensor network, contracted indices are associated with auxiliary vector spaces. An open edge, that is, an edge connected to only one node, represents an uncontracted index of that corresponding tensor factor. Uncontracted indices in a standalone tensor network are typically associated with physical vector spaces. Different tensor network architectures differ by the topology of their representative graphs. Furthermore, one can define even more general tensor network architectures by replacing graphs with hypergraphs, in which case an edge may connect three or more tensors, thus representing a summation over the repeated index in three or more tensors, respectively. In the subsequent discussion, however, we will only deal with conventional graph topologies.

A quantum many-body wave-function, including a multi-qubit wave-function, is essentially a high-rank tensor (its rank is normally equal to the number of simulated quantum particles, quasi-particles, or quantum spins) [29]. A number of different tensor network architectures have been suggested for the purpose of factorizing quantum many-body wave-functions, including the matrix-product state (MPS) [32, 33], the projected entangled pair state (PEPS) [34, 35], the tree tensor network state (TTNS) [36–38], the multiscale entanglement

renormalization ansatz (MERA) [39, 40], as well as somewhat related non-conventional schemes like the complete-graph tensor network (CGTN) [41]. All of the above tensor network *ansaetze* differ in the factorization topology, that is, in how the tensor factors are contracted with each other to form the final quantum many-body wave-function tensor. In a good tensor network factorization, the graph topology is induced by the entanglement structure of the quantum many-body state under study. Many physical systems are described by many-body Hamiltonians with only local interactions—in many cases, nearest neighbor only—with correlation functions decaying exponentially for non-critical states. In such cases, the locality structure of the many-body Hamiltonian induces the necessary topology required to properly capture the quantum correlations present in the system of interest. The factorization topology also strongly affects the computational cost associated with the numerical evaluation/optimization of a specific tensor network architecture. Another important characteristic of a tensor network is its so-called maximal bond dimension, χ, that is, the maximal dimension of auxiliary vector spaces (auxiliary vector spaces are those contracted over). Provided that the maximal bond dimension is bounded, many tensor network factorizations can be efficiently evaluated due to a moderate polynomial computational cost (in the bond dimension) associated with the computed physical expectation values. In practice, the entanglement structure of the underlying quantum many-body state determines the maximal bond dimension in a given tensor network for a given error tolerance. A poorly chosen tensor network topology will necessarily lead to rapidly increasing (exponentially at worst) bond dimensions in order to keep the factorization error within the tolerable error threshold [30].

The entanglement structure in a multi-qubit wave-function is determined by the quantum circuit and may be unknown in general. Consequently, there is no well-defined choice of a tensor network architecture (topology) that could work equally well for all quantum circuits, unless it is some kind of an adaptive topology. In practice, the choice of a tensor network architecture for representing a multi-qubit wave-function is often dictated by numerical convenience and ease of implementation. For example, one of the simplest tensor network architectures, the MPS ansatz illustrated in Fig 1, was used to simulate Shor's algorithm for integer factorization [42]. Although the inherently one-dimensional chain topology of the MPS ansatz

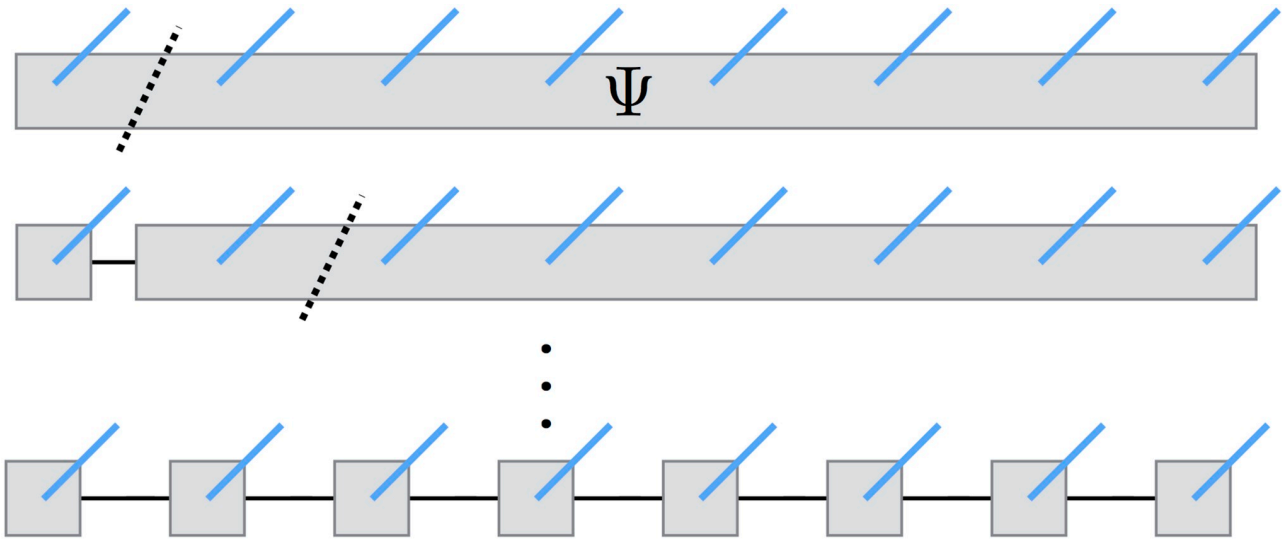

Fig 1. Decomposition of the multi-qubit wave-function tensor into the matrix-product state (MPS) tensor network, replacing a single node (tensor) having *N* (open) edges with *N* nodes (tensor factors) having at most three edges each.

https://doi.org/10.1371/journal.pone.0206704.g001

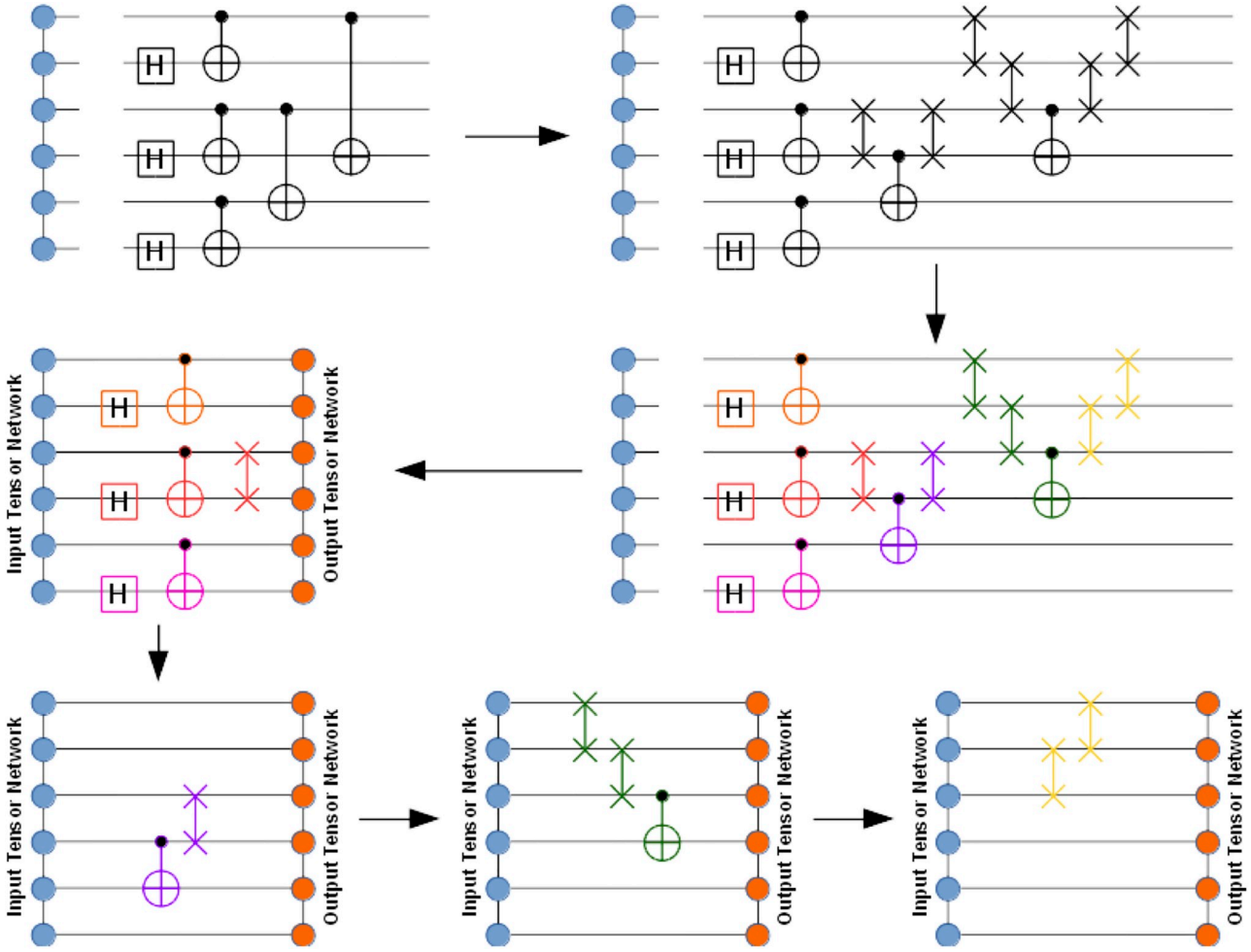

Fig 2. Graphical illustration of the general quantum circuit simulation algorithm with the multi-qubit wave-function factorized as the MPS tensor network. Gate coloring represents aggregation of individual gates into super-gates, which act as a whole. The figure shows progression of super-gate actions on the MPS tensor network. Multiple super-gates may be involved in a single action. Each action results in an updated MPS tensor network (output tensor network). Note that the application of a super-gate does not necessarily affect all output MPS tensors, only requiring an update of a subset of them that is actually affected by the super-gate. In general, the affected tensors are determined by the qubits involved in the super-gate as well as the specific tensor network architecture.

https://doi.org/10.1371/journal.pone.0206704.g002

often results in severely growing bond dimensions, and this can be remedied by a more judicious tensor network form [38], its computational convenience and well understood theory makes the MPS factorization an appealing first candidate for our quantum virtual machine (quantum circuit simulator). In future, we plan on adding more advanced tensor network architectures, however.

In order to simulate a general quantum circuit over an N-qubit register with the tensor network machinery the following steps will be necessary (see Fig 2):

1. Specify the chosen tensor network graph that factorizes the rank-N wave-function tensor into a contracted product of lower-rank tensors (factors). For example, one may choose the MPS factorization as done in Fig 2.
2. Transform the quantum circuit into an equivalent quantum circuit augmented with SWAP gates in order to maximize the number of accelerated gate applications (see below). This is an optional step.

3. Group quantum gates into larger aggregates (super-gates) which will act as a whole on the relevant part of the multi-qubit wave-function. In the simplest case, all elementary quantum gates will be distinguished individually, with no aggregation. The purpose of the aggregation step is to reduce the total number of gates in the quantum circuit and to increase the compute intensity associated with their action on the wave-function tensor network. This is an optional step.
4. Apply aggregated super-gates (or individual gates when no aggregation occurred) to the relevant parts of the wave-function tensor network, thus evolving towards the output state. Multiple super-gates can be applied simultaneously. In general, the application of a super-gate will affect the qubits associated with that super-gate as well as possibly other qubits affected indirectly because of the specific form of the tensor network. For example, in the MPS factorization case the application of a 2-body super-gate to qubits i and j may need to also update the MPS tensors associated with the internal qubits located between the qubits i and j.

In the above general algorithm, the action of the super-gates (or just individual gates) on a multi-qubit wave-function tensor network consists of the following steps:

1. Append a given set of super-gates (or just individual gates) to the input wave-function tensor network TN_{inp}, thus obtaining a larger tensor network TN_{mid}. An n-body super-gate, represented by a rank-$2n$ tensor, is appended to the tensor network graph by pairing its n edges with the corresponding edges of the tensor network that are associated with the qubits the super-gate acts on. This is illustrated in the bottom part of Fig 2 where one or more super-gates are appended to the input tensor network depicted as a graph of circles on the left (followed by a closure by the output tensor network depicted as a graph of circles on the right).
2. If there are 2- or higher-body super-gates present, check whether they are applied to the qubit pairs or triples, etc. that allow accelerated gate application (for example, in MPS factorization, these would be the adjacent qubit pairs, triples, and so on). If yes, evaluate their action in an accelerated fashion (see below). Otherwise, resort to the general algorithm in the next steps.
3. Instantiate a new tensor network TN_{out} by cloning TN_{inp} and complex conjugating all constituent tensors.
4. Close TN_{mid} with TN_{out} by pairing all edges between the two tensor networks, thus obtaining a closed tensor network TN_{opt} which does not have open edges (see the bottom part of Fig 2). TN_{opt} evaluates to a scalar since all tensor indices have been contracted. It represents the inner product between the state obtained by the action of the super-gate(s) on the input tensor network and the state parameterized by the output tensor network. Thus, we can approximate the state obtained by the action of the super-gate(s) on the input tensor network by maximizing the obtained inner product while keeping the output tensor network normalized.
5. In the obtained inner product, contract away some or, if possible, all tensors that will not undergo any changes in value, thus reducing the total number of tensors in TN_{opt} (see Fig 3). As mentioned above, depending on the tensor network architecture, only a subset of output tensors may need to be updated for a given super-gate(s) application. Note that such a simplification of TN_{opt} is not always practical as the contraction of the unaffected tensors

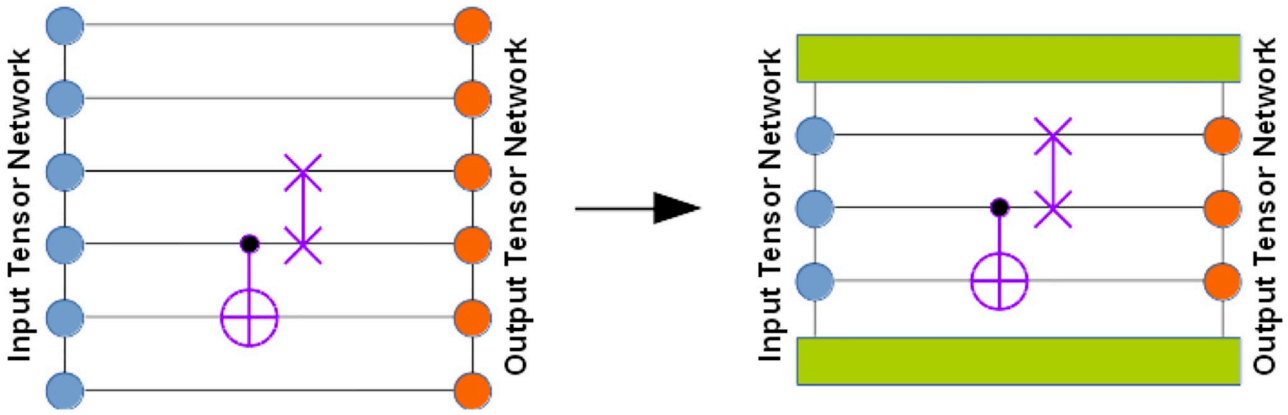

Fig 3. Graphical illustration of the simplification of the inner product being optimized: Some tensors that do not undergo an update can be contracted in order to reduce the total number of tensors while keeping memory requirements mostly intact.

https://doi.org/10.1371/journal.pone.0206704.g003

may yield unmanageably large higher-rank tensors, in which case it should be abandoned, either partially or fully.

6. Optimize the affected tensors of TN_{out} to maximize the TN_{opt} scalar, subject to keeping TN_{out} normalized. For example, one can follow a general algorithm suggested in Ref. [40] where the optimization procedure is based on the evaluation of the gradients of TN_{opt} with respect to each optimized tensor of TN_{out} and using SVD for keeping the updated constituent tensor factors isometric, which should preserve their normalization as a byproduct (isometric tensors are composed of orthonormal columns when they are flattened in a matrix over specific tensor modes). Alternatively, one can add the output tensor network normalization condition (as well as other necessary conditions on tensor factors) to the inner product TN_{opt} to get a constrained optimization problem, as shown in Fig 4. In practice, the optimized output tensors are updated one or two (if adjacent) at a time, based on the corresponding constrained gradients, and the full optimization epoch includes a sweep over all optimized tensors. The advantage of optimizing a pair of connected tensor factors at a time is that one can perform an SVD on the combined tensor to get the updated tensor factors.
7. If the maximum TN_{opt} value is not acceptable after some predefined number of iterations, increase dimensions of the auxiliary spaces in the affected tensors of TN_{out} and repeat Step 6.

In cases where an accelerated gate application is possible (for example, a 2-body gate is applied to the adjacent qubits in the MPS-factorized wave-function), one can restrict the update procedure only to the tensor factors directly affected by the gate action. In case of MPS factorization, in order to apply a 2-body gate to two adjacent qubits one can contract the gate tensor with the two MPS tensors representing the affected qubits and then perform a singular value decomposition (SVD) on the tensor-result, thus immediately obtaining new (updated) MPS tensors as illustrated in Fig 5.

The above general algorithm demonstrates the procedure the TNQVM simulator is designed to use for approximate simulation of quantum circuits based on the tensor network factorizations. For the sake of completeness, we should also mention quantum circuit simulators which use tensor representations for a brute-force simulation of quantum circuits with no approximations [27, 43]. This is different from our approach which is based on the explicit

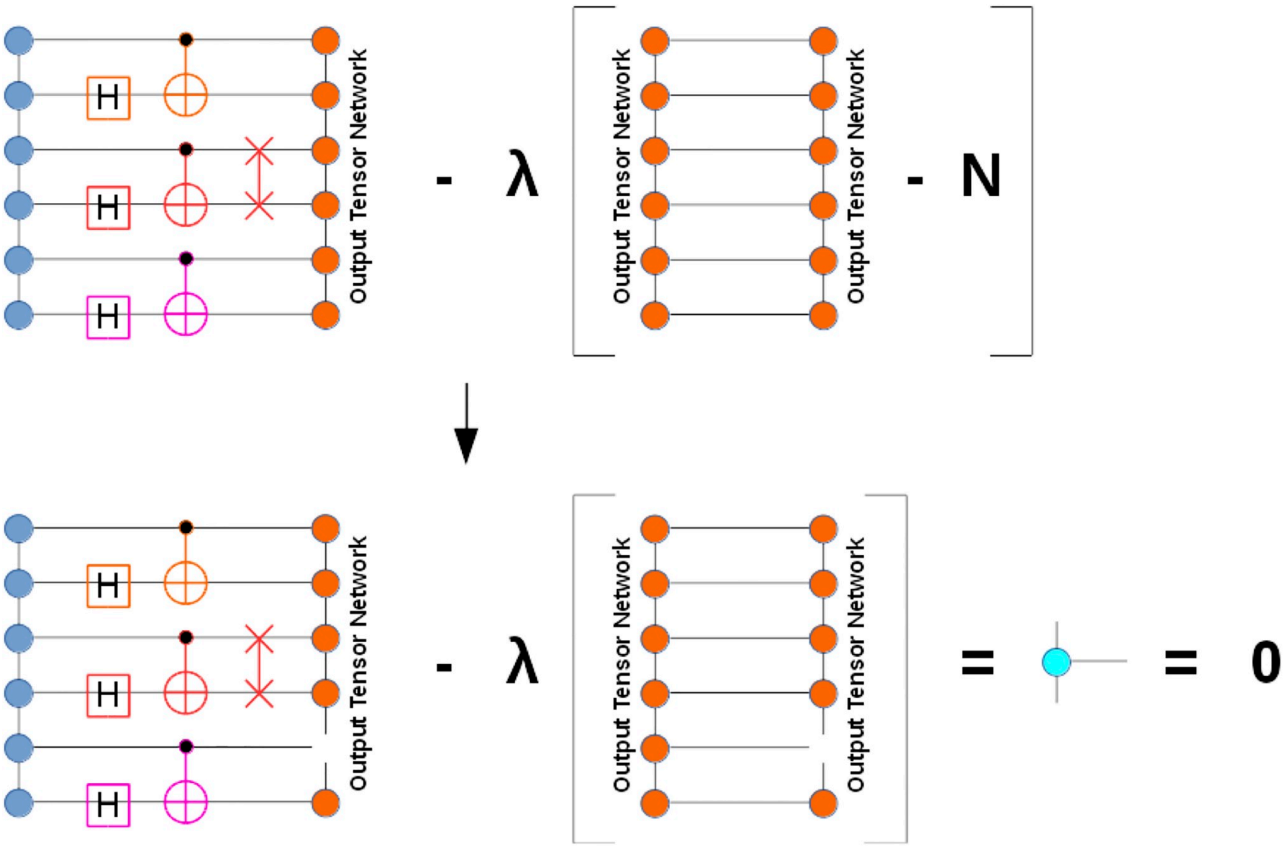

Fig 4. Graphical illustration of the constrained optimization problem for the output MPS tensor network with a simple constraint of normalization to N. In general, additional constraints can apply to the tensor network factors, for example requirement of isometry or unitarity (see Ref. [40]).

https://doi.org/10.1371/journal.pone.0206704.g004

factorization of the multi-qubit wave-function tensor. In these other tensor-based schemes the entire quantum circuit as a collection of gate tensors is considered as a tensor network which is subsequently contracted over in order to compute observables or evaluate output probability distributions. In Ref. [27], a clever tensor slicing technique was introduced that avoided the evaluation of the full wave-function tensor, thus reducing the memory footprint and bypassing the existing 45-qubit limit on large-scale HPC systems. Yet, despite enabling simulations of

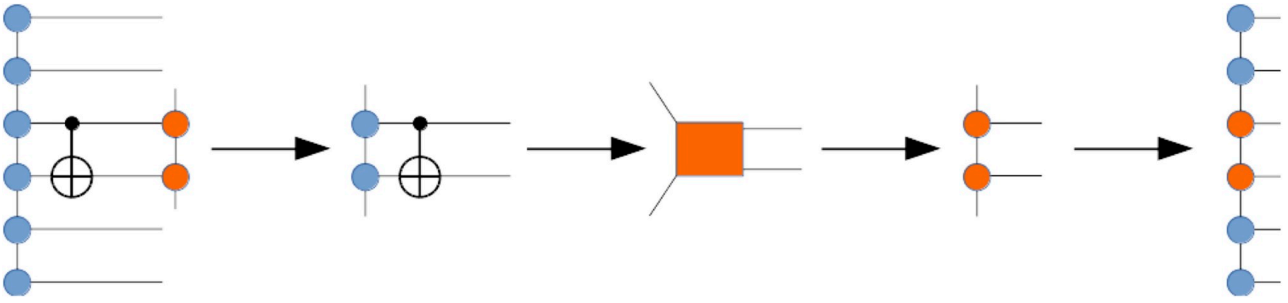

Fig 5. Graphical illustration of an accelerated evaluation of the action of a two-body gate on a pair of adjacent qubits in the MPS representation. The SVD procedure decomposes the rank-4 tensor into a pair of contracted rank-3 tensors, thus immediately producing the updated MPS tensors.

https://doi.org/10.1371/journal.pone.0206704.g005

somewhat larger qubit counts, this technique does not lift the asymptotic bounds of the exact simulation cost.

3 Quantum virtual machines

In order to evaluate the correctness of a quantum program and its implementation via a decomposition into primitive gate operations, it is necessary to model both the conventional computing and quantum computing elements of the system architecture. In particular, it is necessary to expose the interface to the available instruction set architecture (ISA) and methods to support quantum program execution, scheduling, and layout. There are currently many different technologies available for testing and evaluating quantum processing units, and each of these technologies presents different ISAs and methods for program execution [44].

As shown in Fig 6, a quantum virtual machine (QVM) provides a portable abstraction of technology-specific details for a broad variety of heterogeneous quantum-classical computing architectures. The hardware abstraction layer (HAL) defines a portable interface by which the underlying quantum processor technology as well as other hardware components such as memory are exposed to system libraries, runtimes and drivers running on the host conventional computer. The implementation of the HAL provides an explicit translation of quantum program instructions into native, hardware-specific syntax, which may be subsequently executed by the underlying quantum processor. The HAL serves as a convenience to ensure portability of programs across different QPU platforms, while the QVM encapsulates the environment in which applications can be developed independently from explicit knowledge of QPU details. This environment is provided by the integration of the HAL with programming tools, libraries, and frameworks as well as the host operating system.

Application performance within a QVM depends strongly on the efficiency with which host programs are translated into hardware-specific instructions. This includes the communication overhead between the HAL and hardware layers as well as the overhead costs for managing these interactions by the host operating system. Both algorithmic and hardware designs impact this performance by deciding when and how to allocate computational burden to specific devices. Presently, there is an emphasis on the development and validation of hybrid programs, which loosely integrates quantum processing with conventional post-processing tasks. This algorithmic design introduces a requirement for transferring memory buffers between the host and QPU systems. Memory management therefore becomes an important task for application behavior. While current QPUs are often accessed remotely through network interfaces, long-term improvements in application performance will require fine grain control over memory management.

4 Tensor network quantum virtual machine

Our implementation of a QVM presented in this work is based on a previously developed hybrid quantum-classical programming framework, called XACC [25], combined with a quantum circuit simulator that uses tensor network theory for compressing the multi-qubit wavefunction. We provide an overview of the Tensor Network Quantum Virtual Machine (TNQVM) and its applications, including its software architecture and integration with the XACC programming framework. Since XACC integrates directly with TNQVM, compiled programs can in principle be verified instantaneously on any classical computer including workstations as well as HPC clusters and supercomputers. The support of different classical computer architectures (single-core, multi-core, GPU, distributed) for performing numerical simulations is achieved by interchangeability of the numerical backends in our TNQVM simulator. These backends are numerical tensor algebra libraries which perform all underlying

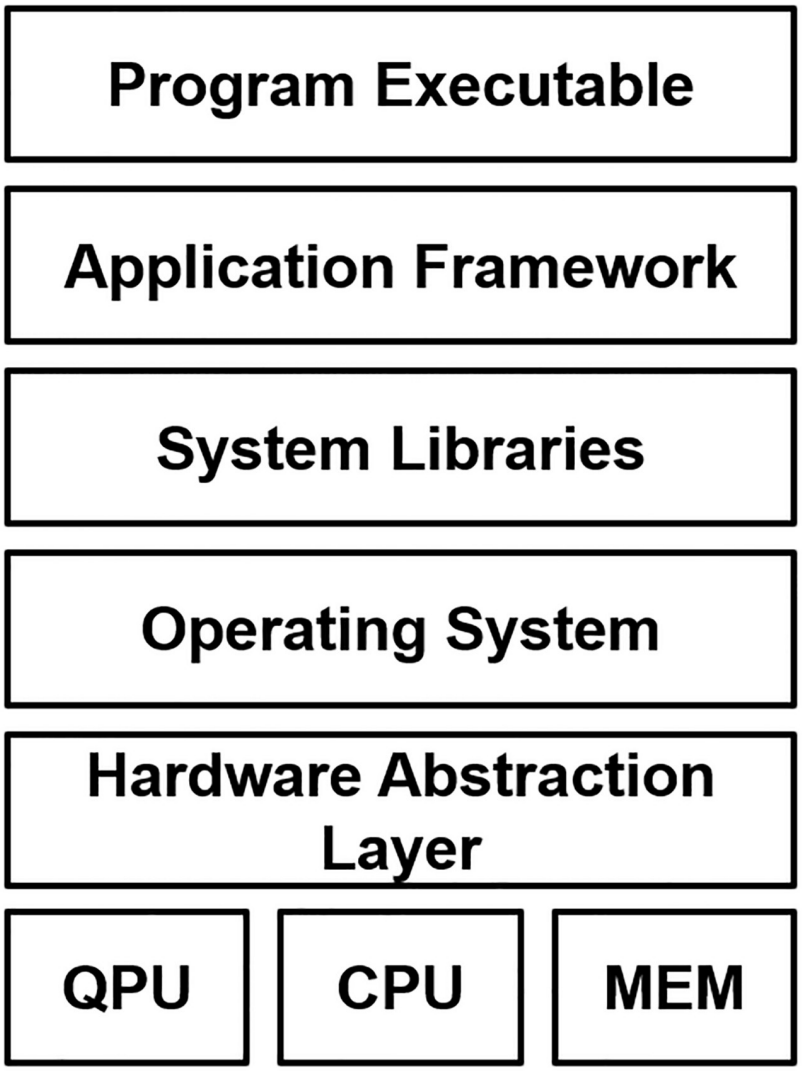

Fig 6. A schematic design how a quantum virtual machine (QVM) manages access to an underlying QPU through the hardware abstraction layer. A program binary exists within an application framework that accesses system resources through libraries. Library calls are managed by the host operating system, which manages and schedules requests to access hardware devices including attached QPUs. The hardware abstraction layer (HAL) provides a portable interface by which these requests are made to the underlying QPU technology.

https://doi.org/10.1371/journal.pone.0206704.g006

tensor computations on a supported classical computer. In this work, we detail the HAL implementation of TNQVM using ITensor [45] for serial simulations, with some example applications demonstrating the utility of TNQVM. We also sketch some details on the upcoming ExaTENSOR backend that will enable large-scale quantum circuit simulations on distributed homo- and heterogeneous HPC systems. Independent verification of hybrid programs within TNQVM provides an increased confidence in the use of these codes to characterize and validate actual QPUs.

4.1 XACC

The eXtreme-scale ACCelerator programming model (XACC) has been specifically designed for enabling near-term quantum acceleration within existing classical high-performance

computing applications and workflows [25, 46]. This programming model and associated open-source reference implementation follow the traditional co-processor model, akin to OpenCL or CUDA for GPUs, but takes into account the subtleties and complexities arising from the interplay between classical and quantum hardware. XACC provides a high-level application programming interface (API) that enables classical applications to offload quantum programs (represented as quantum kernels, similar in structure to GPU kernels) to an attached quantum accelerator in a manner that is agnostic to both the quantum programming language and the quantum hardware. Hardware agnosticism enables quantum code portability and also aids in benchmarking, verification and validation, and performance studies for a wide array of virtual (simulators) and physical quantum platforms.

To achieve language and hardware interoperability, XACC defines three important abstractions: the quantum intermediate representation (IR), compilers, and accelerators. XACC compiler implementations map quantum source code to the IR—the in-memory object key to integrating a diverse set of languages to a diverse set of hardware. IR instances (and therefore compiled kernels) are executed by sub-types of the accelerator, which defines an interface for injecting physical or virtual quantum hardware. Accelerators take this IR as input and delegate execution to vendor-supplied APIs for the QPU, or an associated API for a simulator. This forms the hardware abstraction layer, or abstract device driver, necessary for a general quantum (virtual) machine.

The IR itself can be further decomposed into instruction and function abstractions, with instructions forming the foundation of the IR infrastructure and functions serving as compositions of instructions (see Fig 7). Each instruction exposes a unique name and the set of qubits that it operates on. Functions are a sub-type of the instruction abstraction that can contain further instructions. This setup, the familiar composite design pattern [47], forms an *n-ary* tree of

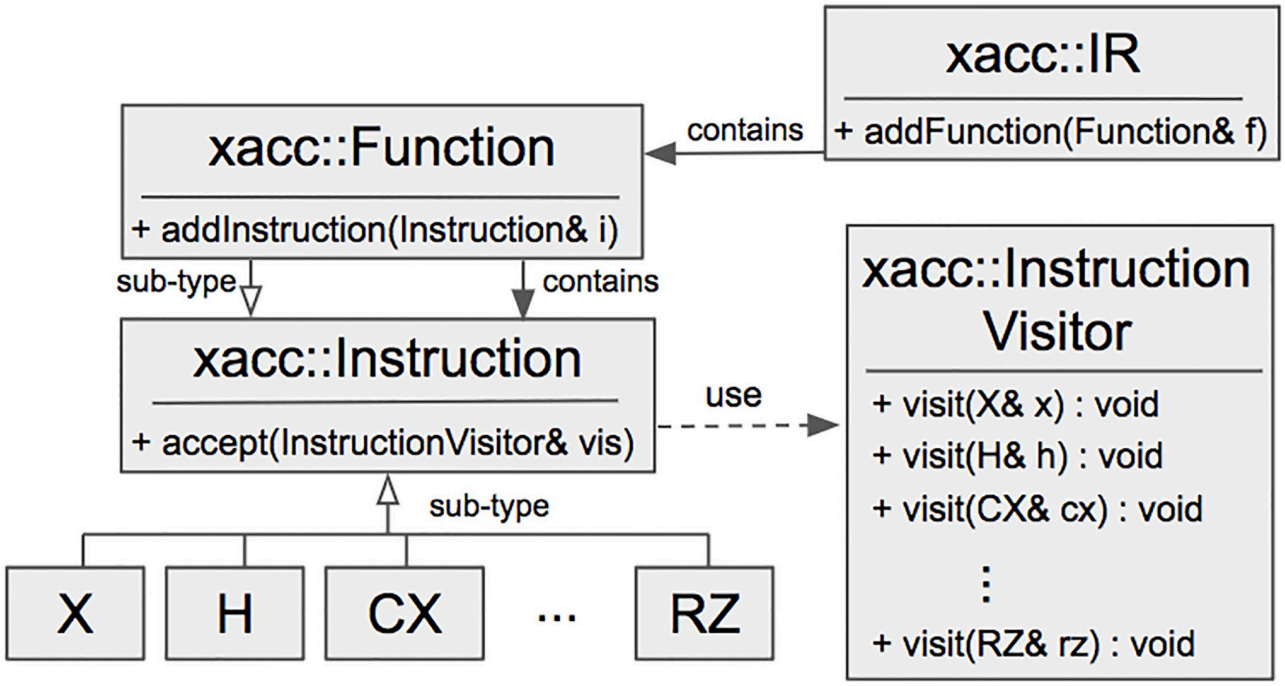

Fig 7. Architecture of the XACC intermediate representation demonstrating sub-type extensibility for instructions, and the associated instruction visitor abstraction, enabling runtime extension of concrete instruction functionality.

https://doi.org/10.1371/journal.pone.0206704.g007

instructions where function instances serve as nodes and concrete instruction instances serve as leaves.

Operating on this tree and executing program instructions is a simple pre-order traversal on the IR tree. In order to enhance this tree of instructions with additional functionality, XACC provides a dynamic double-dispatch mechanism, specifically an implementation of the familiar visitor pattern [48]. The visitor pattern provides a mechanism for adding virtual functions to a hierarchy of common data structures dynamically, at runtime, and without modifying the underlying type. This is accomplished via the introduction of a visitor type that exposes a public set of visit functions, each one taking a single argument that is a concrete sub-type of the hierarchical data structure composition (see Fig 7). For gate model quantum computing, XACC exposes a visitor class that exposes a visit method for all concrete gate instructions (X, H, RZ, CX, etc. . .). All instructions expose an `accept` method that takes as input a general visitor instance, and invokes the appropriate visit method on the visitor through double-dispatch. XACC instruction visitors thereby provide an extensible mechanism for dynamically operating on, analyzing, and transforming compiled IR instances at runtime.

4.2 Tensor network accelerator and instruction visitors

The integration of a tensor network quantum circuit simulator with XACC can be accomplished through extensions of appropriate XACC components. In essence, this is an extension of the quantum virtual machine hardware abstraction layer that enables existing high-level programs and libraries to target a new virtual hardware instance. Injecting new simulators into the XACC framework requires a new implementation of the accelerator. Enabling that simulator to be extensible in the type of tensor networks, algorithmic execution, and the library backend requires different mappings of the IR to appropriate simulation data structures. This can be accomplished through individual implementations of the instruction visitor.

Our open-source implementation of the Tensor Network Quantum Virtual Machine (TNQVM) library extends the XACC accelerator with a new derived class that simulates pure-state quantum computation via tensor network theory [49]. This library provides the TNAccelerator (Tensor Network Accelerator) that exposes an `execute` method that takes as input the XACC IR function to be executed. Generality in the tensor network graph structure and the simulation algorithm is enabled through appropriate implementations of the instruction visitor. For example, an instruction visitor can be implemented to map the incoming XACC IR tree to tensor operations on a matrix product state (MPS) ansatz. Walking the IR tree via pre-order traversal and invoking the instruction visitor `accept` mechanism on each instruction triggers invocation of the appropriate visit function via double dispatch. The implementation of these visit methods provides an extensible mechanism for performing instruction-specific tensor operations on a specific tensor network graph structure.

Furthermore, this visitor extension mechanism can be leveraged to not only provide new tensor network structures and operations, but also provide the means to leverage different tensor algebra backend libraries, and therefore introduce a classical parallel execution context. Different visitor implementations may provide a strictly serial simulation approach, while others can enable a massively parallel or heterogeneous simulation approach (incorporating the Message Passing Interface, OpenMP, and/or GPU acceleration via CUDA or OpenCL).

To date we have implemented two instruction visitor backends for the TNQVM and the TNAccelerator. We have leveraged the ITensor library [45] to provide a serial matrix product state simulator, and the ExaTENSOR library from the Oak Ridge Leadership Computing Facility (OLCF) to provide a matrix product state simulator that leverages MPI, OpenMP and

CUDA for distributed parallel execution on GPU-accelerated heterogeneous HPC platforms. However, the ExaTENSOR visitor is not fully available yet since the ExaTENSOR library is currently undergoing final testing before its public release and the implementation of the generic tensor optimization procedure is still in progress. Thus, it has not been utilized yet as a fully functional backend of TNQVM. Nevertheless, we will provide some details on the ExaTENSOR backend below in order to highlight its design and our future plans.

4.2.1 ITensor MPS implementation. The ITensor MPS instruction visitor implementation provides a mechanism for the simulation of an N-qubit wavefunction via a matrix product state tensor network decomposition. The MPS provides a way to restrict the entanglement entropy through SVD and associated truncation of Schmidt coefficients to reduce the overall Schmidt rank. With these MPS states, we need $O(n\chi^2)$ numbers to represent n qubits, where χ is the largest Schmidt rank we keep. As long as χ is not too large (grows polynomially with system size), the space complexity is feasible for classical simulation. For example, if the quantum register is used to store the gapped ground states of systems with local interactions, we can simulate larger number of qubits and still adequately approximate the wavefunction by keeping χ small enough.

Our ITensor MPS visitor implementation begins by initializing a matrix product state tensor network using the serial tensor data structures provided by the ITensor library [45]. Simulation of the compiled IR program is run through a pre-order tree traversal of the instruction tree. At each leaf of this tree (a concrete instruction), the `accept` method on the instruction is invoked (see Fig 7) which dispatches a call to the correct `visit` method of the instruction visitor.

At this point, the appropriate gate tensor is contracted into the MPS representation, which maps onto itself under local quantum gates. Updating the MPS according to two-body entanglers involves two-qubit gates which act on two rank-3 tensors, and the full contraction results in a rank-4 tensor (see Fig 5). We maintain the MPS structure by decomposing the rank-4 tensor into two rank-3 tensors and a diagonal matrix between them. Note that when the two qubits are not adjacent we apply SWAP gates on intermediary qubits to bring them together (see Fig 2). The gate is then applied and reverse SWAPs bring the qubits back to the original positions. Otherwise, applying a gate to non-adjacent qubits would require using the general tensor network optimization algorithm described in the previous section, which we do not do yet.

The SVD is used to return the resulting rank-4 tensor to the canonical MPS form (n rank-3 tensors and $n-1$ diagonal matrices), with the singular values below a cutoff threshold ϵ (e.g., default is $\epsilon = 10^{-4}$) being truncated. The truncation over subspaces supporting exponentially small components of the wave-function allows our MPS-based TNQVM to simulate large numbers of qubits, contingent on the level of entanglement in the system. Examples and discussion may be found in the demonstrations in Sec. 5.

4.2.2 ExaTENSOR MPS implementation. The ExaTENSOR numerical tensor algebra backend will enable larger-scale TNQVM quantum circuit simulations on distributed, GPU-enabled and other accelerated as well as conventional multicore HPC platforms. ExaTENSOR stores tensors in distributed memory (on multiple/many nodes) as a generally sparse collection of tensor slices in a hierarchical fashion, that is, a tensor is defined recursively as a collection of constituent tensors (the hierarchical storage is the key when dealing with heterogeneous HPC architectures). Distributed tensor storage lifts the memory limitations pertinent to a single node, thus extending the maximal number of simulated qubits. Although we currently target the (distributed) MPS implementation, ExaTENSOR already provides a generic tensor network builder that can be used for constructing an arbitrary tensor network in future. The ExaTENSOR MPS visitor implementation will provide a constructor that creates the MPS

representation of the simulated multi-qubit wave-function with all constituent MPS tensors being distributed now. Then the XACC IR tree traversal will invoke the ExaTENSOR MPS `visit` method for each traversed node (instruction). The `visit` method implements lazy visiting, namely it only caches the corresponding instruction (gate) in the instruction cache of the ExaTENSOR MPS visitor. At some point, once the instruction cache has enough work to perform, the `evaluate` method of the ExaTENSOR visitor will be invoked, implementing the generic gate action algorithm described in Section II. Specifically, it will allocate the output MPS tensor network, that is, the result of the action of the cached gates on the input MPS tensor network. Then it will create the closed (inner product) tensor network by joining the gate tensors to the input MPS tensor network, subsequently closing it with the output tensor network (see Fig 2). This closed tensor network is a scalar whose value needs to be maximized, subject to normalization condition on the output tensor network. The ExaTENSOR MPS visitor will utilize the standard gradient descent algorithm by evaluating the gradients with respect to each tensor constituting the output tensor network. Each of these gradients is itself an open tensor network that needs to be fully contracted into a single tensor. Importantly, the computational cost of this contraction of many tensors strongly depends on the order in which the pairwise tensor contractions are performed. Finding the optimal tensor contraction sequence is an NP-hard problem. Instead, ExaTENSOR implements a heuristic algorithm that delivers a pseudo-optimal sequence of pairwise tensor contractions in a reasonable amount of time (subseconds). Then this pseudo-optimal sequence of pairwise tensor contractions is cached for a subsequent reuse, if needed. Given a sequence of pairwise tensor contractions, the ExaTENSOR library numerically evaluates all of them and returns the gradients that will subsequently be used for updating the output tensor network tensors, until the optimized inner product scalar reaches the desired threshold. In case it does not reach the desired value, the tensors constituting the output tensor network will be reallocated with increased dimensions of the auxiliary spaces and the entire procedure is to be repeated. As of now, an early prototype implementation of the ExaTENSOR MPS visitor in TNQVM is based on the single-node version of the ExaTENSOR library and we are currently finishing the integration of TNQVM with the fully distributed, GPU-accelerated version of the ExaTENSOR library as well as performing the final testing of the ExaTENSOR library itself before its public release later this year. Also, we plan to have a full support for the generic tensor optimization procedure described in Fig 2 soon.

5 Demonstration

Here we demonstrate the utility of TNQVM by describing the overall memory scaling of our matrix product state TNQVM based on the ITensor *visitor* for varying levels of entanglement and system size. Our demonstrations show how TNQVM can be leveraged for validating hybrid quantum-classical programming models. Specifically we focus on random circuit simulations and the variational quantum eigensolver (VQE) hybrid algorithm.

5.1 Profiling random circuit simulations with MPS

We demonstrate the improved resource cost of representing quantum states ($O(n\chi^2)$ vs $O(2^n)$) with TNQVM by using an MPS formulation and by profiling the memory usage of simulating randomly generated circuits. We vary the entanglement structure of our random circuits by constructing time slices defined as *rounds*. The first round begins with a layer of Hadamard operations on all qubits, followed by a layer of single qubit gates (Pauli gates and other general rotations), followed by a set of nearest-neighbor CNOT entangling operations. Multiple rounds constitute multiple iterations of generating these layers (excluding the Hadamards,

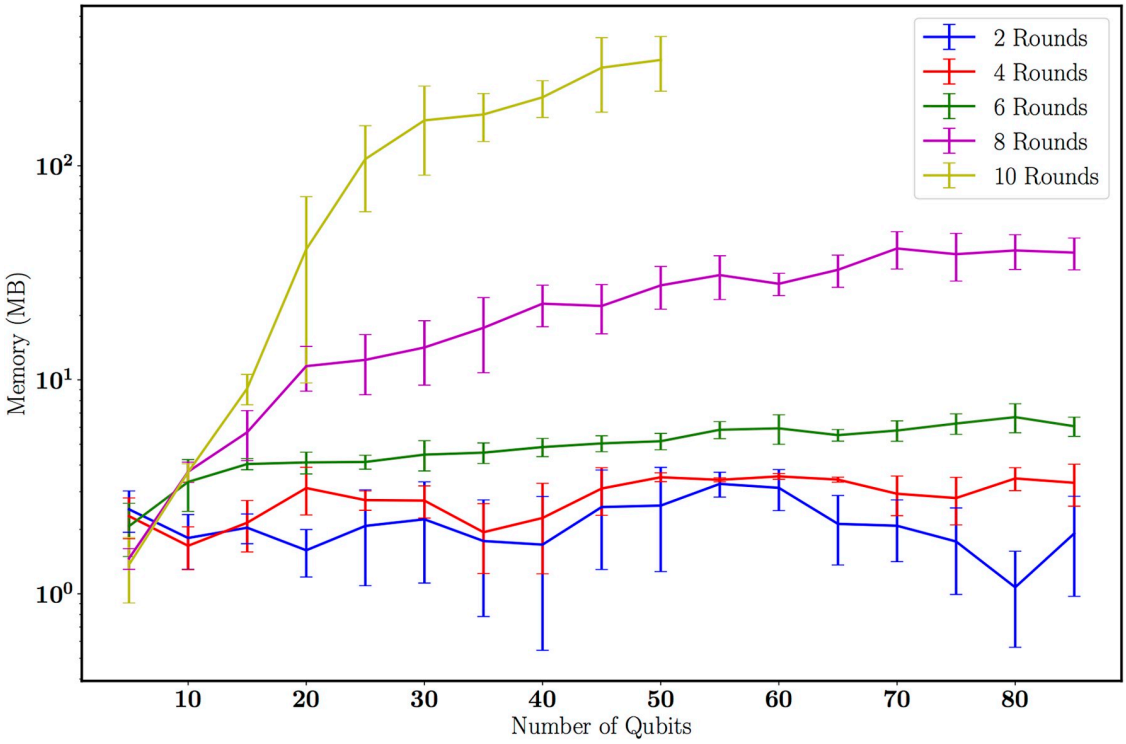

Fig 8. Memory usage as a function of the number of rounds (circuit depth) and with increasing number of qubits. Memory usage is constant for a small number of rounds but rapidly increases as the total circuit depth and number of qubits increases.

https://doi.org/10.1371/journal.pone.0206704.g008

which only appear in the first layer). Clearly, later rounds add layers of entangling CNOT operations and therefore generate states with a more complicated entanglement structure.

We generate these random circuits for 5 through 85 qubits in increments of 5, and for numbers of rounds ranging from 2 through 10 in increments of 2. For each (*round*, $n - qubits$) pair, we generate 10 random circuits, compute the heap memory usage, and compute the mean and standard deviation of the memory usage. The results are plotted in Fig 8. For lightly-entangled systems (i.e. those generated by a small number of random rounds) we see that the MPS structure is able to encode the wavefunction of the system efficiently with a small cost. For example, for only two rounds the maximum bond dimension is $\chi = 4$, which is independent of system size. As we increase the entanglement in our random circuits, the computational cost of the MPS simulations increases exponentially. This is because the circuits we have sampled from are designed to exponentially increase the entanglement which saturates at $\chi_{max} = 2^{n/2}$ for an n-qubit system undergoing $m > n$ random rounds [50].

5.2 Variational quantum eigensolver

Finally, we demonstrate the utility of our tensor network simulation XACC Accelerator backend (the TNQVM library) in validating quantum-classical algorithms. It is this rapid feedback mechanism that is critical to understanding intended algorithmic results, and enables confidence in the programming of larger systems. Here we demonstrate this programmability and its verification and validation through a simple simulation of diatomic hydrogen via the variational quantum eigensolver algorithm. The quantum-classical program leveraging the TNQVM library is shown in the listing in Fig 9. This code listing demonstrates the integration of XACC and our tensor network accelerator implementation. The code shows how to

```
h2_src = """
__qpu__ ansatz(AcceleratorBuffer b,
                                double t0) {
  RX(3.1415926) 0
  RY(1.57079) 1
  RX(7.85397) 0
  CNOT 1 0
  RZ(t0) 0
  CNOT 1 0
  RY(7.8539752) 1
  RX(1.57079) 0
}
__qpu__ term0(AcceleratorBuffer b, double t0) {
  ansatz(b, t0)
  MEASURE 0 [0]
}
... (rest of measurement kernels)
"""
qpu = xacc.getAccelerator('tnqvm')
buffer = qpu.createBuffer('q',2)

p = xacc.Program(qpu, h2_src)
p.build()
kernels = p.getKernels()

for t0 in np.linspace(-np.pi,np.pi,100):
    for k in kernels[1:]:
        k.execute(buffer,
                  [xacc.InstructionParameter(t0)])
```

Fig 9. XACC program compiling and executing the variational quantum eigensolver for the H_2 molecule.

https://doi.org/10.1371/journal.pone.0206704.g009

program, compile, and execute the VQE algorithm to compute expectation values for the simplified (symmetry-reduced), two qubit H_2 Hamiltonian (see [51]). We start off by defining the quantum source code as XACC quantum kernels (note—we have left out a few measurement kernels for brevity). Each of these kernels is parameterized by a single `double` representing the variational parameter for the problem ansatz circuit (the `ansatz` kernel in the `h2_src` string). Integration with the TNQVM simulation library is done through a public XACC API

function (`getAccelerator`). This accelerator reference is used to compile the program and get reference to executable kernels that delegate work to the TN Accelerator. We then loop over all θ and compute the expectation values for each Hamiltonian measurement term. Notice that this execution mechanism is agnostic to the accelerator sub-type. This provides a way to quickly swap between validation and verification with TNQVM, and physical hardware execution on quantum computers from IBM, Rigetti, etc.

6 Conclusion

In this work we have discussed the concept of a general quantum virtual machine and introduced a concrete implementation of the QVM that enables quantum-classical programming with validation through an extensible tensor network quantum circuit simulator (TNQVM). We have discussed the applicability and scalability of a matrix product state backend implementation for TNQVM and discussed the role of TNQVM in benchmarking quantum algorithms and hybrid quantum-classical applications including random circuit sequences used in quantum supremacy [50] and the variational quantum eigensolver [7]. We have chosen a tensor network based quantum virtual machine due to the complexity reduction such a formalism provides for a broad range of problems. In general TNQVM enables large-scale simulation of quantum circuits which generate states characterized by short-range entanglement. Studying systems with long-range entanglement interactions will require further developments in implementing more advanced tensor network decomposition types. We plan to investigate the applicability of the tree tensor network and the multiscale entanglement renormalization ansatz in future work, in an effort to scale simulation capabilities to a larger number of qubits. We will also finish the deployment of the massively parallel ExaTENSOR visitor backend in order to exploit large-scale HPC platforms in the future.

Acknowledgments

This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC0500OR2 2725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for the United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

This work has been supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, the US Department of Energy (DOE) Office of Science Advanced Scientific Computing Research (ASCR) Early Career Research Award, and the DOE Office of Science ASCR quantum algorithms and testbed programs, under field work proposal numbers ERKJ332 and ERKJ335. This work was also supported by the ORNL Undergraduate Research Participation Program, which is sponsored by ORNL and administered jointly by ORNL and the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the US Department of Energy under contract no. DE-AC05-00OR22750.

Author Contributions

Writing – original draft: Alexander McCaskey, Eugene Dumitrescu, Mengsu Chen, Dmitry Lyakh, Travis Humble.

References

1. Childs AM, van Dam W. Quantum algorithms for algebraic problems. Rev Mod Phys. 2010; 82:1–52. https://doi.org/10.1103/RevModPhys.82.1
2. Montanaro A. Quantum algorithms: an overview. npj Quantum Information. 2016; 2(1):15023. https://doi.org/10.1038/npjqi.2015.23
3. Biamonte J, Bergholm V. Tensor Networks in a Nutshell. 2017;.
4. Linke NM, Maslov D, Roetteler M, Debnath S, Figgatt C, Landsman KA, et al. Experimental comparison of two quantum computing architectures. Proceedings of the National Academy of Sciences. 2017; p. 201618020.
5. Friis N, Marty O, Maier C, Hempel C, Holzäpfel M, Jurcevic P, et al. Observation of Entangled States of a Fully Controlled 20-Qubit System. Phys Rev X. 2018; 8:021012.
6. Preskill J. Quantum Computing in the NISQ era and beyond. arXiv preprint arXiv:180100862. 2018;.
7. Peruzzo A, McClean J, Shadbolt P, Yung MH, Zhou XQ, Love PJ, et al. A variational eigenvalue solver on a photonic quantum processor. Nature Communications. 2014; 5:4213. https://doi.org/10.1038/ncomms5213 PMID: 25055053
8. O'Malley PJJ, Babbush R, Kivlichan ID, Romero J, McClean JR, Barends R, et al. Scalable Quantum Simulation of Molecular Energies. Phys Rev X. 2016; 6:031007.
9. Kandala A, Mezzacapo A, Temme K, Takita M, Brink M, Chow JM, et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. Nature. 2017; 549:242. https://doi.org/10.1038/nature23879 PMID: 28905916
10. Otterbach J, Manenti R, Alidoust N, Bestwick A, Block M, Bloom B, et al. Unsupervised Machine Learning on a Hybrid Quantum Computer. arXiv preprint arXiv:171205771. 2017;.
11. Dumitrescu EF, McCaskey AJ, Hagen G, Jansen GR, Morris TD, Papenbrock T, et al. Cloud Quantum Computing of an Atomic Nucleus. Phys Rev Lett. 2018; 120:210501. https://doi.org/10.1103/PhysRevLett.120.210501 PMID: 29883142
12. Humble TS, Britt KA. Software systems for high-performance quantum computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC); 2016. p. 1–8.
13. Britt KA, Humble TS. High-performance computing with quantum processing units. ACM Journal on Emerging Technologies in Computing Systems (JETC). 2017; 13(3):39.
14. Green AS, Lumsdaine PL, Ross NJ, Selinger P, Valiron B. Quipper: a scalable quantum programming language. In: ACM SIGPLAN Notices. vol. 48. ACM; 2013. p. 333–342.
15. Javadi-Abhari A, Patil S, Kudrow D, Heckey J, Lvov A, Chong FT, et al. ScaffCC: a framework for compilation and analysis of quantum computing programs. In: Proceedings of the 11th ACM Conference on Computing Frontiers. ACM; 2014. p. 1.
16. Wecker D, Svore KM. LIQUid: A software design architecture and domain-specific language for quantum computing. arXiv preprint arXiv:14024467. 2014.
17. Humble TS, McCaskey AJ, Bennink RS, Billings JJ, D'Azevedo EF, Sullivan BD, et al. An integrated programming and development environment for adiabatic quantum optimization. Computational Science and Discovery. 2014, 7(1):015006. https://doi.org/10.1088/1749-4680/7/1/015006
18. Smith RS, Curtis MJ, Zeng WJ. A practical quantum instruction set architecture. arXiv preprint arXiv:160803355. 2016.
19. Liu S, Wang X, Zhou L, Guan J, Li Y, He Y, et al. QSI: a quantum programming environment. arXiv:171009500. 2017.
20. Svore K, Geller A, Troyer M, Azariah J, Granade C, Heim B, et al. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018. New York, NY, USA: ACM; 2018. p. 7:1–7:10. Available from: http://doi.acm.org/10.1145/3183895.3183901.
21. Pakin S. Performing fully parallel constraint logic programming on a quantum annealer. Theory and Practice of Logic Programming. 2018; p. 1–22.
22. Smith RS, Curtis MJ, Zeng WJ. A Practical Quantum Instruction Set Architecture; 2016.
23. Steiger DS, Häner T, Troyer M. ProjectQ: An Open Source Software Framework for Quantum Computing. ArXiv e-prints. 2016.
24. Cross AW, Bishop LS, Smolin JA, Gambetta JM. Open Quantum Assembly Language. ArXiv e-prints. 2017.
25. McCaskey AJ, Dumitrescu EF, Liakh D, Chen M, Feng W, Humble TS. A language and hardware independent approach to quantum-classical computing; 2018. Available from: http://www.sciencedirect.com/science/article/pii/S2352711018300700.

26. Häner T, Steiger DS. 0.5 Petabyte Simulation of a 45-Qubit Quantum Circuit. ArXiv e-prints. 2017.
27. Pednault E, Gunnels JA, Nannicini G, Horesh L, Magerlein T, Solomonik E, et al. Breaking the 49-Qubit Barrier in the Simulation of Quantum Circuits. 2017.
28. Aaronson S, Gottesman D. Improved simulation of stabilizer circuits. Phys Rev A. 2004; 70:052328. https://doi.org/10.1103/PhysRevA.70.052328
29. Orús R. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. Annals of Physics. 2014; 349:117–158. https://doi.org/10.1016/j.aop.2014.06.013
30. Ye K, Lim LH. Tensor network ranks. ArXiv e-prints. 2018.
31. Kolda TG. Numerical optimization for symmetric tensor decomposition. Mathematical Programming. 2015; 151(1):225–248. https://doi.org/10.1007/s10107-015-0895-0
32. White SR. Density matrix formulation for quantum renormalization groups. Physical Review Letters. 1992; 69(19):2863–2866. https://doi.org/10.1103/PhysRevLett.69.2863 PMID: 10046608
33. Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Annals of Physics. 2011; 326(1):96–192. https://doi.org/10.1016/j.aop.2010.09.012
34. Schuch N, Wolf MM, Verstraete F, Cirac JI. Computational complexity of projected entangled pair states. Physical Review Letters. 2007; 98(14):140506. https://doi.org/10.1103/PhysRevLett.98.140506 PMID: 17501258
35. Verstraete F, Murg V, Cirac JI. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. Advances in Physics. 2008; 57(2):143–224. https://doi.org/10.1080/14789940801912366
36. Murg V, Verstraete F, Legeza Ö, Noack RM. Simulating strongly correlated quantum systems with tree tensor networks. Physical Review B—Condensed Matter and Materials Physics. 2010; 82(20):205105. https://doi.org/10.1103/PhysRevB.82.205105
37. Nakatani N, Chan GKL. Efficient tree tensor network states (TTNS) for quantum chemistry: Generalizations of the density matrix renormalization group algorithm. Journal of Chemical Physics. 2013; 138(13):134113. https://doi.org/10.1063/1.4798639 PMID: 23574214
38. Dumitrescu E. Tree tensor network approach to simulating Shor's algorithm. Physical Review A. 2017; 96(6):062322. https://doi.org/10.1103/PhysRevA.96.062322
39. Vidal G. A class of quantum many-body states that can be efficiently simulated. Phys Rev Lett. 2006; 101(11):110501. https://doi.org/10.1103/PhysRevLett.101.110501
40. Evenbly G, Vidal G. Algorithms for entanglement renormalization ver 2. Physical Review B. 2009; 79(December 2008):1–17.
41. Marti KH, Bauer B, Reiher M, Troyer M, Verstraete F. Complete-graph tensor network states: a new fermionic wave function ansatz for molecules. New Journal of Physics. 2010; 12(10):103008. https://doi.org/10.1088/1367-2630/12/10/103008
42. Dang A, Hill CD, Hollenberg LCL. Optimising Matrix Product State Simulations of Shor's Algorithm. 2017.
43. Fried ES, Sawaya NPD, Cao Y, Kivlichan ID, Romero J, Aspuru-Guzik A. qTorch: The Quantum Tensor Contraction Handler. arXiv preprint arXiv:170903636. 2017.
44. Britt KA, Humble TS. Instruction Set Architectures for Quantum Processing Units. In: Kunkel JM, Yokota R, Taufer M, Shalf J, editors. High Performance Computing. Cham: Springer International Publishing; 2017. p. 98–105.
45. iTensor. Available from: itensor.org.
46. McCaskey A, Dumitrescu E, Liakh D, Humble T. Hybrid Programming for Near-term Quantum Computing Systems. ArXiv e-prints. 2018.
47. A Look at the Composite Design Pattern;. https://www.javaworld.com/article/2074564/learn-java/a-look-at-the-composite-design-pattern.html.
48. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1995.
49. McCaskey A, Chen M. TNQVM—Tensor Network Quantum Virtual Machine; 2017. https://github.com/ORNL-QCI/tnqvm.
50. Boixo S, Isakov SV, Smelyanskiy VN, Babbush R, Ding N, Jiang Z, et al. Characterizing quantum supremacy in near-term devices. Nature Physics. 2018; 14(6):1–6. https://doi.org/10.1038/s41567-018-0124-x
51. O'Malley PJJ, Babbush R, Kivlichan ID, Romero J, McClean JR, Barends R, et al. Scalable Quantum Simulation of Molecular Energies. Phys Rev X. 2016; 6:031007.