# TorchQuantumDistributed

**Oliver Knitter** **Jonathan Mei** **Masako Yamada** **Martin Roetteler**
IonQ, Applications Team
College Park, MD 20740
{oliver.knitter,jmei,yamada,martin.roetteler}@ionq.co

## Abstract

TorchQuantumDistributed (`tqd`)[1] is a PyTorch-based [Paszke et al., 2019] library for accelerator-agnostic differentiable quantum state vector simulation *at scale*. This enables studying the behavior of learnable parameterized near-term and fault-tolerant quantum circuits with high qubit counts.

| | Quantum Simulation | Differentiable | Accelerator Agnostic | Scalable |
|---|---|---|---|---|
| `torch` | ✗ | ✓ | ✓ | ✓ |
| `qiskit` | ✓ | ✗ | ✗ | ✗ |
| `cirq` | ✓ | ✗ | ✗ | ✓ |
| `tensorflowquantum` | ✓ | ✓ | ✗ | ✓ |
| `pennylane` | ✓ | ✓ | — | — |
| `torchquantum` | ✓ | ✓ | ✓ | ✗ |
| `tqd` | ✓ | ✓ | ✓ | ✓ |

Table 1: Features of different frameworks for quantum simulation and differentiable programming. A checkmark (✓) indicates full or near-full support; a cross (✗) indicates lacking support; and a dash (—) indicates partial support for some features individually but not together in all combinations.

## 1 Introduction

With the increased interest in quantum computing in recent years, there has been an explosion of quantum circuit simulation tools, particularly those that facilitate quantum machine learning. Many popular frameworks such as Qiskit [Javadi-Abhari et al., 2024], Cirq [Developers, 2025], Pennylane [Bergholm et al., 2022], and TorchQuantum [Wang et al., 2022], offer quantum simulation that can be incorporated into existing Python data processing workflows. However, when using state vector simulation, these frameworks do not enable distribution of the state vector across multiple accelerators, limiting scalability and thus the use of these frameworks for studying circuits large enough to potentially yield an advantage over purely classical methods. Furthermore, many of these frameworks use software backends that are largely locked into one particular hardware ecosystem.

Leveraging the power, scaling, and hardware extensibility of PyTorch [Paszke et al., 2019], this paper introduces TorchQuantumDistributed, a differentiable quantum simulation library that enables the study of larger quantum machine learning models using distributed state vector simulator backends.

[1]The code repository for `tqd` is available here: https://github.com/ionq/torchquantum-dist/tree/main

Accepted at the AI for Science Workshop (NeurIPS 2025).

## 2 Background

### 2.1 Development of computing for subfields

**Quantum Computing** Among the first notable open-source python-based frameworks for quantum simulation was ProjectQ [Steiger et al., 2018]. Later entrants were developed by larger institutions for their own roadmaps and ecosystems: Qiskit [Javadi-Abhari et al., 2024] and Cirq [Developers, 2025] provided python interfaces for defining gate-based circuits and features specific to each of their respective hardware platforms, while Q# [Svore et al., 2018] defined its own new language for quantum execution and algorithmic development. Cirq has hardware acceleration capabilities, but they are largely tied to the NVIDIA CUDA ecosystem [Bayraktar et al., 2023].

**Deep learning** Deep learning libraries started with the enablement of automatic differentiation on static graphs, as in Theano [Bergstra et al., 2011], LuaTorch [Collobert et al., 2002], and Caffe [Jia et al., 2014]. Again, big companies released their own frameworks that have since captured a majority of the share of open-source development[2]: TensorFlow [Abadi et al., 2016] featured strong support for deployment, while PyTorch [Paszke et al., 2019] gained popularity due to its extensibility and eager execution creating dynamic computation graphs, which TensorFlow eventually adopted as well.

**Quantum Machine Learning** TensorFlow Quantum [Broughton et al., 2020] interfaces Cirq and TensorFlow, inheriting the capabilities and drawbacks of each. TorchQuantum [Wang et al., 2022] is a native PyTorch implementation of statevector simulation, so it carries some overheads but can support any hardware that PyTorch does, and it does not utilize 'torch.distributed' to allow scaling across multiple accelerators. Pennylane [Bergholm et al., 2022] also provides a platform for statevector simulation supporting several different computational backend libraries and even hardware environments, placing it closest to our desired level of functionality, but ultimately also lacks the ability to distribute the state vector across multiple accelerators when using non-CUDA devices [Asadi et al., 2024].

### 2.2 Mathematical preliminaries

We assume a baseline familiarity with linear algebra, probability, and statistics, and the terminology of ML, but not necessarily knowledge of quantum computing. Thus, we provide an overview of relevant quantum computing concepts.

**Notation** Since we deal with a variety of mathematical objects, notation is treated with care. We use serif to denote a deterministic value (e.g. $x$), sans-serif to denote a random variable (e.g. $\mathsf{x}$), bold lower case to denote a (column) vector (e.g. $\mathbf{x}$ for deterministic vectors or $\mathbf{x}$ for random vectors), bold capital to denote matrices or tensors (e.g. $\mathbf{X}$), and blackboard to denote sets (e.g. $\mathbb{R}$). Calligraphic is used for other objects not previously listed, like distributions or operators, and their use will be clarified depending on the context (e.g. $\mathcal{M}$ as a multinomial distribution or $\mathcal{G}$ as a quantum gate). We also use teletype to refer to specific code libraries or functions (e.g. `torch.permute`). We denote the $m$-th canonical coordinate basis for an $n$-dimensional vector space $\mathbf{e}_m^{(n)}$ as all 0's except a 1 in the $m$-th coordinate, and $\mathbf{I}$ as the identity matrix. To define the operations we use, let $\mathbf{W}$, $\mathbf{Y}$, and $\mathbf{Z}$ be tensors: $\mathbf{x}^*$ denotes the conjugate transpose of $\mathbf{x}$; $\otimes$ is an outer product; given a matrix $\mathbf{Y}$, we take $\mathbf{W} = \mathbf{Y} \times_i \mathbf{Z}$ to denote the contraction of dimension index $i$ of tensor $\mathbf{Z}$, or more explicitly, $\mathbf{W}_{\dots j \dots} = \sum_\ell \mathbf{Y}_{j\ell} \mathbf{Z}_{\dots \ell \dots}$, where only the $i$-th index is shown for both $\mathbf{W}$ and $\mathbf{Z}$. This can be generalized to higher dimensional tensors $\mathbf{Y}$ and ordered index sets, where for $\mathbf{W} = \mathbf{Y} \times_{\mathbb{I}} \mathbf{Z}$ we have $\mathbf{W}_{\dots \mathbb{J} \dots} = \sum_{\mathbb{L}} \mathbf{Y}_{\mathbb{J}\mathbb{L}} \mathbf{Z}_{\dots \mathbb{L} \dots}$. While the Dirac "bra-ket" notation commonly used in quantum computing could be applied here, we aim to only use notation more familiar to an ML audience.

#### 2.2.1 Quantum computing basics

Qubits are the basic unit of computation in quantum computing. The state of a qubit $\boldsymbol{\psi}$ can be represented as a complex vector, $\boldsymbol{\psi} \in \mathbb{C}^2$. The basis vectors $\mathbf{e}_0^{(2)}$ and $\mathbf{e}_1^{(2)}$ are known as the computational basis for a single qubit, and naturally the state can be represented in this basis as

[2] as measured by `paperswithcode.com` implementations up to Oct 2024

$\boldsymbol{\psi} = \sum_{i=0}^{1} \alpha_i \mathbf{e}_i^{(2)}$ for some $\alpha_i \in \mathbb{C}$. There is a constraint that $\|\boldsymbol{\alpha}\|_2 = 1$, and we will return to this later. A $q$-qubit state is represented in the tensor product space $\bigotimes_{i=0}^{q-1} \mathbb{C}^2 \cong \mathbb{C}^{2^q}$.

Quantum measurement is famously stochastic. Letting $\boldsymbol{\phi}$ be an eigenvector of a given measurement operator, then a system in the state $\boldsymbol{\psi}$ is observed to be in the state $\boldsymbol{\phi}$ with probability given by the square magnitude of their scalar product $|\boldsymbol{\phi}^* \boldsymbol{\psi}|^2$. In many algorithms, the Pauli-$Z$ operator, which can be described by the matrix $\mathbf{Z} = \begin{pmatrix} \mathbf{e}_0^{(2)} & -\mathbf{e}_1^{(2)} \end{pmatrix}$, is a natural choice for measurement as it is diagonalized by the computational basis. In Dirac notation, we may refer to $\mathbf{e}_0^{(2)}$ as being state "$|0\rangle$" and $\mathbf{e}_1^{(2)}$ as being state "$|1\rangle$". Returning to the constraint on $\boldsymbol{\alpha}$, we can examine the behavior of measurements of $\boldsymbol{\psi} = \sum_{i=0}^{1} \alpha_i \mathbf{e}_i^{(2)}$ in the computational basis. The probability of measuring state $|0\rangle$ is given by $|(\mathbf{e}_0^{(2)})^* \boldsymbol{\psi}|^2 = |(\mathbf{e}_0^{(2)})^* \sum_{i=0}^{1} \alpha_i \mathbf{e}_i^{(2)}|^2 = |\alpha_0|^2$. Similarly, the probability of measuring state $|1\rangle$ is $|(\mathbf{e}_1^{(2)})^* \boldsymbol{\psi}|^2 = |\alpha_1|^2$. Thus we see that the constraint $|\alpha_0|^2 + |\alpha_1|^2 = 1$ is required to satisfy the laws of probability and describe a valid distribution.

A quantum computation can be specified in the form of a "circuit" consisting of a register of qubits, a sequence of unitary gate operations acting on the register, and some measurements. A comprehensive treatment of the mathematics of quantum computation can be found in [Nielsen and Chuang, 2010].

### 2.2.2 QML primer

The field of QML describes the use of quantum computing to perform ML tasks. Early work focused on using quantum primitives to implement subroutines commonly found in traditional ML algorithms, but the field has also evolved to include full models and even optimization algorithms that can be used to replace large portions of the classical computing workload. A brief survey of the field is useful to provide context for our work; a more complete review of the QML field can be found in [Schuld and Petruccione, 2021].

Initially, QML evolved from a start in custom-designed models and methods for specific tasks [Guţă and Kotłowski, 2010, Schuld et al., 2014]. Then, the variational eigensolver [Peruzzo et al., 2014] from chemistry appeared alongside analytical formulas for estimating gradients [Li et al., 2017] of parameterized quantum circuits, giving rise to general-purpose algorithms for which learning could be handled by the same optimization methods used to train classical neural networks [Krizhevsky et al., 2012].

Another related field can be neatly summarized as, "quantum-inspired ML," which borrows concepts and tools from quantum mechanics and computing, for the purpose of running algorithms on classical hardware. The methods have been applied successfully to classic ML problems such as learning time series models on graphs [Mei and Moura, 2017] and image classification [Stoudenmire and Schwab, 2016], to further push the boundaries on quantum chemistry [Knitter et al., 2025], and even in conjunction with QML on actual quantum hardware for text classification [Kim et al., 2025]. While it may not yet be clear whether quantum-inspired ML should be considered part of QML or exist as a separate area, our `tqd` framework provides a toolbox for developing both.

## 3 Design Principles

TorchQuantum provides a pythonic statevector simulator. As suggested by the name, TorchQuantumDistributed borrows the nice features and usage style from TorchQuantum, while utilizing distributed tensors to shard the statevector across multiple accelerators, allowing us to scale statevector simulation to more qubits than previously possible.

**Object Oriented and Functional** Mirroring PyTorch, `tqd` provides object oriented and functional interfaces to deep learning computation. This offers the flexibility to design model architecture and also extend the framework with custom quantum operations.

**Extensible** As a starting point, a universal gate set using single-qubit Pauli rotation gates and the two-qubit "CNOT" is provided, which can in principle, given infinite numerical precision, perform any quantum operation [Nielsen and Chuang, 2010]. Of course, it may be more efficient to define certain other commonly encountered gates, reducing the number of computations needed to perform

operations and simulate circuits. As in many other frameworks, `tqd` allows for easy definition of custom quantum gate operations, though users should take care to ensure they are unitary.

**Modularity** This ties into extensibility. For ease of development, different functionalities are grouped together according to the aspects of the framework they affect. For example, the logic for implementing custom operators lives separately from the logic for tracking the dimension order of the statevector.

**Code Parsimony** The usage of `tq` is very intuitive. However, the implementation of `tq` includes a separate file for the `nn.functional` version of a gate as well as for the stateful `nn.Module` version. `tqd` condenses this, defining gates only once in a central location and generating both function and object forms of gates operations in a programmatic fashion following a consistent schema, without sacrificing readability.

# 4 Implementation Details

Here we describe the salient aspects of the implementation needed to enable state vector sharding. For expositional clarity, we will mostly deal with a single statevector corresponding to an implicit batch size of 1 as the concepts generalize naturally to multiple statevectors. Where the batch dimension is relevant to the discussion, we will make explicit note of it.

## 4.1 Bookkeeping

A large part of the work required for simulating quantum computation, especially in scaling it up across many hardware nodes, is tracking how different dimensions of the statevector, corresponding to qubits in the circuit, are being represented and stored at any given intermediate point during the computation.

First, we describe how 'tq' utilizes compute and data movement primitives in PyTorch. Then we detail speed improvements that 'tqd' makes over 'tq'. We initially ignore considerations for sharding the statevector, focusing on them in the later section.

### 4.1.1 Tensor dimension order

To describe the computation, we introduce some more notation. Let $\mathbf{X} \in \mathbb{C}^{\overbrace{2 \times 2 \times \cdots \times 2}^{q}}$ be the complex statevector for a $q$-qubit circuit. We take a gate $\mathcal{G}$ as defined by the ordered pair of 1) a unitary matrix $\mathbf{M}$; and 2) an ordered set of qubits on which it operates $\mathbb{Q}$. A gate operating on a statevector is denoted $\mathcal{G}(\mathbf{X})$.

Suppose we are given $\mathbb{Q}$, the set of qubits on which a gate operates. Consider the operation $\text{MoveDim}(\mathbf{X}, \mathbb{Q})$, which moves the dimensions specified in the ordered set $\mathbb{Q}$ to the front. This can be implemented as `torch.movedim(X, Q, destination=torch.arange(len(Q))`. Note that if $\mathbb{Q}$ contains all integers up to $q$, then $\text{MoveDim}(\cdot, \cdot)$ is equivalent to permuting the dimensions (`torch.permute`).

Further, we define its inversion $\text{MoveDim}^{-1}(\mathbf{X}, \mathbb{Q})$ to reorder dimensions from the front such that

$$\mathbf{X} = \text{MoveDim}^{-1}(\text{MoveDim}(\mathbf{X}, \mathbb{Q}), \mathbb{Q}). \tag{1}$$

To perform the computation of the gate on the statevector $\mathbf{X}$, `tq` rearranges the dimensions, performs the computation using a (broadcasting) matrix multiplication primitive (denoted "mm"), and then rearranges the dimensions back. This is summarized in Algorithm 1.

The computation of `tqd` follows Algorithm 2.

While if implemented using `torch.permute` there is no additional data movement overhead from the additional MoveDim, it is nonetheless helpful for implementing data sharding to track the actual tensor layout in memory.

**Algorithm 1** `tq` computation pseudocode

**Require:** $\mathbf{X}$, $(\mathbf{M}, \mathbb{Q})$
**Ensure:** $Y = \mathcal{G}(X)$
  $\mathbf{X} \leftarrow \text{MoveDim}(\mathbf{X}, \mathbb{Q})$
  $\mathbf{X} \leftarrow \text{mm}(\mathbf{M}, \mathbf{X})$
  $\mathbf{Y} \leftarrow \text{MoveDim}^{-1}(\mathbf{X}, \mathbb{Q})$

**Algorithm 2** `tqd` computation pseudocode

**Require:** $\mathbf{X}$, $\mathbb{I}$, $(\mathbf{M}, \mathbb{Q})$
**Ensure:** $\text{MoveDim}^{-1}(\mathbf{Y}, \mathbb{J}) = \mathcal{G}(\text{MoveDim}^{-1}(\mathbf{X}, \mathbb{I}))$
  $\mathbf{X} \leftarrow \text{MoveDim}(\mathbf{X}, \mathbb{Q})$
  $\mathbb{J} \leftarrow [\mathbb{Q},\ \mathbb{I} \backslash \mathbb{Q}]$ ▷ List concatenation
  $\mathbf{Y} \leftarrow \text{mm}(\mathbf{M}, \mathbf{X})$

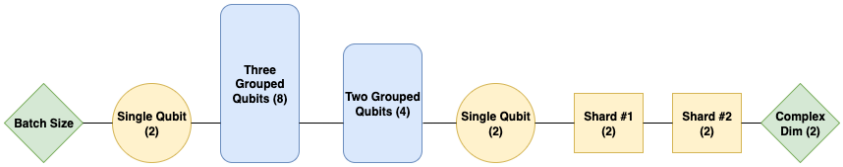


Figure 1: An example dimensional arrangement for a TQD distributed tensor representating a nine qubit statevector, with two sharded qubits. TQD always reserves the first and last dimension for batching and to contain the real and imaginary parts. Sharded qubits correspond with the dimensions preceding the final one, and at least two unsharded qubits are always kept ungrouped.

### 4.2 Sharding and dimension grouping

While section 4.1.1 provides a high level mathematical view of the computation, there are further details to track when distributing the statevector across multiple accelerator devices. In particular, while PyTorch places no intrinsic limitation on the number of dimensions a tensor can have, it was not intended to work with arbitrary tensor ranks, and certain fundamental subroutines within the underlying architecture may place their own dimension limitations. Therefore, unlike TorchQuantum, which shapes the statevector so that each tensor dimension encodes a single qubit, `tqd` arranges statevector DTensors so that some dimensions correspond with groups of multiple qubits.

Figure 1 illustrates how `tqd` arranges a 9-qubit statevector, with 2 sharded dimensions, and a total tensor rank of 8. In practice, we find our implementation works with tensor ranks as high as 16. The first and final dimensions are always reserved for batching and for the real and imaginary parts of the statevector. Distributing across $d$ accelerators allows for sharding $\log_2 d$ dimensions, each corresponding with 1 qubit. From there, `tqd` assumes at least two unsharded dimensions correspond with single qubits, and that either all qubits are ungrouped, or there exist at least two grouped qubit dimensions.

This arrangement corresponds with a linear permutation of the qubits, where continuous subsets of qubits can be grouped together. A second tensor keeps track of exactly how the qubits are arranged within this permutation. Reshaping the tensor corresponds with regrouping qubits without changing their order, while permuting the tensor dimensions corresponds with rearranging groupings of tensors in the permutation. Since reshaping and permuting tensors stored in contiguous memory is not computationally expensive, `tqd` is built on a framework of interchanging qubit positions within the grouping permutation. This allows us to easily retrieve the qubit dimensions needed to perform gate multiplications and redistribute across devices.

### 4.3 Shot noise

Shot noise describes the randomness inherent to the process of quantum measurement. Specifically, each run of the circuit ending in measurement yields a categorical variable. Typically, we run many shots of the circuit, and the many independent realizations of the categorical variable can

be summarized by a multinomial variable. Because this is inherent to the way we retrieve useful information from the quantum device, we view shot noise as an essential component of quantum simulation. We describe two different ways we incorporate this stochasticity into our framework along with their benefits and drawbacks.

We can perform sampling from the exact multinomial distribution. This is slow and not differentiable. Thus, this may be preferred during inference to get more accurate characterization of the models.

In the high-shot limit, the multinomial variable can be well-approximated by a degenerate multivariate Gaussian variable [Georgii, 2012]. While not exact, this is fast and differentiable, as it is an application of the reparameterization trick [Kingma and Welling, 2014]. Thus, this may be more useful during training, where we still wish to run the model in a robust, noise-aware manner [Kim et al., 2025], but also value optimization iteration speed.

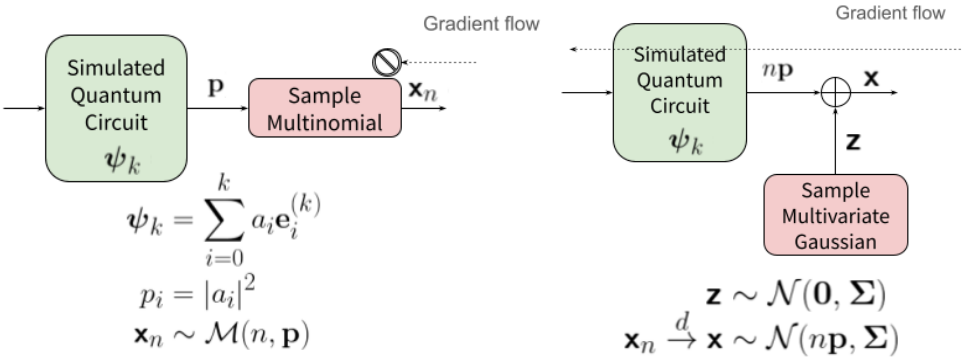


Figure 2: Left: exact sampling breaks differentiability. Right: approximate sampling uses the reparameterization trick to maintain differentiability.

Figure 2 shows the breaking of the computational graph caused by exact sampling, which is a non-differentiable operation (left), and the application of the reparameterization trick to approximate sampling from the quantum circuit in a differentiable way (right).

#### 4.3.1 Exact

Sampling from a multinomial can be accomplished on a single accelerator in a straightforward call to `torch.multinomial()`. However, when the statevector is split across multiple accelerators, the same mathematical operation is not so simple to implement faithfully. Interestingly, we can utilize several statistical properties of the multinomial to perform this in a distributed fashion.

Take $\mathbf{x} \sim \mathcal{M}(n, \mathbf{p})$ to be jointly multinomial variables, for parameters $0 \leq p_i \leq 1$ that satisfy $\sum_{i=0}^{k-1} p_i = 1$ and integer $n$. That is, $\{\mathsf{x}_i\}$ has the PMF

$$P(\mathsf{x}_0 = x_0, \mathsf{x}_1 = x_1, \ldots, \mathsf{x}_{k-1} = x_{k-1}) = p(x_0, x_1, \ldots, x_{k-1}) = \frac{n!}{\prod_{i=0}^{k-1} x_i!} \prod_{i=0}^{k-1} p_i^{x_i} \tag{2}$$

supported on integers $x_i$ such that $\sum_{i=0}^{k-1} x_i = n$.

Suppose we have a set of $m$ index sets $\{\mathbb{I}_i\}$ such that $\bigcup_{i=0}^{m-1} \mathbb{I}_i = \{0, \ldots, k-1\}$ and $\mathbb{I}_i \bigcap \mathbb{I}_j = \varnothing \; \forall i \neq j$. That is, the index set is non-overlapping and covers the integer indices up to but excluding $k$. Then consider the variables $\mathsf{y}_j = \sum_{i \in \mathbb{I}_j} \mathsf{x}_i$. Then the variables $\mathsf{y}_j$ are themselves multinomial with parameters $q_j = \sum_{i \in \mathbb{I}_j} p_i$ [Georgii, 2012],

$$\mathbf{y} \sim \mathcal{M}(n, \mathbf{q}). \tag{3}$$

Next, we note that the variables $\{\mathsf{x}_i : i \in \mathbb{I}_j\} | \mathsf{y}_j$ are also multinomial Georgii [2012]. This points to a hierarchical sampling scheme, which groups individual local variables $\mathsf{x}_i$ to form intermediate $\mathsf{y}_j$, then samples values of $\mathsf{y}_j$, and finally samples the $\mathsf{x}_i$'s conditional upon the $\mathsf{y}_j$'s. Practically, we can associate each accelerator with its own $\mathbb{I}_j$ with the indices for variables for which it locally contains

the parameters. One small round of initial communication is required where each accelerator shares its $q_j$ so that they can all can assemble the full vector $\mathbf{q}$. With a shared seed, each accelerator can then sample $\mathsf{y}$ in parallel, then taking $\mathsf{y}_j$ to sample $\mathsf{x}_i$.

#### 4.3.2 Approximate

Let the diagonal matrix $\mathbf{D}$ have diagonal elements $\mathbf{D}_{ii} = \sqrt{p_i}$. As the number of shots $n$ grows, we have a certain convergence in distribution to a multivariate Gaussian

$$\mathbf{y}_n \sim \mathcal{M}(\mathbf{p}, n) \tag{4}$$

$$\mathbf{x}_n = \frac{1}{\sqrt{n}}\mathbf{D}^{-1}(\mathbf{y}_n - n\mathbf{p}) \tag{5}$$

$$\mathbf{x} \sim \mathcal{N}(0, \mathbf{\Sigma}) \tag{6}$$

$$\Rightarrow \mathbf{x}_n \xrightarrow{d} \mathbf{x} \tag{7}$$

where $\mathbf{\Sigma}_{ii} = p_i(1 - p_i)$ and $\mathbf{\Sigma}_{ij} = -p_i p_j$ for $i \neq j$ [Georgii, 2012]. One way to sample from the Gaussian is to have access to $\mathbf{S}$, a matrix factorization of the covariance satisfying $\mathbf{\Sigma} = \mathbf{S}\mathbf{S}^\top$. Then we can undo the transformation from $\mathbf{x}_n$ to $\mathbf{y}_n$ to approximate the multinomial sample. We use a factorization based on a Householder transformation that can be fairly easily computed. Let the unnormalized vector $\widetilde{\mathbf{v}} = \mathbf{e}_k^{(k)} - \mathbf{u}$ and its normalization $\mathbf{v} = \widetilde{\mathbf{v}}/\|\widetilde{\mathbf{v}}\|_2$ (in the degenerate case where $\mathbf{u} = \mathbf{e}_k^{(k)}$, we take $\mathbf{v} = \mathbf{0}$). Then we can take

$$\mathbf{S} = \mathbf{I} - \mathbf{v}\mathbf{v}^\top. \tag{8}$$

Finally, for completeness, taking i.i.d. unit normals $\mathsf{z}_i \sim \mathcal{N}(0, 1)$ for $i = 0, \ldots, k-2$ and $\mathbf{z} = \begin{pmatrix}\mathsf{z}_0 & \ldots & \mathsf{z}_{k-2} & 0\end{pmatrix}^\top$, we have

$$\mathbf{y}_n = n\mathbf{p} + \sqrt{n}\mathbf{D}\mathbf{x}_n \tag{9}$$

$$\mathbf{x}_n = \mathbf{S}\mathbf{z} \tag{10}$$

$$\Rightarrow \mathbf{y}_n = n\mathbf{p} + \sqrt{n}\mathbf{D}\mathbf{S}\mathbf{z}. \tag{11}$$

### 4.4 Backpropagating Invertible Computations

Since quantum computing is built on unitary operations, the computations are invertible. This means that for backpropagation, we may recompute intermediate activations instead of storing them. For memory-intensive settings such as quantum simulation, this can be used to ameliorate some of the memory requirements for enabling differentiability.

Concretely, let $\partial_{\mathbf{x}}$ ($\partial_{\mathbf{U}}$) denote the gradient of the loss function with respect to vector (matrix) variable $\mathbf{x}$ ($\mathbf{U}$). Then consider a layer implementing unitary matrix multiplication,

$$\mathbf{y} = \mathbf{U}\mathbf{x} \tag{12}$$

$$\Rightarrow \begin{cases}\partial_{\mathbf{x}} = \mathbf{U}^\top \partial_{\mathbf{y}} \\ \partial_{\mathbf{U}} = \partial_{\mathbf{y}} \mathbf{x}^\top.\end{cases} \tag{13}$$

Generally, our automatic differentiation framework would "see" Equation (13) and store the intermediate activation $\mathbf{x}$ to compute the gradient $\partial_{\mathbf{U}}$. However, since the inverse of a unitary is its conjugate transpose, we can instead implement the backward pass of Equation (13) using

$$\mathbf{x} = \mathbf{U}^* \mathbf{y}. \tag{14}$$

This removes the need to store the layer input $\mathbf{x}$ at the expense of recomputing it from layer output $\mathbf{y}$.

## 5 Profiling

As a fundamental proof of concept, we conduct a basic profiling test of `tqd` by simulating a rudimentary ansatz using a multi-node HPC cluster. Each node within the cluster contains 4 AMD MI250X

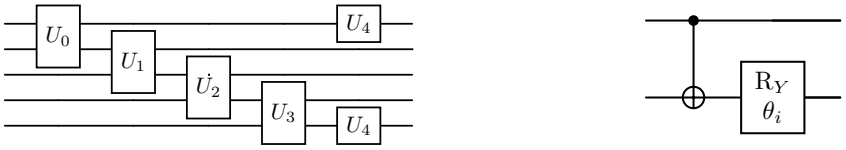


Figure 3: The primary building block for the ansatz (left) consists of unitaries with a ladder structure across the width of the circuit. Each block $U_i$ is defined as a combination of a controlled NOT and a single-qubit rotation $R_Y$ of angle $\theta_i$ (right).

accelerators, each containing 2 graphics compute dies, which act effectively as separate 64GB GPUs. For these profiling experiments, we use the circuit structure shown in figure 3: a ladder structure consisting of entangling CNOT and Pauli Y rotation gates, since it is a commonly used ansatz component throughout QML models [Kim et al., 2025]. With a batch size of 16, we run 5 iterations of Adam [Kingma and Ba, 2015] to optimize the initial circuit input, but not the ansatz parameters. Information from the final gradient update step is recorded for profiling. More specifically, for a single GPU, the first according to the world topology, we record the total walltime, the total time dedicated to all-to-all communication, and the total memory used, though not simultaneously utilized, for all GPU operations within the final update step. These three benchmarks were chosen as qualitative measures of the leading compute costs with respect to problem size and accelerator count.

### 5.1 Scaling

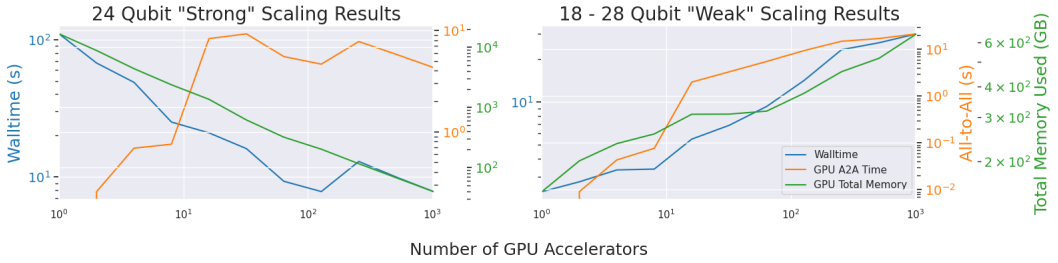


Figure 4: Basic "strong" and "weak" scaling tests of `tqd`, applying the ansatz from figure 3 to qubit sizes between 18 and 28. We vary the number of accelerators between 1 and 1024. We perform benchmarking by collecting the walltime, total NCCL all-to-all communication time, and total memory usage recorded by a single GPU for one forward–backward pass through the ansatz.

The results of these benchmarking experiments are depicted in figure 4. Here, we perform a "strong" and "weak" scaling test as the total number of GPUs is doubled incrementally from 1 to 1024. Our use of quotations is intended to indicate that these are only basic proof of concept tests, not thorough explorations of the full extent of `tqd`'s capabilities. For the "strong" scaling test, we perform a 24-qubit test at all world sizes, while for the "weak" scaling test, we incrementally increase problem size from 18 to 28 qubits. Presented on a log–log scale, these results in both cases indicate favorable power law trends between world size and nearly all benchmarks: for the "strong" scaling test, the additional overhead incurred by inter-GPU communication does not appear to substantially hinder the expected improvement in walltime.

## 6 Conclusion

We have introduced `tqd`, a tool for scalable hardware-accelerated, differentiable quantum statevector simulation that is extensible. We have shown basic proofs of concept indicating favorable scaling behavior when applying `tqd` to circuit simulations inspired by common QML ansatze, and it is our hope that this tool may find itself incorporated into many future QML research pipelines.

Potential future work on this simulator includes profiling peak GPU usage, network traffic, and incorporation of circuit-cutting and knitting techniques to ameliorate the communication costs.

## Acknowledgments and Disclosure of Funding

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

## References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

Ali Asadi, Amintor Dusko, Chae-Yeun Park, Vincent Michaud-Rioux, Isidor Schoch, Shuli Shu, Trevor Vincent, and Lee James O'Riordan. Hybrid quantum programming with Pennylane Lightning on HPC platforms. *arXiv preprint arXiv:2403.02512*, 2024.

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

Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. AkashNarayanan, Ali Asadi, Juan Miguel Arrazola, Utkarsh Azad, Sam Banning, Carsten Blank, Thomas R. Bromley, Benjamin A. Cordier, Jack Ceroni, Alain Delgado, Olivia Di Matteo, Amintor Dusko, Tanya Garg, Diego Guala, Anthony Hayes, Ryan Hill, Aroosa Ijaz, Theodor Isacsson, David Ittah, Soran Jahangiri, Prateek Jain, Edward Jiang, Ankit Khandelwal, Korbinian Kottmann, Robert A. Lang, Christina Lee, Thomas Loke, Angus Lowe, Keri McKiernan, Johannes Jakob Meyer, J. A. Montañez-Barrera, Romain Moyard, Zeyue Niu, Lee James O'Riordan, Steven Oud, Ashish Panigrahi, Chae-Yeun Park, Daniel Polatajko, Nicolás Quesada, Chase Roberts, Nahum Sá, Isidor Schoch, Borun Shi, Shuli Shu, Sukin Sim, Arshpreet Singh, Ingrid Strandberg, Jay Soni, Antal Száva, Slimane Thabet, Rodrigo A. Vargas-Hernández, Trevor Vincent, Nicola Vitucci, Maurice Weber, David Wierichs, Roeland Wiersema, Moritz Willmann, Vincent Wong, Shaoming Zhang, and Nathan Killoran. PennyLane: Automatic differentiation of hybrid quantum-classical computations, July 2022. URL `http://arxiv.org/abs/1811.04968`. arXiv:1811.04968 [quant-ph].

James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on GPUs with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3. Citeseer, 2011.

Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J Martinez, Jae Hyeon Yoo, Sergei V Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, et al. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989*, 2020.

Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. *Technical Report IDIAP-RR 02-46, IDIAP*, 2002.

Cirq Developers. *Cirq*. Zenodo, July 2025. doi: 10.5281/ZENODO.4062499. URL `https://zenodo.org/doi/10.5281/zenodo.4062499`.

Hans-Otto Georgii. *Stochastics: introduction to probability and statistics*. Walter de Gruyter, 2012.

Mădălin Guţă and Wojciech Kotłowski. Quantum learning: asymptotically optimal classification of qubit states. *New Journal of Physics*, 12(12):123032, December 2010. ISSN 1367-2630. doi: 10.1088/1367-2630/12/12/123032. URL `https://dx.doi.org/10.1088/1367-2630/12/12/123032`.

Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, June 2024. URL http://arxiv.org/abs/2405.08810. arXiv:2405.08810 [quant-ph].

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Sang Kim, Jonathan Mei, Claudio Girotto, Masako Yamada, and Martin Roetteler. Quantum language model fine tuning. *IEEE International Conference on Quantum Computing and Engineering*, 2025.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *ICLR*, Banff, AB, Canada, April 2014. URL https://openreview.net/forum?id=33X9fd2-9FyZd.

Oliver Knitter, Dan Zhao, James Stokes, Martin Ganahl, Stefan Leichenauer, and Shravan Veerapaneni. Retentive neural quantum states: efficient ansätze for ab initio quantum chemistry. *Machine Learning: Science and Technology*, 6(2):025022, 2025.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

Jun Li, Xiaodong Yang, Xinhua Peng, and Chang-Pu Sun. Hybrid Quantum-Classical Approach to Quantum Optimal Control. *Physical Review Letters*, 118(15):150503, April 2017. doi: 10.1103/PhysRevLett.118.150503. URL https://link.aps.org/doi/10.1103/PhysRevLett.118.150503. Publisher: American Physical Society.

Jonathan Mei and José M. F. Moura. Signal Processing on Graphs: Causal Modeling of Unstructured Data. *IEEE Transactions on Signal Processing*, 65(8):2077–2092, April 2017. ISSN 1941-0476. doi: 10.1109/TSP.2016.2634543. URL https://ieeexplore.ieee.org/abstract/document/7763882. Conference Name: IEEE Transactions on Signal Processing.

Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1):4213, July 2014. ISSN 2041-1723. doi: 10.1038/ncomms5213. URL https://www.nature.com/articles/ncomms5213. Publisher: Nature Publishing Group.

Maria Schuld and Francesco Petruccione. *Machine Learning with Quantum Computers*. Springer, 2021.

Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. Quantum Computing for Pattern Classification. In Duc-Nghia Pham and Seong-Bae Park, editors, *PRICAI 2014: Trends in Artificial Intelligence*, pages 208–220, Cham, 2014. Springer International Publishing. ISBN 978-3-319-13560-1. doi: 10.1007/978-3-319-13560-1_17.

Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.

Edwin Stoudenmire and David J Schwab. Supervised Learning with Tensor Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/hash/5314b9674c86e3f9d1ba25ef9bb32895-Abstract.html.

Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the real world domain specific languages workshop 2018*, pages 1–10, 2018.

Hanrui Wang, Yongshan Ding, Jiaqi Gu, Zirui Li, Yujun Lin, David Z Pan, Frederic T Chong, and Song Han. QuantumNAS: Noise-adaptive search for robust quantum circuits. In *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA-28)*, 2022.

## A API Example Usage

We will now show some simple examples of how to use `tqd`.

First, in Listing 1, we see the basic building blocks for circuit construction: initializing a quantum circuit, applying gates in 3 different paradigms, and finally taking a measurement.

**Listing 1** Basic usage example: `basic_example.py`

```python
import torch
import tqd

nq = 6  # number of qubits
qdev = tqd.DistributedQuantumDevice(nq)

# functional on the qdev
tqd.z(qdev)

# create a stateful RY gate module that tracks its own parameters
ry = tqd.RY(wires=[0], params=torch.pi/3)
ry(qdev)

# operate directly using qdev's own methods
qdev.cx(wires=[0,1])

exact = tqd.measure_allZ(qdev)
```

This snippet could be run with 2 GPUs as

```
torchrun --nproc-per-node=2 basic_example.py
```

For a more sophisticated example, we now show the differentiability, composability, and invertibility in Listing 2.

**Listing 2** Typical usage: `main_example.py`

```python
import torch
import tqd
import tqd.module

rank = os.environ['RANK']

qdev = tqd.DistributedQuantumDevice(
    nq,
    bsz=batch,
    device=f'cuda',
    world_sz=world_sz,
    invertible=True,
)

func_list = [
    {'func': 'ry', 'wires': [i], 'input_idx': [i]}
    for i in range(nq)
]
enc = tqd.GeneralEncoder(func_list)
base_mod = [enc]
for _ in range(3):  # 3 sets of alternating CNOT ladders and RYs
    base_mod = base_mod + \
        [tqd.CX(wires=[i, (i+1) % nq]) for i in range(nq)] + \
        [tqd.RY(wires=[i]) for i in range(nq)]
mod = tqd.module.InvertibleUnitary(base_mod)
mod.train()

def fun(qdev, inp):
    qdev.reset_states()
    mod(qdev, inp)
    meas_approx = tqd.measure_allZ(qdev, shots=0, training=True)
    return meas_approx

# test backprop:
x_i = torch.nn.Parameter(torch.rand([batch, nq], device=f'cuda:{rank}') * torch.pi /
↪  3)

opt = torch.optim.Adam([x_i])
loss_s = []
for i in range(10):

    opt.zero_grad()
    out_i = fun(qdev, x_i)

    loss = out_i.abs().sum()
    loss_s.append(loss.item())
    loss.backward()
    opt.step()
```