Can Trainium Run Quantum Circuit Simulation?

Quan Do
biquando@cs.ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

ABSTRACT

Because access to quantum computers is limited, quantum computing researchers often test their algorithms on quantum circuit simulators. However, running such simulations on classical computers requires an exponential amount of memory and execution time. We present two implementations of quantum circuit state vector simulators that make use of AWS Trainium, an AI accelerator designed for machine learning training and inference. The first implementation is a hybrid simulator, performing some computation on Trainium and some in the host CPU. The second implementation runs entirely within the Trainium device. We compare our implementations to Qiskit, and find that they do not provide a performance advantage over CPU-only simulators. We discuss four limitations of the Trainium hardware that make it ill-suited for state vector simulation.

1 INTRODUCTION

Quantum computing is an active area of research that seeks to take advantage of quantum phenomena to efficiently compute classically difficult problems. For some important algorithms, quantum computers can gain an exponential advantage over classical computers. Many researchers focus on the "software" of a quantum computer, which includes algorithm development, quantum error correction, and quantum circuit compilation. Because access to quantum computers is limited, researchers often do quantum circuit simulation on classical computers. However, simulating large quantum circuits is often difficult due to exponential memory requirements and execution time.

A lot of research has been dedicated to accelerating quantum circuit simulation. Several novel simulation algorithms have been developed that scale subexponentially in certain conditions (see Section 3.3). Quantum computing software stacks such as Qiskit [12] provide tools to easily build, compile, optimize, and execute circuits. cuQuantum [5] provides a framework to accelerate quantum circuit simulation on GPUs by orders of magnitude.

In this work, we write a quantum circuit simulator for the Amazon Web Services (AWS) Trainium devices, and we evaluate whether they are suitable for quantum simulation. AWS Trainium is a family of artificial intelligence (AI) accelerators that are designed to deliver high performance in training and inference at a low cost compared to AWS GPU instances. They are programmed using Python APIs, allowing the user to use either standard high-level frameworks such as PyTorch [14] or the provided low-level kernel API, allowing for fine-grained control over the hardware. We aim to take advantage of Trainium's tensor engine, parallelism, and high memory bandwidth in order to efficiently perform the calculations necessary for quantum circuit simulation.

The rest of this report is structured as follows. Section 2 provides an overview of the hardware and software features of the Trainium devices. Section 3 gives an introduction to the standard mathematical formulation and circuit model of quantum computation, as well as classical algorithms for simulating quantum circuits. Section 4 shows a standard CPU-only implementation of a quantum circuit simulator. Section 5 gives a hybrid implementation that utilizes the Trainium chip as well as the host device's CPU and memory. Section 6 gives an implementation that runs entirely on the Trainium device. Section 7 presents experimental results for each implementation. We conclude in Section 8.

2 BACKGROUND ON TRAINIUM

This section describes the hardware of Trainium and the software stack used to program the device.

2.1 Hardware: NeuronCores

AWS offers two kinds of AI accelerators—Trainium [2] and Inferentia [1]. Inferentia chips are designed for high-performance, low-cost AI inference, and Trainium is designed for both training and inference. This project targets the Trainium1 chip as a part of the AWS Build on Trainium research program [3]. Although these chips are primariliy targeted towards AI applications, it offers some hardware features that may be useful in quantum circuit simulation.

Each Trainium chip consists of two NeuronCores with a combined 32 GB of high bandwidth memory (HBM). Figure 1 shows the structure of each neuron core. The State Buffer (SBUF) has 25 MB of memory used for storing currently-used variables. The Partial Sum Buffer (PSUM) is 2 MB dedicated to storing the results of matrix multiplication computed by the Tensor Engine. The Tensor Engine is a systolic array that can perform $128 \times 128 \times 512$ matrix multiplications. The Vector Engine can reduce a 128×512 matrix across the second dimension, yielding a 128×1 vector. The Scalar Engine can perform element-wise computation on a 128×64K matrix. The GPSIMD Engine contains 8 general-purpose SIMD processors, each supporting computation on 512-bit vectors.

2.2 Software: Neuron SDK

Neuron is the software development kit (SDK) that enables programmers to run machine learning models or custom kernels on Trainium. It consists of a compiler, a runtime, training and inference libraries for distributed models, and developer tools to help with profiling and debugging. Neuron's interface is written in Python, allowing it to integrate with existing high-level machine frameworks like PyTorch [14] and JAX [8]. The SDK also provides the Neuron Kernel Interface (NKI), which allows users to write their own kernels using a lower-level API.

High-level framework compilation. The Neuron compiler transforms a model from PyTorch or JAX into an XLA HLO intermediate representation [13]. The compiler performs optimizations, and

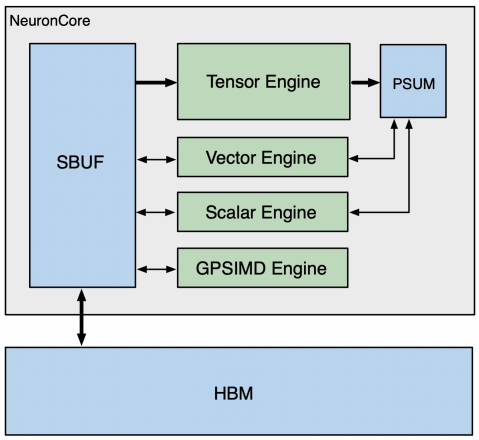

Figure 1: NeuronCore architecture. From Ref. [4]

outputs a binary that will be loaded by the Neuron runtime into the Trainium device. The programmer is able to define their model using normal high-level operations, such as PyTorch's `torch.matmul`. This enables users to quickly start using Trainium for their models, but it does not always give the best performance.

NKI kernels. In order to optimize certain operations in their models or create custom operations to run on the Trainium device, the user can write a NKI kernel. The Neuron SDK provides two APIs for NKI that can be used in tandem.

`nki.language` provides high-level operations that operate on tensors within the Trainium device's memory. For example, it provides load and store operations to move data between HBM and SBUF, a matrix multiplication operation that utilizes the Tensor Engine, a summation operation that runs on the Vector Engine, and a variety of element-wise activation functions that use the Scalar Engine.

`nki.isa` provides a lower-level interface that resembles the hardware instructions on the NeuronDevice ISA. It allows the programmer to specify all input parameters available in the hardware instructions, giving fine-grained control over each engine in the NeuronCore. It also enforces tensor size and layout constraints present in the ISA.

3 BACKGROUND ON QUANTUM CIRCUIT SIMULATION

Quantum computing is commonly modeled in the language of linear algebra. To understand this model, it is useful to compare it to classical computing under a similar lens.

3.1 State vectors

The state of a classical computer can be represented as a bit string B. If all of a computer's memory and registers consist of n bits, then the total state is a bit string of length n. Equivalently, we can represent the state as a column vector (denoted $|B\rangle$) of length 2^n, where the Bth element is set to 1, and all other elements are 0. For example, if $B = 00\ldots010 = 2$, then the classical state vector is

$$\begin{aligned} S_{\text{classical}} &= |B\rangle = |2\rangle \\ &= \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & \ldots & 0 \end{pmatrix}^{\mathsf{T}}. \end{aligned}$$

With this mathematical model, one can try setting the elements of the state vector to numbers other than 0 or 1. In probabalistic programming, each element of the state vector is a probability that the computer is in that state. Note that this requires all elements of the state vector to be between 0 and 1, and their sum must be 1. For example, a two-bit state that has a 70% probability of being 00 and a 30% probability of being 11 can be represented with the following vector:

$$\begin{aligned} S_{\text{probabalistic}} &= \begin{pmatrix} 0.7 & 0 & 0 & 0.3 \end{pmatrix}^{\mathsf{T}} \\ &= 0.7\,|00\rangle + 0.3\,|11\rangle\,. \end{aligned}$$

In quantum computing, the state vector (denoted ψ) can contain complex numbers called amplitudes. Instead of bits, a quantum computer has *qubits*, which allow the state vector to be complex-valued. When a qubit is measured, it probabilistically collapses to some classical state (either 0 or 1). Consider the following quantum state:

$$\begin{aligned} S_{\text{quantum}} \equiv \psi &= \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{i}{\sqrt{2}} \end{pmatrix}^{\mathsf{T}} \\ &= \frac{1}{\sqrt{2}}\,|00\rangle + \frac{i}{\sqrt{2}}\,|11\rangle \end{aligned}$$

To determine the probability of measuring a certain classical state, we apply the Born rule (Figure 2), which states that the probability of measuring the qubits to be bitstring B is equal to the magnitude squared of the amplitude of $|B\rangle$. Thus in the above example, the

$$P(B) = |\psi \cdot |B\rangle\,|^2$$

Figure 2: The Born rule. This gives the probability of measuring a quantum state to be bitstring B.

probability of measuring 00 is $|1/\sqrt{2}|^2 = 1/2$, and the probability of measuring 11 is $|i/\sqrt{2}|^2 = (1/\sqrt{2})^2 = 1/2$. From the Born rule, we derive a constraint: $|\psi| = 1$.

3.2 Quantum circuits

In a single-qubit system, the state is $\alpha\,|0\rangle + \beta\,|1\rangle = (\alpha\ \beta)^{\mathsf{T}}$, where $|\alpha|^2 + |\beta|^2 = 1$. To create a system of two (or more) independent qubits, we use the tensor product:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}$$

One can verify that this operation preserves the state vectors' unit-length. If a given state vector cannot be represented as a tensor product like this, then we say its qubits are *entangled*.

In classical computing, bits are operated on with logical gates—AND, OR, NOT, etc. In quantum computing, qubits are operated

on with quantum gates. A quantum gate is represented mathematically as a unitary matrix (i.e. $UU^\dagger = U^\dagger U = I$). A gate that applies to m qubits is of size $2^m \times 2^m$. We apply a gate U as follows:

$$\psi' = U\psi$$

It is common to represent quantum programs visually as circuits. Figure 3 shows a simple circuit with two qubits (drawn as horizontal lines), the commonly-used one-qubit Hadamard gate (labeled as H), and a two-qubit CX gate. The circuit is read from left to right, and each gate is applied using matrix multiplication:

$$\psi' = (CX)(I \otimes H)\,|00\rangle$$

In this circuit, the state is initialized to $|0\rangle \otimes |0\rangle = |00\rangle$. Then an H gate is applied to the first qubit. We represent this operation as $I \otimes H$, where I is the 2×2 identity matrix, because this gate does not affect the second qubit. Lastly, we apply the CX gate to both qubits.

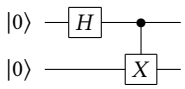

Figure 3: Quantum circuit with two qubits initialized to $|00\rangle$, a one-qubit gate, and a two-qubit gate.

3.3 Simulation algorithms

In the absence of a quantum computer, researchers perform quantum circuit simulation on classical computers using a variety of methods.

State vector simulation. The simplest method is to store the entire state vector in memory and perform gate operations as matrix multiplications. For a circuit with n qubits, we store 2^n complex numbers in memory. An n-qubit gate would involve multiplying the state vector by a $2^n \times 2^n$ matrix, which is not feasible in general. Instead, gates are usually limited to a small number of qubits—any gate can be decomposed into one and two-qubit gates. To apply an m qubit gate to an n qubit state vector (where $m < n$), one can use indexing tricks to reduce the time complexity from $O(2^{2n})$ to $O(2^{m+n})$. See Section 6 for such an algorithm implemented on Trainium.

For this project, we focus on state vector simulation because it is the most general simulation method, is simple to implement, performs exact simulation of the quantum circuit, and is reliant on matrix multiplication, with which Trainium excels. It is also possible to use a sparse representation of the state vector to achieve large performance gains [18], but Trainium does not have good support for sparse data structures; thus we only consider the dense representation.

Matrix product state simulation. Matrix product state (MPS) simulation is a method that can efficiently simulate quantum circuits whose states are only slightly entangled [19]. It represents the quantum circuit state as a product of $O(n)$ tensors—each with a size depending on the maximum entanglement χ of the circuit. Applying a gate to one or two qubits only requires modifying the tensors associated with those qubits. For small χ, the circuit can be simulated in polynomial time.

However, this method is not well-suited for Trainium. In particular, MPS relies on singular value decomposition (SVD) to apply two-qubit gates. Trainium's compiler can handle most common PyTorch operations, but it cannot compile `torch.linalg.svd`. Specifically, compiling SVD requires the `mhlo.while` IR operation, which Trainium does not support. This means we must manually implement an SVD algorithm on Trainium. This can be done with either a NKI kernel or a custom C++ operator that runs on Trainium's GPSIMD engine. We leave this for future work.

Stabilizer simulation. Under certain conditions, quantum circuits can be simulated in polynomial time. The Gottesman-Knill theorem [10] states that quantum circuits that contain only "Clifford" gates can be perfectly simulated in polynomial time on a classical computer. Although not all quantum circuits can be represented with only Clifford gates, Clifford-only circuits are useful in quantum error correction and some communication problems such as quantum teleportation. We do not attempt to run stabilizer simulation on Trainium because projects such as Stim [9] are highly optimized and can already simulate circuits with thousands of qubits and millions of gates in a matter of seconds.

3.4 Running example: QFT

One important quantum algorithm that promises an exponential advantage over its classical counterpart is the Quantum Fourier Transform (QFT) [6]. The Fourier Transform analyzes a signal (e.g. a sound wave) and outputs its frequency components (e.g. the pitches contained within a sound). The Discrete Fourier Transform (DFT) takes a sampled complex-valued signal as input, and outputs the frequency components of that signal. The input to QFT is encoded as the state of the quantum circuit—the amplitudes of the state vector are interpreted as samples of a signal. After the QFT circuit, the new state vector amplitudes correspond to the frequency components of the original signal.

The classical DFT is used for many applications, including spectral analysis, data compression, and audio processing. Given an input of $N = 2^n$ samples, the Fast Fourier Transform algorithm can compute the DFT in $O(N \log N) = O(n2^n)$ time. The QFT requires $O(n^2)$ gates, yielding an exponential speedup. The QFT is also used in Shor's algorithm [15], which gives an exponential speedup for factoring integers and threatens to break RSA encryption.

4 ALL-CPU SIMULATION

As a baseline, we use two CPU-only quantum circuit simulators to run QFT.

4.1 Qiskit implementation

Qiskit is a commonly-used framework for building and simulating quantum circuits. It provides an implementation of the QFT circuit that we use for the rest of this project. Figure 4 shows how to create a QFT circuit in Qiskit with $n = 20$ qubits, as well as how to use Qiskit's state vector simulator to get the output state.

In line 9, we transpile the circuit, decomposing the general QFT gate into a sequence of U and CX gates. We do this because quantum computers are limited in the gates they can handle; no quantum computer supports QFT as a single operation. In particular, the gate set $\{U, CX\}$ is used in many IBM quantum computers, where

U represents any one-qubit gate, and CX is a particular two-qubit gate. This gate set is universal, so any quantum circuit can be decomposed into it. Even though we are not running the circuit on a real quantum computer, it is still useful to have the circuit in a common form that is supported by all simulation methods. For the rest of this project, we consider the transpiled circuit `tqc` to be the "source circuit," and the sequence of U and CX gates to be the "source-level gates."

```python
import qiskit
from qiskit.circuit.library import QFTGate
from qiskit_aer import StatevectorSimulator

n = 20
qc = qiskit.QuantumCircuit(n)
qc.append(QFTGate(n), range(n))

tqc = qiskit.transpile(qc, basis_gates=['u', 'cx'])

backend = StatevectorSimulator()
job = backend.run(tqc, shots=1)
print(job.result())
```

Figure 4: Qiskit implementation and simulation of QFT.

4.2 Custom implementation

In order to compare to the hybrid CPU+Trainium implementation that will be described in Section 5, we also have a CPU-only circuit simulator that uses the same algorithm as the hybrid implementation. We simply replace all Neuron operations in the hybrid code with their PyTorch equivalents. This allows us to more fairly evaluate the advantages of the Trainium hardware without bias from how our algorithm differs from Qiskit.

5 HYBRID SIMULATION

In this section, we present a quantum circuit state vector simulator that uses both the CPU and the Trainium device. In general, applying a quantum gate in a state vector simulator consists of two operations: qubit indexing and matrix multiplication. For an m-qubit gate U on an n-qubit circuit, we must do 2^{n-m} matrix multiplications, each multiplication being with U and 2^m elements of the state vector. These 2^m elements are generally scattered throughout the state vector, so one approach is to gather them, multiply by U, and store them back in their original locations. We use this approach for the all-Trainium implementation in Section 6. However, for the hybrid implementation, we use a different method based on permuting qubits.

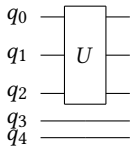

Figure 5: A gate that only applies to the first m qubits is relatively easy to compute.

If the gate we are trying to apply acts on only the first m qubits (Figure 5), then the matrix multiplication has the following form:

$$\begin{aligned}\psi' &= (I \otimes U)\psi \\ &= \begin{pmatrix} U & & & \\ & U & & \\ & & \ddots & \\ & & & U \end{pmatrix}\psi \\ &= \begin{pmatrix} U\psi_1 \\ U\psi_2 \\ \vdots \\ U\psi_{n-m} \end{pmatrix}\end{aligned}$$

The gate operation takes the form of a block-diagonal matrix, where each block U is $2^m \times 2^m$, and there are 2^{n-m} copies of U along the diagonal. If we can do a $2^m \times 2^m \times 1$ matrix multiplication in a single operation, then applying this gate takes $O(2^{n-m})$ time. This is because we only need to multiply U by each contiguous tile of 2^m elements of ψ.

For this to work, we need to do three operations: fuse multiple source-level gates into a single $2^m \times 2^m$ unitary (Section 5.1), shuffle the qubits so that the gate applies only to the first m qubits (Section 5.2), and apply the matrix multiplication to each length-2^m tile of ψ (Section 5.3). In Section 5.4, we discuss some limitations of this method and motivate the all-Trainium implementation in Section 6.

5.1 Gate fusion

Recall from Section 4.1 that the source circuit is given as a sequence of one and two-qubit gates (specifically U and CX gates). Simulating a circuit with these gates would require 2×2 and 4×4 matrix multiplication. In order to take advantage of Trainium's Tensor Engine, we *fuse* the source-level gates into a larger m-qubit gate, corresponding to a $2^m \times 2^m$ matrix multiplication.

In order to fuse a sequence of source-level gates into an m-qubit gate, they must all operate on a shared set of m qubits. Then we can extend each source-level gate's unitary matrix to be $2^m \times 2^m$ by taking the tensor product with the identity matrix. Finally, we multiply each extended unitary into a single $2^m \times 2^m$ matrix. Because each gate fusion works on m qubits, we are essentially simulating m-qubit subcircuits.

Algorithm 1 shows our implementation of gate fusion. We loop through all the source-level gates, adding them to an m-qubit subcircuit. We keep track of the qubit indices that are included in the subcircuit in currIdcs. If adding a gate would result in the subcircuit having more than m qubits, then we combine the current m-qubit subcircuit into a single m qubit gate by simulating the subcircuit.

Algorithm 1 Gate fusion

```
Input: list of source-level gates
Output: list of m-qubit gates
outputGates ← ()
subcircuit ← ()
currIdcs ← {}
for gate in srcGates do
    if |gate.idcs ∪ currIdcs| > m then
        outputGate ← SIMULATE(subcircuit)
        outputGates.append(outputGate)
        subcircuit ← ()
        currIdcs ← {}
    end if
    subcircuit.append(gate)
    currIdcs ← currIdcs ∪ gate.idcs
end for
outputGate ← SIMULATE(subcircuit)
outputGates.append(outputGate)
return outputGates
```

In this case we don't have a state vector, so "simulate" means matrix multiplying the source-level gates in the subcircuit to get a $2^m \times 2^m$ unitary matrix.

The time complexity of gate fusion is $O(s \cdot 2^{3m})$, where s is the number of source-level gates and m is the output gate size. Since m is a constant (for this project we set $m \leq 7$), the algorithm is linear in the number of source-level gates. This means we can pre-fuse the gates in CPU before we pass them into Trainium.

5.2 Shuffling qubits

For reasons described in Section 5.4, we perform qubit shuffling using the host machine's CPU.

In order to shuffle qubits around, we need to permute the state vector. Recall that each element of the length-2^n state vector corresponds to a unique bit string, where each bit corresponds to a qubit. We can instead view the state vector as an n-dimensional tensor, where each dimension is size two. Then we can directly relate each element of the state vector to its corresponding bit string by simply looking at its n-dimensional index.

For example, suppose we have a 5-qubit state vector, and we want to find the element corresponding to the bitstring 01001. With the flat state vector representation, we would look at index 9 (`psi[9]`). But with the n-dimensional tensor representation, we index using the bitstring itself: `psi[0,1,0,0,1]`. This representation has the advantage of making qubit permutations simple—we simply need to permute the dimensions of the n-dimensional state vector. We use PyTorch's `torch.permute` function to do this.

Figure 6 shows a simplified version of the permutation operation required before applying a gate. Given a list of qubit indices that the gate operates on, we create an list of indices called `permutation` that contains our target indices at the beginning, and all other indices after it. For the sake of convention, we reverse this ordering. We also need the inverse-permutation to restore the original qubit ordering later. Overall, for each gate, we permute the qubits, apply the block-diagonal matrix multiplication, and invert the permutation to restore the qubit ordering.

```
1  idcs = [...]
2  permutation = idcs + \
3    [i for i in range(n) if i not in idcs]
4  permutation = permutation[::-1]
5
6  inv_permutation = [0] * n
7  for i, dim in enumerate(permutation):
8      inv_permutation[dim] = i
9
10 # Shuffle qubits
11 state = state.reshape([2] * n)
12 state = torch.permute(state, permutation)
13
14 # Block-diagonal matrix multiplication
15 state = state.reshape([-1]).to('xla')
16 state = multiply_block_diag(U, state).to('cpu')
17
18 # Un-shuffle qubits
19 state = state.reshape([2] * n)
20 state = torch.permute(state, inv_permutation)
```

Figure 6: Qubit shuffling implementation. "`idcs`" is a list of qubit indices that the gate operates on.

5.3 Block-diagonal matrix multiplication

As mentioned in Section 2.1, Trainium is able to perform a $128 \times 128 \times 512$ matrix multiplication in a single operation. Thus a natural choice for gate size is 7 qubits.

To perform the block-diagonal matrix multiplication, we write a NKI kernel that takes the initial state vector ψ and 128×128 matrix U, and returns $(I \otimes U)\psi$. Figure 7 shows a simplified version of the kernel.

First, we need handle complex numbers. Trainium does not support complex values natively, so we add another dimension to our state vector ψ and unitary U to store the real and imaginary components of the vector separately. This also requires slight modifications to the permutation code in Figure 6. The new dimensions of ψ and U are shown in lines 6–7.

Next, we need to loop through each length-128 tile ψ_{128} of the state vector and multiply it by U. A naive implementation would simply perform the $128 \times 128 \times 1$ matrix multiplication, but we can batch up to 512 tiles together and multiply them in a single $128 \times 128 \times 512$ matrix multiplication. To do this, we first need to reshape the state vector so that the real and imaginary components are each size $2^{n-7} \times 128$. Then we load a 512×128 tile into SBUF, simultaneously transposing it, which yields a 128×512 tile of the state vector in column-major format.

In order to perform complex matrix multiplication, with each operand's real and imaginary components separated in the first dimension, we use the 3M method presented in Ref. [16].

```python
from neuronxcc import nki
import neuronxcc.nki.language as nl

@nki.jit
def multiply_block_diag(state, U):
  assert state.shape == (2, 2**n)
  assert U.shape == (2, 128, 128)

  state = state.reshape([2, -1, 128])
  U_tile_real = nl.load(U[0])
  U_tile_imag = nl.load(U[1])

  for i in nl.affine_range(state.shape[1] // 512):
    offset = i * 512

    # Load
    state_tile_real = nl.load_transpose2d(
      state[0, offset:offset+512, 0:128]
    )
    state_tile_imag = nl.load_transpose2d(
      state[1, offset:offset+512, 0:128]
    )


    # Multiply U by the state tile
    W_real = nl.matmul(U_tile_real, state_tile_real)
    W_imag = nl.matmul(U_tile_imag, state_tile_imag)
    SA = nl.add(U_tile_real, U_tile_imag)
    SB = nl.add(state_tile_real, state_tile_imag)
    SASB = nl.matmul(SA, SB)
    state_tile_real = nl.subtract(W_real, W_imag)
    state_tile_imag = nl.subtract(
      nl.subtract(SASB, W_real),
      W_imag
    )

    # Store
    nl.store_transpose2d(
      state[0, offset:offset+512, 0:128],
      value=state_tile_real
    )
    nl.store_transpose2d(
      state[1, offset:offset+512, 0:128],
      value=state_tile_imag
    )
  return state
```

Figure 7: NKI kernel for performing block-diagonal matrix multiplication.

5.4 Limitations of the hybrid implementation

The main limitation of this method is the qubit shuffling. This operation requires permuting the entire 2^n state vector, which Trainium is not well-suited for. This is why we do the operation in the host device's CPU. In fact, if we try to compile `torch.permute` using the Neuron compiler so that we can run the permutation on Trainium, compilation is too slow to be feasible. In a 20 qubit circuit, it takes around 15 minutes to compile a single state vector permutation. Additionally, because each gate has a different qubit permutation that results in a different compilation graph, we need to do this 15 minute compilation for every gate in the circuit.

Instead, we perform the permutation in the host device using CPU. This avoids most of the compilation overhead—we now only need to compile the block diagonal matrix multiplication kernel once for each circuit. However, this incurs a large overhead of transferring the entire state vector between host and Trainium for each gate. The exact cost of this memory transfer is shown in Section 7.

This raises a question: can we perform qubit shuffling within Trainium in a way that reduces the compilation overhead? We discuss an all-Trainium implementation in Section 6.

6 ALL-TRAINIUM SIMULATION

As discussed in Section 5.4, using `torch.permute` to perform qubit shuffling in Trainium is infeasible due to long compilation times. In this section, we explore a method of manually performing the permutation operation from within a NKI kernel in an attempt to reduce compilation time.

For each gate in Section 5's hybrid implementation, we moved the state vector from host to HBM in the Trainium device. Then for each tile of the state vector, we would load the tile into SBUF, multiply with U, and store the result back into HBM. Finally, the state vector is moved back to host. For the all-Trainium implementation, we want to avoid as much memory-transfer overhead as possible. This means the state vector must stay in the Trainium device between gate applications.

To accomplish this, we make use of Neuron's indirect load operation, which supports dynamic index values. Our unpermuted state vector stays in HBM, and we use the dynamic indexing to load certain elements into SBUF that correspond to the correct indices if we had actually done a permutation. Then we multiply these elements with U and store them back to HBM, once again using dynamic indexing to ensure the elements return to their original indices.

6.1 Algorithm overview

Algorithm 2 All-trainium simulation

1: **Input:** ψ, m-qubit gates
2: **Output:** ψ after applying gates
3: **for** g in gates **do**
4: **for** i_{tile} in $0, 1, \ldots, 2^{n-m} - 1$ **do**
5: $j_{orig} \leftarrow [i_{tile}, i_{tile} + 1, \ldots, i_{tile} + 2^m - 1]$
6: $j_{perm} \leftarrow \text{PERMUTE}(j_{orig}, g.\text{idcs})$
7: $\psi_{tile} \leftarrow \text{LOAD}(\psi[j_{perm}])$
8: $\psi_{tile} \leftarrow U\psi_{tile}$
9: $\psi[j_{perm}] \leftarrow \text{STORE}(\psi_{tile})$
10: **end for**
11: **end for**
12: **return** ψ

Algorithm 2 shows pseudocode for the NKI kernel that implements our all-Trainium simulator. The inner loop on line 4 corresponds to the `for` loop in the block-diagonal matrix multiplication kernel in Figure 7. On line 6, we calculate the permuted state

vector indices based on which qubits are acted on by the gate. On lines 7 and 9, we use Trainium's dynamic indexing to load and store elements from arbitrary locations in ψ. Notice that if we remove j_{perm} from the algorithm and only use j_{orig}, the algorithm would simply compute the block-diagonal matrix multiplication for each gate.

6.2 Implementing PERMUTE, LOAD, and STORE

```
1  j_orig = ...
2  j_perm = nl.copy(j_perm)
3  for i in nl.sequential_range(len(gate.qubitSwaps)):
4    a  = nl.load(gate.qubitSwaps[i].src)
5    b  = nl.load(gate.qubitSwaps[i].dst)
6    ma = 1 << a
7    mb = 1 << b
8
9    j_perm[:] = j_perm & ~mb
10   j_perm[:] = j_perm | ((j_orig & ma) << (b-a))
11   j_perm[:] = j_perm | ((j_orig & ma) << (a-b))
```

Figure 8: Implementation of PERMUTE in the NKI kernel.

Figure 8 shows a simplified version of the state vector index permutation code. This implements the PERMUTE function from line 6 of Algorithm 2. The only difference between this code and the PERMUTE function in the pseudocode is that this function expects the gate's qubit indices to be provided in the form of a sequence of qubit swaps. For example, in a circuit with $n = 3$ and $m = 2$, a gate that acts on qubits $\{q_1, q_2\}$ will require swaps $q_1 \rightarrow q_0$, $q_2 \rightarrow q_1$, and $q_0 \rightarrow q_2$. This results in a qubit permutation where qubits $\{q_1, q_2\}$ end up at the top of the circuit (i.e. the first m qubits).

The bitwise operations from lines 9–11 implement the following logic: "For each (integer) element of `j_perm`, copy the ath bit of that element to its bth bit." This has the effect of moving the qubit at index a in the circuit to index b. In the actual kernel, the bitwise operators are replaced with NKI API calls such as `nl.bitwise_and` and `nl.left_shift`. These operations apply element-wise, utilizing Trainium's Scalar Engine.

The last part of Algorithm 2 (the load/matmul/store) is implemented in Figure 9. Once we have the permuted state vector indices `j_perm`, we can simply use them to index the state vector while we are loading and storing.

```
1  state_tile = nl.load(state[j_perm])
2  state_tile = nl.matmul(U, state_tile)
3  nl.store(state[j_perm], value=state_tile)
```

Figure 9: Performing a dynamically-indexed load/store with NKI (real numbers only).

6.3 Limitations of the all-Trainium implementation

In exchange for solving the Host↔Trainium memory-bandwidth limitation of the hybrid implementation mentioned in Section 5.4, the all-Trainium implementation introduces two new limitations. The first limitation is the PERMUTE operation. This adds a lot of bitwise operations that we need to compute within Trainium. The second limitation is the dynamically-indexed load and store. Each of these operations access elements from all over the state vector, so this method will only perform well if these accesses are well-supported by Trainium.

In addition, we still need to compile the qubit permutations for each gate separately. This is the same problem as `torch.permute`, but not quite as severe. Instead of taking around 15 minutes to compile each gate for a 20 qubit circuit, it takes around 20 seconds. Exact measurements are provided in Section 7. For most circuits, it is a fatal limitation to take 20 seconds per gate for $n = 20$. However, there are some variational algorithms that require many runs of the same circuit layout, but with each gate's matrix having different elements between each run. For example, the quantum approximate optimization algorithm (QAOA) [7] is an algorithm for combinatorial optimization that involves a circuit whose gates are parametrized. Finding the optimal solution involves running the circuit many times with different parameters each time. In this case, the circuit only needs to be compiled once, and it can be run many times without recompilation.

7 EXPERIMENTAL RESULTS

In this section, we present measurements of each implementation's execution time and discuss our findings.

7.1 Experimental setup

We evaluate four quantum circuit simulators: Qiskit (Section 4.1), a custom CPU-only implementation (Section 4.2), our hybrid implementation (Section 5), and the Trainium-only implementation (Section 6).

For the CPU-only implementations, we ran on the "Bora" server at UCLA. This machine has an Intel(R) Xeon(R) Silver 4116 CPU with 24 cores, as well as 188 GB of memory. For the hybrid and all-Trainium implementations, we ran on the AWS EC2 trn1.2xlarge instance, which provides one Trainium chip, 32 GB of device memory (HBM), 8 host vCPUs, and 32 GB of host memory.

7.2 Comparison of simulation times

Figure 10 shows a comparison of the execution of each simulation method per source-level gate. For the CPU-only and hybrid implementations, we simulate QFT. We measure the amount of time each simulator takes to run (excluding compile times), and divide by the original number of source-level (U and CX) gates.

For the all-Trainium method, for reasons explained in Section 7.4, we do not simulate QFT. Instead, we simulate a single 7-qubit gate, measure its execution time, and divide by the QFT gate fusion ratio found in Figure 11. To mitigate the effects of circuit-constant overhead, we only start measuring execution time after all input tensors (i.e. initial state vector, gates, and parameters) are loaded into the Trainium device.

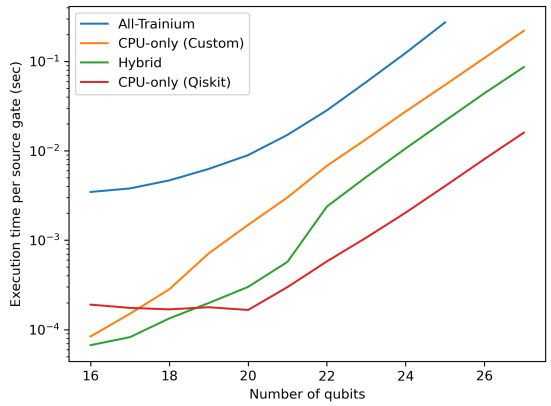

Figure 10: Execution time per source-level gate for each implementation.

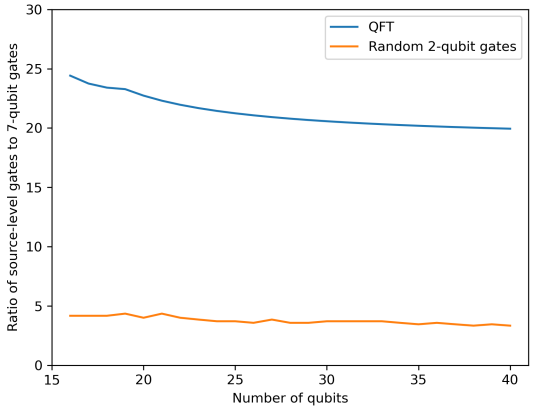

Figure 11: Gate fusion ratios for QFT and circuits with random 2-qubit gates.

Figure 11 shows the effectiveness of our gate fusion described in Section 5.1. For circuits composed of random 2-qubit gates, there is not much benefit. But for QFT in particular, the source-level gates have a lot of qubit-locality, so we can fuse many more gates together. On average, for a large number of qubits, we can fuse about 20 source-level QFT gates into a single 7-qubit gate.

7.3 Hybrid implementation analysis

As discussed in Section 5, the main limitation of the hybrid implementation is that we must move the state vector to the host's memory and permute in CPU for each gate. Figure 12 shows a breakdown of the hybrid simulator's execution time, separating it into three operations: (1) transferring the state vector between host and Trainium device, (2) permuting the state vector in CPU, and (3) performing the matrix multiplication in Trainium. From the figure, we see that for a low number of qubits, the matrix multiplication is the main bottleneck; transferring and permuting the small state vector does not take much time. For large numbers of qubits, the matrix multiplication becomes the fastest operation, with transferring and permuting being the main bottlenecks. There is a clear crossover point at 19 qubits.

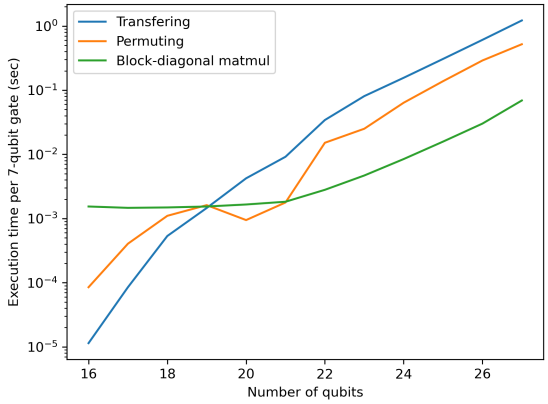

Figure 12: Breakdown of execution time overheads for the hybrid implementation.

7.4 All-trainium simulator compilation

Figure 13 breaks down the time to compile and execute the all-Trainium simulator for a single gate. We can see that the compilation time, while much faster than trying to compile `torch.permute`, is still a major bottleneck.

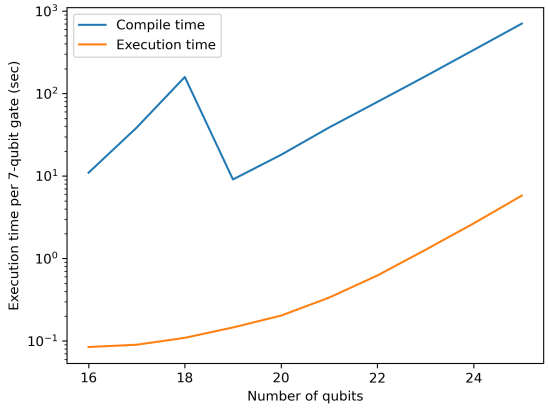

Figure 13: Compile time for a single gate in the all-Trainium implementation.

The memory requirement of compiling the all-Trainium simulator is another significant restriction. Attempting to compile a 26-qubit single-gate circuit consumed all the memory (32 GB) of

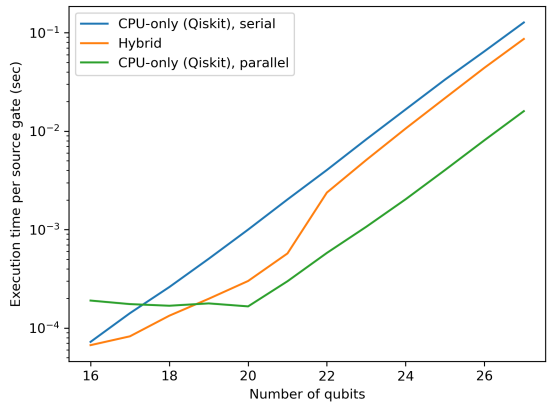

Figure 14: Performance of the hybrid implementation compared to Qiskit with and without parallelism.

our trn1 instance. We expect this memory requirement to scale linearly with the number of gates and exponentially with the number of qubits. This also prevented us from running the full QFT benchmark. As a whole, compiling the all-Trainium implementation is infeasible for any sizable circuit because of both compilation time and memory bottlenecks.

For $n < 19$, the compilation time is higher than expected. We suspect that this is due to some compiler optimizations that only apply when the size of the compiled model is smaller than a certain size.

7.5 Impact of parallelism

One limitation of our hybrid and Trainium implementations is their lack of parallelism. We only target a single NeuronCore, so the only parallelism comes from the different engines (e.g. DMA, Tensor, Scalar) working simultaneously. Figure 14 shows how our hybrid implementation compares to Qiskit with and without parallelism. We see that it is faster than single-threaded Qiskit, but is slower than multi-threaded Qiskit.

7.6 Discussion

From these results, it is clear that Trainium is not well-suited to quantum circuit state vector simulation. Based on the results, we discuss four limitations of the Trainium hardware and software stack that make it incompatible with state vector simulation.

Low memory bandwidth between host and device. The hybrid implementation's largest bottleneck is the overhead of transferring the state vector between the host and Trainium device for each gate. Although Trainium has high memory bandwidth within the device (e.g. transferring between HBM and SBUF), it suffers when a lot of data must move back and forth to host memory.

Simulators written for CPUs do not suffer from this problem because the state vector can simply stay in main memory. The all-Trainium simulator also does not suffer from this, but it does suffer from the other limitations.

Poor performance with nonlocal accesses. One key difference between quantum circuit simulation and machine learning is data locality. For example, large language models (LLMs) use transformer models [17], which consist of a sequence of layers that apply many matrix multiplications. When training a transformer, each sequence of input tokens is passed in and multiplied with the layers. These are dense matrix multiplications that do not require any data reordering or permutations, so there is high spatial locality. The layers are reused between inputs, so there is also high temporal locality.

In contrast, quantum circuit state vector simulation has bad spatial and temporal locality. In order to apply a gate, we must gather elements from throughout the state vector and multiply them with the unitary matrix. As we increase the number of qubits, these elements get exponentially more scattered throughout the state vector. There is also poor temporal locality, as each gate is applied to the state vector only once. We cannot reuse them like we do for layers in a deep neural network. While a neural network can run many times with different inputs each time, this is uncommon for quantum circuit state vector simulation, with the exception of variational algorithms.

Poor spatial locality is the main source of overhead in the all-Trainium simulator. Trainium is bad at cross-partition lane operations in SBUF, which makes dynamic indexing for permutations create many small DMA transfers, causing a large overhead [11].

Compilation overhead. With the all-Trainium simulator, the biggest overhead is in its compilation time. The Neuron compiler must convert all the qubit permutations and gate applications into an XLA computation graph and represent it using HLO IR operations. Because the Neuron SDK was designed for machine learning applications, it does not have good support for the operations involved in permuting a large state vector. The compiler must handle the large amount of bitwise operations involved in creating the permutation indices, multiplied by the number of gates in the circuit. Neuron has good support high-level operations that are common in machine learning, such as matrix multiplication, activation functions, and normalization.

Additionally, machine learning models fall under the "compile once, run many times" paradigm. Once a model is compiled, we can run training and inference on it many times without having to recompile. In contrast, quantum circuit state vector simulation is often "compile once, run once," because for non-variational algorithms, there is little purpose in simulating the circuit twice. This means compilation time becomes much more important, as we cannot amortize the cost by running the compiled model many times.

Lack of parallelism within a NeuronCore. Because each NeuronCore has only one TensorCore, we can only do one matrix multiplication at a time. From Section 7.5, we can see that Qiskit benefits greatly from multi-threading. It would not be easy to distribute the all-Trainium simulator over multiple NeuronCores because each core has its own memory space. It would be easy to distribute the hybrid implementation using SPMD, but this is not likely to result in much improvement because the hybrid implementation's main overhead is in transferring data between device and host, as well as permuting in CPU.

8 CONCLUSION

We have presented two implementations of quantum circuit state vector simulators that utilize the AWS Trainium chip and compared them with existing CPU-only implementations. We evalutated circuit execution time on the Quantum Fourier Transform and found that our implementations are not faster than state-of-the-art CPU simulators. We discuss the properties of Trainium that make it ill-suited for state vector simulation—low memory bandwidth between the Trainium device and its host, poor performance with dynamically-indexed accesses that have poor spatial locality, heavy compilation overhead, and lack of parallelism within each NeuronCore.

ACKNOWLEDGEMENTS

Many thanks to Richard Yin and Runzhao Guo for their contributions; Professor Jens Palsberg for his guidance; and Adam Stanley, John Gray, and the Trainium compiler team for their assistance and insight.

REFERENCES

[1] Amazon Web Services, Inc. 2025. AWS Inferentia. https://aws.amazon.com/ai/machine-learning/inferentia/ Accessed: November 03, 2025.
[2] Amazon Web Services, Inc. 2025. AWS Trainium. https://aws.amazon.com/ai/machine-learning/trainium/ Accessed: November 03, 2025.
[3] Amazon Web Services, Inc. 2025. Build on Trainium. https://aws.amazon.com/ai/machine-learning/trainium/research/ Accessed: November 03, 2025.
[4] Amazon Web Services, Inc. 2025. NKI Programming Model. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/nki/programming_model.html Accessed: November 03, 2025.
[5] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, et al. 2023. cuquantum sdk: A high-performance library for accelerating quantum science. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Vol. 1. IEEE, 1050–1061.
[6] Don Coppersmith. 2002. An approximate Fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067* (2002).
[7] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028* (2014).
[8] Roy Frostig, Matthew James Johnson, and Chris Leary. 2019. Compiling machine learning programs via high-level tracing. In *SysML conference 2018*.
[9] Craig Gidney. 2021. Stim: a fast stabilizer circuit simulator. *Quantum* 5 (2021), 497.
[10] Daniel Gottesman. 1998. The Heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006* (1998).
[11] John Gray. 2025. Private communication.
[12] 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, et al. 2024. Quantum computing with Qiskit. *arXiv preprint arXiv:2405.08810* (2024).
[13] OpenXLA. 2022. XLA. https://github.com/openxla/xla.
[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[15] Peter W Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 124–134.
[16] Field G Van Zee and Tyler M Smith. 2017. Implementing high-performance complex matrix multiplication via the 3m and 4m methods. *ACM Transactions on Mathematical Software (TOMS)* 44, 1 (2017), 1–36.
[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[18] Hristo Venev, Thien Udomsrirungruang, Dimitar Dimitrov, Timon Gehr, and Martin Vechev. 2025. qblaze: An Efficient and Scalable Sparse Quantum Simulator. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 444–470.
[19] Guifré Vidal. 2003. Efficient classical simulation of slightly entangled quantum computations. *Physical review letters* 91, 14 (2003), 147902.