ABSTRACT

CHUNDURY, SRIKAR. DiaQ: A Novel Quantum-Tailored Numerical Format. (Under the direction of Dr. Frank Mueller).

In the current NISQ era of Quantum Computing, efficient digital quantum simulation is pivotal in algorithm development and verification. A major bottleneck in classical simulation is the exponential growth in Hilbert space. Hence, the need for efficient storage and computation of quantum states and unitaries is paramount. Although there has been prior research in the sparsity seen in the state-vector, there has been limited research in the sparsity seen in the unitary matrices, more so for structured sparsity patterns.

In this thesis, we first analyze the structured sparsity patterns seen in unitary matrices and propose a novel sparse numerical format called DiaQ specifically designing for quantum simulations. We demonstrate the efficiency of DiaQ in terms of storage and computation by comparing it with dense numerical formats. We then present algorithms for key kernels required in simulations, e.g. the format conversion from dense to DiaQ, sparse matrix multiplication, sparse tensor dot, among others. We implement the format and these kernels with performance optimizations like OpenMP multi-threading, SIMD vectorization and GPU acceleration in a C++ library called `libdiaq`. We then evaluate `libdiaq`'s performance for unitary and tensor network simulations. Apart from DiaQ, in the context of tensor network simulations, we also develop a new plugin for TNQVM [1] that enables PEPS [2] simulation via XACC [3].

We evaluate the performance of DiaQ empirically and experimentally, showing that it outperforms dense numerical formats by many orders of magnitude. We also show that the parallelized library demonstrates superior scalability for tensor network simulations. We conclude that a sparse numerical format specifically designed for quantum simulations can be more efficient in performance and storage than dense general-purpose numerical formats. For the PEPS plugin, we perform a proof of functionality test by comparing results and performance with TNQVM's existing MPS plugin.

[1] Tensor Network Quantum Virtual Machine, a library for simulating quantum states using tensor networks.
[2] Projected Entangled Pair States, a 2D graph.
[3] eXascale ACCelerator programming framework.

DiaQ: A Novel Quantum-Tailored Numerical Format

by
Srikar Chundury

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina
2024

APPROVED BY:

Dr. Jiajia Li

Dr. Xipeng Shen

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Srikar hails from India. He completed his schooling at DAV Public School, Safilguda, Hyderabad. He then moved to Bangalore to obtain his bachelor's degree in Computer Science with a specialization in Systems and Core Computing from PES University in 2019. His undergraduate thesis, titled "Program Phase Characterization for Big Data Workloads", was conducted under the guidance of Dr. Subramaniam Kalambur. Following that, he worked as a Software Engineer at Walmart Labs, Bangalore, where he developed tools for Walmart's Infrastructure team. In 2022, he joined North Carolina State University to pursue his master's degree and began his research journey under the guidance of Dr. Frank Mueller. Srikar's research interests lie at the intersection of High-Performance Computing and Quantum Computing. After completing his master's degree, he plans to continue his research and pursue a Ph.D. degree with the support and guidance of Dr. Mueller.

In his leisure time, Srikar enjoys watching sports, playing cricket and table tennis, and exploring new places.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to Dr. Frank Mueller for his unwavering support, invaluable ideas, exceptional guidance, and encouraging faith in me. I hold him in the highest regard both professionally and personally, as his perspective on life has been truly inspiring. I would also like to extend my heartfelt thanks to Dr. Jiajia Li and Dr. In-Saeng Suh for their insightful feedback and valuable suggestions throughout my master's degree journey. Additionally, I am immensely grateful to my close friends (some in India, some in the US, and some I made at NC State) and my family (including my sister & brother-in-law, maternal uncles, and cousin brothers) for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Dr. Richard Feynman in 1982 suggested that the best chance of simulating physics using computers is by using quantum computers that leverage quantum mechanical properties as nature is not classical (Feynman 1982).

He also suggested the possibility of a universal system (quantum simulator) which could be “programmed" in a way that it could simulate any other quantum system.

The simulators that are general purpose and, hence, can be programmed to target any quantum system are called digital quantum simulators. These simulators are based on the principles of quantum computation and can be programmed to simulate any quantum system. Analog quantum simulators, on the other hand, are designed to simulate a specific quantum system. They are based on the principles of quantum mechanics and are designed to simulate a specific quantum system.

Quantum simulators today can be realized on a variety of platforms such as trapped ions, ultracold quantum gases, photonic systems, superconducting circuits, and quantum dots.

Using classical techniques to simulate the operations that quantum simulators perform but using linear algebra is also referred to as simulation but is classical in nature.

1.1 Motivation

We are currently in the Noisy Intermediate-Scale Quantum (NISQ) era, where quantum hardware exhibits errors, noise, limited qubits, and shorter decoherence times. Consequently, utilizing quantum hardware for every minor program modification, as we do with classical computers, is not a viable option. This is where the significance of efficient classical simulation arises.

But it comes with the curse of dimensionality, where the size of the Hilbert space [1] grows exponentially with the number of qubits. This exponential growth in the Hilbert space makes it infeasible to simulate large quantum systems on classical computers. This is pointed out by many researchers, including Dr. Yuri Manin, who stressed on the exponential cost that a classical computer would incur to simulate many-particle systems (Preskill 2021).

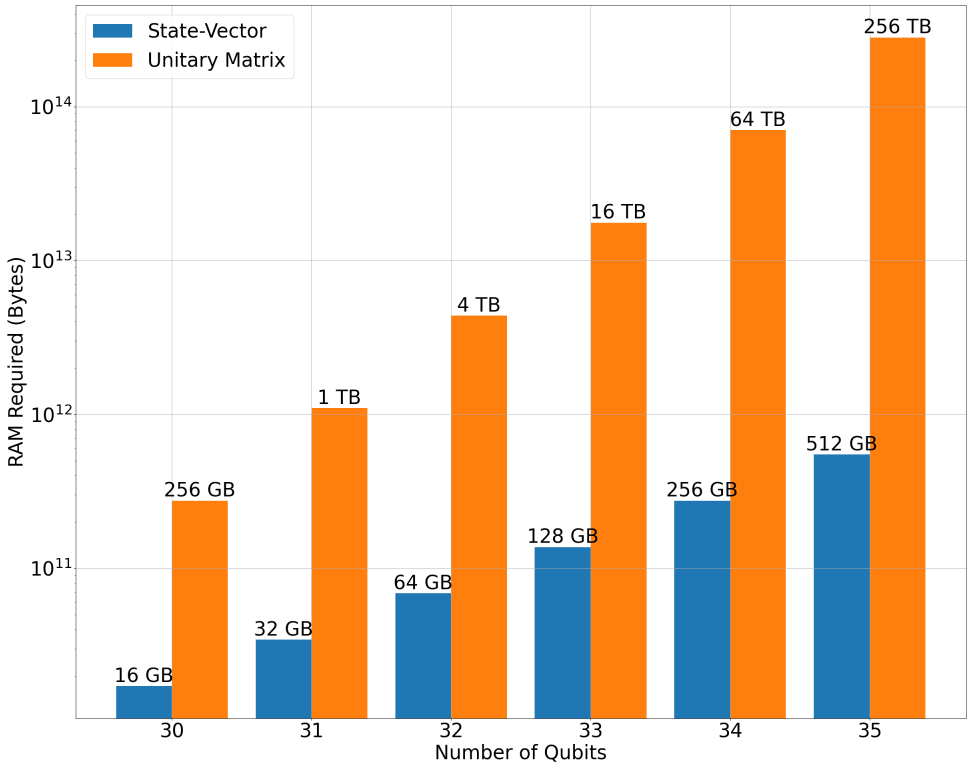


Figure 1.1: RAM requirements for simulating qubit systems

Figure 1.1 demonstrates the amount of RAM required to simulate various qubit systems classically. As shown, the memory requirements double and increase by four times for the state-vector and unitary matrix, respectively, with each additional qubit, assuming complex double precision values.

Because of this curse of dimensionality, it is important to develop efficient numerical formats that can store and compute quantum states and unitaries in a more efficient manner. Hence, there is immense scope to improve both memory and computation efficiency in quantum simulations and is not explored in depth thus far.

[1] The mathematical space that describes all possible states of a quantum system.

1.2 Hypothesis

We hypothesize that a sparse numerical format specifically designed for quantum simulations can be more efficient in performance and storage than dense general purpose numerical formats.

1.3 Contributions

This work makes the following contributions:

- Formulation of a new numerical format specifically designed for quantum simulations and implementation as a C++ library.
- Testing the efficacy of the numerical format on unitary simulations and tensor network simulations.
- Implementation and integration of a Projected Entangled Pair State (PEPS) topology into TNQVM.

CHAPTER

2

BACKGROUND

In this chapter, we will discuss the different types of quantum simulations and existing sparse matrix formats. Assume an n-qubit system, where N is the dimension of the Hilbert space ($N = 2^n$).

A **quantum gate** is a unitary operator that acts on one or more qubits. It is used to manipulate the quantum state of the qubits. It can be represented by a unitary matrix.

A **quantum circuit** is a sequence of quantum gates that act on qubits to perform a quantum computation. It is represented as a directed acyclic graph, where the nodes represent the quantum gates, and the edges represent the qubits on which the gates act.

For instance, the Figure 2.4 shows a quantum circuit where H is the Hadamard gate (a single-qubit gate), and the dot to X notation represents a $CNOT$ gate (a two-qubit gate). Their unitary matrices are as follows:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Since quantum circuits are represented as a sequence of quantum gates, we refer to the point in time (x-axis in a quantum circuit) when a gate is applied as a **time-step**. The quantum circuit is executed by applying the gates sequentially, one time-step at a time. For instance, there are 5 timesteps in Figure 2.4.

2.1 Types of Quantum Simulations

Different types of quantum simulations exist and are distinguished by their purpose. The most common types of quantum simulations are:

2.1.1 Unitary Simulation

Every quantum gate is represented by a unitary matrix. And the entire quantum circuit can also be represented as a single unitary matrix by multiplying the unitary matrices of the individual gates. This process of arriving at the circuit unitary by chain multiplication of individual gate unitaries is called Unitary Simulation. Mathematically, unitary simulation is represented as:

$$\mathrm{U}_{\text{quantum circuit}} = U_1 \cdot U_2 \cdot U_3 \cdot \ldots \cdot U_n$$

It is important to note that, as with any chain multiplication, since it is associative, the order in which the matrices are multiplied impacts the performance of the simulation.

2.1.2 State-Vector Simulation

In state-vector simulation, the initial quantum state is represented as a vector of size N. Each value in this vector corresponds to the amplitude of the corresponding state. For instance, in a 2-qubit system, the state-vector is a vector of size 4, where each value corresponds to the amplitude of the states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. The initial is always $|0\rangle$, which mathematically corresponds to the vector $[1,0,\ldots,0]$. This state evolves as gates are applied (vector times matrix operation) to it, arriving at a final state after the entire circuit is executed. This process of evolving the state-vector is called State-vector Simulation. Mathematically, one way to perform state-vector simulation is as follows:

$$\text{Final State} = U_n \cdot (U_{n-1} \cdot (\ldots (U_2 \cdot (U_1 \cdot \text{Initial State})) \ldots))$$

2.1.3 Density-Matrix Simulation

In density-matrix simulation, the quantum state is represented as a density matrix. The density matrix is a matrix of size $N \times N$, which can represent pure and mixed states, where the quantum system is in a superposition of states. Unlike the state-vector, the density matrix can represent statistical mixture of states, implying that the matrix captures inherent noise in the system.

The density matrix is evolved by applying the gates (matrix times matrix operation) to it, similar to the state-vector simulation.

2.1.4 Tensor-Network Simulation

Tensors are multi-dimensional maps or functions, with a 2-dimensional tensor equivalent to a matrix. Tensor-networks are modelled as graphs, with tensors as vertices and tensor methods, like contraction

and decomposition, as edges. Tensor-networks are used to represent quantum states and unitaries in a compact form, and are used in quantum simulations to reduce the memory requirements.

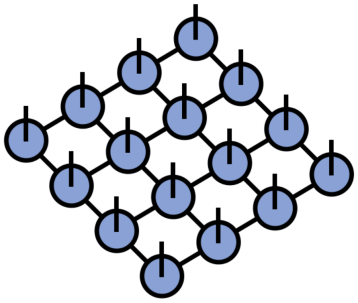


Figure 2.1: Projected Entangled Pair States (PEPS) (tensornetworks 2023).

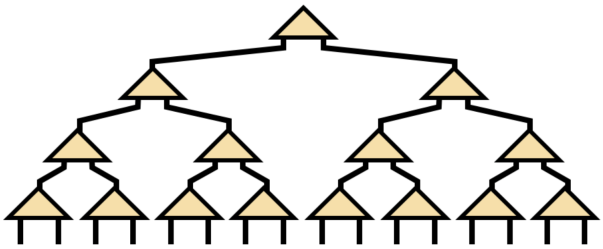


Figure 2.2: Tree Tensor Network (TTN) (tensornetworks 2023).

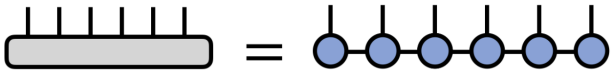


Figure 2.3: Matrix Product State (MPS) (tensornetworks 2023).

There are different types of tensor networks based on their dimensionality and adjacency (tensornetworks 2023). One-dimensional tensor networks, also known as Matrix Product States (MPS) or Tensor Trains (TT), and higher-dimensional tensor networks, like Projected Entangled Pair States (PEPS), have mainly been employed as a proposed quantum wavefunction for the ground states of two-dimensional Hamiltonians (Low and Chuang 2019), (Perez-Garcia et al. 2007). A pictorial depiction of MPS and PEPS can be seen in Figures 2.3 and 2.1, respectively. There exist more tensor networks, e.g. Tree Tensor Networks (TTN) (tensornetworks 2023), which are used to represent the quantum state in a tree-like

structure, as shown in Figure 2.2.

A **tensor contraction** is an operation on a tensor in a finite-dimensional vector space. Therefore, it can be expressed as a sum of products of scalar components of different tensors. A tensor network is a countable collection of tensors connected by "contractions".

In simulating quantum circuits using tensor networks, there are several general steps involved:

- First, the tensor network representation of the circuit is built by adding tensors representing the gates to the initial state, which is usually the product state $|000...00\rangle$. Low-rank decompositions may also be performed on the tensors if necessary.
- Second, the entire tensor network of the desired quantity is formed, which could be the full wavefunction, a local expectation value or reduced density matrix, a marginal probability distribution, or the fidelity with a target state or unitary (Gray 2018).
- Local simplifications are then performed to make the tensor network easier to contract, followed by finding a contraction path to turn the tensor network into a single tensor. The contraction may also be sliced or chunked to fit memory constraints or introduce parallelism.
- Finally, the resulting tensor network can be contracted using specialized libraries, such as our numerical library, to efficiently perform the necessary calculations.

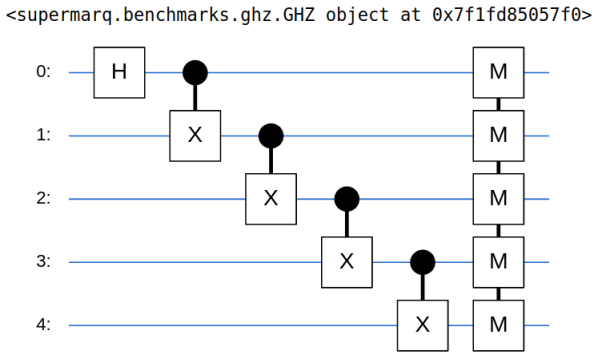


Figure 2.4: SupermarQ (Tomesh et al. 2022) GHZ quantum circuit.

As an example, the Figure 2.4 shows a quantum circuit and Figure 2.5 shows its tensor network representation.

2.2 Sparse Matrix Formats

Considering two-dimensional tensors (i.e., matrices), there are several well-known methods for sparse storage (as outlined in Table 2.1), including Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), and Coordinate Format (COO). However, it is important to choose a data structure that

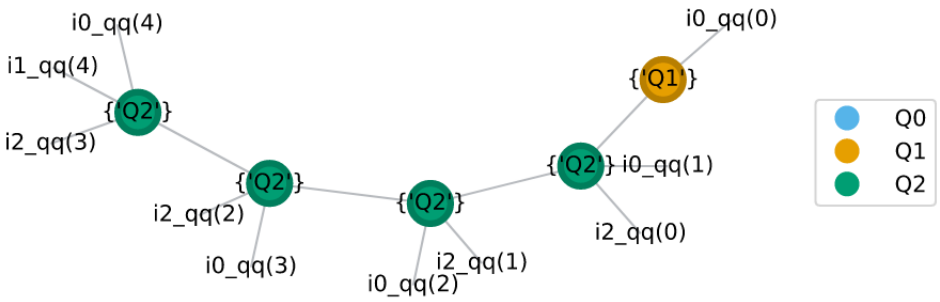


Figure 2.5: SupermarQ (Tomesh et al. 2022) GHZ Tensor-Network Representation. The circles represent tensors in the network, for instance, the yellow circle represents the Hadamard gate and all the green ones represent CNOT gates. The lines (edges) represent bond interactions. The measurements have been removed, since this is a unitary simulation. NOTE: Generated using QUIMB (Gray 2018).

Table 2.1: Comparison of Sparse Matrix Formats

Matrix Format	**Suitable for**	**Extendable to Higher Dimensions?**
COO	Sparse matrices with arbitrary sparsity patterns	Yes
CSR	Row-wise operations	No
CSC	Column-wise operations	No
BSR	Block matrices	No
DIA	Diagonal matrices	No

aligns with our specific needs. For instance, the CSR format is more efficient for row-wise operations, while the CSC format is more efficient for column-wise operations. The COO format is more efficient for constructing the sparse matrix, but not for matrix-vector multiplication.

The Scipy's DIA (Diagonal) format (Virtanen et al. 2020) serves as a representation method for sparse matrices by arranging elements in a diagonal-major pattern, employing offsets to denote positions beyond the matrix's immediate scope.

The COO (Coordinate List) format (PyData 2023) is another sparse matrix representation, utilizing three arrays to store non-zero elements' coordinates and values. For instance, considering the same matrix in COO format, the non-zero elements would be stored as:

$$\text{row\_indices} = [0, 0, 1, 2, 3, 3]$$
$$\text{col\_indices} = [0, 3, 1, 2, 0, 3]$$
$$\text{values} = [a, b, c, d, e, f]$$

CHAPTER

3

DESIGN

In this chapter, we first analyze the structured sparsity of unitary matrices visually, and then design a novel sparse tensor format, DiaQ, to represent these matrices. We then discuss important kernels using DiaQ, such as matrix multiplication and format conversion from dense to DiaQ. Finally, we discuss our other contribution, induction of the PEPS topology into TNQVM.

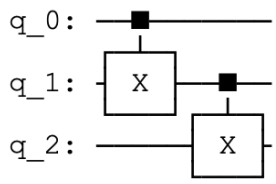
3.1 Sparsity Patterns in Quantum Simulations

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 3.1: The Toffoli gate in matrix form. Notice, the upper left 6x6 submatrix is a diagonal and the lower right 2x2 is a single anti-diagonal.

We observe that quantum gates, represented as matrices (a 2x2 matrix for a 1-qubit gate, and a 4x4 matrix for a 2-qubit gate and so on) are sparse and in a peculiar way. For example, the Toffoli gate can be represented by a $4x4$ matrix, and more specifically by only the places where we find values of 1. Namely, on three diagonals- one principal diagonal and two adjacent to it as shown in Figure 3.1. Many of the simplest gates we work with also follow this trend of values only falling on a select few diagonals.

Circuit unitaries follow a similar trend. For example, the GHZ circuit unitary (Figure 3.2), the Hamiltonian circuit unitary (Figure 3.3), and the Mermin-Bell circuit unitary (Figure 3.4) all exhibit sparsity patterns with non-zero values highlighted in red.



(+1.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+1.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+1.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+1.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+1.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+1.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+1.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+1.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)

Figure 3.2: GHZ Unitary

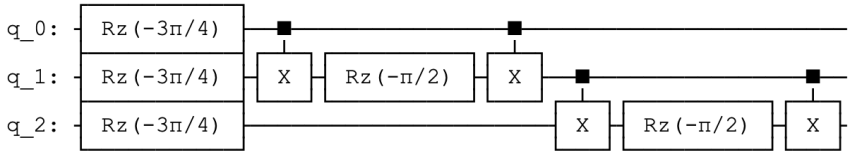
We find that this pattern is consistent across various quantum circuits and unitaries. This structured sparsity pattern can be leveraged to design a novel sparse tensor format that can store and compute these unitaries more efficiently.

3.2 DiaQ: A novel sparse tensor format

Leveraging the structured sparsity patterns seen in the previous section, we design a novel sparse matrix format, DiaQ, specifically catering to quantum simulations. It stores non-zero elements in a diagonal-major pattern. It does so by storing a hashmap of diagonal indices to diagonal array of elements. And since there are limited number of diagonals, we anticipate step-linear retrieval times.

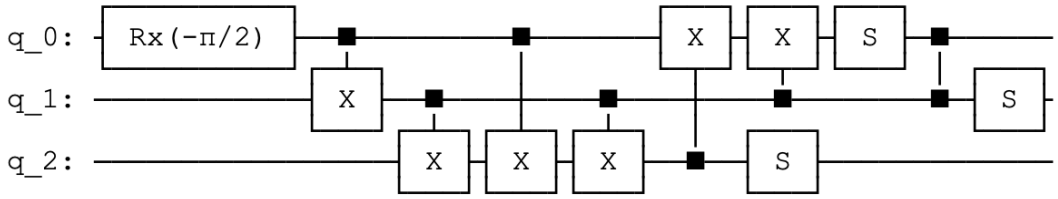
We storage the non-zeros in the below structure:

```
data[diagonal index] ← diagonal elements
```



(+0.4,-0.9) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.4,+0.9) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.9,-0.4) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.4,-0.9) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.4,+0.9) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (-0.9,-0.4) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.4,-0.9) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (-0.4,-0.9)

Figure 3.3: Hamiltonian Unitary



(+0.7,+0.0) (+0.0,+0.7) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (-0.7,+0.0) (+0.0,+0.7)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (-0.7,+0.0) (+0.0,+0.7) (+0.0,+0.0) (+0.0,+0.0)
(-0.0,+0.0) (-0.0,+0.0) (+0.7,+0.0) (+0.0,+0.7) (-0.0,+0.0) (-0.0,+0.0) (-0.0,+0.0) (-0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (-0.7,+0.0) (+0.0,+0.7) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (-0.7,+0.0) (-0.0,-0.7) (+0.0,+0.0) (+0.0,+0.0)
(+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (+0.0,+0.0) (-0.7,+0.0) (-0.0,-0.7)
(-0.7,+0.0) (+0.0,+0.7) (-0.0,+0.0) (-0.0,+0.0) (-0.0,+0.0) (-0.0,+0.0) (-0.0,+0.0) (-0.0,+0.0)

Figure 3.4: Mermin-Bell Unitary

Difference from DIA format

Because the diagonal lengths are fixed for a given matrix shape, we need not store offsets as in the DIA format. This makes DiaQ more efficient in terms of storage. For instance, using the same example as in 2.2, the DiaQ format would store the non-zero elements as:

$$\begin{bmatrix} a & 0 & 0 & b \\ 0 & c & 0 & 0 \\ 0 & 0 & d & 0 \\ e & 0 & 0 & f \end{bmatrix}$$

DIA Format:

values $= [a, c, d, f, *, *, *, e, b, *, *, *]$

offsets $= [0, -3, 3]$

DiaQ Format:

$(dIndex = -3) \quad [e]$

$(dIndex = 0) \quad [a, c, d, f]$

$(dIndex = 3) \quad [b]$

3.3 Matrix Multiplication in DiaQ

The bulk of calculating a matrix product can be collapsed to appropriately multiplying pairs of diagonals. This collapses even further, as a matrix which is nonzero only on some diagonal is simply an operator which performs an element-wise shift to matrix elements followed by some element-wise multiplication to appropriately scale. An example of this operation over an entire matrix (a collection of diagonals) can be found in Figure 3.5.

Let

$$S = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}; \quad A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 3 & 2 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Then,

$$SA = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 3 & 2 & 1 \\ 1 & 2 & 2 & 2 & 1 \end{pmatrix}; \quad AS = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 2 & 2 & 2 & 1 & 0 \\ 2 & 3 & 2 & 1 & 0 \\ 2 & 2 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Figure 3.5: Shift operations encoded in matrices. Notice That the order of the matrix multiplication corresponds to vertical or horizontal shift. Depending on the order.

It is worth recognizing that even dense matrices can still be represented using this approach. Analogous to the thinking that a matrix is simply a collection of column vectors, we will assume that the matrices we work with are collections of diagonals instead. However, we choose this diagonal representation as it fits our assumption that many quantum operations can be encoded uses a minimal set of diagonals.

3.4 Format Conversion from Dense to DiaQ

Most quantum simulators use dense matrices for their needs and, hence, it is important to have an efficient conversion algorithm from dense to DiaQ. We can convert a dense matrix to DiaQ by iterating over the matrix and storing non-zero elements in the hashmap of diagonal indices to diagonal array of elements. The time complexity of this conversion is $O(n^2)$, where n is the number of rows or columns in the matrix.

3.5 DiaQ for Tensor Networks

DiaQ can be extended to represent tensors as well. This is done by storing the tensor shape along with the underlying matrix shape. This technique is very popular and sometimes referred to as matricised tensor format.

While storing the underlying matrix, we ensure that the x dimension and the y dimension are mutually maximized. We found that this approach produces fewer diagonals and, hence, requires less storage.

Accessing an element in DiaQ-Tensor is done by first converting the tensor indices to matrix indices and then accessing the element in the underlying matrix. This is an $O(1)$ operation. Similarly, updating an element in DiaQ-Tensor is also an $O(1)$ operation.

We utilize the QUIMB framework (Gray 2018) to perform arbitrary tensor contractions using DiaQ as a backend option. There exist other backend options like numpy (Harris et al. 2020), tensorflow (Abadi et al. 2015) etc. It utilizes `opt_einsum` (a. Smith and Gray 2018), an optimal path finding python libary for a given einsum expression.

The sequence of steps to interact with QUIMB is as follows:

1. When a tensor network is created in QUIMB, it is passed to `opt_einsum` for contraction along with a choice of backend to use.
2. Based on the backend choice, `opt_einsum` calls the corresponding backend's functions to perform the contraction.
3. Format conversion from `numpy` to the backend's format (DiaQ in our case). This is done for all the tensors in the graph.
4. `opt_einsum` creates the optimal path for contraction. This path is a sequence of pair-wise contractions that need to be performed to get the final result.
5. The chosen backend's tensor product function is called for each pair-wise contraction.
6. Format conversion from the backend's format to `numpy`. This is done for the final result tensor.

Hence, the two required functions that `libdiaq` has to expose are: **format conversion** and **tensor dot**. The transpose and reshape kernels are also required at times for the tensor dot operation.

We perform tensor dot operations using our spGEMM, reshape, and transpose algorithms. This technique is commonly used for matricized tensors and is sometimes referred to as “GEMM via tensor-transpose". Since we store tensors as matrices with the x and y dimensions mutually maximized, we can perform the tensor dot operation by first reshaping the tensors and then performing the matrix multiplication.

3.6 PEPS topology for TNQVM

Quantum Circuit simulation broadly involves 2 steps:

1. Circuit to tensor network conversion.
2. Manipulating/contracting the tensor network for various observables and more.

We reuse the truncated SVD functionality that already exists within the TNQVM library for the PEPS visitor as well.

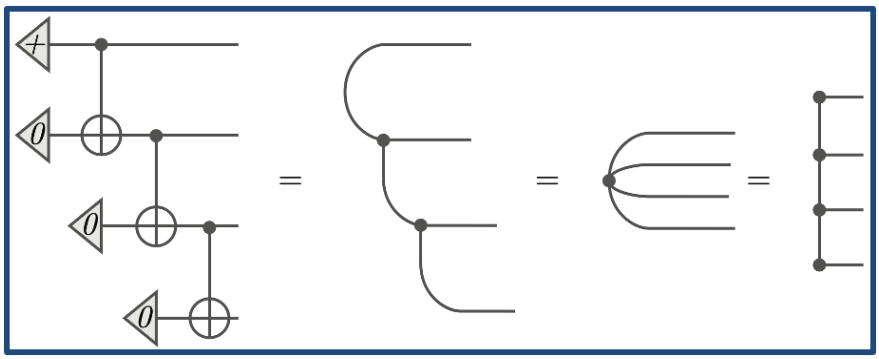


Figure 3.6: Conversion of a Quantum Circuit to MPS

For the MPS topology, the conversion from a quantum circuit is straightforward with step-wise contractions to form a single tensor, and then decomposition of this possibly high-dimensional tensor into a product of smaller tensors (a Matrix Product State). This is shown in Figure 3.6.

But visualizing such a conversion for PEPS is not as straighforward, as the PEPS tensor network is a 2D grid of tensors, and the conversion involves merging two qubit tensors into a single tensor, followed by a decomposition into two qubit tensors. This is shown in Algorithm 1.

We first convert a given quantum circuit to a PEPS tensor network as described in Algorithm 1. TNQVM already provides implementations of multiple “visitors”, e.g., MPS, MPO, and TTN. We leverage the MPS tensor network to implement the snake (zig-zag) boundary contraction algorithm (shown in Figure 3.7) inspired by Guo et al. (2019). Our contraction algorithm first forms a boundary sequence starting from top-left corner and follows a zig-zag path to cover all the nodes in the PEPS network. We

Algorithm 1 Conversion of a Quantum Circuit to PEPS

```
1: pepsNetwork ← exatn::builder("PEPS").initialize(X,Y);
2: for gate_i in circuit.gates() do
3:     gateTensor ← tnqvm::getGateTensor(gate_i);
4:     if gate_i.rank() == 2 then
5:         mergedTensor ← pepsNetwork.getQubitTensor(i) ⊗ gateTensor;
6:         pepsNetwork.setQubitTensor(a, mergedTensor);
7:     end if
8:     if gate_i.rank() == 4 then
9:         mergedTensor ← pepsNetwork.getQubitTensor(a) ⊗ pepsNetwork.getQubitTensor(b);
10:        contractedTensor ← mergedTensor ⊗ gateTensor;
11:        U, V ← TruncatedSVD(contractedTensor);
12:        pepsNetwork.setQubitTensor(a, U);
13:        pepsNetwork.setQubitTensor(b, V);
14:    end if
15: end for
```

then construct an MPS tensor network using this node traversal sequence. TNQVM's existing MPS-visitor is re-used for observables or density matrices.

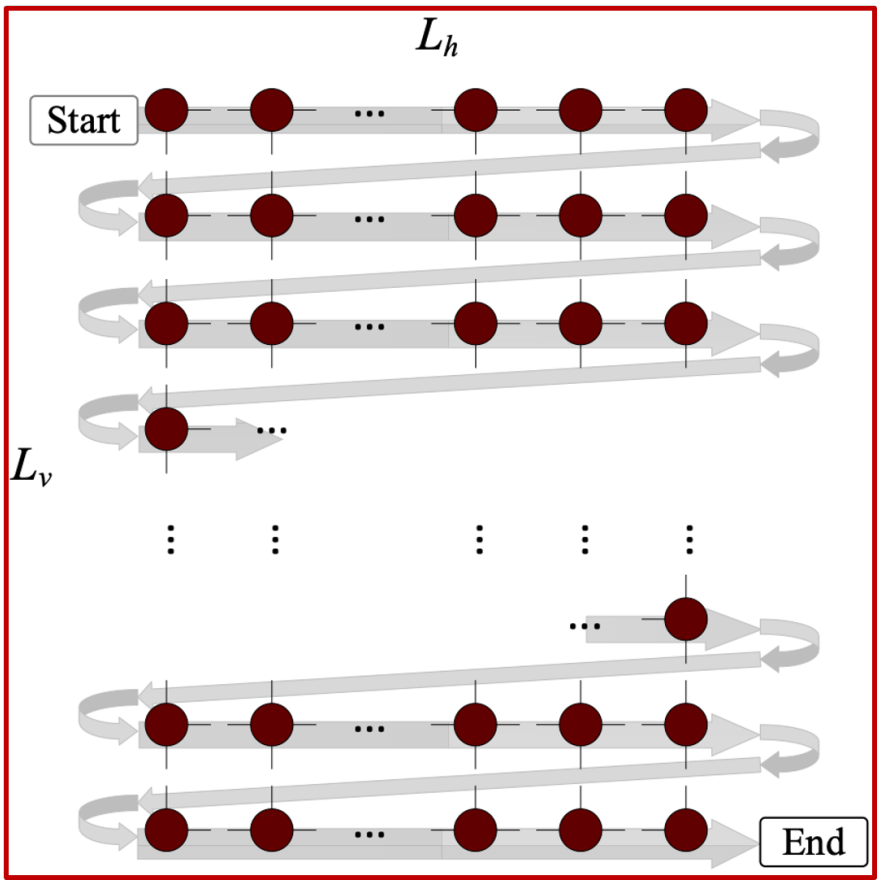


Figure 3.7: Snake boundary (Guo et al. 2019) contraction algorithm for PEPS

CHAPTER

4

IMPLEMENTATION

In this section, we will discuss the implementation of key algorithms and parallelization approaches used in the numerical library for performance optimization. Further, we will discuss the implementation of a home-grown benchmarking tool for the library. We will also discuss the integration of the PEPS topology into TNQVM for quantum circuit simulation.

4.1 libdiaq: A C++ library for DiaQ

4.1.1 OMP Parallelization

We leverage OpenMP (OMP) to parallelize the spGEMM Algorithm 2. We parallelize the outer loop of the algorithm, which iterates over the diagonals of the input matrices. We use the #pragma omp parallel for directive to parallelize the outer loop. This directive instructs the compiler to parallelize the loop by distributing the iterations among the available threads. We set the number of threads to the number of available cores on the system.

4.1.2 SIMD Vectorization

Quantum simulations are often represented as complex numbers. However, the choice of SIMD instructions depends on the configuration setting. If SSE4.2 is enabled, we split an array of complex numbers into two arrays- a real number array and a complex number array. This approach allows us to use two SSE4.2 '__m128' vectors to perform operations on the arrays.

To perform complex vector times complex vector, we use two real vectors and two imaginary vectors. We multiply each element of the real vector with the corresponding element of the other real vector and

Algorithm 2 Matrix Multiplication ($A \times B$)

```
1:  if A.columns ≠ B.rows then
2:      throw InvalidArgument("not multiplyable")
3:  end if
4:  result ← InitializeResultMatrix()
5:  mapA ← A.getDiagonalMap()
6:  mapB ← B.getDiagonalMap()
7:  #pragma omp parallel for
8:  for diagA ∈ mapA do
9:      for diagB ∈ mapB do
10:         diagNew ← MultiplyDiagonals(diagA, diagB)          ▷ SIMD vectorized
11:         if diagNew.isValid() then
12:             newIndex ← diagA.index + diagB.index
13:             result[newIndex] ← diagNew
14:         end if
15:     end for
16: end for
17: return result
```

each element of the imaginary vector with the corresponding element of the other imaginary vector. We then subtract the product of the imaginary vectors from the product of the real vectors to get the real part of the result. Similarly, we add the product of the real and imaginary vectors and multiply it by -1 to get the imaginary part of the result.

After performing the complex vector operation, we stitch the resulting real and imaginary vectors back into a complex vector again. We take care of the left-over values (values that are not pure multiples of 4) by using partial loads.

If AVX2 is chosen pre-compile time using `src/config.hpp`, we use the AVX2 '__m256' vectors instead of SSE4.2 '__m128' vectors to perform the same operations on the arrays. The AVX2 instructions allow us to process 8 complex numbers at a time, resulting in improved performance.

Later, we moved away from using complex `std::vector` and started using regular arrays to improve performance. We observed a substantial performance improvement when transitioning from using inbuilt C++ vectors to regular arrays in our library optimization efforts.

4.1.3 GPU Acceleration

We have implemented a CUDA kernel to convert dense matrices to DIAQ (another significant bottleneck identified). As outlined in Figure 4.1, we use the GPU to convert row index and column index in the dense version to diagonal index and the position in that diagonal.

Since spGEMM is memory bound, formulating a GPU kernel for spGEMM is not straightforward. We are currently exploring the possibility of using the GPU for the spGEMM operation.

```
__global__ void convertToDQ(
    long size, long matrixSizeX, long matrixSizeY,
    long* devDiagIdx, long* devDiagPos
) {
    // Calculate the global index of the current thread
    long idx = blockIdx.x * (blockDim.x * blockDim.y * blockDim.z)
                + threadIdx.z * (blockDim.x * blockDim.y)
                + threadIdx.y * blockDim.x + threadIdx.x;

    // Check if the current index is within the valid range
    if (idx < size) {
        // Calculate the diagonal index for the current index
        long dIdx = ((idx - matrixSizeY * (idx / matrixSizeY)) %
           matrixSizeY)
                    - (idx / matrixSizeY);

        long dPos;
        // Calculate the diagonal position based on the diagonal
           index
        if (dIdx <= 0) {
            dPos = ((idx - matrixSizeY * (idx / matrixSizeY)) %
               matrixSizeY);
        } else {
            dPos = ((idx - matrixSizeY * (idx / matrixSizeY)) %
               matrixSizeY)
                    - dIdx;
        }

        // Store the calculated diagonal index and position in the
           corresponding arrays
        devDiagIdx[idx] = dIdx;
        devDiagPos[idx] = dPos;
    }
}
```

Figure 4.1: CUDA Kernel: convertToDQ

4.2 bench: Benchmarking tool for DiaQ

We first export the intermediate matrices generated in qiskit (Qiskit contributors 2023) as `.npz` files. We then import these files (using cnpy (Reininghaus 2023), a third party library) into our new data structure and then perform chain matrix multiplication using our kernels. This mimics unitary simulation for quantum circuits. `bench` is a tool that we developed to do this. It can easily be compiled using CMake scripts we provide.

4.3 QUIMB with DiaQ

QUIMB (Gray 2018) interacts with `opt_einsum` (a. Smith and Gray 2018) for its numerical contraction requirements. Since `opt_einsum` is written in a way that new backends can be added, by implementing their expected interface (a set of function declarations), we were able to integrate QUIMB with DiaQ.

We also integrate the `sparse` (PyData 2023) numerical library with `opt_einsum` by linking the interface that `opt_einsum` expects with the `sparse` library's function calls. This allows us to compare the performance of DiaQ with the `sparse` library for tensor network simulations.

Figure 4.2 shows the integration of QUIMB with DiaQ. We first create a GHZ state using the supermarq library. We then remove the measurements from the circuit and convert the circuit to tensors using the `cirq.contrib.quimb` library. We then visualize the tensor network. We contract the tensor network using the numpy backend and the DiaQ backend and finally print them out. The results represent the final circuit unitary matrix.

4.4 TNQVM-PEPS Visitor

We implement the PEPS-builder class in ExaTN and the PEPS-visitor class in TNQVM, and integrate them to perform quantum circuit simulation using the PEPS backend.

PEPS-builder

This component implements the quantum circuit to PEPS conversion described in Algorithm 1. It initializes the PEPS tensor network with the given row and column sizes, and then applies the quantum gates to the PEPS tensor network. The PEPS tensor network is then ready for the TNQVM-PEPS visitor to perform the quantum circuit simulation.

PEPS-visitor

This component first calls the PEPS-builder to initialize the PEPS tensor network. It then contracts this network using the snake boundary contraction described in Figure 3.7. Each pair-wise contraction is approximate because of the truncated SVD step. Since the contraction path is fixed for a given algorithm (snake boundary), we re-use the MPS-Visitor to perform the pair-wise contractions. In this manner,

```
import numpy as np
import diaq as dq
import supermarq as sm
import cirq.contrib.quimb as ccq
import quimb
import quimb.tensor as qtn

# Create GHZ state
ghz = sm.ghz.GHZ(9)
ghz_circuit = ghz.circuit()
ghz_circuit = cirq.drop_empty_moments(ghz_circuit)

# Remove measurements
all_ops = list(ghz_circuit.all_operations())
all_qubits = ghz_circuit.all_qubits()
ghz_circuit.batch_remove([(len(all_ops) - 1, all_ops[-1])])

# Convert circuit to tensors
tensors, qubit_frontier, fix = ccq.circuit_to_tensors(ghz_circuit,
   all_qubits, initial_state=None)
tn = qtn.TensorNetwork(tensors)

# Visualize the tensor network
tn.graph(color=['Q0', 'Q1', 'Q2'])

# Contract the tensor network using numpy backend
contracted_tn_numpy = tn.contract(backend="numpy")
print(contracted_tn_numpy)

# Contract the tensor network using DiaQ backend
contracted_tn_diaq = tn.contract(backend="diaq")
print(contracted_tn_diaq)
```

Figure 4.2: QUIMB with DiaQ Example

pre-implemented methods are leveraged to extend TNQVM to support PEPS-based quantum circuit simulation.

PEPS can be used with TNQVM just like other visitors, but with extra parameters (row and column sizes) for the 2D tensor network initialization. We make an attempt to choose them automatically if these parameters are not provided by the user. An example of GHZ(9) with TNQVM is shown in Figure 4.3.

```
xacc::Initialize();
auto xasmCompiler = xacc::getCompiler("xasm");
auto ir = xasmCompiler->compile(R"(__qpu__
void test2(qbit q) {
    H(q[0]);
    for(int i = 0; i < 8; i++) {
        CNOT(q[i], q[i+1]);
    }
    for (int i = 0; i < 9; i++) {
        Measure(q[i]);
    }}
)");
std::vector<int> bitstring(9, 0);
auto program = ir->getComposite("test2");
auto accelerator = xacc::getAccelerator(
"tnqvm", {
    std::make_pair(
        "tnqvm-visitor", "exatn-peps"
    ),
    std::make_pair("shots", 10),
    std::make_pair("lx", 3),
    std::make_pair("ly", 3)
});
auto qreg = xacc::qalloc(9);
accelerator->execute(qreg, program);
qreg->print();
```

Figure 4.3: TNQVM-PEPS Plugin Usage Example (GHZ-9)

CHAPTER

5

EVALUATION

In this section, we first showcase the performance of DiaQ with unitary simulations using chain matrix multiplication, tensor network contraction using QUIMB. We showcase multiple HPC optimizations that are implemented in the `libdiaq` library.

We also showcase the integration of PEPS topology with TNQVM.

5.1 Experimental Setup

DiaQ Tests

We first perform an empirical FLOP analysis proving the performance gains that DiaQ provides over the naive approach. We then perform a performance analysis of the chain matrix multiplication using the `libdiaq` library. We utilized Qiskit's unitary simulator to gather intermediate unitary matrices, which were subsequently subjected to chain multiplications.

Since collecting intermediate matrices for wider quantum circuits is exponentially complex in terms of memory requirements, we scale up the relatively smaller intermediates to generate similar data for larger number of qubits (>=16). For instance, we stitch four 8-qubit matrices to form a 16-qubit unitary.

These tests have been run on a single node of Intel's Cascade (2.50 GHz) processor. This node has 192GB of DDR4 RAM and 32 logical cores. It has 64KB of L1 cache, 1MB of L2 cache, and ~10MB of L3 cache. This is part of the ARC cluster (Cluster 2023) in the CSC department at NC State University.

The experiments are summarized in Table 5.1.

We undertook experiments assessing AVX2's performance compared to SSE on SupermarQ's benchmark circuits. Initial observations revealed no significant performance gains. To delve deeper into these

Table 5.1: Summary of DiaQ experiments performed.

#Qubits	Operations	Numerical Libraries compared	#Iterations
4,6,8,10,12,16	Tensor-Network Simulation, Chain Matrix Multiplication	Numpy, Jax, Tensorflow, Scipy (COO Format), Our Library	50
(Generated data) 16,18,20,22,24,26	Chain Matrix Multiplication, Reshape Operation, Transpose Operation	Dense O(n^3) Version, Our Library	50

results, we employed likwid (Treibig et al. 2010) to measure retired instructions and Single Precision FLOP/s across all circuits. This analysis was conducted while employing byte-aligned memory allocation to optimize AVX2 utilization.

SIMD experiments were conducted on a single node equipped with an AMD EPYC 7302P 16-Core Processor operating at 3.00 GHz. The CPU specifications comprise 32KB of L1 data cache, 32KB of L1 instruction cache, 512KB of L2 cache, and 16MB of L3 cache per CPU. Each CPU encompasses 16 cores and supports 2 threads per core, organized into 4 NUMA nodes. This node is part of the ARC cluster (Cluster 2023) located in the CSC department at NC State University, boasting 192GB of DDR4 RAM and 32 logical cores.

Tests involving GPU acceleration were conducted on a single node equipped with an NVIDIA GeForce RTX 4060 Ti GPU. The experiments utilized GPU driver version 535.86.10 and CUDA version 12.2 for GPU-based computations.

TNQVM Tests

We perform sample tests on an Intel(R) Core(TM) i7-4820K at 3.70GHz with 8 cores, L1 data and instruction caches of 128 KiB each, an L2 cache of 1 MiB, an L3 cache of 10 MiB and 8GB of DRAM.

5.2 Results

Since the DiaQ format is designed specifically for the structured sparsity in unitary matrices, a significant amount of storage is saved, as expected. This is seen in Figure 5.1.

Our approach (DiaQ, diagonal arithmetic) will compute the final unitary with significantly fewer number of additions and multiplications. This is confirmed in Figure 5.2b. Especially in case of Supermarq's Hamiltonian circuit, which has a unitary representation of a perfect singular diagonal, we see from Figure 5.3b that the dense kernel can be improved by around 4 orders of magnitude.

We first compare the dense kernel with our sparse kernel for a sequence of matrix multiplications (unitary simulation) using the C++ library. We observe in Figure 5.5 that most of the time is spent in the conversion step from the dense format to our format. Hence, we employ OpenMP to improve the performance of this kernel.

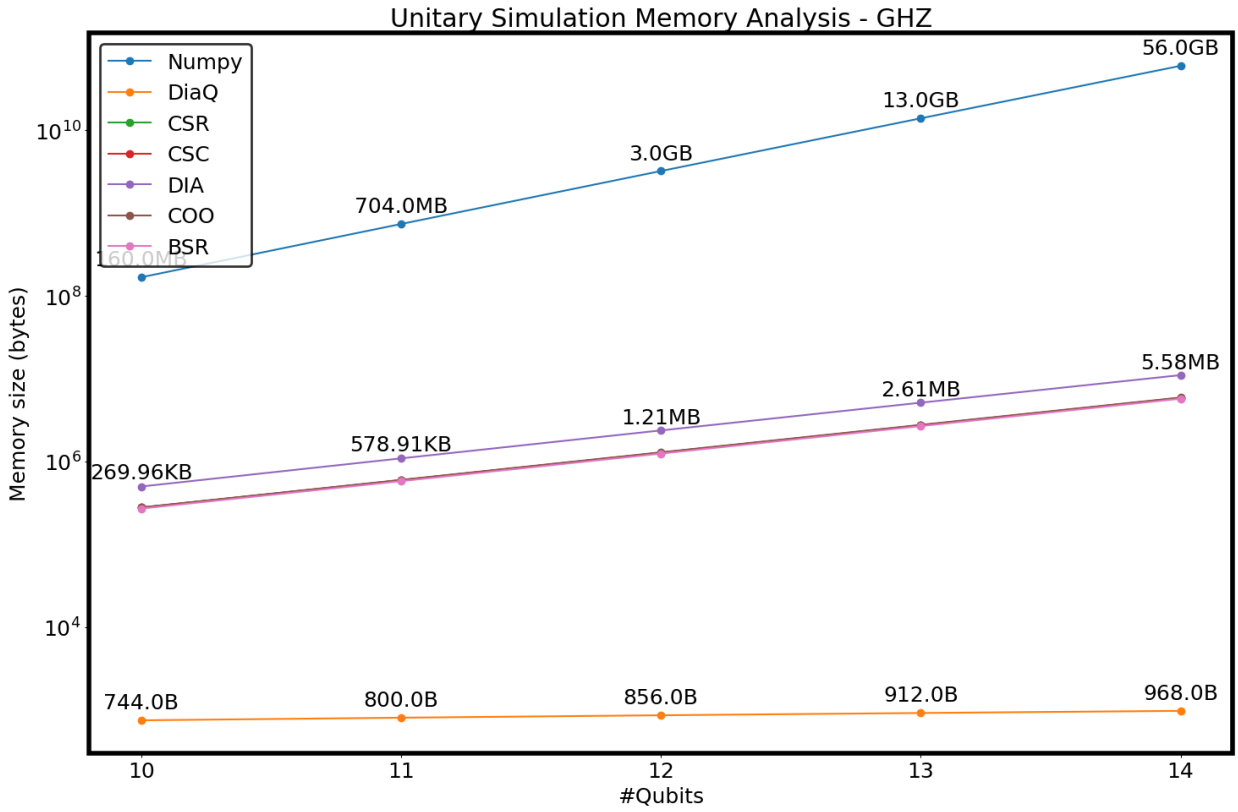
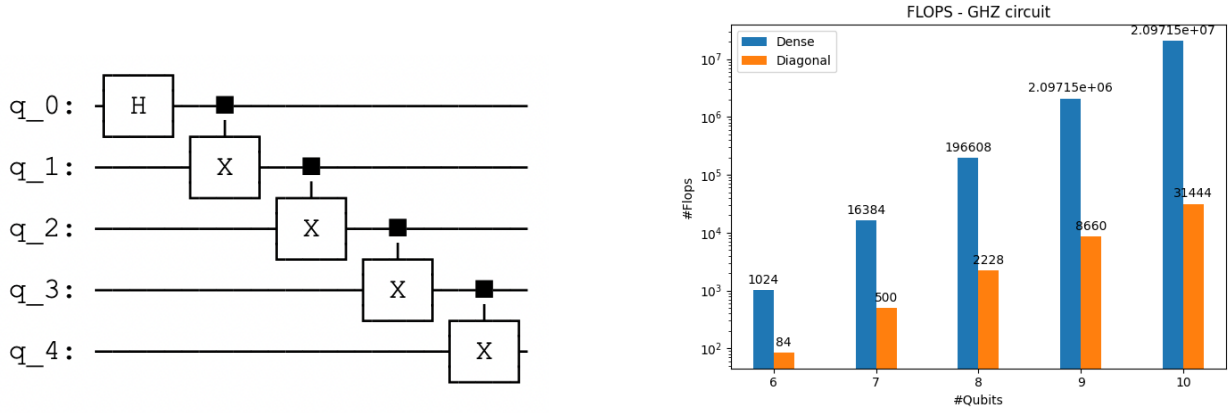


Figure 5.1: DIAQ Memory Savings.

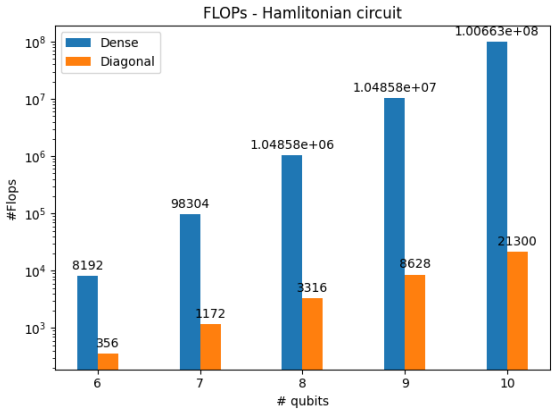


(a) GHZ circuit (4 qubit version)

(b) Comparison (Dense/Numpy vs Sparse Diagonal)

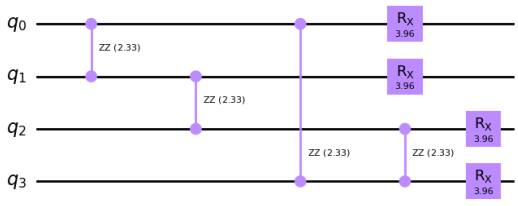
Figure 5.2: Supermarq's GHZ circuit unitary simulation for different #qubits

(a) Hamiltonian circuit (3 qubit version)

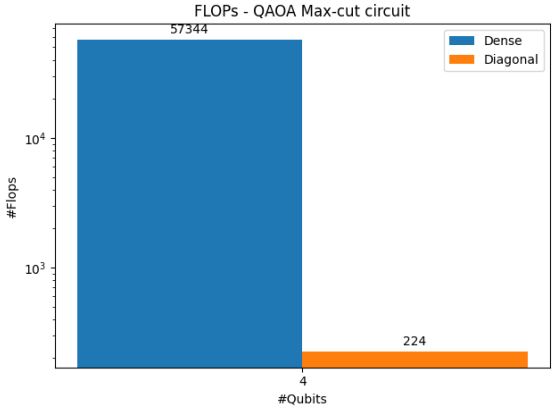


(b) Comparison (Dense/Numpy vs Sparse Diagonal)

Figure 5.3: Supermarq's Hamiltonian circuit unitary simulation for different #qubits



(a) Max cut circuit - without Hs



(b) Comparison (Dense/Numpy vs Sparse Diagonal)

Figure 5.4: QAOA Max-cut for a square

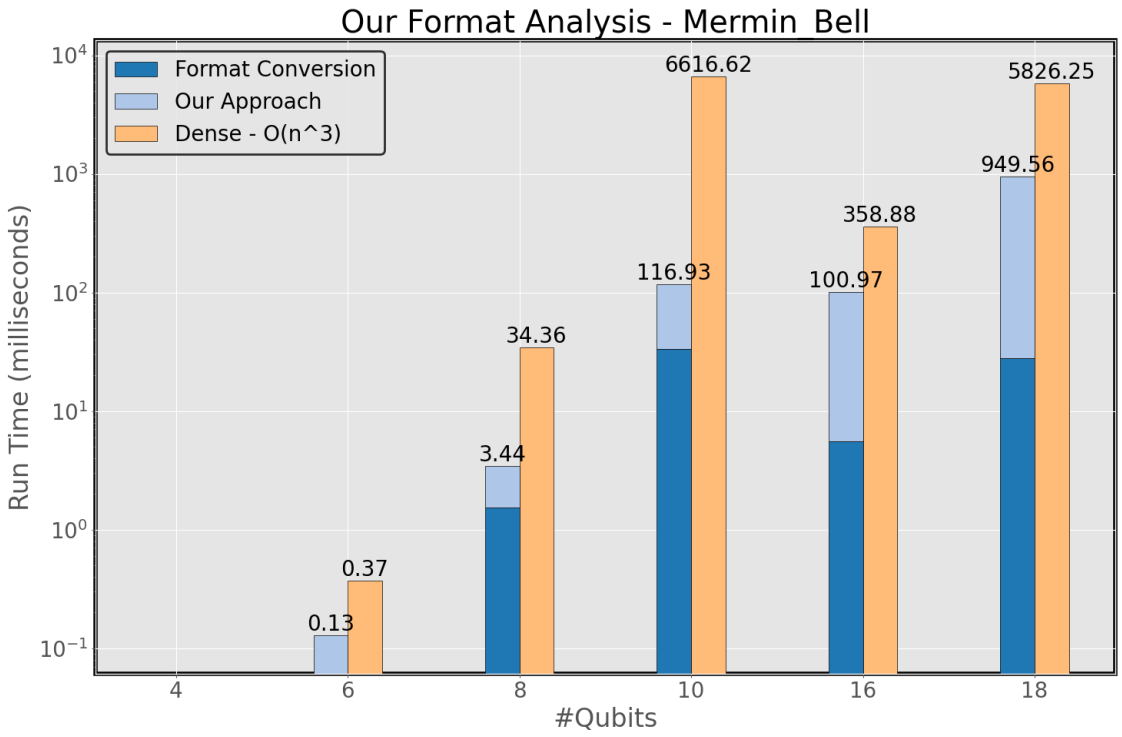


Figure 5.5: Dense Kernel vs Sparse Kernel for SupermarQ's mermin_bell benchmark. Note that 16 and 18 qubit simulations are based on generated data.

As shown in Figure 5.6, our optimized implementation using OpenMP with 16 threads in parallel outperforms all other implementations, with a 65% improvement in format conversion for a unitary matrix with dimensions are $2^{26} \times 2^{26}$ from the 26-qubit GHZ circuit.

We further analyze the performance of modern numerical libraries for tensor network-based simulations using opt_einsum, a package used internally by Quimb. Our implementation provides an interface to link to different libraries, and we compare its performance with other libraries in Figure 5.7. We see here that while numpy works better for smaller number of qubits, it does not scale like sparse numerical formats do for larger qubits and runs into memory issues.

Also, in Figure 5.7, we observe that Scipy's COO format is faster than our format even for the 20 qubit HAM circuit. We have identified that the transpose operation (performed within the tensordot kernel) is the bottleneck in our library. We plan to optimize this operation in the future. But, while our library is slower than COO, we see that we do not run into errors like COO does for larger (≥ 20) qubit circuits.

In Figure 5.8, we evaluate the impact of SIMD on performance and observe that the runtime is in the order of microseconds. While we cannot definitively conclude that SIMD always provides an advantage, the results suggest that it can benefit immensely massive matrices, such as those in 50-qubit systems. We make this argument because the SIMD version outperformed the sequential version for 26 and 28 qubit systems, but only by ~13.2%. Therefore, the results indicate that SIMD may provide substantial benefits for larger systems, and its effectiveness may vary depending on the size of the system.

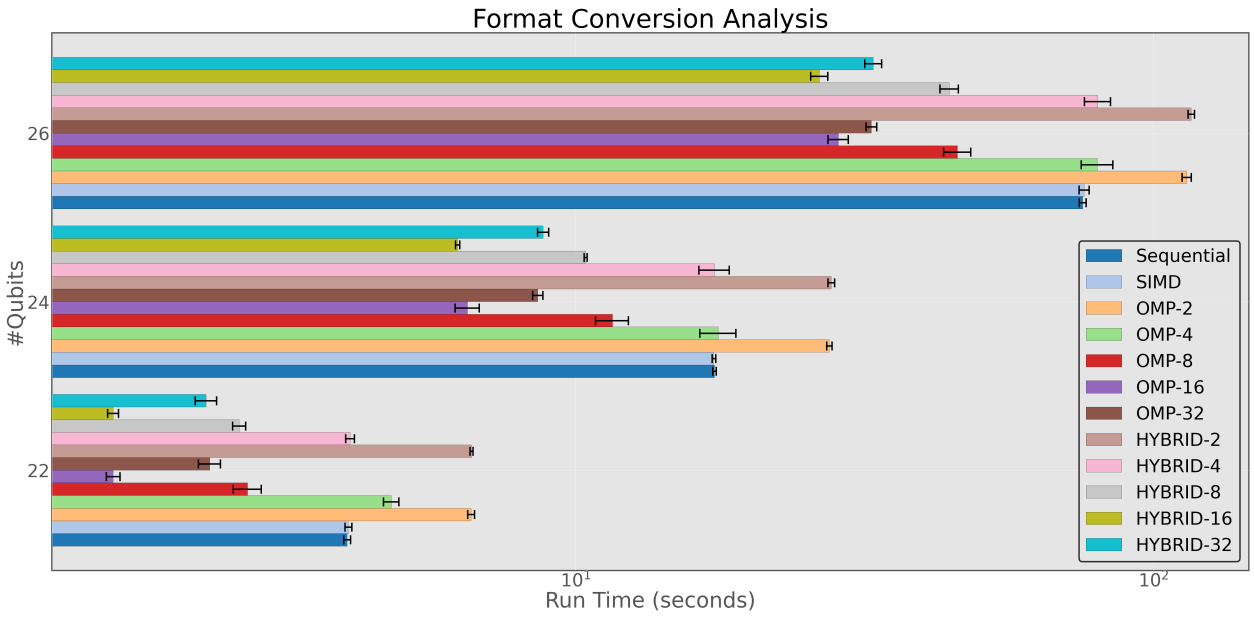


Figure 5.6: Comparison of various versions that run the format conversion operation.

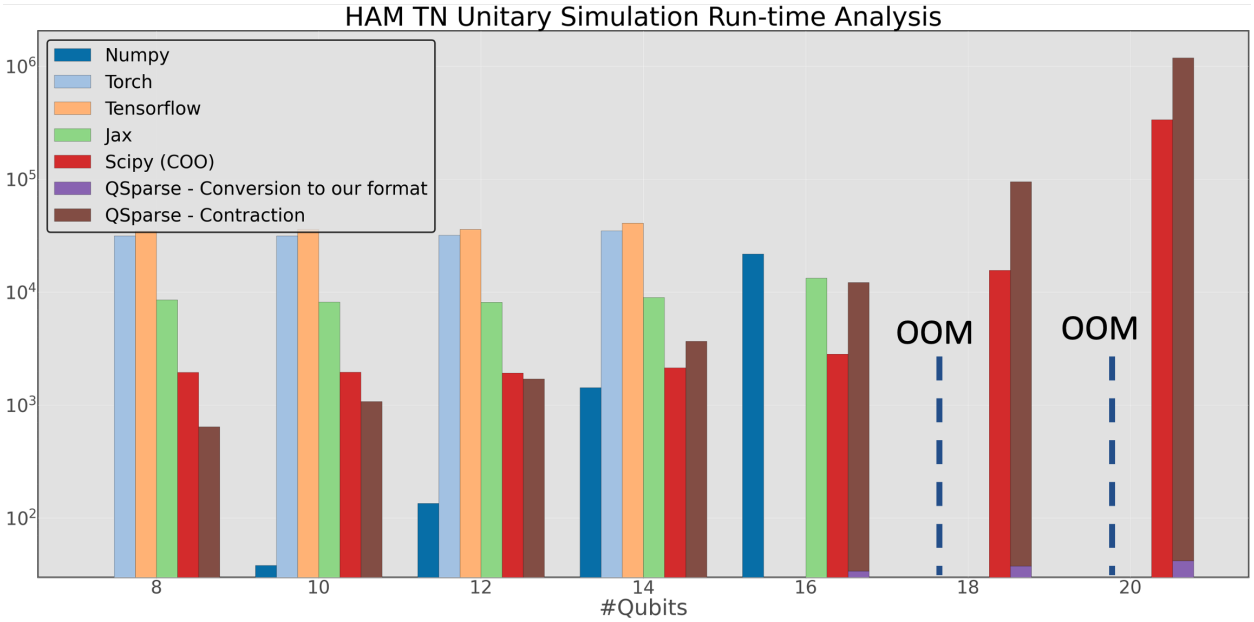


Figure 5.7: Performance comparison of modern numerical libraries for tensor network-based simulations using opt_einsum. Note: QSparse here refers to our library, `libdiaq`.

Our approach's novelty lies in its potential to simulate circuits with more than 20 qubits significantly faster using our format. This capability sets us apart from other packages such as "'sparse" (which throws exceptions "too many dimensions passed to ravel_multi_index") and "numpy" (which throws exception "numpy.core._exceptions.MemoryError" due to an inability to store such large matrices in memory). In the future, we expect that DiaQ will allow us to study larger quantum systems and gain insights into their behavior, making our approach a valuable tool for quantum researchers and practitioners.

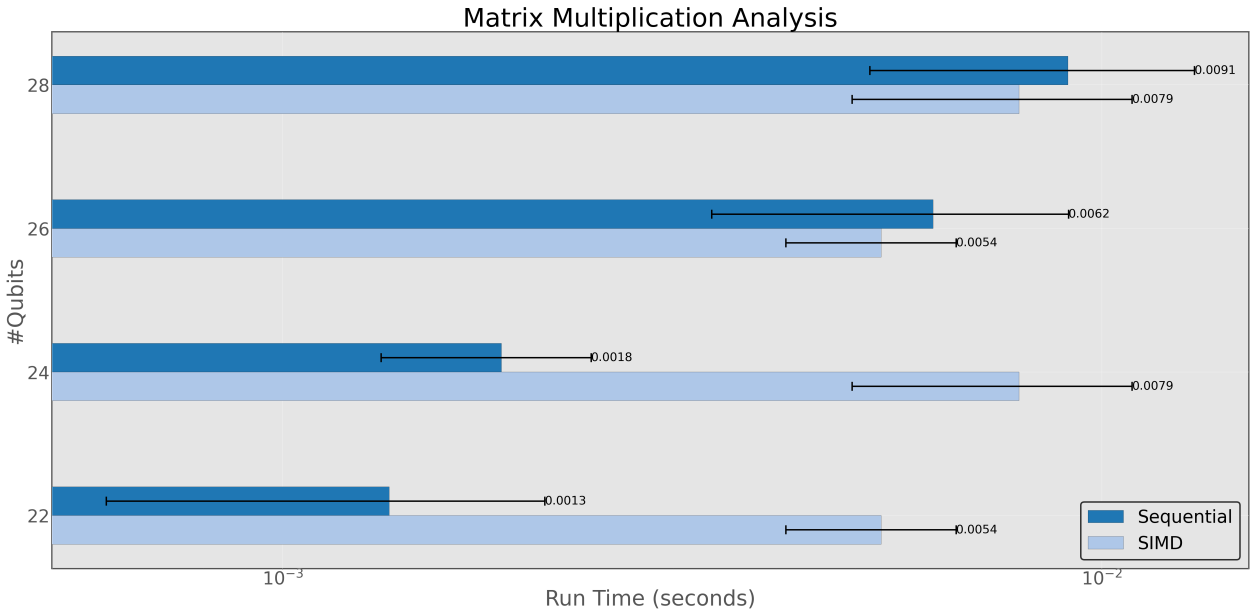


Figure 5.8: Multiplication performance analysis of SIMD on large GHZ intermediate matrices. As seen, SIMD gives us benefits only from 26 qubits onwards.

The results depicted in Figure 5.11 indicate that AVX2 demonstrates enhanced performance, particularly for larger lengths and with proper byte-alignment assistance during memory allocation. We conducted an analysis of the instruction count across various alignment strategies during a benchmark run of SupermarQ's GHZ for 8 qubits. Surprisingly, we observed that SSE exhibited a lower count of retired instructions, as depicted in Figure 5.9. In response, we offer detailed considerations regarding the potential reasons behind this discrepancy:

- Despite all code being compiled at optimization level 03, it appears that optimization strategies might not solely account for the variance in retired instructions between SSE and other alignment strategies.
- Differences in the instruction sets employed by SSE and alternative alignment strategies could contribute to the observed differences in retired instructions. A more in-depth analysis is imperative to elucidate these differences.
- Variances in memory access patterns influenced by SSE and alternative alignment strategies may

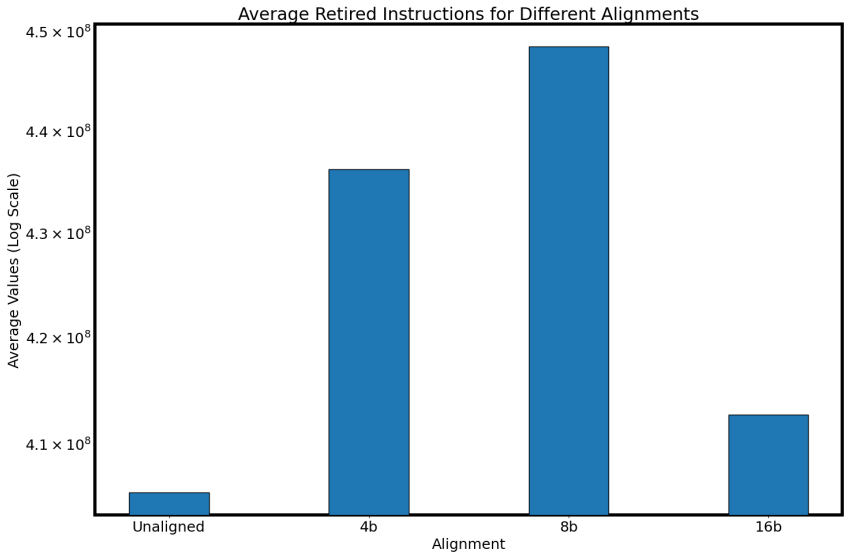


Figure 5.9: Number of Instruction comparison for various alignment sizes.

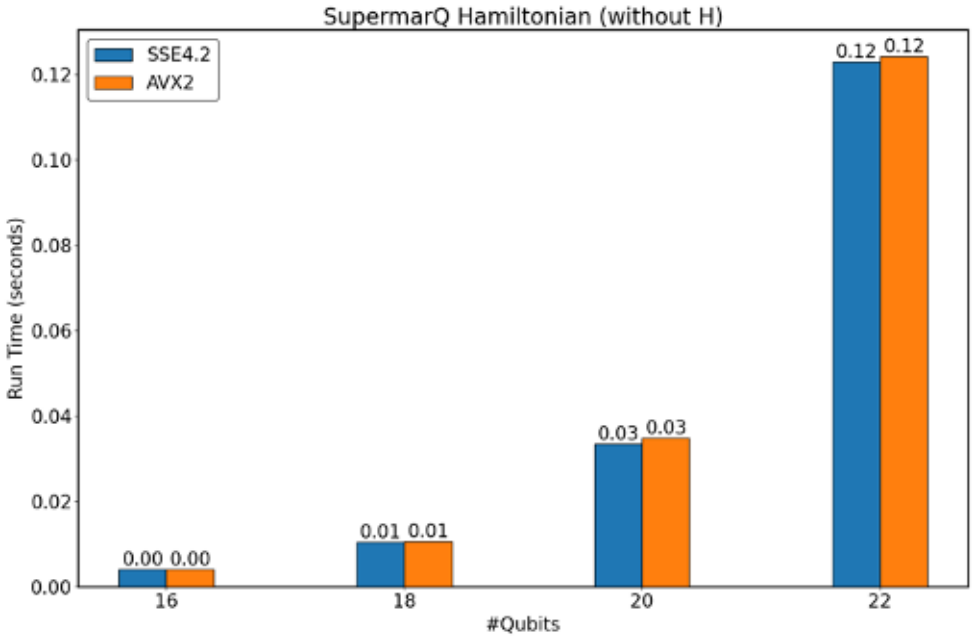


Figure 5.10: SSE vs AVX (HAM chain multiplication – spGEMM)

potentially contribute to the differences in retired instruction counts. Further investigation is essential to substantiate this hypothesis.

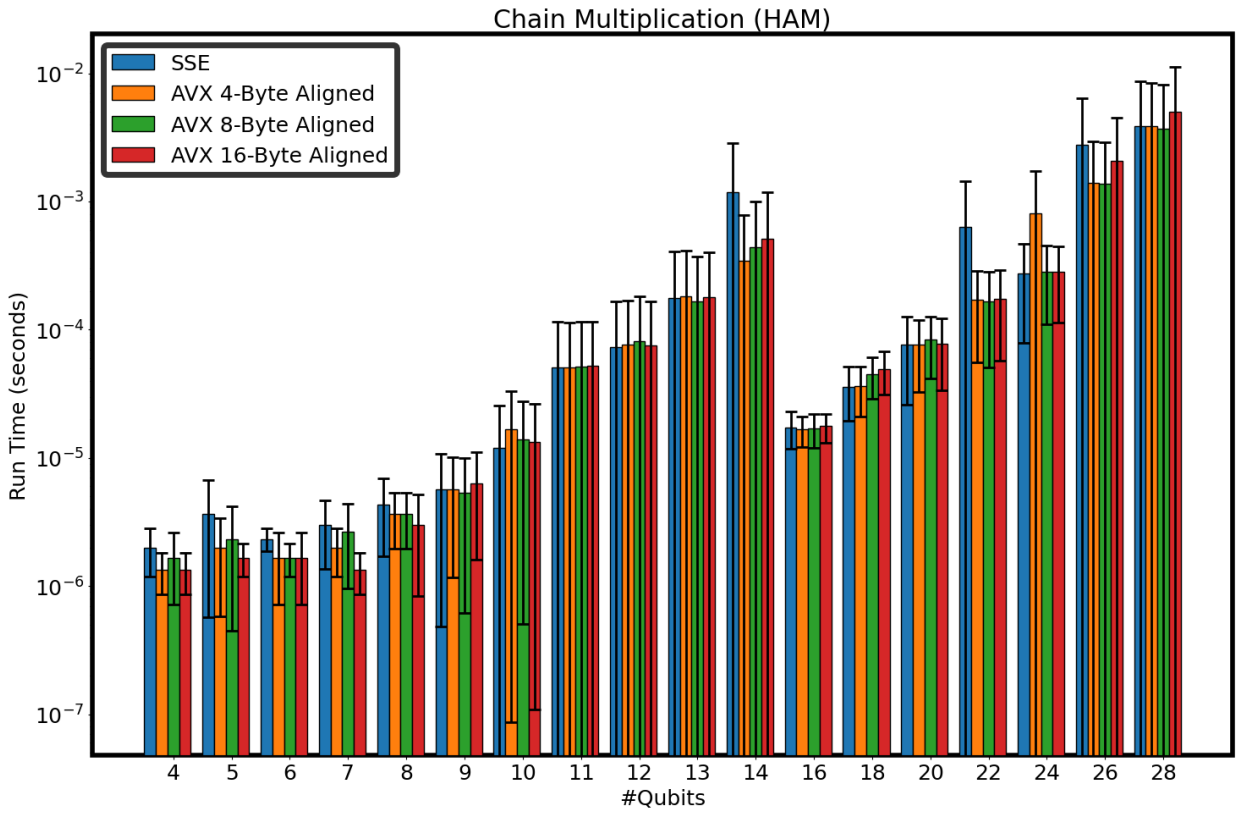


Figure 5.11: Chain spGEMM – Hamiltonian quantum circuit simulation

Performance-wise, SSE4.2 appears to outperform AVX2, achieving better average FLOP/s as observed in Figure 5.12. However, we speculate that AVX2 should excel over SSE4.2 for Double Precision since AVX2 (8-vectorization) inherently supports double precision, while SSE4.2 might face challenges with 4-vectorization.

Regarding the utilization of CUDA for format conversion, Figure 5.13 displays no significant improvement over the CPU version. This may be attributed to the lack of density for the conversion. In the future, we plan to implement the DASP (Lu and Liu 2023) idea and enforce bandwidth improvements for format conversion.

While the outcomes with CUDA may not directly indicate positive results, we believe that the tooling, scripting, and linking around the relatively new library DIAQ are now solid. This allows us to seamlessly implement and test new CUDA kernels.

Figure 5.14 demonstrates the functionality of the newly added PEPS plugin to TNQVM by comparing the results using a GHZ circuit while maintaining competitive runtime performance.

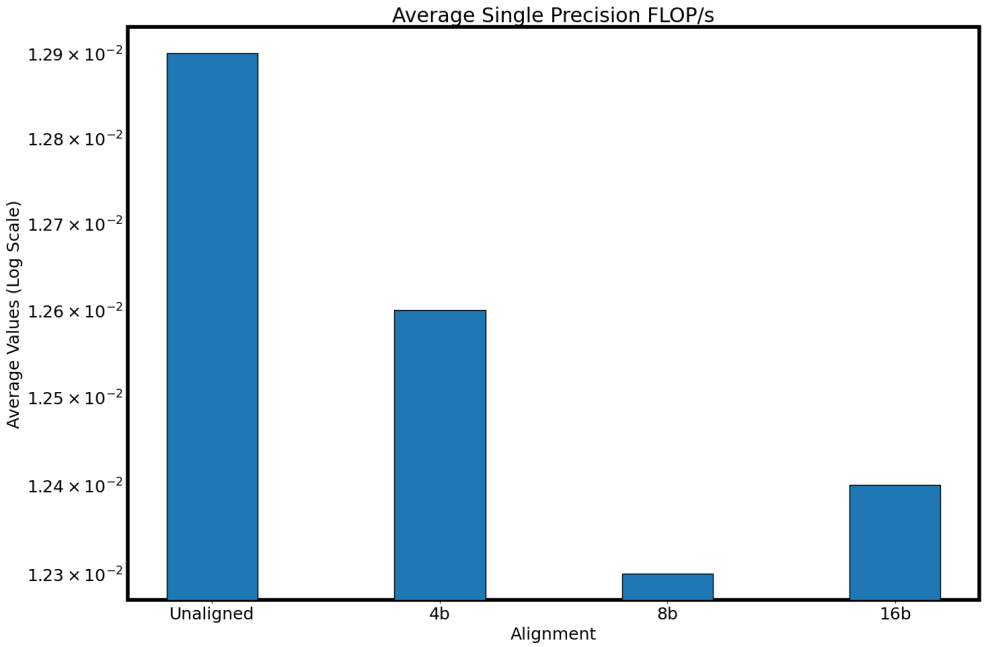


Figure 5.12: Single Precision FLOP/s for various alignment sizes.

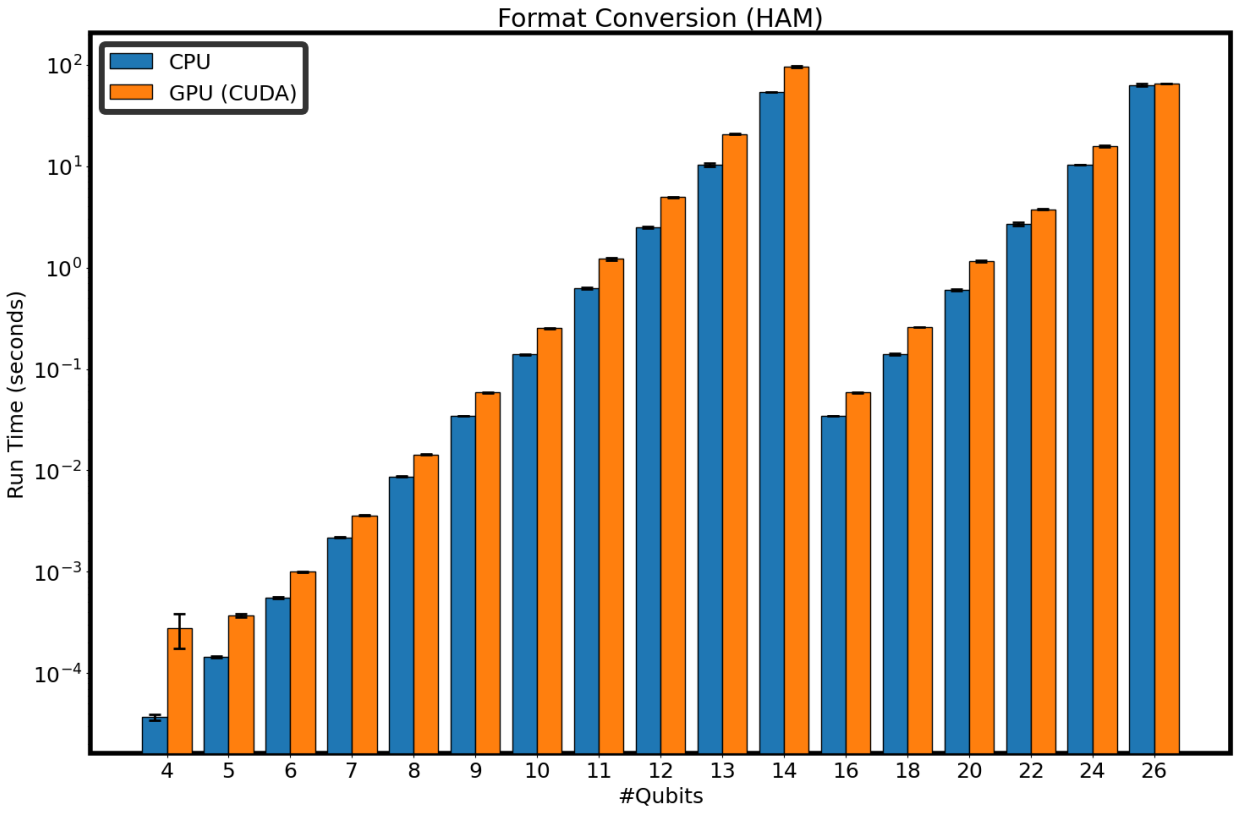


Figure 5.13: Format Conversion (HAM) – CPU vs GPU

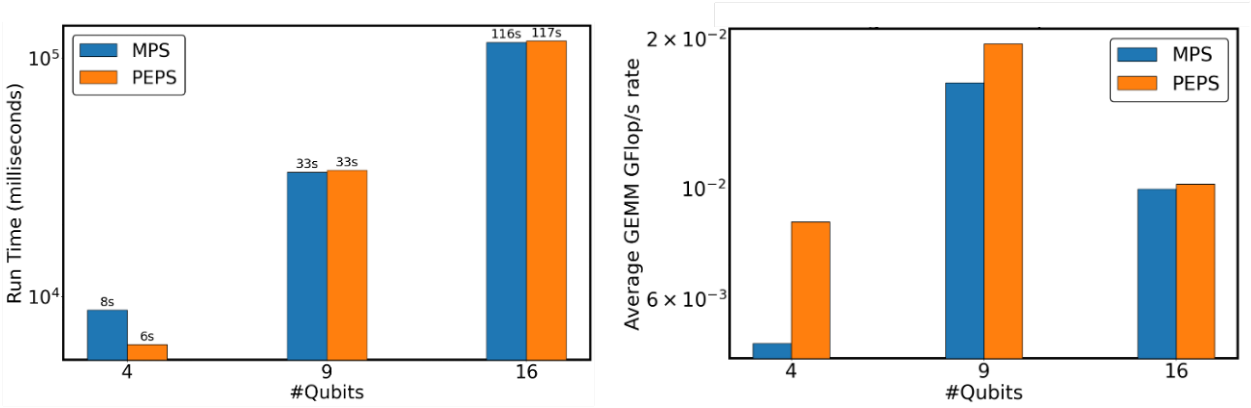


Figure 5.14: GHZ Tensor Network Simulation with TNQVM

5.3 Discussion

In our investigations of performance optimizations, we observed that SIMD operations over complex number arrays incurred certain performance overheads. Specifically, we noted the necessity of splitting complex number arrays into their real and imaginary components prior to vectorization. To mitigate this issue, we chose to store the real and imaginary components of complex numbers separately from the outset. This modification resulted in some improvement in performance.

In addition, we opted to shift away from using dynamic vectors (i.e., the std::vector data structure of C++11) in favor of regular arrays. Notably, this alteration directly enhanced the performance of the OpenMP version. For instance, the runtime for a 24 qubit GHZ circuit chain multiplication with std::vector was measured to be approximately 0.63745 seconds, while the runtime with regular arrays was measured to be approximately 0.00185 seconds, representing a reduction in time by almost 3 orders of magnitude.

We attribute this improvement to the internal dynamic behavior and implementation of `std::vector` and the lack of optimizations to accommodate this format.

CHAPTER

6

RELATED WORK

This section presents a comparative analysis of our high-performance numerical library for quantum computing with other existing quantum libraries and their supported numerical libraries.

Cyclops - CTF/Koala

Cyclops Tensor Framework (CTF) (Solomonik et al. 2014) is a high-performance numerical library for multidimensional arrays (tensors) in C++ and Python designed for parallel computing. The library utilizes MPI communication to coordinate tensor operations across all processes executing the program. Cyclops supports near sparsity using the COO (coordinate list) format, various symmetries, and user-defined element types, making it a versatile tool for scientific applications such as physics, chemistry, engineering, and machine learning. Its support for near sparsity allows it to efficiently handle large-scale data sets with complex data structures, while its support for symmetries can significantly reduce computational costs for certain operations.

Cyclops is interoperable with other scientific computing libraries such as ScaLAPACK and numpy, allowing for seamless integration into existing workflows and the ability to leverage its capabilities alongside other tools. These features make Cyclops an efficient and versatile option for researchers and scientists working with large-scale data sets and complex tensor operations.

Koala (Pang et al. 2020) is a quantum circuit simulation library that is based on projected entangled-pair states (PEPS) tensor networks. It is also developed by the same group. It currently supports numpy and ctf as tensor backends.

Like CTF, this work also involves the development of a novel high-performance numerical library for quantum computing. Our library is aimed at presenting sparsity through the utilization of innovative

data structures and specialized kernels. Since our work is similar to CTF, it can be integrated with Koala, which supports PEPS tensor networks also.

Qiskit - Numpy/Jax

Qiskit (Qiskit contributors 2023) is a high-performance quantum simulation library that provides users with a streamlined workflow consisting of four high-level steps: Build, Compile, Run, and Analyze. Users design a quantum circuit(s) to represent their problem, compile it for a specific quantum service, and then run the compiled circuits on a specified quantum service(s) that can be cloud-based or local. Finally, users analyze the results by computing summary statistics and visualizing the experiments. This workflow enables users to efficiently perform quantum simulations and computations using Qiskit. As per qiskit textbook, the only available backends for qiskit are numpy and jax. Both of which only have support for dense arrays.

Our work is primarily different from these numerical libraries with respect to the way they are stored, utilizing the seen sparsity pattern and hence reducing memory requirements.

In this work, we utilize Qiskit library to read and collect intermediate unitaries. These unitaries are then arranged in sequence, using chain matrix multiplications, to measure specific metrics that will be discussed in detail in the methodology section.

Quimb - opt_einsum

The Quimb (Gray 2018) library offers strong support for the simulation of quantum circuits, leveraging its capacity to depict and collapse arbitrary geometry tensor networks. However, its depiction typically falls outside the category of representing the complete wavefunction (like many other simulators) or a specific type of tensor network (such as an MPS or PEPS, as with some other simulators), which necessitates a unique approach to utilizing it that may require additional consideration. Google's Cirq also uses this package internally for its tensor-network based simulation needs. The opt_einsum package (a. Smith and Gray 2018) has an option to switch between various numerical libraries such as torch, jax, numpy, tensorflow and more, for the contraction step.

In this work, we leverage this package to compare the performance of tensor-network simulation of supermarQ benchmark circuits for these libraries against our new library.

CHAPTER

7

CONCLUSION

We observed that most unitary matrices involved in the process of quantum circuit simulation are “diagonal sparse", i.e., most non-zeros are present close to the main diagonal. We made use of this property to create a custom data structure (DiaQ sparse format) to store these matrices saving space, and developed custom Matrix product kernels to reduce computational complexity from $O(n^3)$ to $O(d*d*n)$ where d is a constant. We then performed multiple experiments (chain matrix multiplications) to simulate unitary evolution. Our approach beats the dense version by many orders of magnitude. We exposed the DiaQ format and its kernels as a numerical library called `libdiaq`, also providing python wrappers for use with other python packages.

We then focussed on 2D tensor network-based quantum circuit simulations. While employing parallelization approaches, including multi-threading via OpenMP and SIMD vectorization using SSE4.2 and AVX2, to improve the performance of the library, we also made progress towards GPU acceleration for certain bottleneck kernels like the format conversion step and sparse matrix times sparse matrix kernel. We also designed kernels for reshape and transpose operations.

We saw that AVX did not perform as well as SSE. Further investigation revealed that implementing byte-aligned memory better suited AVX2, enabling it to match SSE4.2 for all qubits and surpass it at the 14 qubit mark for the Hamiltonian benchmark quantum circuit.

We used quimb’s 2D tensor networks to evaluate the performance of the parallelized library in comparison to the sequential version. Our results showed that OpenMP significantly improves format conversion, while SIMD vectorization has little impact on overall run-time performance. Additionally, we discovered a substantial performance improvement when transitioning from using std::vector to regular arrays in our library optimization efforts.

We hence corroborate our hypothesis that a sparse numerical format specifically designed for

quantum simulations can be more efficient in performance and storage than dense general purpose numerical formats. This work provides a basis to investigate further in this direction (diagonal/anti-diagonal) research.

Furthermore, as a standalone project related to tensor networks, we developed a new “visitor” for TNQVM that enables PEPS simulation capturing 2D entanglement in Hamiltonian problems, among others. We leveraged pre-existing tensor network topologies, such as MPS and MPO, to facilitate the contraction of PEPS. It is experimental and requires extensive tests and benchmarking at scale before which comparisons can be drawn against other quantum simulation libraries. Other approximate PEPS contraction can should be explored, leading to efficient simulation of more complicated quantum circuits.

REFERENCES

a. Smith, D. G. and Gray, J. (2018). opt_einsum - a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Cluster, A. (2023). Arc cluster at north carolina state university.

Feynman, R. P. (1982). Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488.

Gray, J. (2018). quimb: A python package for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819.

Guo, C., Liu, Y., Xiong, M., et al. (2019). General-purpose quantum circuit simulator with projected entangled-pair states and the quantum supremacy frontier. *Physical Review Letters*, 123(19):190501.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

Low, G. H. and Chuang, I. L. (2019). Hamiltonian simulation by qubitization. *Quantum*, 3:163.

Lu, Y. and Liu, W. (2023). Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA. Association for Computing Machinery.

Pang, Y., Hao, T., Dugad, A., Zhou, Y., and Solomonik, E. (2020). Efficient 2d tensor network simulation of quantum systems. pages 1–14.

Perez-Garcia, D., Verstraete, F., Cirac, J. I., and Wolf, M. M. (2007). Peps as unique ground states of local hamiltonians.

Preskill, J. (2021). Quantum computing 40 years later. *arXiv e-prints*, page arXiv:2106.10522.

PyData (2023). Sparse: A library for sparse matrix formats in python.

Qiskit contributors (2023). Qiskit: An open-source framework for quantum computing.

Reininghaus, M. (2023). cnpy++: A c++17 library for reading and writing .npy/.npz files. *SoftwareX*, 21:101324.

Solomonik, E., Matthews, D., Hammond, J. R., Stanton, J. F., and Demmel, J. (2014). A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190.

tensornetworks (2023). Tensor networks.

Tomesh, T., Gokhale, P., Omole, V., Ravi, G. S., Smith, K. N., Viszlai, J., Wu, X.-C., Hardavellas, N., Martonosi, M. R., and Chong, F. T. (2022). Supermarq: A scalable quantum benchmark suite.

Treibig, J., Hager, G., and Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, I., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy: Open source scientific tools for python.