Quantum Circuit Simulations Leveraging NvLink and Qubit Swapping

Jin Bai

4th Year Project Report
Computer Science and Physics
School of Informatics
University of Edinburgh

2024

Abstract

In this dissertation, study on accelerating quantum circuit simulation with multi-GPU set up is carried out. As the advancement of quantum computing, quantum circuit simulation is becoming increasingly important. While distributed GPU simulation suffers from memory and communication bottleneck, optimization from enabling NvLink as well as qubit swapping algorithms tailored for such platform is able to speed up the simulation. Experimentation and analysis on the runtime data showed that these two methods proposed are highly effective, especially at larger numbers of qubit, where distributed GPU simulation used to fell short at, the speedup over pre-optimzied multi-GPU simulation is upto 48 times, and potentially higher with increased qubit size. To achieve this speed up, I designed Four different qubit swapping algorithms using different principles, two of which were proved to be better at handling different sizes of circuit.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jin Bai*)

Acknowledgements

I would like to express my utmost gradtitude for my supervisor Dr.Luo Mai for his instruction and support; I'd like to thank Yeqi Huang for his contribution to the baseline foundation of this project. I appreciate the feedback provided by my second marker Chris. I'd like to thank my friends and family for all of their whole-heartily support.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Classical computer has been finding more and more problems which it cannot solve in polynomial time, namely the NP problems. Quantum computing promises potential for accelerating many of the NP problems that are becoming increasingly relevant in many fields of computer science, even tackling classically impossible problems. [2] While quantum hardwares still suffers from many challenges including noise and decoherence, simulating quantum computers on classical computers plays a critical role for prototyping, assessing and understanding quantum algorithms.

With near-term quantum hardware soon reaching sizes and complexity to be useful, it also becomes increasingly challenging to directly simulate. However, the work to realise more complex quantum computers needs to be guided by exploration on quantum algorithm's scaling at such scale. Therefore, the demand for efficient, scalable simulation tools becomes increasingly urgent.

General purpose graphical processing unit (GPGPU) emerged within many areas of computations ranging from molecular dynamics [25] to training machine learning models [30]. GPU's ability to perform parallel processing is able to address the computation bottlenecks in simulating quantum computers. However, whilst showing promises for accelerating quantum computer simulations, GPU also has been shown to suffer from scaling due to exponential increases in memory consumption for increased qubit size. [6] [8] [32] [10]

This limitation leads to the question: are GPUs a key enabling technology for the acceleration of increasingly larger quantum computer simulations? And if so, what are needed for GPUs to provide maximal acceleration?

1.2 My Project

For my dissertation, I investigated the major issue quantum simulation is facing with multi-GPU set up, and provided my solution to a multi-GPU quantum circuit simulator

that scales better with increasing qubit sizes. I demonstrated my design choices by creating a prototype for such a simulator and benchmarked its scaling behavior. After that, I investigated the optimization methods proposed to speed up simulation, created several algorithms aimed to make optimization best suits simulators with multiple GPUs, and explored their performances under different circumstances.

This study bore the goal of reducing the simulation time of larger sized quantum circuit simulation on high performance computational platforms accessible to students, educators and researchers, namely, multi-GPU servers with high speed GPU interconnects. My dissertation aimed to contribute to the advancement of quantum computer simulations, in turn contributing to the board study of quantum computing.

1.3 Structure

Chapter 2: In this chapter, I will look into the basics of quantum computing, as well as how it translates to quantum computer simulations. Specifically, I will introduce the quantum circuit model, and the optimizations it allows to be made: gate fusion and qubit swapping. I will introduce how these optimizations are implemented in mainstream simulators including Qiskit, Cirq, QuEST and cuQuantum. I will then introduce the GPGPU model, the emerging high speed interconnect technology NvLink, as well as the GPU network topology.

Chapter 3: In this chapter, I will detail the design choices I made for a prototype on leveraging NvLink for multi-GPU quantum circuit simulation. I will justify the decisions on aspects such as memory allocation, gate implementations and peer access utilizations.

Chapter 4: In this chapter, I will present several algorithms used to optimize quantum circuits with qubit swapping, these algorithms are tailored for multi-GPU simulations, they are implemented in a prototype of the circuit transpiler. I will present the pseudo codes as well as time complexity analysis, and examples on its effectiveness.

Chapter 5: In this chapter, I will present the experiments conducted to evaluate my methods for its effectiveness. I will provide a comprehensive analysis of the speedup brought with my optimizations. In the end, I will discuss the results on the topic and potential improvements.

Chapter 2

Background

2.1 Quantum Circuits

2.1.1 Qubits

The basic unit in quantum computation is a qubit, it is a quantum counterpart of a classical bit. Just like how a classical bit can be in one of the two states: 0 or 1. A qubit can be measured to be in one of the two orthogonal computational basis states $|0\rangle$ and $|1\rangle$ [21] [12]

Unlike classical bits, qubits can be in a superposition of the two basis state, as shown,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{2.1}$$

Where α and β are complex probability amplitudes, the qubit collapse to $|0\rangle$ and $|1\rangle$ with probability of α^2 and β^2 upon being measured. And from the Born rule, the probability amplitudes must satisfy: $\alpha^2 + \beta^2 = 1$.

The notation used here is called dirac notation,in particular ket vectors representing column vectors. For quantum computing, a ket vector can either represent a single state, or the state of a system composed of one or more qubits:

$$\begin{aligned} |BA\rangle = |B\rangle \otimes |A\rangle &= \begin{pmatrix} b_0 \times \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \\ b_1 \times \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} b_0 a_0 \\ b_0 a_1 \\ b_1 a_0 \\ b_1 a_1 \end{pmatrix} \\ &= b_0 a_0 |00\rangle + b_0 a_1 |01\rangle + b_1 a_0 |10\rangle + b_1 a_1 |11\rangle \end{aligned} \tag{2.2}$$

The state of the system is the tensor product of all of its qubits' states, the resultant vector is called a state-vector. The indexing within the ket vectors in the rightmost representation each represent the basis state a qubit is in. Following little endian order, the rightmost bit is the first qubit.

2.1.2 Quantum Gates

Quantum computation is achieved by applying a sequence of unitary operations on the state-vector; these unitary operations are also called quantum gates; they can be represented in unitary matrix forms.[21] [12]

The rule for matrix multiplication enforces that a quantum gate applying to a single qubit needs to be of dimension 2×2.

Study on the universal set of quantum gates has shown that any unitary transformations can be written as a sequence of single-qubit quantum gates and CNOT gate.

The pauliX gate is a quantum analogy to the NOT gate in classical computers, it acted on a single qubit, hence it is a single-qubit gate.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

The CNOT gate is an example of multi-qubit gate, precisely, it is a controlled gate, where the state of the control qubit determines if the gate is applied to the target qubit.

$$CNOT = |0\rangle\langle 0| I + |1\rangle|0\rangle X$$

The right hand side represents that if the control qubit is in $|0\rangle$, an identity gate would be applied to the target qubit, whereas a pauli X gate is applied when the control qubit is in $|1\rangle$ state.

When applying a single-qubit gate to a qubit inside a multi-qubit system, the tensor product relation ensures that it has no effect on any other qubits in the system.

2.1.3 Circuit Model

Below is a quantum circuit:

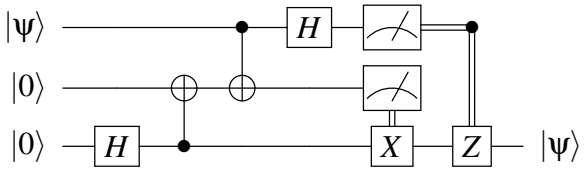

Table 2.1: Quantum Circuit Example

Each horizontal wire in the circuit represents a qubit, whereas the boxes represent quantum gates. Time flows from left to right in a quantum circuit, a quantum gate needs to be executed after the gate to its left has executed. controlled gates are represented by the dot on the control bit connecting to the gate on target qubits. [21] [12] The symbols near the end represent measurement, after which the qubit collapses to a classical outcome which is drawn with double lines.

Quantum circuit is a way to model quantum computation, every symbol in a quantum circuit has its corresponding mathematical representation:

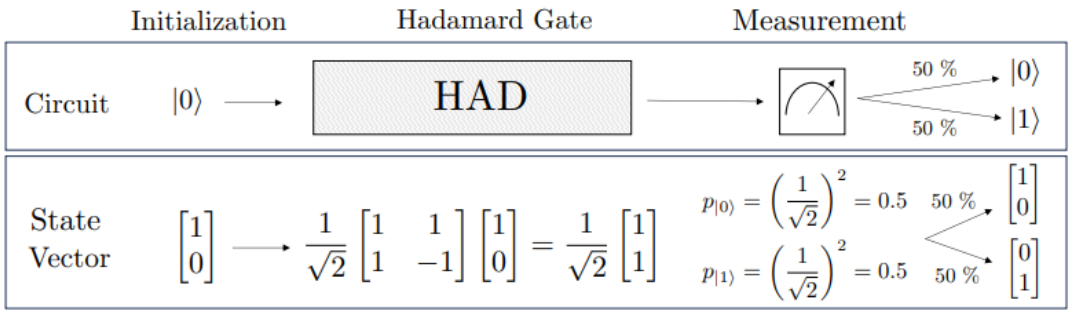

Figure 2.1: The mapping between quantum circuit and state-vector representations. A single qubit in $|0\rangle$ states is represented by the column vector, applying hadamard gate is doing matrix-vector multiplication, measurements corresponds to random sampling of a state according to its probability amplitudes[8]

2.2 Quantum Simulation

Development of real quantum hardware is facing challenges such as noise and coherence time, limiting the practicality and scalability of existing quantum computers, therefore, being able to simulate quantum computation on classical machines becomes increasingly important for purposes such as education and studying quantum algorithms.

Simulations of quantum computing on classical machines are widely modelled by quantum circuits, the circuit model provides universal, intuitive representation of quantum operations that can be directly mapped to real quantum hardware. [5]

Many mainstream simulators adopt full-state simulation, also called state-vector simulation, using the concept mentioned in the previous part, the simulator represents the state of the quantum system with its state vector, this method of simulation allow for fully capturing the evolution of highly entangled system. [14] [17] [29]

2.2.1 GPU Acceleration

As discussed in previous section, quantum gates can be represented by matrices, applying quantum gates to quantum systems can therefore be simulated by matrix multiplications. The highly parallel nature of matrix multiplication is able to utilize GPU extensively. [20]

From the early 2010s, attempts were made to accelerate quantum circuit simulation with GPU. More recently, many mainstream quantum simulators including Qiskit [6], QuEST [17] and Cirq [15] all supported GPU acceleration to different degrees.

Cirq and QuEST implemented simulation backends supporting single GPU, in their papers, the maximal qubits able to fit onto a single Nvidia A100 (40Gb) and a Nvidia Tesla K40m (12Gb) were 32 and 29 respectively. Cirq was benchmarked with noisy simulation at different noise probabilities, the GPU backends running on Nvidia A100 outperforms CPU backends significantly, providing 1.2× - 1.7× and 3× to 9× speedups at 20 qubits and 27 qubits. QuEST showed 5× speedups over already highly-parallelised 24-threaded CPU simulation. Thus, GPU is able to provide tremendous speedups to

quantum circuit simulations, however, the limited device memory size becomes the bottleneck for GPU simulations. [17] [15]

Qiskit, on the other hand, allowed use of multiple GPU, from benchmarking quantum volume circuits and quantum fourier transform circuits on machine with 6 Nvidia Tesla V100 (16Gb), compared to the linear scaling with multi-threaded CPU backend, GPU backend scales much slower between 25 to 29 qubits where one GPU's memory is sufficient, the speedup was reduced noticeably after using multiple GPUs. [6]

Additional benchmarking using qiskit-aer showed at most 12× speed up over cpu at lower number of qubits, at 32 qubits and higher the gap starts to close. The introduction of more gpu brings huge communication time to the total runtime. It was also discovered that with multi-GPU setup, the communication between 2 GPU took up over 90% of the total GPU time, making communication the top limiting factor of simulation runtime with multi-GPU simulation. [8]

2.2.2 Existing Approaches

The quantum circuit model also benefits from allowing certain optimizations, which includes gate fusion and qubit reordering.

Gate fusion is a technique to combine multiple smaller gates into one large gate, it is adopted by many simulators such as QHipster and its predecessor Cirq, as well as nvidia custatevec sdk. Gate fusion can save considerable computation time, but does not help with the communication bottleneck of Multi-GPU simulation. [3] [15]

Qubit reordering, this technique can be found in Project Q [6] , Qiskit [6] as well as nvidia custatevec sdk [3]. When the state vector is distributed to multiple nodes, each node stores only part of the state-vector, making some qubits local and others global, in the case of executing a gate acting on a global qubit, the node needs to access state-vectors stored on other nodes, thus incur communications between nodes. Qubit reordering instead maps the target qubit to another local qubit's location, swapping all their states and making the target qubit local, and thus eliminates the need to access other nodes' memory when executing a global gate. The policy all of them proposed can be summarized as "All global to local", the goal is to swap all global qubits to local positions when needed.

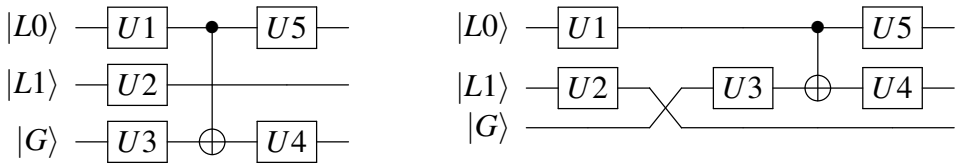

Table 2.2: Qubit Swapping Example, the left hand side is a circuit with local qubits L0,L1 and global qubit G, on the right hand side, L1 and G are swapped, gates applied to G are mapped to local qubit's position

However, one important realization needs to be made clear, swapping qubits does come

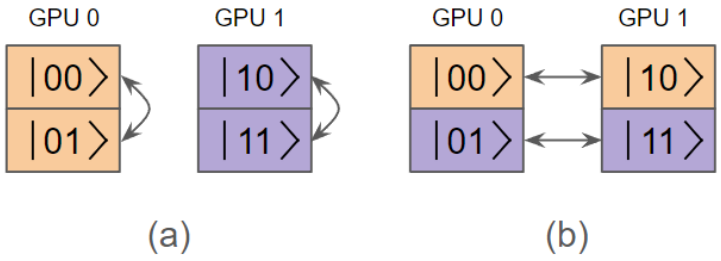

Figure 2.2: The state-vector distribution of 2 qubits with 2 GPUs. When a pauliX(bit flip) gate is applied, the states with opposite target qubit indices exchange probability amplitudes. (a) Apply X gate to local qubit at position 0 (b) Apply X gate to global qubit at position 1 where cross GPU data transfer is needed

at a price, since the swapped qubit's state is exchanged with that of a local qubit's, the process of swapping itself requires the node to communicate with other nodes.

Swapping as a technique goes beyond quantum simulation, it is also widely used in real quantum computation, since real quantum hardware often requires two qubits to be close by in order for multi-qubit gates to be executed. Swapping is performed to allow for this. And to reduce the number of swaps needed, there have been studies on algorithms making swaps in a circuit. Graph partitioning and hypergraph partitioning has been used for making swaps. The circuit is first decomposed into single-qubit gates and CZ gate, then optimized for better gate placements and translated into hypergraphs and optimized for swaps. [1]

2.3 Multi-GPU Server

2.3.1 GPU Programming

In the realm of high-performance computing and advanced data processing, GPU programming stands as a pivotal technology that has revolutionized the way computational tasks are executed. GPUs (Graphics Processing Units) were initially designed to accelerate graphics rendering but have evolved to become highly efficient parallel computing devices. Unlike traditional CPUs (Central Processing Units), GPUs are composed of hundreds or thousands of smaller cores, making them exceptionally well-suited for tasks that can be performed in parallel. This capability allows for significant reductions in computational time for a wide range of applications, including machine learning, scientific simulations, and data analysis.

The advent of multi-GPU servers has further enhanced the potential of GPU programming. By housing multiple GPUs, these servers can tackle complex, large-scale computational problems more efficiently than single-GPU systems. The parallel processing capabilities of multi-GPU servers enable the handling of larger datasets and more demanding simulations, making them indispensable tools in research and industry. However, leveraging the full power of multi-GPU architectures requires specialized

programming techniques and an understanding of how to distribute tasks and manage data across the GPUs effectively. [7] [19]

At its core are three key abstractions — a hierarchy of thread groups, shared memories, and barrier synchronization

2.3.2 Communication topology

GPU has its own memory, and GPU programming requires the data to be accessible either by storing on the GPU or by transfer. The data transfer between host and device, as well as device and device go through interconnects.

Peripheral-Component-Interconnect-Express-Bus (PCIe), is a high-speed serial computer expansion bus standard. Traditionally, a GPU-integrated system connect one or multiple GPU devices to the CPUs via PCIe. However, compared to the interconnect between CPU and DRAM, PCIe is much slower. It often becomes a major performance bottleneck for GPU-acceleration. [31] Such a condition exacerbates when PCIe based GPU P2P communication is adopted [22]

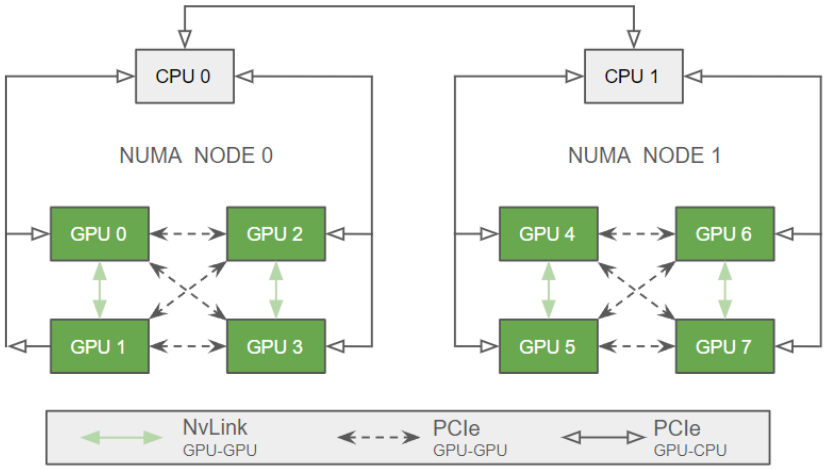

Figure 2.3: Illustration of the communication network topology in a multi-GPU server

Known as the first generation of NVLink, NVLink-V1 is a wire-based communication interface for near-range devices based on High-Speed-Signaling-Interconnect (NVHS) . It supports P2P communication that enables CPU-GPU or GPU-GPU linking. It allows direct read and write on remote CPU's host-memory and/or peer GPU's device-memory. Remote atomic operations are also feasible. NVLink is bidirectional; each link consists of two sublinks — one for each direction. Each sublink further contains eight differential NVHS lanes. An embedded clock is integrated for transmission. The packet size varies from 16 bytes to 256 bytes (one 128-bit flit to sixteen 128-bit flits). The communication efficiency is strongly correlated to the packet size. Overall, it is reported to be twice as efficient as PCIe [31]. An NVLink can be viewed as a cable with two terminalplugs whereas each GPU incorporates several NVLink slots. How these slots are connected via the NVLink cables dictate the topology and bandwidth

of the GPU network. Multiple cables can be ganged together to enhance bandwidth when they are linking the same endpoints. A Pascal-P100 GPU has quad NVLink slots. Therefore, for a dual-GPU system, a direct setting would be two GPUs connected by four NVLinks, leading to 4× bandwidth of a single link. [18]

The second generation of NVLink improves per-link bandwidth and adds more link-slots per GPU: in addition to 4 link-slots in P100, each V100 GPU features 6 NVLink slots; the bandwidth of each link is also enhanced by 25%. Besides, a low-power operating mode is introduced for saving power in case a link is not being heavily exploited. The extra two link-slots have enabled certain alternation to the original network topology.

Multi-GPU execution has become an increasing popular way to handle heavy computation tasks. The emerging new interconnect technology like NvLink also changed the communication topology of a multi-gpu server.

Benchmarks showed that NvLink connection, compared to PCIe, shoes strong NUMA effect, and in general achieve higher bandwidth and start-up latency comparing to PCIe. [18]

Results from testing the message size efficiency shows that NVLink's bandwidth starts to saturate strating from around 4mb, with smaller message size the gap between PCIe and NVLink is smaller. [18]

Chapter 3

Multi-GPU Simulator Leveraging NvLink

3.1 Overview

In order to study how quantum circuit simulators can exploit multiple GPUs and NvLink effectively, I developed a prototype multi-GPU simulator that can rapidly adapt to different data structures, memory allocation policy and data transfer methods.

GPU distributed prototype for quantum circuit simulation(GDP) is a prototype for full state vector quantum circuit simulator that supports GPU simulation exclusively. It was designed for the purposes to study and benchmark the speedup brought by uses of multiple GPUs as well as high speed direct interconnects such as NVLink. While GDP does not have the ability to simulate any quantum circuits like other simulators, it is fully capable of demonstrating runtime scaling as a universal simulator would. [26]

As mentioned in the background chapter, quantum circuit simulation quickly runs into memory bottlenecks with GPU, when adding GPUs to solve the memory bottleneck, inter-GPU communication become the major limiting factor for scaling: taking up more than 90% of the GPU time when applying a quantum gate. [8]

Therefore, GPU was designed to bring down communication overhead to its best capability as a demonstration on how to improve scaling with multi-GPU setup, and in turns accelerate quantum circuit simulation at larger qubit sizes.

3.2 Programming Languages

3.2.1 CUDA

Stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach known as GPGPU (General-Purpose computing on Graphics Processing Units). [19] CUDA C extends the C programming language with

NVIDIA's CUDA APIs, enabling direct access to the GPU's virtual instruction set and parallel computational elements. This allows for the development of highly efficient GPU-accelerated applications. Choosing CUDA C for a GPU programming project offers several benefits:

Direct GPU Control: Offers fine-grained control over GPU hardware, optimizing for performance and efficiency.

Wide Support: Extensively supported across NVIDIA's GPUs, ensuring broad compatibility and performance optimization.

Rich Ecosystem: Access to a comprehensive ecosystem of libraries, tools, and frameworks that accelerate development.

High Performance: Enables massive parallelism, significantly outperforming CPU-only computations for suitable tasks.

Mature Tooling: Benefits from NVIDIA's continuous improvements in compiler technology, debuggers, and profilers for GPU computing.

Flexibility: Allows mixing of GPU and CPU code, facilitating incremental optimization of existing applications.

3.2.2 Python

Pybind11 is a lightweight, header-only library that facilitates the creation of Python bindings for C++ code, enabling seamless integration of C++ libraries into Python applications.[16] By adding a Python interface to a CUDA C backend program using pybind11, developers can unlock several key benefits:

Ease of Use: Python's simplicity and readability make it easier to use and deploy complex C++/CUDA applications.

Rapid Prototyping: Developers can quickly prototype and test their CUDA C applications in Python, speeding up the development cycle.

Interoperability: Facilitates interaction between Python and CUDA C code, allowing developers to leverage Python's extensive ecosystem of libraries and tools alongside CUDA's computational power.

Accessibility: Makes high-performance computing accessible to a broader audience, including those not proficient in C++ or CUDA.

Performance: Python acts as a flexible wrapper, maintaining the high performance of the underlying CUDA C code while offering the convenience of Python.

Integrating pybind11 to bridge CUDA C with Python offers a powerful approach to developing and deploying high-performance applications that benefit from both CUDA's computational efficiency and Python's ease of use and versatility.

3.3 Qubit Indexing

In 3.1, n_local represents the number of qubits of the sub state vector. By combining multiple sub state vectors distributed to GPUs, one can add qubits to the state vector. In Figure 1, four sub state vectors are combined to add 2 (=$\log_2 4$) qubits [24]

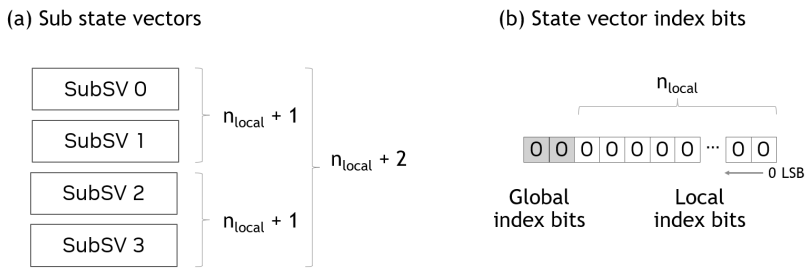

Figure 3.1: An example of the state vector distribution and the indexing [24]

As a state vector simulator, GDP maintains the pure state of a n qubit system with 2^n probability amplitudes in a state-vector with the same size.The bits of the state vector index directly map to qubits (or wires) in quantum circuits. There are two types of index bits. The first is the index bits of state vectors local to GPUs. The second is the index bits that correspond to the index of partitions. These types of index bits are called local and global index bits as shown in 3.1, the qubits mapped by these index bits are called local and global qubits.

When using multiple GPUs, GDP partitioned the state-vector evenly across all GPUs. GDP also requires the number of GPU being used to be power of 2. These two policies ensured a straight forward indexing of the global and local qubits, as shown in the figure, $\log_2 devices$ qubits are always global.

3.4 Gate Implementation

Shown in background chapter, If a gate acts on local index bits, the gate is applied in each sub state vector concurrently without accessing other GPUs. However, if a gate acts on global index bits, gate application needs accesses to multiple sub state vectors. As sub state vectors are distributed to multiple GPUs, gate applications on global index bits results in data transfers between GPUs, which is the performance limiter for the distributed state vector simulation.

It is natural to simulate quantum gates by performing matrix-vector multiplication. This requires O(N^2) memory on the device for storing the quantum gate. This added memory consumption is less ideal, when memory consumption is already the major limiting factor of GPU simulation.

For better memory consumption scaling, GDP adopts the gate implementation used by QuEST, Instead of performing the traditional matrix-vector multiplication, vector is modified in-place [17], The tensor product decomposition guarantees a pairwise relation:

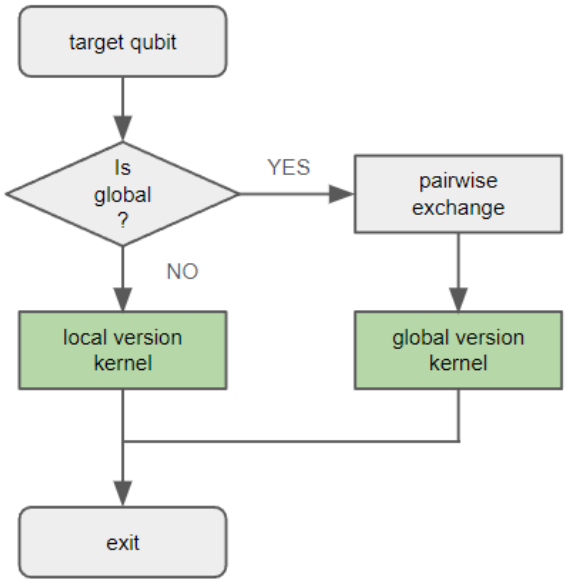

Figure 3.2: Flow chart for executing a quantum gate, the green blocks represent device code

$$\begin{pmatrix} \alpha_{n_i} \\ \alpha_{n_i+2^q} \end{pmatrix} \rightarrow G \begin{pmatrix} \alpha_{n_i} \\ \alpha_{n_i+2^q} \end{pmatrix} \tag{3.1}$$

With this approach, the computation no longer requires additional O(N^2) memory, but O(N).

Each quantum gate is implemented with one CUDA kernel that updates the state-vectors on any given device, and a wrapper function that handles all the pre-computation and others

3.5 States Management

As previous mentioned, GDP maintains the pure state of a n qubit system with 2^n complex floating points. With the default precision double, the state-vector alone takes up $2 \times 8 \times 2^n$ Bytes, where n is the qubit size.

GDP include a header cudarray implementing GPU-distributed arrays. cudarray provides abstraction over state management for GDP.Allowing GDP to oversee the implementation details and manipulate the states as one contiguous array, this provides GDP the ability to quickly adapt to different internal data structures as well as state managing policies. [13]

3.5.1 Even Partitions

To better study the scaling behavior with multiple-GPU setups, cudarray allocates states directly to device memory evenly. When the number of partitions N is power of 2, it can be proven that the number of states on each device is also power of 2. Combined with the pairwise relation equation, having power of 2 states in any partition allows for $(M - \log_2 N)$ qubits to stay local when simulating M qubits. Without these two assumptions, the pairwise relation cannot be easily used to determine if a gate is executed globally or locally.

3.5.2 Exchange Reserves

In the case of a gate executed globally, all states in a GPU need to access states stored on another GPU, hence requiring peer ot peer data transfer. This also means that each GPU needs reserved memory space to store these states temporarily.

The portion of the size of this reserving space over the size of the partition S and the transfer time T needed fits a reciprocal relation $S \times T = 1$. Meaning that if space equal to the partition size is reserved, all computation can be done concurrently incurring one data transfer, on the other hand, a simulator such as Qiskit that only reserves $1/10$ of the partition space needs to fetch for data 10 times. [6]

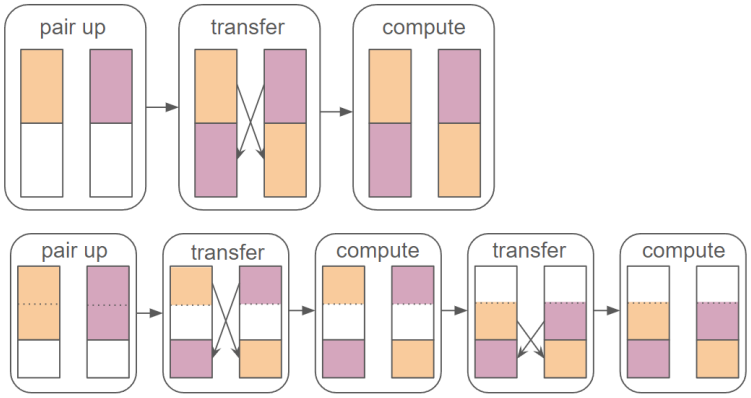

Figure 3.3: Upper half shows the execution of a global gate with reservation of 1× the partition size. Lower half shows that of reservation of 0.5× the partition size, needing 2 computation to update all states in a partition

Due to the exponential increase in memory consumption, reserving the size of the partition double the space needed by the simulator, resulting in one fewer qubit being able to fit into the memory. The benefit however, comes from not having to do many computations sequentially when they are highly parallelizable, as well as saving communication time.

Therefore, the memory consumption for this simulator is 32×2^n bytes at double precision. Far less than $16 \times 2^n + 16 \times 2^n \times 2^n$ for the matrix multiplication approach.

3.6 Leveraging NvLink

3.6.1 Transfer vs Qubits

In the case of a gate executed globally, enabling peer access between GPUs can provide massive speedup especially at larger numbers of qubits.

When peer access is disabled, device to device transfer involves copying the data from source device to host memory first, then copying the data from host memory to destination device. The double transfer and host's involvement can significantly increase latency for the transfer.

On the other hand, enabling peer access allows transfer between device memory directly through NvLink or PCIe bus. As shown in the figure, as the number of qubits increased, the runtime for peer access disabled and enabled both increased. However, the runtime with peer access enabled was consistently lower than with it disabled, the gap between the runtimes also increased with the number of qubits, especially from 25 qubits onwards, where the runtime without peer access increased substantially.

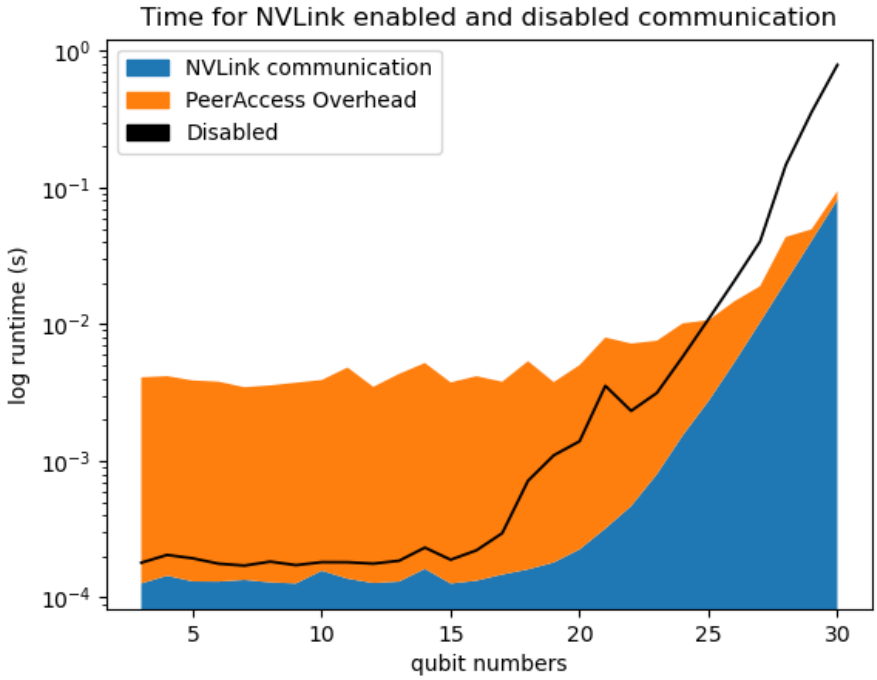

Figure 3.4: Results from testing communication same as in a gate with and without NvLink enabled, results is log scaled for better visibility

Although enabling peer access is always beneficial for transfer time, it can also be seen from the plot that enabling and disabling peer incurred near constant overhead, this arises from setting up the hardware and drivers for subsequent direct accesses. At lower numbers of qubits, this overhead was significant compared to the transfer time, only starting from 25 qubits, the combined time with peer access enabled outperformed that without.

3.6.2 Compute vs Transfer

Runtime of a gate executed globally is composed of compute time and transfer time, whereas that of a gate executed locally involves only computation. As the number of qubits increased, runtime for global gates eventually scaled exponentially as shown in figure, whereas the runtime for local gates stayed constant. Comparing the runtime of global gates to the transfer time in figure, it can be seen that transfer time is the dominant term of global gate runtime.

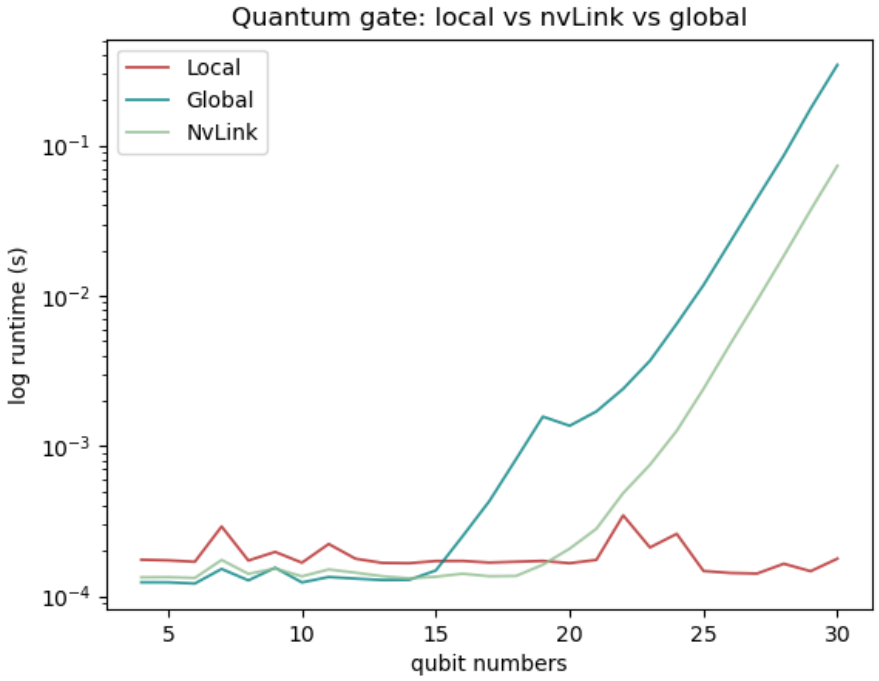

Figure 3.5: Logrithmic scaled runtime results from executing pauliX gate 100times at qubits with local, global and indices that utilize NvLink

The constant scaling of compute time arises from GPU's parallel computing capabilities, the computation increase is insignificant compared to modern professional GPU's computing power. In the case of CPU simulation, compute time would visibly increase as the number of qubits increases. From the figure, compute time has also been consistently higher for local gates, this was due to added computation from the index pairing calculation needed when a gate is executed locally, another possible factor is the memory access latency within a single GPU due to its memory hierarchy.

3.6.3 Peer Access Policy

With the added overhead, it is not always beneficial to enable peer access, at sufficiently low number of qubits, the states transferred is too small to facilitate transfer time reduction from Higher bandwidth that can outweighs the overhead. As shown in previous tests, enabling peer access only boosts performance at 25+ qubits simulating on gala with 8 GPU, the minimal number of qubits for peer access to be beneficial depends

on the simulation environment. Model of GPU, number of GPU used, communication topology can all affect this number significantly. As a solution, SIM provided a function to determine the minimal number of qubits that would benefit from peer access.

Chapter 4

Circuit Transpiler

4.1 Overview

With GDP being able to leverage NvLink using multiple GPUs, I decided to investigate the qubit swapping strategy. Quantum circuit transiple and optimization kit (QcTok) is a prototype designed to demonstrate and study the most beneficial qubit swapping strategy in a setup with multi-GPU and fast interconnects.

QcTok is implemented to cope with GDP, it has the capability to perform optimizations on circuits that can mock practical circuits' computation pattern with the core functions. Using QcTok as a baseline, it is straightforward to develop a complete circuit transpiler that supports most simulation syntax.

Within QcTok, I designed several algorithms to optimize circuits, these algorithms aimed to produce the best results without being limited by a specific simulator's syntax, making their performance gain translatable with other simulators.

4.2 Workflow

QcTok generates ready to execute source files in GDP syntax with below steps.

1. Reads all lines from the input file and tokenizes them.
2. Tokenized circuits are then optimized with qubit swapping.
3. Write the optimized circuits with selected simulator's syntax.

For the purposes of this project, QcTok currently supports the GDP syntax as well as QuEST syntax.

4.3 Optimizer

When a gate is executed globally, a round of communication is needed, where all devices forms pairs and transfer its state-vector to the paired device. The number of

data transfer varies depending on the system configuration, but this process can be done concurrently, hence it is simplified to be considered as a single round of peer to peer communications.

Given a quantum circuit with N gates, where G_0 of them are global, after S qubit swapping, having G_s the total runtime cost T of the optimized circuit is given by:

$$T(N) = compute(N) + transfer(G_0) \tag{4.1}$$
$$T(N+S) = compute(N+S) + transfer(G_s + S) \tag{4.2}$$
$$\delta T = compute(S) + transfer(G_0 - G_s - S) \simeq transfer(G_0 - G_s - S) \tag{4.3}$$

From the background chapter, it has been shown that especially at larger numbers of qubits, the data transfer time is the dominating term, therefore the focus on the qubit swapping should be minimizing the number of global gates in the circuits.

This problem is a complex optimization problem, the ability to insert swap gate at arbitrary positions makes the optimal solution difficult to find. Implemented optimization such as in qiskit and cuquantum all follow the described "all global to local policy" where all global gates are swapped to local before execution. An important realization is that swapping itself counts as a global gate, hence it may not provide the best runtime cost.

Below is a circuit, it has 5 qubits, qubit 0,1 in set Local, qubit 2,3,4 in set Global. There are 30 gates in the circuit, among them 10 are controlled gates, the rest are single qubit gates.

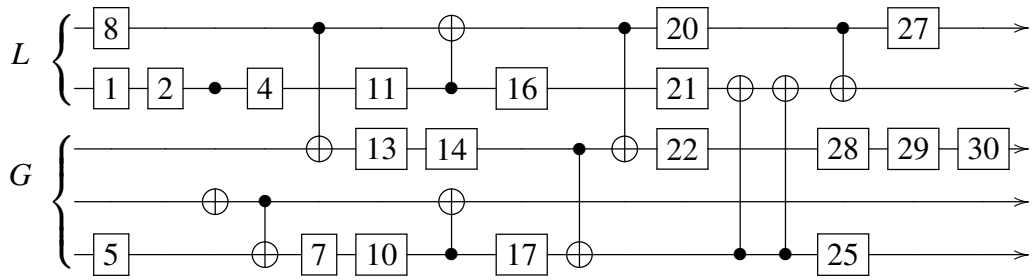

Table 4.1: Original Circuit, with $T = compute(30) + transfer(17) \simeq 17$

After swapped with "all global to local" policy [6] [14] [3], the circuits ended up with 17 additional swaps to put all global qubits to local whenever encountered, resulting in unchanged cost of the circuit.

The "all global to local" policy clearly does not perform well at minimizing the runtime cost function, there is much room for improvements.

4.3.1 Update Circuits

After a swap is made, all subsequent gates in the circuits need updating if the swapped qubits are involved. The intuitive approach is to iterate through the circuit from where the swap is inserted, check and update each gate. The time complexity for updating each gate is O(t+c), where t is the number of target qubits and c is the number of control qubits. Since most gates used in quantum circuits only have 1 to 3 qubits, this term can be seen as O(1). To traverse the circuit with n total gates starting after p gates where swap is inserted, the total updating complexity is O(n-p), making the overall time complexity of updating circuit O((n-p)*1), in the worst case where the swap is inserted at the beginning of the circuit, this function has time complexity O(n).

4.3.2 Global Optimization Algorithm

Inspiration: The optimization problem can be reduced in such a way, instead of inserting swaps at any position in the circuit, swaps can only be inserted to the beginning of the circuit. The assumption is that circuits with fewest global gates have the fastest runtime, and to achieve that, local indices should map to the set of qubits with highest occurrences in the circuit. This algorithm finds the maximized optimization for the circuit with an added constraint of only making global swaps(where it affects the entire circuit).

The GOA algorithm works as follows:

1. Counting the how many time each qubit occurred as target qubit
2. Select a set of qubits with the lowest occurrences to be the optimal local qubits
3. Pair up members that are not in the intersection of optimal local qubtis and current local qubits.
4. Make swaps and update the circuit

Time complexity.

- m, g are the number of qubits, and global qubits respectively
- n is the number of gates in the circuit
- Counting qubit occurrences is of O(n), sorting has time complexity of O(m log m) with efficient algorithms like quick sort or merge sort.
- Populating the optimal local qubits is of O(m)
- The set operations is of O(m), zip operation is O(g) at worst case where all the global qubits are in optimal local qubits
- Updating the circuit is O(m) as discussed in earlier section
- The overall time complexity is O(m log m) + O(m) + O(n), in practice, n is usually many times larger than m, which is limited by memory consumption.

The circuits processed by GOA is as shown, GOA found the set of qubits that have the most targeted-gates to be qubit 0 and 2, hence swapped qubit 1 and 2 a the beginning of

the circuit. This mapped qubit 2 to index of qubit 1, making all qubit 2 targeted gate local, and all qubit 1 targeted gates global. GOA has reduced 3 rounds of communication in the circuit

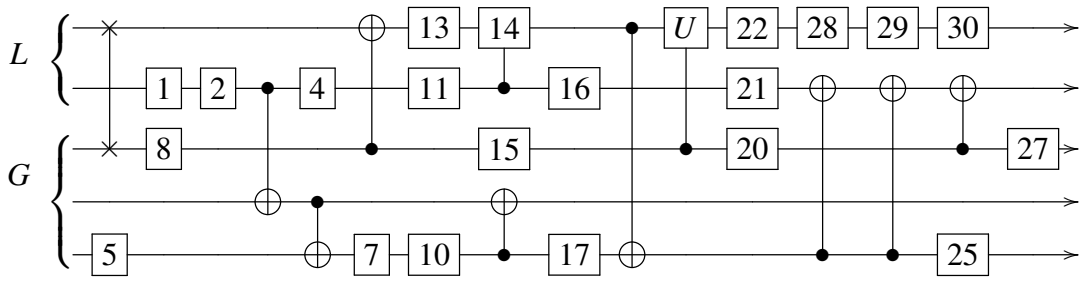

Table 4.2: Circuit swapped with GOA, with $T = compute(30+1) + transfer(13+1) \simeq 14$

Conclusion. GOA works on the reduced optimization problem, it can achieve optimal results within its constraint, but how much potential room for optimization is lost due to the constraint. Being able to insert swaps at arbitrary position of the circuit can potentially achieve better result if the local behavior in sections of circuit does not agree with the overall behaviour.

4.3.3 Greedy Tracker Algorithm

Inspired by GOA, this algorithm also keeps track of the occurrences of qubits, it is designed with a greedy algorithm aspect, where swaps are made greedily if it will benefit the transfer cost in the already seen section of the circuit. The assumption is making swaps that lowered the runtime cost within seen parts of circuits will be beneficial overall.

The GTA algorithm works as follows:

1. Employ a tracker that track the qubits seen
2. Check all gates, for those that are single-targeted, if the target is local, compute the cost changes brought to the tracked section of circuit if swapping with all other qubits that are global, store the results in a grid if the cost is reduced, vice versa if target is global.
3. Sort the grid such that the first item in each row represents the pair of qubits that, if swapped, will reduce the cost the most. Find pairs of qubits that provide approximately the best cost reduction.
4. Make swaps and update the circuit if any pair is found, clear the tracker and start over for the remaining gates in the circuit.

Time complexity.

- m, g , l are the number of qubits, and global qubits respectively

Algorithm 1 Global Optimization Algorithm (GOA)

```
1:  procedure GOA(Gates, Locals, Globals, All)
2:      Swaps ← empty list
3:      Occurrences ← empty map
4:      for all Qubit ∈ all do
5:          Occurrences[Qubit] ← 0.
6:      end for
7:      for all Gate ∈ Gates do
8:          for all Target ∈ Gate.targets do
9:              Occurrences[Target] ← Occurrences[Target] + 1.
10:         end for
11:     end for
12:     Sort Occurrences by value in descending order.
13:     Opt_Gs ← empty list
14:     Opt_Ls ← empty list
15:     for all (Qubit, value) ∈ Occurrences do
16:         if SIZE(Opt_Ls) < SIZE(locals) then
17:             Opt_Ls.APPEND(Qubit)
18:         else if SIZE(Op_Gs) < SIZE(globals) then
19:             Opt_Gs.APPEND(Qubit)
20:         end if
21:     end for
22:     Swap_Ins ← SET(Opt_Gs) - SET(Globals)
23:     Swap_Outs ← SET(Globals) - SET(Opt_Gs)
24:     Pairs ← ZIP(Swap_Outs, Swap_Ins)
25:     for all (In, Out) ∈ Pairs do
26:         ADDSWAP(Swaps, In, Out, Position ← 0)
27:         UPDATEGATES(Gates, In, Out, Position ← 0)
28:     end for
29:     return Swaps, Gates
30: end procedure
```

- n is the number of gates in the circuit
- Initializing the tracker has time complexity O(m)
- The nested loop calculating hypothetical cost changes has worst case complexity of O($g \times l$), sorting the lists with in grid has worst case complexity of O($g \times$ l log l), sorting the grid itself is of O(g log g)
- The double loop for selecting swaps is bounded by the size of the rid, it has complexity of O($n \times g \times l$) in the worst case where all gates are global
- Above process is repeated at worst case O(n) times
- The overall time complexity is O($n \times g \times$ l log l) + O($n \times$ g log g) + O($n^2 \times g \times l$) + O($n \times g \times l$) + O($n \times m$), the most dominant term is O($n \times g \times$ l log l)

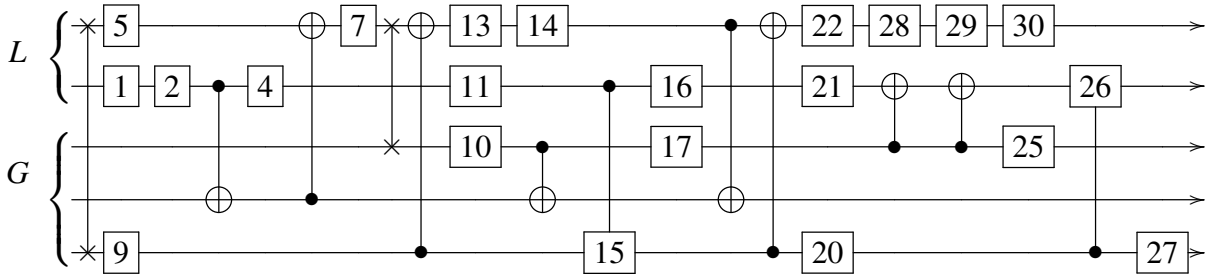

Table 4.3: Circuit swapped with GTA, with $T = compute(30+2)+transfer(10+2) \simeq 12$

Analysing time complexity has shown this algorithm to be slower than GOA without showing significant advantage on cost reduction. Examples on the circuit reflects the circuit partitioning done by this algorithm, GTA attempts to split the circuit into subcircuit and optimize separately. It is limited to focusing only on the parts seen, and does not necessarily make the best partitioning.

4.3.4 Look Ahead Algorithm

In contrast to GTA, where circuit splitting is based on the examined part of the circuit, the look ahead algorithm is designed with the assumption that, If a swap is only made when it lowers the overall cost of the circuit, the overall results must be better than original.

1. Keep track of how many occurrences each qubit has left in the circuit
2. For each gate in the circuit, calculate the cost of the entire circuit if no swap is made
3. Calculate the cost of the circuit with all possible swaps that can happen to the target qubit using the tracked occurrences, record the swaps that produce the lowest cost.
4. Make swaps and update the circuit if a swap is found

Algorithm 2 Greedy Tracker Algorithm (GTA)

procedure GTA(*Gates*, *Locals*, *Globals*, *All*)
    $Swaps, Gates \leftarrow \text{GOA}(Gates, Locals, Globals, All)$
    Occurrences ← empty map
    for all *Qubit* ∈ *all* **do**
        $Occurrences[Qubit] \leftarrow 0.$
    end for
    for all *Gate* ∈ *Gates* **do**
        if *Gate* is *single_target* **then**
            $Target \leftarrow Gate.target$
            $Occurrences[Target] \leftarrow Occurrences[Target] + 1$
            Grid ← empty list
            for all *G* ∈ *Globals* **do**
                if *G* in *Occurrences* **then**
                    Cost_G ← empty list
                    for all *L* ∈ *Locals* **do**
                        if *L* in *Occurrences* and $L \neq G$ **then**
                            $Cost\_Change \leftarrow Occurrences[L] - Occurences[G] + 1$
                            $Cost\_G.\text{APPEND}((G, L), Cost\_Change)$
                        end if
                    end for
                    Sort *Cost_G* by *Cost_Change* in ascending order
                    $Grid.\text{APPEND}(Cost\_G)$
                end if
            end for
            Sort *Grid* by first item's *Cost_Change*
            $Selected, Seen\_G, Seen\_L \leftarrow$ empty list
            for all $Cost_G \in Grid$ **do**
                for all $(G, L), Cost\_Change \in Cost\_G$ **do**
                    if $G \notin Seen\_G$ and $L \notin Seen\_L$ and $Cost\_Change < -1$ **then**
                        $Selected.\text{APPEND}(G, L)$
                        $Seen\_G.\text{APPEND}(G)$
                        $Seen\_L.\text{APPEND}(L)$
                        break
                    end if
                end for
            end for
            if $\text{SIZE}(Selected) \geq 1$ **then**
                for all $(In, Out) \in Selected$ **do**
                    $Position \leftarrow \text{SUM}(Occurences.\text{VALUES}())$
                    ADDSWAP(*Swaps*, *In*, *Out*, *Position*)
                    UPDATEGATES(*Gates*, *In*, *Out*, *Position*)
                    Occurrences ← 0, for all qubits
                end for
            end if
        end if
    end for
    return $(Swaps, Gates)$
end procedure

Time complexity.

- m, g , l are the number of qubits, and global qubits respectively
- n is the number of gates in the circuit
- Counting qubit occurrences is of O(n)
- Calculating the cost of the entire circuit is of O(g), this is done O(l) times if target is global and O(g) if target is local, making the combined time complexity O($l \times g$).
- Above process is repeated at worst case O(n) times
- The overall time complexity is O($l \times g \times n$) + O(n)

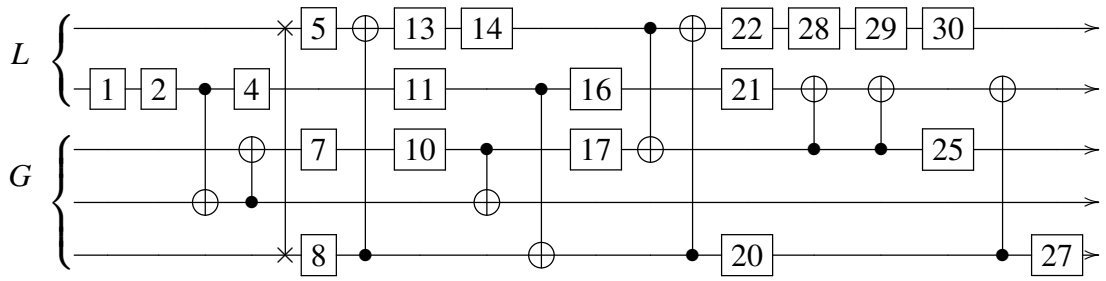

Table 4.4: Circuit swapped with LAA, with $T = compute(30+1)+transfer(12+1) \simeq 13$

The LAA is able to split the circuit in a more optimal way than GTA, it is also faster. Moreover, LAA(and GTA) can be combined with GOA, having the circuits optimized before supplying it to LAA, the two algorithms complement each other.

4.3.5 Ideal Locals Algorithm

Unlike GTA and LAA, this algorithm focus not on directly lowering the cost of the circuit. But to partition the circuit into subcircuits. The assumption is that, If a circuit can be split into subparts that only has local qubits, it has no communication cost. The added communication cost with swap insertion is accounted for with inspiration from dynamic programming.

This algorithm works as follows:

1. Employ a tracker that track the qubits seen, and keep track of how many occurrences each qubit has left in the circuit
2. For each gate seen in the circuit, updates the tracker until the number of qubits examined is at local qubits' limit. The set of qubits tracked is called candidates.
3. Remove qubits from candidates if it occurred less than twice, this is to counter the cost increase brought by insertion of swap gate.

Algorithm 3 Look Ahead Algorithm (LAA)

```
1:  procedure LAA(Gates, Locals, Globals, All)
2:      Swaps, Gates, Remaining ← GOA(Gates, Locals, Globals, All)
3:      for all Gate ∈ Gates do
4:          if Gate is single_target then
5:              Target ← Gate.target
6:              Cost_0 ← sum of Remaining[G] for each G in Globals
7:              Optimal ← Target
8:              if Target ∈ Globals then
9:                  for all L ∈ Locals do
10:                     Cost_H ← Cost_0 − Remaining[Target]
11:                     Cost_H ← Cost_H + Remaining[L] + 1
12:                     if Cost_H < Cost_0 then
13:                         Cost_0 ← Cost_H
14:                         Optimal ← L
15:                     end if
16:                 end for
17:             else if Target ∈ Locals then
18:                 for all G ∈ Globals do
19:                     Cost_H ← Cost_0 − Remaining[G]
20:                     Cost_H ← Cost_H + Remaining[Target] + 1
21:                     if Cost_H < Cost_0 then
22:                         Cost_0 ← Cost_H
23:                         Optimal ← G
24:                     end if
25:                 end for
26:             end if
27:             if Optimal ≠ Target then
28:                 In, Out ← Target, Optimal
29:                 Position ← index of Gate in Gates
30:                 ADDSWAP(Swaps, In, Out, Position)
31:                 UPDATEGATES(Gates, In, Out, Position)
32:                 Exchange remaining[In] and remaining[Out]
33:             else
34:                 Remaining[Target] ← Remaining[Target] − 1
35:             end if
36:         end if
37:     end for
38:     return (Swaps, Gates)
39: end procedure
```

4. Pair up members that are not in the intersection of candidates and current local qubits, the qubit that has the least future occurrences is prioritized to be swapped out.
5. Make swaps and update the circuit if found any pairs. Clear the tracker and start over for the remaining circuit.

Time complexity:

- m, g , l are the number of qubits, and global qubits respectively
- n is the number of gates in the circuit
- Counting qubit occurrences is of O(n)
- Updating tracker and remaining occurrences are all O(1), the set operations to select candidates is of O(l), picking out the qubit with least remaining occurrences with sorting is of complexity O(m log m)
- The examination process is repeated at worst case O($\frac{n}{l}$) times
- The overall time complexity is O($\frac{n}{l}$ m log m) + O(n)

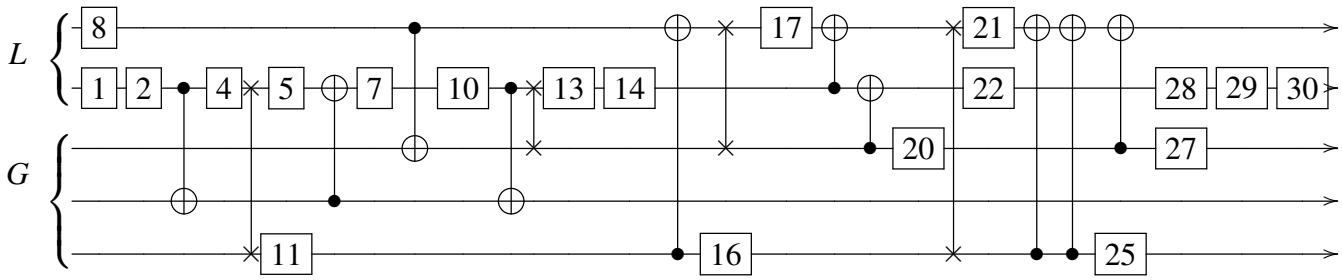

Table 4.5: Circuit swapped with ILA, with $T = compute(30+4) + transfer(8+4) \simeq 12$

As seen, ILA made swaps very differently comparing to the rest of the algorithms, and achieved a better result, ILA made use of many of the mechanisms from previous algorithms. Although it's partitioning heuristic relies on sufficiently large circuit size to work as intended.

Algorithm 4 Ideal Locals Algorithm (ILA)

```
1:  procedure ILTA(Gates, Locals, All)
2:      Swaps ← empty list
3:      Occurences ← empty map
4:      Remaining ←COUNTQUBITS(Gates,All)
5:      for all Gate ∈ Gates do
6:          if Gate is single_target then
7:              Target ← Gate.target
8:              remaining[Target] ← remaining[Target] − 1
9:              if Target ∈ Occurences then
10:                 Occurences[Target] ← Occurences[Target] + 1
11:             else if SIZE(Occurences) < SIZE(Globals) then
12:                 Occurences[Target] ← 1
13:             else
14:                 Not_Seen ← SET(Locals) − SET(Occurences.KEYS())
15:                 Candidate ← SET(Occurences.KEYS()) − SET(Locals)
16:                 Sort Not_Seen based on Remaining with descending order
17:                 for G ∈ Candidate do
18:                     if Occurences[G] ≥ 2 then
19:                         In, Out ← G, Not_Seen.POP()
20:                         Position ← index of Gate in Gates
21:                         ADDSWAP(Swaps, In, Out, Position)
22:                         UPDATEGATES(Gates, In, Out, Position)
23:                         Exchange remaining[In] and remaining[Out]
24:                     end if
25:                 end for
26:                 Occurences.CLEAR()
27:             end if
28:         else
29:             Repeat Iteration for Gate
30:         end if
31:     end for
32:     return (Swaps, Gates)
33: end procedure
```

Chapter 5

Experiments and Evaluations

5.1 Overview

5.1.1 Data Collection

Four experiments were designed to evaluate the performance of SIM and Q-TOK, the focus of these experiments was to determine, respectively, the effectiveness at enhancing the scalability of multi-GPU by leveraging NvLink and qubit swapping. All experiments were conducted on the server gala1, see 2.3 for gala1's network communciation topology.

Randomly genreated circuits have long been used to test the peformance of quantum computer simulators. [4] Two types of randomly generated quantum circuits were used for sampling. Uniform circuits: all qubits in uniform have equal probability to be targeted by quantum gates, quantum algorithms that produce near-uniformly distributed circuits include: grover's algorithm, bernstein- vazirani quantum, approximation optimization algorithm,etc. [11] [28] [9] Biased circuits: a range of qubits are targeted by much more quantum gates than others, usually with controlled gates, quantum algorithms that produce biased circuits include: quantum fourier transform, quantum phase estimation, shor's algorithm, etc. In reality, such a circuit is usually the combination of two qubit registers; by convention, the more targeted registers are usually placed at the bottom of the circuits. Hence it is sensible to evaluate circuits where global qubits are targeted more frequently. [27] [23]

SIM was first evaluated for the additional scalability gained by enabling NvLink. At a range of qubits: 10 to 30, ten randomly generated uniform circuits were simulated with Peer access enabled and disabled respectively. The same experiment was carried out using 1, 2, 4 and 8 GPUs.

Q-tok was then evaluated for its ability to reduce the communication cost with uniform circuits as well as biased circuits. At the same range of qubits: 10 to 30, ten randomly generated circuits from each category were transpiled with different swapping algorithms. These circuits were then simulated on SIM using 8 GPUs with NvLink enabled.

Q-tok's swapping algorithms were then evaluated for their performance when handling different circuit depth. Five uniform circuits were generated for each combination from 10 different qubit numbers and 10 different circuit depths. All 500 circuits were processed by GOA, GTA, LLA and ILA respectively, the transpiled circuits adding the original circuits were then simulated on SIM using 8 GPUs with NvLink enabled.

SIM and Q-tok together, was lastly evaluated against the multi-threaded CPU based QuEST.

5.1.2 Hypothesis

From previous results on gates runtime and communication time, enabling NvLink was expected to provide significant speedup starting from around 25 qubits with 8 GPUs, the number of qubits where enabling NvLink started being beneficial was expected to decrease as the number of GPUs used decrease, due to decreased overhead from enabling and disabling.

Comparing to the previous demonstration on qubit swapping, the 'all global to local' policy was expected to provide little to no speedup over the original circuits, where the rest of the algorithms were expected to reduce the communications by around 20

With biased circuits' denser global gates distribution, the algorithms were expected to provide much higher speedups than uniform circuits.

5.2 Evaluating: Simulator

5.2.1 At 8 GPU

It can be seen that at all number of qubits, the time taken to simulate the circuit with enabled peer access was shorter than that without, from 10 to 27 qubits, NVLink provided 1.2× to 1.5× speedup over disabled version, the speedup became substantial at 28 qubits with 2× speedup at 29 qubits and 3× speedup at 30 qubits.

Although the gate execution time of enabled version had always outperformed the disabled version, the simulation time using SIM had to include the preparation time to enable and disable peer access on all devices, this overhead stayed near constant, and was significant at lower qubit numbers, making the enabled version slower than disabled version even with a faster execution time, the turning point was at 25 qubits, where the total simulation time of enabled version outperformed the disabled simulation time, and the preparation time become negligible comparing to the total simulation time.

Analysing the runtime for disabled versions,the runtime stays relatively constant from 10 to 17 qubits, then shows a 'step' increase pattern every two qubits for the next 4 qubits, each step with a 30% to 50% increase from the previous. From 23 qubits onward, it shows a steady increase every qubit from 40% to 80% until 26 qubit, where the runtime more than doubled, from 27 qubit onward, the increase in runtime gets closer and closer to that of an exponential scaling, resultanted in 800× the runtime at 30 qubit comparing to 10 qubit.

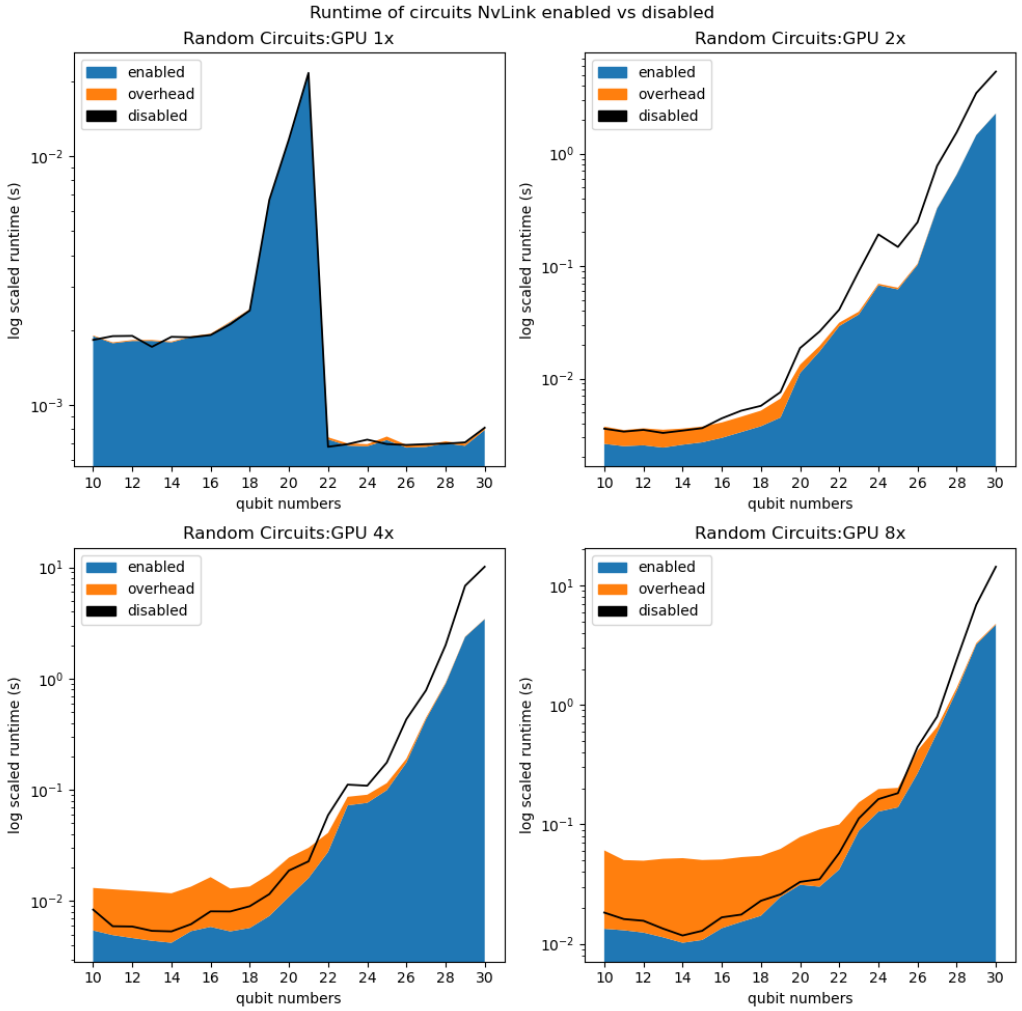

Figure 5.1: Experimental results of (depth=200) random uniform circuits' runtime in logarithmic scale, the stack plots' blue region represent the runtime on executing the gates, the orange region represents time taken for setting up peer access. Runtime without peer access is plotted with the line.

As for the enabled versions, the overhead from enabling and disabling peer access has stayed nearly constant at all qubits ranging from 0.04 to 0.08 seconds. For the execution time, qubit 10-15 has been constant time, from 16 qubits to 25 qubits, the runtime increased in around 30%'steps' of 2 to 3 qubits with runtime doubling from 22 to 23. After 25 qubits, the runtime nearly doubled for every added qubit. Overall, the scaling on the NVLink enabled version is a lot better, with only 350x the runtime at 30 qubit compared to 10 qubit.

Concluding this experiment, it can be seen that utilizing nvlink's higher bandwidth will always be beneficial in terms of execution time, and it provides much better scaling for larger numbers of qubits where multi-gpu simulator felt short at. The overhead of enabling and disabling peer access is negligible at larger number of qubits where NVLink shows more significant improvement over disabled version but rather significant at smaller number of qubits, making SIM overall performance without peer access better

at 25 qubits and lower, this fits the behaviour of the communication time in earlier section. This overhead is incurred by cudaDeviceEnablePeerAccess, this function sets up the peer access between devices, allowing the communication to leverage NVLink, the overhead comes from setting up the hardware and drivers for subsequent direct accesses for all existing allocations. The scaling behaviours arise from the bandwidth differences, since the transfer size scales exponentially with number of qubits, at small number of qubits, the transfer size is very small, where the setup time contributes to the majority of communication time, making NvLink only slightly faster at transferring. With larger number of qubits, the transfer size was significantly larger, where NvLink's much higher bandwidth was able to provide significant speedup over PCIe.

5.2.2 8 vs 4 vs 2 vs 1 GPU

From the figures, the scaling of NvLink transfer time for 8, 4 and 2 GPU were nearly identical, the runtime without peer access went up when using more GPUs, investigating the raw data, at lower number of qubits, the execution time scales linearly with the number of GPU, which suggests this increase is likely coming from the increase in the number of transfers inside each round of communication. The biggest differences across different numbers of GPUs lie in the intersecting point. It can be seen that the overhead for enabling peer access drastically decrease with fewer number of GPUs used, as a result, at 8 GPUs, peer access became faster at 25 qubits, at 4 GPUs, this number reduced to 21 qubits, at 2 GPUs, it went further down to 15 qubits.

5.3 Evaluating Remapping Algorithms

The average number of communication in a circuit before and after applying the above remapping algorithm is shown in 5.2.

From the plot, all the algorithms had on average reduced at least 15% of the communications in the circuit.

With the random circuit where each qubit has equal probability of being targeted by a quantum gate, the proportion of global gates decreases as the number of qubits increases in uniform circuits, this is due to the uniform probability of qubits being targeted and the fixed number of global qubits $\frac{g}{N}$. From previous testing, the time taken to complete one communication round increase exponentially at larger number of qubits, making the performance of algorithms at higher qubits more important.

Analysing the plot, the GOA, due to its mechanisms, always insert the minimal number of swaps. At lower number of qubits from 10 to 18, GTA and LAA inserts more swaps to the circuit, with GTA inserting on average 20% fewer than LAA, although the overall communications is about the same for both algorithms, with one out performing the other at some situations, both algorithms shows better results comparing to GOA. ILA inserts the most swaps to the circuits, it is the only algorithms that has more communications coming from swaps than global gates, ILA also shows a decent amount of advantage in terms of communication reduction over the rest by around 20% comparing to GOA and 10% to GTA and LAA.

With 18 qubits and more, the performance gap is starting to close up between all four algorithms, while the overall performance comparing to unprocessed circuits has increased, the algorithm on averages now reduce half of the communications, including GOA, while the number of swaps made stays the same. As for LAA and GTA, both algorithms now makes the similar number of swaps, but LAA on average performs better than GTA. The ratio of swap vs global gates for ILA stays the same, but ILA has lost its advantage in terms of overall communication reduction against the rest.

This suggests that for uniform distributed circuits, ILA has massive advantage with smaller number of qubits, at 20 qubits and more, LLA's results becomes comparable to that of the ILA, at 26 qubits and higher, GOA becomes comparable to ILA and LLA, leaving GTA the least effective overall.

5.3.1 Runtime:Uniform Circuits

Looking at 5.3, It can be seen that swapping under the "all local" principle provides terrible performance, especially with NVLink enabled, the average speedup is negative half the time, the shaded area has 40% under the speedup=1 line, combined with average speedup plot, it can be concluded that such strategy is not suitable for device-stored simulator like SIM.

Looking at subplot5, the average speedup provided by four algorithms is showing a similar overall trend, corresponding to the communication reduction plot. From 10 to 15 qubits, the average speedup ranges from 1.1x to 1.2x, from 15 to 25 qubits, the average speedup ranges from 1.2x to 1.5x, from 25 qubits onwards, the speedup is more than 1.6x, with maximal average speedup of 2.4x at 30 qubits, with the trend of likely going up sharply.

All the speedups are on top of the speedups already provided by enabling NVLink, judging by previous evaluation, at 30 qubits, NVLink + remapping can achieve as high as 12x speedup over unprocessed circuits without NVLink enabled at 8 gpu server.

At lower number of qubits, ILA shows better performance than all other algorithms which agrees to the communication reduction plots. From 10 to 18 qubits, ILA on average provides 10% more speedup compared to the others, confirming its effectiveness with smaller numbers of qubits GOA, GTA and LAA have similar performances with GTA slightly better. With 25 and more qubits, ILA suddenly dropped to the same level of speedup provided by GOA and LAA, with LAA overtaken to be the most effective algorithm, providing on average 15% added speedups.

This makes ILA ideal for smaller number qubits, whereas LLA is ideal for higher number of qubits.

How consistent are the algorithms? Looking at all algorithms, they tend to stay in the middle of maximal and minimal speedups, comparing the shaded areas, it can be seen that GOA is more consistent than others, with the narrower areas. ILA, GTA, LAA has increasingly more cases of negative speedups, making ILA and LAA having better worst case performances.

ILTA and GTA, on the other hand, each has cases where one is more stable than the other.

5.3.2 Runtime:Biased Circuits

See 5.4, In biased circuits generated, global qubits had constant probability of being targeted, unlike the reciprocal relation from uniform circuits. This overall increase in the number of global gates drastically improved the speedups qubit swapping provided. Compared to uniform circuits, GOA is more consistently effective at a larger range of qubit numbers, the nature of biased circuits ensured that swapping global qubits with local qubits is much more beneficial, agreeing with GOA's assumption. On the other hand, ILA's no longer out perform all other algorithms, this largely has to do with GTA and LAA taking advantage of GOA's results.

For highly swap-benefitbale biased circuits,m GOA,GTA and TLAA has near identical average speedups due to GOA being highly effective. ILTA, however, suffered from inconsistency.

To conclude, for highly concentrated circuits, GOA becomes increasingly effective, where ILTA is better at handling sparse circuits.

5.3.3 Circuit Size vs Depth

Looking at 5.5, Throughout the entire range of qubit numbers, ILA's speedup increased with increase in circuit depth. When circuit depth was smaller than 50 gates, ILA performed terribly, providing no or even negative speed up over the original circuits. During medium qubit range, ILA outperformed all other algorithms on nearly every depth, but provides slightly lower speedup at large number of qubits.

GOA and GTA had very similar performance, GOA was more consistent across all circuit depths, whereas GTA performed slightly worse on some occasions. LLA had very similar trend with the other two, but performed better in nearly all settings.

In conclusion, with circuit depths ranging from 50 to 200, qubit number ranging from 12 to 26, ILA is the best choice, with qubit size over 26, LLA provides higher speedup. And with in 50 depth and 16 qubits, swapping algorithms are not effective.

5.4 Results

The experiments has shown significant improvement on runtime with both NvLink and qubit swapping, with uniformaly distributed circuit, at 25+ qubits on 8 GPUs, the combined speedup is over $8\times$, this speedup was due to NvLink's high bandwidth, low latency in combination with qubit swapping algorithm's ability to reduce global gates.

In general, the two methods discussed was proved to be effective at improve the scalability and accelrate quantum circuit simulation with multiple GPUs.

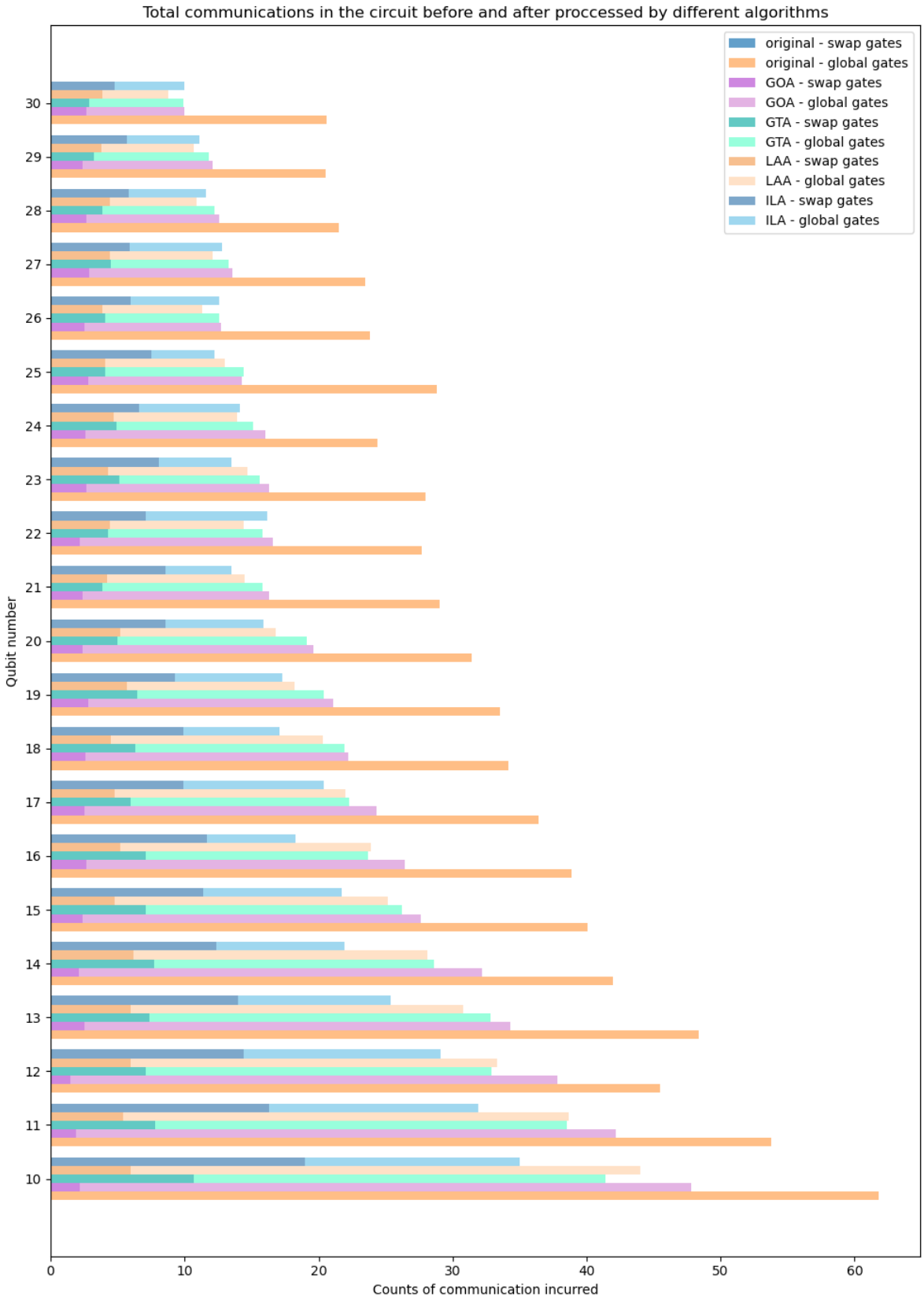

Figure 5.2: the bars represent the average number of communications, incurred by global gates as well as swap gates, at each qubit the number of communications in the original circuit as well as the ones in circuits after remapping is shown.

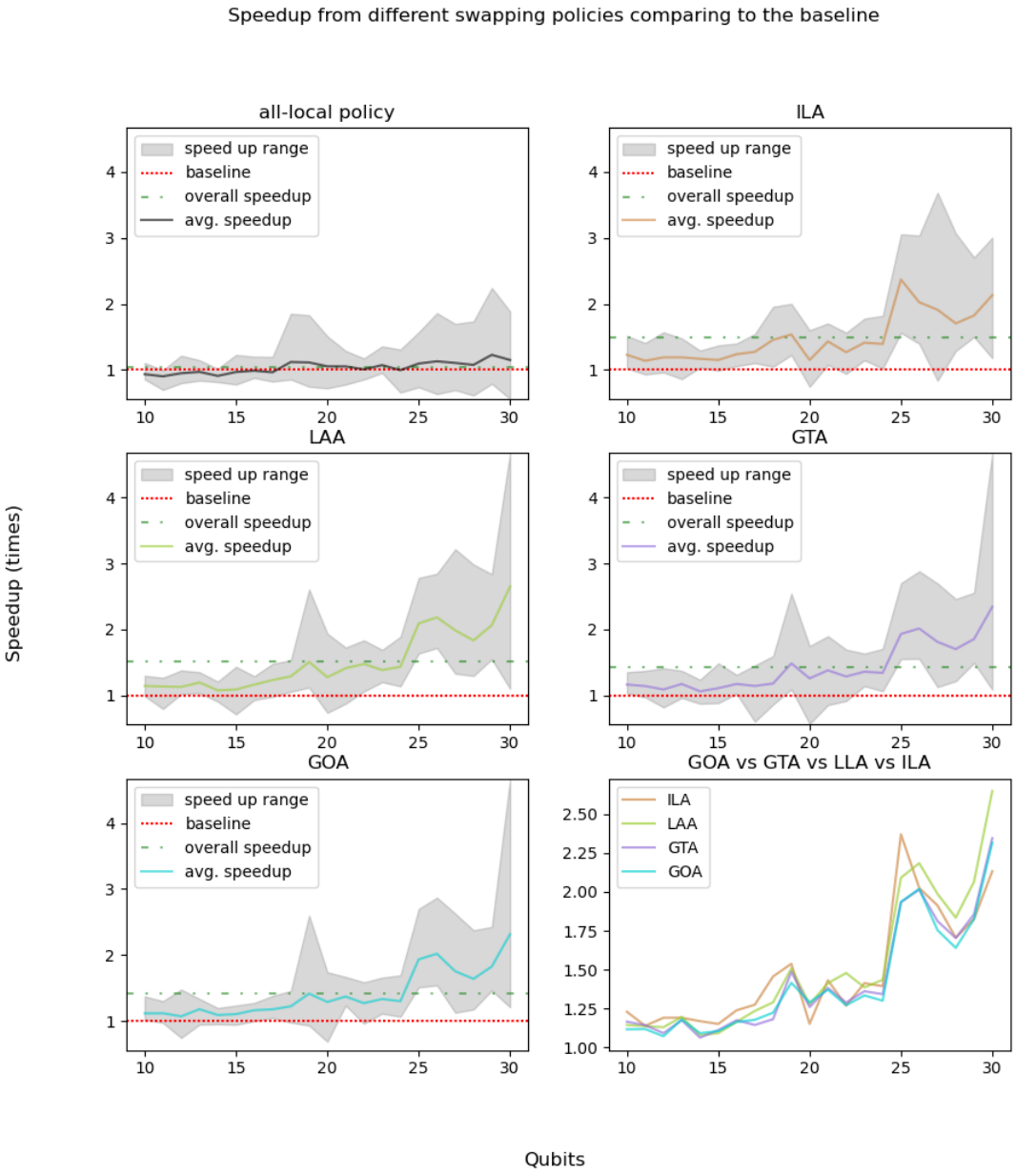

Figure 5.3: first five subplots shows the speedup of processed circuits vs unprocessed circuits, the shaded area is the full area covered by the max speedup and min speedup at a given qubit number, where the line represents the average speedups. The last plot combined average speedups from all algorithms for more direct comparison, horizontal lines each represent the average overall speedup across all qubit numbers and the speedup=1 line representing no performance gainl

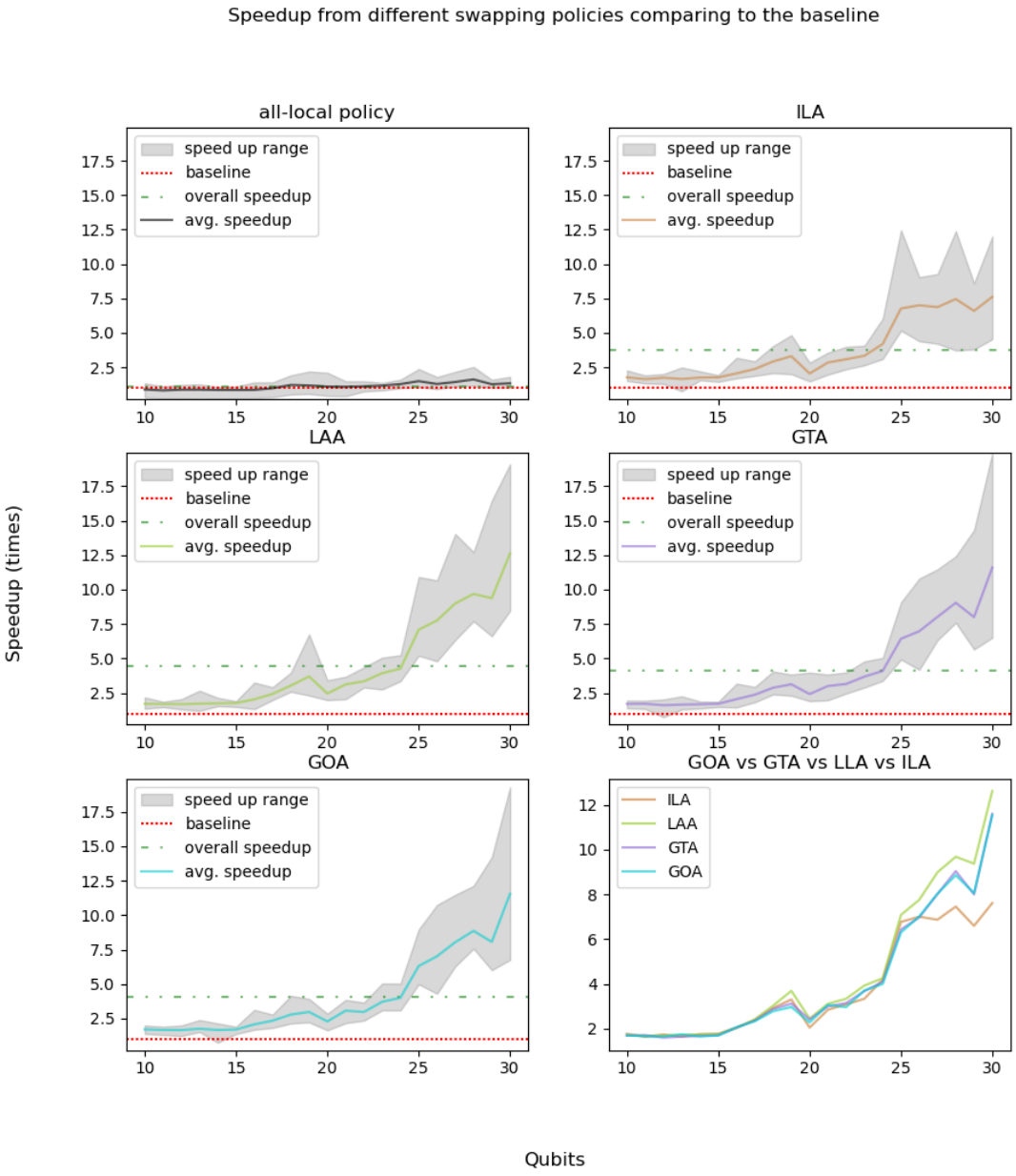

Figure 5.4: first five subplots shows the speedup of processed circuits vs unprocessed circuits, the shaded area is the full area covered by the max speedup and min speedup at a given qubit number, where the line represents the average speedups. The last plot combined average speedups from all algorithms for more direct comparison, horizontal lines each represent the average overall speedup across all qubit numbers and the speedup=1 line representing no performance gainl

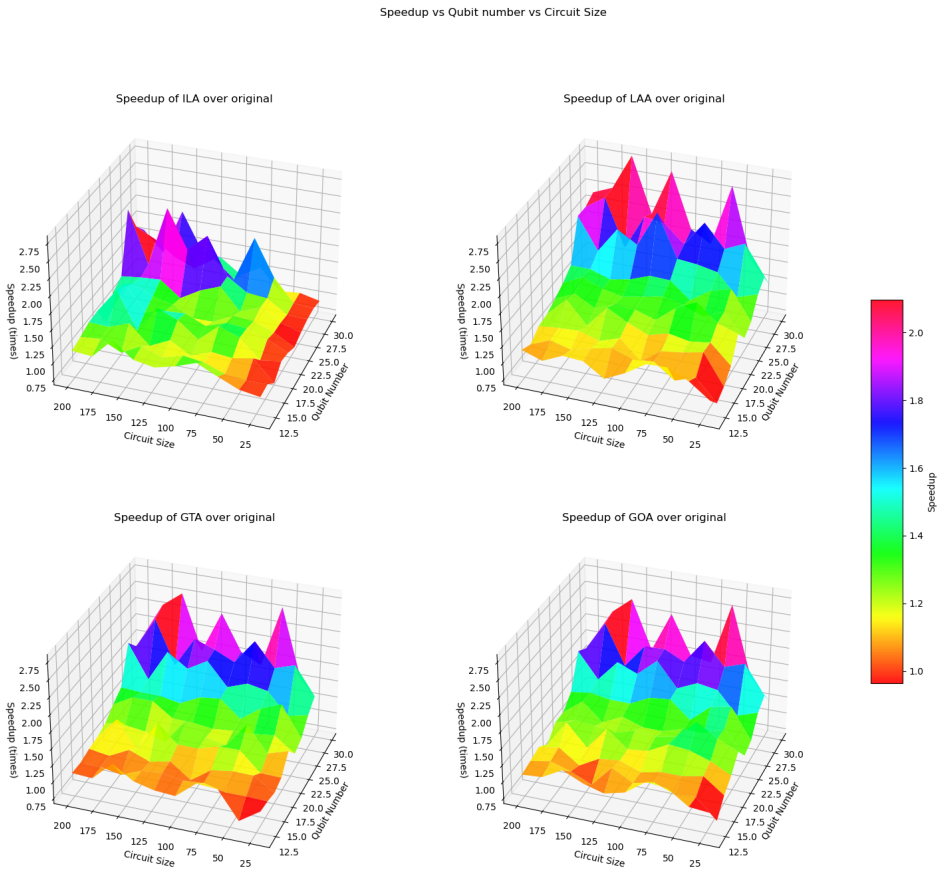

Figure 5.5: The x and y axis each represent circuit depth and qubit number, whereas the z axis represents the speedup over original circuit's runtime.

Chapter 6

Conclusion and future work

In this report, I set out several improvements to quantum circuit simulators with multiple GPUs. Which mainly composed accelerations from the two sources.

Firstly, I designed and justified the necessary features and configurations of scalable multi-GPU simulators, and demonstrated with my prototype GDP. I showed that the number of GPUs used needs to be power of 2, together with even partitions of the state-vector across all devices, these requirements ensures the correctness the qubit indexing systems, in turn excludes majority of the qubits from incurring costly communications. I showed a scalable implementation of quantum gates, which caters to both memory consumption and data transfer through in-place modification of the state-vector as well as the exchange policy for global gates, the two major limiting factors for GPU based simulations to scale. I tested and justified enabling peer access between GPUs accelerates simulations considerably at sufficiently large qubit sizes. I benchmarked the simulator prototype against state-of-art CPU based simulator and found my simulator provides minimal 7x speedup at 30 qubits. I further benchmarked the effect of enabling NvLink and found the scaling to be much smoother especially at larger qubit size, at 30 qubits, the enabled NvLink still provides worst case 4x speedup against simulations without NvLink. These results indicated that the introduction of high speed GPU interconnect technology is crucial for quantum circuit simulations at near-term scale. Secondly, I provided quantum circuit optimization solutions tailored for multi-GPU simulations on a NUMA architecture. I proposed 4 swapping algorithms that maintain the original input circuit maximally, which would be useful for general purposes of optimizing quantum circuits regardless of the simulation tools used. I further adapted a gate reordering algorithm for better efficiency on multi-GPU simulations, as well as increased capability in dealing with multi-qubit gate centric circuits. I experimented with the optimizer under different parameters such as the distribution of target qubits, the length of the circuit and the size of qubits. I found my algorithms capable of reducing up to 50% the communications in a uniformly distributed circuit, and much more with global qubit skewed circuits. The reduction in communications reflects on runtime, with uniform circuits, my best algorithm provides over 2.5 times speedup on average at 30 qubits. With skewed circuits, this number can go up to 12 times and even beyond. Results showed that my proposed LAA algorithm is the most effective with

larger qubit size, whereas ILA provides more consistent speedup at lower qubit sizes. The results on gate reordering algorithms showed its effectiveness goes down with increased multi-qubit gates in the circuit. In other words, my qubit swapping algorithms are able to provide additional speedup over already accelerated NvLink simulation at large qubit size. This indicates qubit swapping can act as an essential component for GPU to efficiently simulate near-term scaled circuits.

Overall, my work indicated the direction to build state-of-art multi-GPU quantum computer simulators. Through my work, the scaling on distributed-GPU simulators improved considerably, contributing to the future quantum computer simulator ready for larger qubit sizes. The techniques and algorithms used can be quickly adopted by mainstream simulators, making GPU even more viable for quantum circuit simulation than it already is.

Future work. With respect to above considerations, I had two things come to mind that can produce further study on the topic of improving the scalability of multi-GPU simulation. One being the internal data structure for state-vector representation. Currently I used struct of array (SOA) for my state-vector, the data accessing pattern may be further benefited from data structure in the form of AOSOA. Furthermore, the limitation on GPU memory can be partially resolved by storing excessive states in host memory, allowing for much larger qubit sizes to be simulated using GPUs, the device to host connection from the latest generation of NvLink could resolve the communication bottleneck considerably. Both these directions extended from this study have the potential to further boost multi-GPU simulation's capabilities, and contribute to the development of quantum computation.

Bibliography

[1] Pablo Andrés-Martínez and Chris Heunen. Automated distribution of quantum circuits via hypergraph partitioning. *Physical Review A*, 100(3), September 2019.

[2] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 10 2019.

[3] Harun Bayraktar, Ali Charara, David Clark, Saul Cohen, Timothy Costa, Yao-Lung L. Fang, Yang Gao, Jack Guan, John Gunnels, Azzam Haidar, Andreas Hehn, Markus Hohnerbach, Matthew Jones, Tom Lubowe, Dmitry Lyakh, Shinya Morino, Paul Springer, Sam Stanwyck, Igor Terentyev, Satya Varadhan, Jonathan Wong, and Takuma Yamaguchi. cuquantum sdk: A high-performance library for accelerating quantum science, 2023.

[4] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3), September 2019.

[5] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, January 2007.

[6] Jun Doi and Hiroshi Horii. Cache blocking technique to large scale quantum computing simulation on supercomputers. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, October 2020.

[7] Georgii Evtushenko. Multi-gpu programming. `https://medium.com/gpgpu/multi-gpu-programming-6768eeb42e2c`, 7 2020. Accessed: insert access date here.

[8] Jennifer Faj, Ivy Peng, Jacob Wahlgren, and Stefano Markidis. Quantum computer simulations at warp speed: Assessing the impact of gpu acceleration, 2023.

[9] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.

[10] Tianyu Feng, Siyan Chen, Xin You, Shuzhang Zhong, Hailong Yang, Zhongzhi

Luan, and Depei Qian. dgquest: Accelerating large scale quantum circuit simulation through hybrid cpu-gpu memory hierarchies. In *Network and Parallel Computing: 18th IFIP WG 10.3 International Conference, NPC 2021, Paris, France, November 3–5, 2021, Proceedings*, page 16–27, Berlin, Heidelberg, 2021. Springer-Verlag.

[11] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

[12] Jack D. Hidary. *Quantum Computing: An Applied Approach*. Springer Cham, 1 edition, 8 2019.

[13] Bai Jin Huang, YeQi. joyce-quantum. `https://github.com/GPU4Science/joyce-quantum.git`, 2024.

[14] Thomas Häner and Damian S. Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17. ACM, November 2017.

[15] Sergei V. Isakov, Dvir Kafri, Orion Martin, Catherine Vollgraff Heidweiller, Wojciech Mruczkiewicz, Matthew P. Harrigan, Nicholas C. Rubin, Ross Thomson, Michael Broughton, Kevin Kissell, Evan Peters, Erik Gustafson, Andy C. Y. Li, Henry Lamm, Gabriel Perdue, Alan K. Ho, Doug Strain, and Sergio Boixo. Simulations of quantum circuits with approximate noise using qsim and cirq, 2021.

[16] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[17] Tyson Jones, Anna Brown, Ian Bush, and Simon C. Benjamin. Quest and high performance simulation of quantum computers. *Scientific Reports*, 9(1), July 2019.

[18] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, January 2020.

[19] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 836–838, 2008.

[20] Adriano Maron, Renata Reiser, Mauricio Pilla, and Adenauer Yamin. Expanding the VPE-qGM Environment Towards a Parallel Quantum Simulation of Quantum Processes Using GPUs. *CLEI Electronic Journal*, 16:3 – 3, 12 2013.

[21] David McMahon. *Quantum Computing Explained*. Wiley-IEEE Press, 2008.

[22] Timothy Prickett Morgan. The system bottleneck shifts to pci-express. `https://example.com/the-system-bottleneck-shifts-to-pci-express`, 7 2017. Accessed: insert access date here.

[23] Hongkang Ni, Haoya Li, and Lexing Ying. On low-depth algorithms for quantum phase estimation. *Quantum*, 7:1165, November 2023.

[24] NVIDIA Corporation. Distributed index bit swap api. `https://docs.nvidia.com/cuda/cuquantum/latest/custatevec/distributed_index_bit_swap.html`, 2024. In cuStateVec: A High-Performance Library for State Vector Quantum Simulators. Accessed: [Your Access Date Here].

[25] James C. Phillips, David J. Hardy, Julio D. C. Maia, John E. Stone, João V. Ribeiro, Rafael C. Bernardi, Ronak Buch, Giacomo Fiorin, Jérôme Hénin, Wei Jiang, Ryan McGreevy, Marcelo C. R. Melo, Brian K. Radak, Robert D. Skeel, Abhishek Singharoy, Yi Wang, Benoît Roux, Aleksei Aksimentiev, Zaida Luthey-Schulten, Laxmikant V. Kalé, Klaus Schulten, Christophe Chipot, and Emad Tajkhorshid. Scalable molecular dynamics on CPU and GPU architectures with NAMD. *The Journal of Chemical Physics*, 153(4):044130, 07 2020.

[26] Michael Reck, Anton Zeilinger, Herbert J. Bernstein, and Philip Bertani. Experimental realization of any discrete unitary operator. *Phys. Rev. Lett.*, 73:58–61, Jul 1994.

[27] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[28] Alok Shukla and Prakash Vedula. A generalization of bernstein–vazirani algorithm with multiple secret keys and a probabilistic oracle. *Quantum Information Processing*, 22(6), June 2023.

[29] Mikhail Smelyanskiy, Nicolas P. D. Sawaya, and Alán Aspuru-Guzik. qhipster: The quantum high performance software testing environment, 2016.

[30] D. Steinkraus, I. Buck, and P.Y. Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120 Vol. 2, 2005.

[31] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on gpus: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149, 2014.

[32] Chen Zhang, Zeyu Song, Haojie Wang, Kaiyuan Rong, and Jidong Zhai. Hyquas: hybrid partitioner based quantum circuit simulation system on gpu. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 443–454, New York, NY, USA, 2021. Association for Computing Machinery.