Approaches to Improve the Execution Time of a Quantum Network Simulation

THESIS

Joseph A. Tippit, First Lieutenant, USAF

AFIT-ENG-MS-21-D-013

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-21-D-013

Approaches to Improve the Execution Time of a Quantum Network Simulation

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Joseph A. Tippit, B.S.E.E.

First Lieutenant, USAF

December 23, 2021

AFIT-ENG-MS-21-D-013

Approaches to Improve the Execution Time of a Quantum Network Simulation

THESIS

Joseph A. Tippit, B.S.E.E.
First Lieutenant, USAF

Committee Membership:

Douglas D. Hodson, Ph.D.
Chair

Maj Richard Dill, Ph.D.
Member

Michael R. Grimaila, Ph.D., CISM, CISSP
Member

Gerald B. Baumgartner, Ph.D.
Member

AFIT-ENG-MS-21-D-013

Abstract

Evaluating quantum networks is an expensive and time-consuming task that benefits from simulation. The scale of computation, however, grows exponentially in relation to the input. In order to better enable the study of such networks, it is desired to have a software framework that is both accurate and performant. A potential improvement is to utilize graphics processing units (GPUs), namely by leveraging NVIDIA's programming framework, CUDA. To avoid performance pitfalls of higher level languages and programming models such as the so called "two language problem," the Julia Programming Language provides the basis for the development effort [1]. This research develops a prototype quantum network simulation framework using GPUs and Julia. Performance of the software is measured and compared against other languages such as MATLAB. In one evaluation, this research simulates the Hong-Ou-Mandel (HOM) Interference Experiment, whereby the coincidence rate of two entangled photons is measured. We consider this an important baseline for further research and development of a successful GPU-accelerated quantum simulation framework due to the experiment's relation to many core quantum networking concepts. A two-module framework is developed, one providing the basic simulation implementation and a second that extends the former with GPU compatibility. This allows developers and quantum researchers to utilize the same framework with consistent syntax regardless of whether they have a CUDA-compatible GPU.

AFIT-ENG-MS-21-D-013

To my wife. Without your love, companionship, and support, I would have never seen the completion of this work.

To my brothers of Theta Tau. You all are, and always will be, my family.

Acknowledgments

I would like to thank my advisor, Dr. Hodson, for all his direction, advice, and fascinating conversations throughout my time at AFIT. He took me on as a part time student even though you already had a busy schedule, and you saw me through to the end of these three years. More often than not, you were able to show me that I was overthinking things and that there was a clear path ahead.

I would also like to thank my sponsors and research partners at the Laboratory For Telecommunication Sciences and the University of Maryland. I would have found myself lost many times without your patient explanations and feedback.

Lastly, I would like to thank my wife for putting up with my countless rants and stress. On so many occasions, you were the reason I was able to make it through. You were a foundation in a turbulent time in my life, and I cannot express to you enough how much I love, cherish, and appreciate you for it.

Table of Contents

I. Introduction

The Department of Defense (DoD) has expressed interest in the field of quantum science [2, 3]. An investment in quantum capabilities will provide the warfighters and decision makers with revolutionary technologies that would provide a competitive edge in a post quantum epoch. Due to significant investment in time and resources required to perform quantum experiments, the DoD has invested in to further the field of study.

The introduction of quantum networks and computing is a significant security threat to any non-quantum enabled adversary. With the promise of securing communication protocols, classic cryptographic breaking prime number factorization, reaching the quantum level has become the arms race of the 21st century [4, 5]. In order to not fall behind in this field, it has become ever more essential to enable the efficient study, development, and experimentation of quantum networks.

This research aims to address these objectives by accelerating the speed at which quantum optical simulations can be performed while maintaining a high degree of accuracy in the output. Through examining core quantum experiments and processes, ways to improve performance are developed. Using these results, software is developed to perform the calculations on graphics processing units (GPUs) to further improve execution runtime.

1.1 Problem Statement

The set up and execution of quantum experiments is an expensive and time-consuming task [6]. The ability to instead perform accurate simulations of these phenomena would be advantageous. However, many existing quantum optics simulation frameworks and libraries either fail to provide a sufficient accuracy or require substantial runtime for quantum systems larger than a couple qubits. Quantum software such as SimulaQron is not aimed at accurate time dependencies, error correcting code, or the study of noise [7]. For this, The Network Simulator for Quantum Information using Discrete events (NetSquid) [8] is recommended. NetSquid is proprietary, however, and an open source alternative would be preferred. Another such library often discussed in the literature is the Quantum Toolbox in Python (QuTiP) [9, 10]. QuTiP, generally favored for its performance and open-source nature, is often benchmarked against by newer libraries.

Our research has selected the Julia Programming Language as the basis to develop performant quantum network simulations. Within the Julia ecosystem of libraries, there are competitors to QuTiP in terms of performance. Both Julia-based libraries, QuantumInformation.jl and QuantumOptics.jl, leverage Julia's capability to write code that supports the use of multiple dispatch, LLVM, and Just-In-Time (JIT) compilation [11, 12].

Larger systems are necessary in order to simulate anything more complex than the most trivial quantum networks. Considering the magnitude of computation grows exponentially as the number of qubits are increased, a rather large barrier is placed. GPUs, however, offer a potential solution, but none of the common quantum simulation frameworks currently available take advantage of GPU acceleration. We define our work as experimenting with the use of the Julia Programming Language to develop a quantum simulation library that is intentionally designed from the ground

up with the use of GPUs in mind. This library should be modular and open-source where possible, and it should ensure the speed and accuracy of the simulation in an easy to use framework. Physicists, and other researchers, should be able to write code as close to normal quantum notation as possible. It is desired for this framework to be well-documented, accurate, performant, and scalable in order to enable the study of real-world quantum networks and phenomena efficiently and cost effectively.

1.2 Research Objectives

The first objective is to compare the benchmarks across multiple versions of a previously written MATLAB script that simulates the effects associated with Hong-Ou-Mandel (HOM) Interence. These benchmarks provide an important baseline to develop and compare further work against. The next objective will be to re-implement the effects of this MATLAB-based simulation in the Julia Programming Language and the execution runtimes compared. These efforts will then lead into the main objective of developing a prototype framework which leverages the execution performance of GPUs to perform calculations.

Progress will be measured in the number of quantum experiments we are able to successfully simulate with the framework, as well as by measurable execution performance improvements. The prototype framework should be able to at least successfully simulate the HOM Interference and Mach-Zehnder experiments.

1.3 Organization

This thesis is organized in a scholarly article format with three papers to be submitted for publication. Chapter II is a published conference paper presented at The 2020 World Congress in Computer Science, Computer Engineering, and Applied Computing [13]. In it, an overview of the Julia Programming Language as well as the

concept of containerization focusing on the high performance computing (HPC) container engine Singularity is provided. The motivation for using the two in conjunction to leverage the computational benefits of GPUs in our research is then established.

Chapter III reprints the second paper. The discussion in this paper focuses on the first real attempted use of the Julia Programming Language to perform simulation of quantum computation in this research. The problem area centered on using Julia's multi-threading capabilities to simulate quantum process tomography. To better understand how robust the language is in this area, we execute our simulation on the Air Force Research Laboratory's supercomputer Mustang. The results of this experiment show that the version of Julia used was able to maintain close to theoretical performance until a significant number of threads, at which point performance was lost.

Chapter IV is our third paper. This paper analyzes and compares an initial conversion of a pre-existing MATLAB script that simulated HOM interference to Julia. After this initial conversion, techniques used to improve runtime and memory usage are discussed. These techniques are applied to the Julia version, and the results are compared to the initial attempt. It is shown that in some cases, Julia is indeed faster than MATLAB.

Chapter V concludes with a summary of the contributions made to the field, the current status of the library in the development, and gives proposals for future work.

Finally, the appendices document the current state of the two-module framework developed as a result of this research. Appendix A focuses on encapsulating documentation and design philosophy of the overall framework. Appendix B provides code listings and examples of the framework. Lastly, Appendix C contains framework benchmarks of the HOM Interference and Mach-Zehnder experiment simulations.

II. Julia and Singularity for High Performance Computing

The following conference paper was presented at The 2020 World Congress in Computer Science, Computer Engineering, & Applied Computing. It was accepted for publication and was included in Springer's Advances in Parallel & Distributed Processing, and Applications [13, 14].

Julia and Singularity for High Performance Computing

Joseph Tippit
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
joseph.tippit@afit.edu

Douglas Hodson, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
douglas.hodson@afit.edu

Michael Grimaila, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
michael.grimaila@afit.edu

***Abstract*—High performance computing (HPC) is pivotal in the advancement of modern science. Scientists, researchers, and engineers are finding an increasing need to process massive amounts of data and calculations faster and more accurately than ever before. This is especially true in our work of developing a general quantum library for researchers to use in their simulations. Much of this effort revolves around getting the maximum performance enhancements offered by GPUs as possible. We have found that the relatively newer programming language Julia has offered us a productive means of development with minimal overhead. Combined with the container engine Singularity, we can ensure maximum distributability and reproducibility.**

***Index Terms*—high performance computing, GPUs, Julia, container engine, Singularity**

I. Introduction

Our research team is focusing on developing a software suite of tools to simulate quantum systems, specifically in regards to quantum teleportation. Our goal is to create a library general enough for researchers to be able to apply our software to many different quantum problems rather than one specific one and to keep it as open source and distributable as possible. Due to the high level of computation needed to fully model these systems, code run-times can easily and exponentially be driven upwards as the matrices involved become increasingly larger.

High level, dynamic languages such as Python, despite their benefits in ease of use and readability, simply do not offer the speed we require. However, lower level languages such as C offer considerably less flexibility and greater difficulty in developing and maintaining code. As a middle grounds to this, we have chosen to use the relatively newer programming language Julia. Julia, while being a dynamic language, was developed with speed in mind, targeting researchers and data scientists hoping to get as much performance out of their code as possible while still maintaining inherit readability and ease of use. This made it an obvious first choice for our research.

Also, due to the high level of matrix calculations involved in quantum mechanics, our research can benefit greatly from the performance gains offered by running as much of our code as possible on GPUs rather than the traditional CPU. Julia offers an extensive amount of support in this area, as it has native GPU programming capabilities offered by the CUDAnative.jl library, as well as multiple other libraries for support. Combined with Julia's Just-In-Time Compiler, these libraries offer a great level of efficiency in the kernel launch sequence.

In keeping distributability in mind, we have also opted to develop our library inside of the container engine Singularity. Collaborators can be limited by local security practices and administrative privileges needed to install dependencies required to run the software and code of others. This is combined with the need to maintain version control of software libraries fundamental to their own workflow. Containerization as a technology has risen to meet these needs. Containers offer similar benefits to virtual machines such as managing library dependencies and running multiple, isolated operating systems (OS) from the same machine. They do, however, have certain advantages more critical to our work.

Specifically, Singularity makes use of a definition file, where software such as the operating system and essential libraries are defined. This allows us to share our work with others, ensuring all versions and libraries will exactly match our own without interfering with their workflow. This also offers a kind of version control for our work and makes it easier to develop on one machine and execute on another. All that is required is to simply build the container from the definition file, and everything that is required will be installed without having to worry about system administration. Companies such as NVIDIA also offer large repositories of containers pre-built to meet many different needs, allowing us and other researchers to further focus on our research.

Another key benefit we see in using containers is their ability to directly share the kernel of the host OS without the need for a hypervisor, defined later, as is required for a virtual machine. This allows them to directly access the resources of a physical machine with minimal overhead. This is crucial, as much of our code will revolve around utilization of GPUs and getting as much of their speed up as possible. Combined with the definition file, containers only need to install what is absolutely essential to our workflow, sharing everything else with the host OS. This is in contrast to virtual machines, which need to install and run a full OS, requiring more overhead.

Singularity has also been developed with researchers in mind, assuming no administrative privileges and targeting high performance computing. These reasons have made it ideal for our research. Throughout the course of this paper, we will provide further justification for why we chose both Singularity

and Julia as fundamental tools to our work.

II. The Julia Programming Language

Traditionally, high level, dynamic languages have lagged behind lower level, static languages in terms of performance. The emphasis on readability, ease of use, and productivity is believed to come at the cost of run-times and execution speeds. Prototyping is thus done in a high level language and then fully implemented in a low level language for speed.

With Julia, this is not the case. From its initial development, the Julia Programming Language was designed with speed in mind while still offering the same productive programming of a high level language. In a blog post about why they created it, Julia's developers are quoted as saying, "We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell" [1].

By examining Julia's Pythonic syntax with natural mathematical notation, as well as the micro-benchmarks in Figure 1 and 2, we can easily see that they have achieved their goals. We will further examine three key features that lend themselves to this success:

- Julia's Just-In-Time Compiler and Type System
- Low-Level Virtual Machine (LLVM)
- Native GPU Support

A. Just-In-Time Compilation

Just-In-Time (JIT) compilation is a technique that converts high level languages into machine code executable directly on the CPU when it is run [3]. This means that Julia, unlike Python, is a compiled language. This is a benefit for its speed, as it drops the overhead of interpretation. The difference between a language such as C, however, is that the code is compiled at run-time rather than beforehand. To adequately benchmark Julia run-times, it is therefore necessary to run

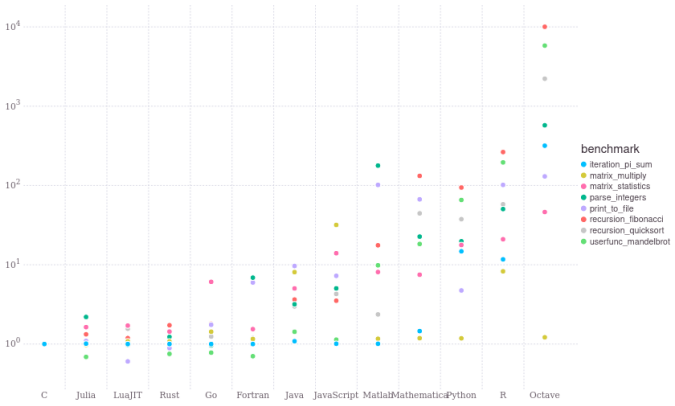


Fig. 1. Julia Micro-Benchmarks [2]

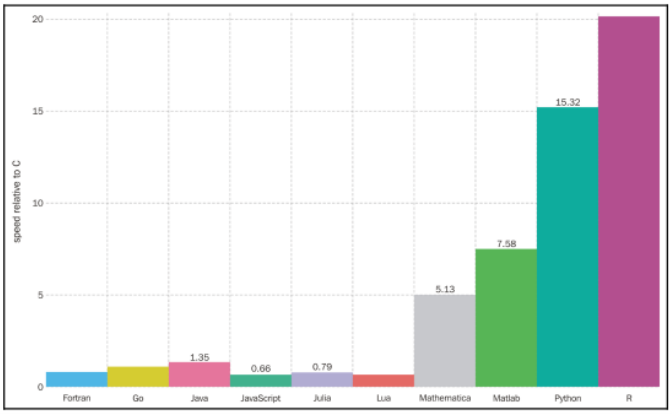


Fig. 2. Language Speedups Relative to C [3]

the code once so that it is compiled, and then a second time to benchmark. Otherwise, you will also time the overhead of compiling the code as well.

We can see the steps involved in the Julia compilation process in Figure 3 below. Julia also offers macros to allow the programmer to see the output of each step. For example, if one is interested in seeing the LLVM bitcode output, the @code_llvm macro could be placed in front of the desired function. If, instead, one wanted to see the actual assembly that Julia compiled to, use the @code_native instead. These macros can give the programmer insight into where optimizations in their code can be made.

The speed ups offered by Julia are also largely due to multiple dispatch. Essentially, this is a method whereby multiple functions are automatically created to perform the same operation, and one is selected based on the types of the

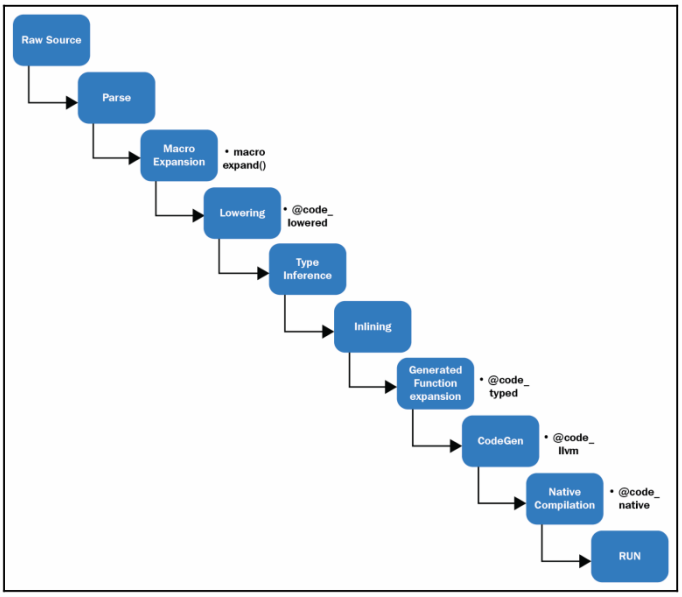


Fig. 3. Julia Compilation Process [3]

individual inputs. This allows the compiler to make certain optimizations and improvements to each function without the user needing to manually define the same operation for every combination of possible input types.

Adding two numbers together is an excellent example of this. On the computer hardware, the value 1.0 is stored uniquely different from 1. The former is stored as a float and the latter as an integer. Summing these together would output 2.0, a float. If instead, both had been kept integers, the resulting output would have also been an integer.

A central idea behind multiple dispatch is type stability. Type stability is the concept that the type of a return value depends only on the types of the individual inputs and not on their values. It is, ultimately, what makes multiple dispatch work by allowing the compiler to choose the correct function to perform the desired operation. The book, Julia High Performance Second Edition by Avik Sengupta has a simple example of this highlighted in the code examples below [3].

```
function pos(x)
    if x < 0
        return 0
    else
        return x
    end
end
```

We can see that inputting float 2.5 will return the float 2.5. However, -2.5 will return zero as an integer. This is type instability. Julia offers many ways to identify and fix type instability, however, Julia's compiler has been optimized to make even code with type instabilities execute almost as fast as code without. The ability to focus more on writing our code while letting the compiler worry about the lower level optimizations is a huge benefit to our work.

B. LLVM

Prior to being passed off to the JIT compiler, Julia uses the compiler infrastructure LLVM to convert its syntax into an intermediate representation in memory. This syntax allows LLVM to make different optimizations before being compiled to machine code [5]. These optimizations are implemented as passes in LLVM, and there are three different categories: Analysis, Transform, and Utility Passes [6].

In the Analysis passes, information is collected for other passes to use in regard to debugging and program visualization, and Transform passes will all mutate the program in some way. Utility passes is a categorical catchall for passes that have some utility but do not fit into the other two [6]. The final output is the intermediate representation, which is a low level language similar to assembly. As previously mentioned, it can be viewed by adding the @code_llvm macro to a function. Figure 4 shows a high level overview of this process.

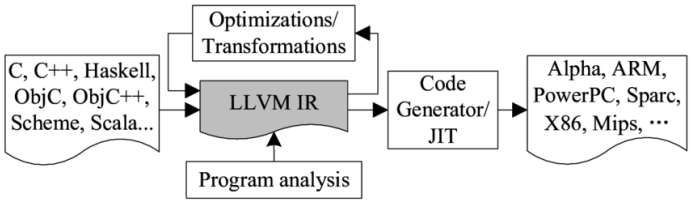


Fig. 4. High Level LLVM Overview [4]

C. Native GPU Support

One of the most influential reasons in using Julia for our research is its native GPU support. Julia has multiple libraries for utilizing the GPU at differing levels of abstraction, as shown in Figure 5. As we will be using NVIDIA GPUs, CUDA support is of the greatest interest to us. Using the CUDAnative.jl library, in conjunction with CUDAdrv.jl and CuArrays.jl, we are capable of doing the same low level GPU programming that could be done in CUDA C++. These libraries provide a means of interfacing with the CUDA driver and run-time libraries, writing kernels, and managing execution [7].

These libraries integrate into Julia's JIT compiler, allowing the code to be compiled directly to GPU assembly. Ultimately, what this means is that you can use Julia for the GPU almost exactly how you would for the CPU. This gives the programmer the same productivity Julia offers and combines it with the inherent parallelism of the GPU, generating efficient PTX code [7]. Similarly to how we can view the LLVM intermediate representation and native assembly code, we can also view the PTX generated by the LLVM PTX compiler backend using the @device_code_sass macro.

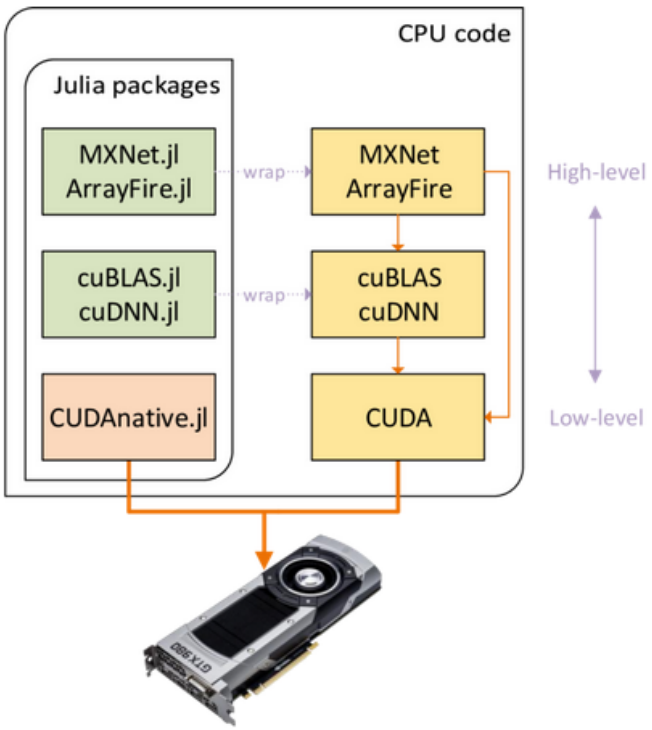


Fig. 5. Julia GPU libraries at differing levels of abstraction [7]

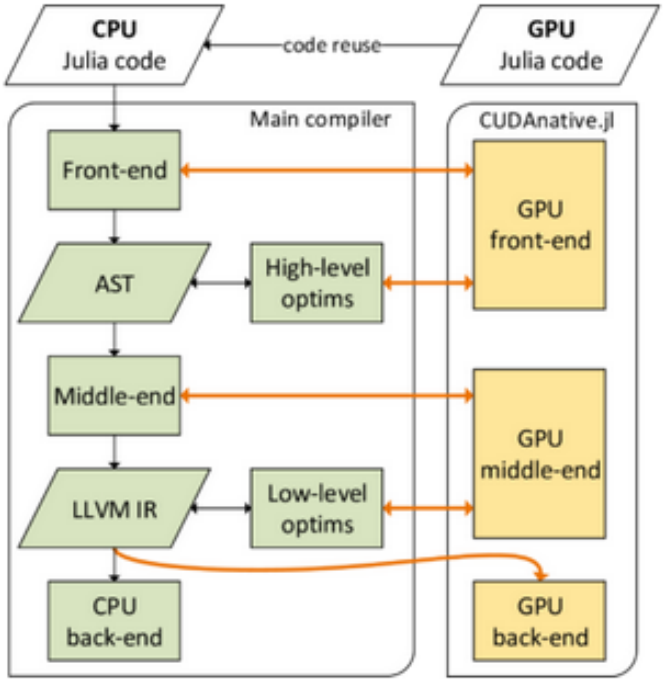


Fig. 6. Overview of the CUDAnative.jl compiler [7]

Additionally, the Julia team has been porting the Rodinia, a benchmark suit for heterogeneous computing, to Julia. The Rodinia benchmark suite works by measuring parallel communication patterns, synchronizations techniques, as well as power consumption in order to provide a standard benchmark to compare platforms [8]. We can see how Julia compares against CUDA C++ in Figure 7.

CUDAnative.jl can be considered the basic library essential to using CUDA in Julia, however, CuArrays.jl is arguably the most signficant. GPU code must be vectorized in order to gain

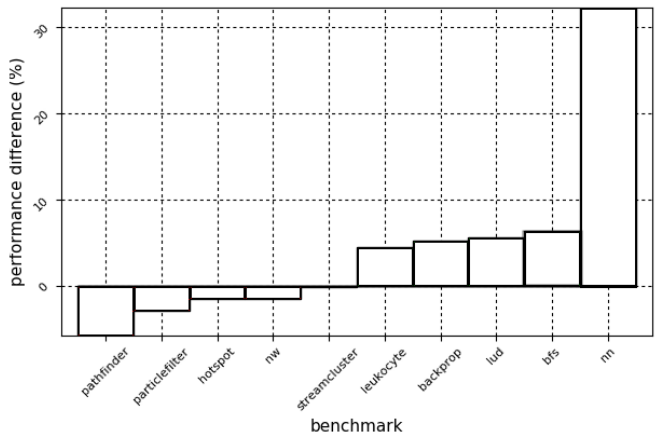


Fig. 7. Performance difference between CUDA C++ and CUDAnative.jl using the Rodinia Benchmark [7]

any performance increases, and the array is the fundamental type for this. The CuArrays data type allows arrays to be created for use on the GPU just like any other array in Julia [3]. Overall, these libraries make Julia just as useful and powerful for programming on the GPU as C++ while still keeping in line with a high level language paradigm. Intuitive mathematical syntax, a highly optimized compiler, and the ability to make low level GPU abstractions have given Julia a competitive edge in high performance computing as well as making it the decisivie choice for our research.

III. Containerization

A. Differences between Containers and Virtual Machines

Virtualization developed as a means of meeting the demands of shared resources amongst a large collection of users [9]. Universities and companies alike utilize virtual machines (VMs) to provide their students and employees with a way of accessing the organization's computer resources remotely from a shared server. This can reduce the costs of having to provide users these same resources physically.

Conceptually, virtualization is an abstraction of the hardware and resources of a physical machine from the operating systems and software running in a virtual machine. Fundamental to this is the software called the hypervisor. A hypervisor, or virtual machine monitor (VMM), isolates the operating system and resources on the physical machine (the host) from the virtual machine (the guest) and oversees all VMs on the host. Users can create many VMs on one machine, allocating different amounts of resources such as memory and storage to each. The hypervisor sees these resources as a pool and manages their utilization amongst every running VM as needed [10].

Essentially, the virtual machine exists on the host as a folder. This allows it to be easily moved and copied around. The image containing the VM includes a complete operating system and its corresponding applications, allowing for different operating systems and environment regardless of what the host OS is. However, this comes at the price of storage space and having potentially significant overhead in terms of CPU resources and RAM [11].

In contrast to this, containers exist on the host OS as either a file or collection of files, allowing them to be highly portable and configurable. Since containers share the same kernel as the host OS, they drop the need for a hypervisor and have direct access to system resources without needing to emulate them. We show a high level view of this difference in Figure 8.

Containers also come with just the necessary run-time files, dependencies, and software to run the required software. Due to this, they are more lightweight than a VM and will run natively on a Linux operating system while still isolating their applications from the OS [12]. The file system is also isolated and will only mount certain prescribed and user directed directories from the host OS. Other than these directories, the file system paths and the files they hold will be different from the host.

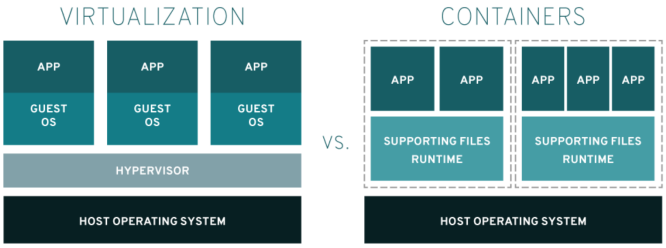


Fig. 8. High Level View of Containers vs Virtualization [12]

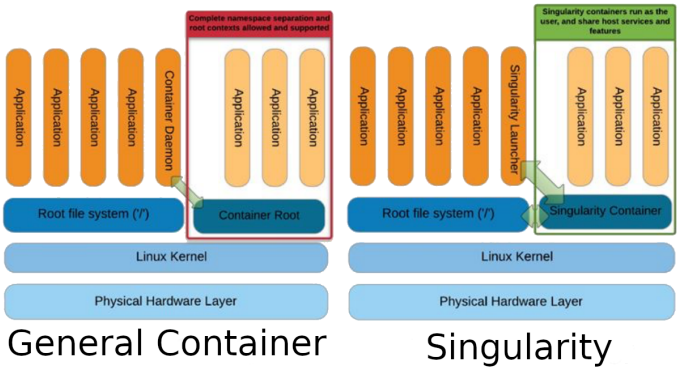


Fig. 9. General Container Architecture compared to Singularity [15], [16]

Both Singularity and Docker, another popular container engine, set up their container environments from a definition file or Dockerfile, respectively. This allows users to maintain a level of version control when sharing their work. These files define the operating system environment and library/software requirements of the application being contained, giving collaborators a straightforward means to ensure all dependencies will be exactly what the original developer intended without needing to have a system administrator install anything. Since containers are also isolated from the host, they will run as just another process and will not interfere with any other workflow or library dependencies.

B. Docker versus Singularity

Docker is one of the most widely known and used container engines available today. There are a plethora of options available to researchers to immediately pull a container suitable to their needs and begin working. NVIDIA's own GPU Cloud, a hub for high performance computing container images, hosts their work environments natively as Docker images.

However, Docker was designed to be an enterprise focused container. No HPC center allows it as it was not intended to support highly distributed parallel applications [13]. Docker also assumes administrative or root privileges and runs applications as such. Not only does this not solve the problem of sharing our work (if collaborators do not have the proper local privileges, they will not be able to run our containers), but this also causes potential security risks as well. Since applications are run as root, any user accessing the container can also be granted the same escalated privileges.

Singularity, on the other hand, assumes no privileges whatsoever and was specifically designed to support HPC and MPI applications. This no trust model makes it ideal for sharing work amongst researchers and other collaborators, as everything run in the image will be executed with the same privileges as the user. The run-time writes the UID and GID information to the files within the container, so the privileges are the same because the user is the same. Singularity has a slightly different philosophy from Docker, one of "Integration over Isolation" [14]. This means it supports being able to map more directories from the host operating system into the guest, integrating and embedding it directly into your workflow. Overall, Singularity is a paradigm targeted at scientific workloads to address the core missions of Mobility of Compute, Reproducibility, HPC support, and Security [14].

It also supports converting Docker containers directly to Singularity containers in a single command (it will even pull directly from Docker Hub and NVIDIA GPU Cloud, as well as its own Singularity Hub), which means every image on the NVIDIA GPU Cloud will also be supported on Singularity as well. Taken collectively, and in keeping in line with other HPC workflows, we have found Singularity to be the best suited container engine for our applications.

C. Singularity and GPUs

Singularity, by default, makes all host devices accessible to the container. This provides for a seamless integration with GPUs and other devices common to HPC. In fact, Singularity comes stock with command line options to control the usage of GPUs, choosing which to run and when [14].

As previously mentioned, NVIDIA offers many different container options for researchers to pull and immediately begin working. These images come set up with everything one needs to access their GPUs from within a container with similar performance to what one would expect from the host (see Figure 10 below), and much of NVIDIA's documentation covers using them from within Singularity [17]. Singularity even has direct access to the host's driver libraries. These reasons led us to choose Singularity as the container engine of choice for our research.

CONCLUSION

Performance and productivity are two areas crucial to our research. The need to perform massive matrix calculations efficiently have turned us to utilizing GPUs to leverage their inherent parallelism. We have found the programming language Julia, with its highly optimized compiler and native GPU support, to be the ideal basis on which to develop our quantum libraries. Providing us with a high level language paradigm combined with speeds comparable to C, as well as essentially equivalent support for CUDA as C++, it was a clear choice as a way forward.

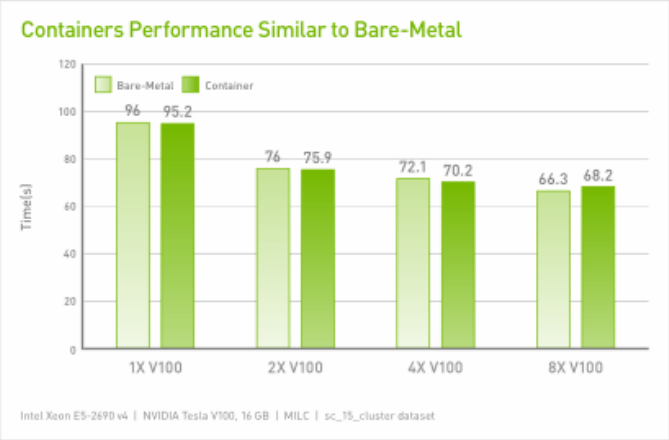


Fig. 10. NGC Performance: Containers vs. Bare-Metal [17]

However, distributability and reproducibility were two other key factors, leading us to delve into container technology as a solution. Designed with HPC and direct host resource access in mind, Singularity provides us with a means to share our work to the highest degree possible with minimal performance impacts.

REFERENCES

[1] J. Bezanson, S. Karpinksi, V. B. Shah, and A. Edelman, "Why We Created Julia," Feb. 2012. [Online]. Available: https://julialang.org/blog/2012/02/why-we-created-julia/. [Accessed Mar. 17, 2020].

[2] The Julia Project, *Julia 1.3 Documentation*, Aug. 2019. [Online]. Available: https://docs.julialang.org/en/v1/. [Accessed Mar. 17, 2020].

[3] A. Sengupta, *Julia High Performance*, 2nd ed. Birmingham, UK: Packt Publishing, 2019.

[4] J. Zhao, "Formalizing the SSA-Based Compiler for Verified Advanced Program Transformations," Ph.D. dissertation, Dept. Comp. and Info. Science, Univ. of Pennsylvania, Philadelphia, PA, 2013.

[5] R. Lakhanpal and A. Joshi, *Learning Julia*, Birmingham, UK: Packt Publishing, 2017.

[6] LLVM Project, *LLVM Documentation*, Mar. 2020. [Online]. Available: https://llvm.org/docs/. [Access Mar. 19, 2020]

[7] T. Besard, "High-Performance GPU Computing in the Julia Programming Language," Oct. 2017. [Online]. Available: https://devblogs.nvidia.com/gpu-computing-julia-programming-language/. [Accessed Mar. 18, 2020].

[8] G. Honan, S. Shivakumar, and A. Siraman, "Rodinia Benchmark Suite," Univ. of Pennsylvania, PA, Tech. Rep., Mar. 2017.

[9] Oracle, "Introduction to Virtualization", Jan. 2013. [Online]. Available: https://docs.oracle.com/. [Access Mar. 18, 2020].

[10] Red Hat, "What is a Hypervisor?", 2020. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor. [Accessed Mar. 17, 2020].

[11] A. Strong, "Containerization vs. Virtualization: What's the Difference," Nov. 2019. [Online]. Available: https://www.burwood.com/blog-archive/containerization-vs-virtualization. [Accessed Mar. 17, 2020]

[12] Red Hat, "What's a Linux Container?," 2020. [Online]. Available: https://www.redhat.com/en/topics/containers/whats-a-linux-container. [Accessed Mar. 17, 2020].

[13] M. Kandes, "An Introduction to Singularity: Containers for Scientific and High-Performance Computing," Univ. of California, San Diego, CA, Tech. Rep., Feb. 2019.

[14] Sylabs Inc., *Singularity Admin Guide*, 2020. [Online]. Available: https://sylabs.io/guides/3.5/admin-guides/. [Accessed Mar. 19, 2020].

[15] "Docker vs Singularity vs Shifter in an HPC environment," 2016. [Online]. Available: http://geekyap.blogspot.com/2016/11/docker-vs-singularity-vs-shifter-in-hpc.html18. [Accessed Mar. 19, 2020].

[16] E. Bollig, "Singularity & Containers," Univ. of Minnesota, MN, Tech. Rep., Nov. 2019.

[17] NVIDIA, "Optimized Containers from NVIDIA GPU Cloud," MO, Tech. Rep., 2018.

III. Parallelization of Quantum Process Tomography Computations with 2 Qubits

The following paper, "Parallelization of Quantum Process Tomography Computations with 2 Qubits," is a planned submission to the Journal of Supercomputing.

Parallelization of Quantum Process Tomography Computations with 2 Qubits

Joseph Tippit
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
joseph.tippit@afit.edu

Laurence Merkle, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
laurence.merkle@afit.edu

Douglas Hodson, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
douglas.hodson@afit.edu

***Abstract*—The simulation of computation is important as the study of quantum computers evolves. Unfortunately, this exciting new area necessitates a quantum computer to examine the benefits offered over classical computing. For now, researchers will continue to use available tools to study and understand quantum computer capabilities. A library of essential components and protocols to model quantum computation is therefore desired to allow researchers the ability to quickly and efficiently develop software to meet their simulation needs.**

Simulating quantum phenomena is expensive, and the complexity and resources needed grow exponentially when modeling larger systems. However, the parallelism inherent to quantum computations allows us to see them as ideal candidates for which to develop parallel algorithms. With speedups subject to Amdahl's Law and software development efficiency subject to the programming language of choice, a balance must be found between the two [1]. To further examine this, the performance of applying parallel algorithm techniques to the simulation of quantum process tomography (a technique used to characterize a quantum system) is measured. The programming language Julia is utilized because of its native parallel computation support, easy-to-learn syntax, and inherent speed [2].

***Index Terms*—quantum computation, quantum computers, parallel algorithms, quantum process tomography, Julia**

I. Introduction

Our research team is focused on developing a software suite of tools to simulate quantum systems, with special consideration of teleportation. Our goal is to create a library general enough for researchers to be able to apply our software to many different quantum problems rather than developing towards one specific problem. Of particular interest to our team is leveraging parallel capabilities as much as possible.

One problem of interest is quantum process tomography, a means of characterizing quantum systems, and can be used for error and noise detection [3]. In any system, a number of things can go wrong that prevent it from performing or working as expected, and this is especially true when working in the quantum realm. Decoherence, or any other interference in the system, can produce errors in an output of a prototype, and detecting these errors is fundamental to a simulation. Thus, an important task for developing quantum networks is to be able to characterize the network [4]. With quantum process tomography's myriad amount of matrix operations, we see it as an ideal candidate for parallelization.

A procedure by which quantum process tomography can be performed is thus examined. Using this procedure, a means in which to parallelize this process is developed and applied to a controlled-NOT gate as an example.

II. Background

This section first describes some quantum theory concepts to provide a working understanding of the methodology outlined in later sections. Starting with the idea of a qubit, the concept of qubit gates is developed. The concepts presented here explain why quantum process tomography is an important task to model and simulate. Finally, this section concludes with an introduction to the Julia Programming Language and an overview of its benefits.

A. Qubits

Analogous to the bit in classical computing is the concept of a quantum bit (qubit) in quantum computation. A qubit, like a bit, has a binary state, and represents values 0 or 1. Quantum theory uses dirac notation (or bra-ket notation) and represents the states 0 and 1 in kets as $|0\rangle$ and $|1\rangle$ respectively. Any combination of two possible values to represent these states could have been used, but using $|0\rangle$ and $|1\rangle$ is special - these represent the computational basis states and form an orthonormal basis for the vector space [3].

Extending this concept to multiple qubits, a qubit register of length n could now be represented as a vector of size 2^nx1. This discussion focuses on a two qubit system, where states are represented as a combination of the individual qubit states. If both qubits are in state $|0\rangle$, in ket notation, it would be written as $|00\rangle$. More generally, the superposition of a single qubit system would be displayed as:

$$\alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where $|\alpha|^2 + |\beta|^2 = 1$ [5]. Therefore, a qubit will be in state $|0\rangle$ with probability $|\alpha|^2$ and state $|1\rangle$ with probability $|\beta|^2$. In vector form, this superposition is written as $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$. For example, a qubit known to be in the state $|0\rangle$ can be written as $\begin{bmatrix}1\\ 0\end{bmatrix}$ [3].

Extending this concept to multiple qubits, we could now represent a qubit register of length n as a vector of size 2^nx1. We focus on a two qubit system, which we represent a state as a combination of the individual qubit states. If both qubits are in state $|0\rangle$, in ket notation, we would write this as $|00\rangle$. More

generally, the superposition of a two qubit system would be displayed as:

$$a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle \tag{2}$$

where $|a|^2+|b|^2+|c|^2+|d|^2 = 1$ and the vector representation is $\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$ [5]. Using this notation, it is trivial to extend this system to more qubits.

The coefficients, which are amplitudes of the quantum systems, could take on any value - real or complex. This is also true for any quantum system.

For a system with n qubits, there are 2^n possible combinations of individual qubit states, each with an amplitude of x_i where $1 \leq i \leq 2^n$ and $x_i \in \mathbb{C}$. With this description, the notion of the bra (the conjugate-transpose of the ket) can be conveyed. For an n-qubit state, $|X\rangle = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{2^n} \end{bmatrix}$, the bra $\langle X| = \begin{bmatrix} X_1^* X_2^* \dots X_{2^n}^* \end{bmatrix}$. With these two notations, ket and bra, we can perform familiar linear algebra operations between two states $|X\rangle$ and $|Y\rangle$ such as the inner product $\langle X|Y\rangle$ and the outer product $|X\rangle\langle Y|$ [3].

B. Qubit Gates

Performing linear algebra operators on qubits allows for the introduction of the concept of a qubit gate. Just as there exists logic gates that operate on classical digital bits, there also exists this quantum counterpart. A qubit gate acting on a single qubit takes input as a 2x1 vector representing the state of the qubit and outputs a 2x1 vector representing the state of the output qubit. We visualize this as:

$$X \begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} = \begin{bmatrix} \alpha_o \\ \beta_o \end{bmatrix} \tag{3}$$

where i and o designate the α and β input and output respectively, and the matrix operator X represents the gate.

X must be a 2x2 matrix. In general, for a gate acting on n qubits, the matrix representing the gate must be of size 2^nx2^n; when the matrix is unitary, it is considered to represent a valid gate. That is, $X^\dagger X = I$, where $X^\dagger$ is the conjugate-transpose of X and I is the identity matrix [3].

Three useful single qubit gates to be familiar with are the NOT, Z, and Hadamard gates. Their matrix representations and operations are defined in Table I for an input of $\alpha\,|0\rangle + \beta\,|1\rangle$.

At this point, the only 2 qubit gate of concern is the CNOT gate. It takes as input two qubits, a control and target, and "flips" the state of the target qubit if the control is in state $|1\rangle$ [4]. Its classical computing counterpart is the XOR gate. Equation 4 shows its matrix and operation.

TABLE I
COMMON SINGLE QUBIT GATES [3]

Gate	Symbol	Matrix	Result
NOT	X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\beta\,\|0\rangle + \alpha\,\|1\rangle$
Z	Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\alpha\,\|0\rangle - \beta\,\|1\rangle$
Hadamard	H	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	$\alpha\frac{\|0\rangle+\|1\rangle}{\sqrt{2}} + \beta\frac{\|0\rangle-\|1\rangle}{\sqrt{2}}$

TABLE II
TRUTH TABLE FOR AN IDEAL CNOT GATE [5]

Input	Ouput
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

$$CNOT \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4}$$

C. Quantum Process Tomography

Quantum tomography, or quantum state tomography, is a procedure used to characterize an unknown quantum state by repeatedly preparing identical states and taking different measurements on them. This process is done in order to develop a "complete description" of the state being measured. Similarly, quantum process tomography is a procedure to describe the dynamics that a (potentially unknown) quantum system undergoes experimentally. Some applications include determining the performance of a quantum gate or measuring the effects of noise on a quantum system [3]. Any quantum operation can be written in the operator-sum representation where the sum is over k Kraus operators, E_1-E_k, for some input state ρ [4]:

$$\varepsilon(\rho) = \sum_k E_k \rho E_k^\dagger \tag{5}$$

For a d dimensional quantum system, d^2 quantum states $|\Psi_1\rangle, \dots\ |\Psi_{d^2}\rangle$ are required such that their density matrices $|\Psi_1\rangle\langle\Psi_1|, \dots, |\Psi_{d^2}\rangle\langle\Psi_{d^2}|$ form a basis set [3]. Each state is then subjected to the operation being characterized. The goal is to ultimately determine the E_k for ε with the condition that $\sum_k E_k^\dagger E_k \leq I$ [4].

These Kraus operators resultantly describe the quantum system, including its measurement and decoherence [4]. This discussion is focused on a two-qubit system (and thus d=2, requiring 4 input states). Reference [4] describes the entire process, which is the basis of our CNOT gate.

Let I, X, Y, and Z represent the Pauli Matrices, and $\rho^{(\alpha\beta)}$ denote a 4x4 matrix where all elements are zero with the

exception of a one at row α and column β. Let $\varepsilon(\rho)$ represent the quantum operation we are attempting to characterize [4]. With $\otimes$ denoting the tensor product, we have:

$$\lambda = (Z \otimes I + X \otimes X) \otimes (Z \otimes I + X \otimes X)/4 \tag{6}$$

$$P = I \otimes [\rho^{(11)} + \rho^{(23)} + \rho^{(32)} + \rho^{(44)}] \otimes I \tag{7}$$

$$K = P\lambda \tag{8}$$

The procedure for this case is thus summarized as needing to calculate the 16x16 χ matrix, where the ε are calculated in Equation 10, of the form:

$$\chi = K^T \begin{bmatrix} \varepsilon(\rho^{(11)}) & \varepsilon(\rho^{(12)}) & \dots \\ \varepsilon(\rho^{(21)}) & \varepsilon(\rho^{(22)}) & \dots \\ \vdots & \vdots & \ddots \end{bmatrix} K \tag{9}$$

We can determine each $\varepsilon(\rho^{(\alpha\beta)})$ by solving the matrix in Equation 10:

$$\begin{pmatrix} \rho^{(11)} \\ \rho^{(12)} \\ \rho^{(13)} \\ \rho^{(14)} \\ \rho^{(21)} \\ \rho^{(22)} \\ \rho^{(23)} \\ \rho^{(24)} \\ \rho^{(31)} \\ \rho^{(32)} \\ \rho^{(33)} \\ \rho^{(34)} \\ \rho^{(41)} \\ \rho^{(42)} \\ \rho^{(43)} \\ \rho^{(44)} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -a & -a & 1 & i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -a & 0 & 0 & 0 & -a & 0 & 0 & 0 & 1 & 0 & 0 & 0 & i & 0 & 0 & 0 \\ \frac{i}{2} & \frac{i}{2} & -a & a^* & \frac{i}{2} & \frac{i}{2} & -a & a^* & -a & -a & 1 & i & a^* & a^* & i & -1 \\ -a^* & -a^* & 1 & -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & -a & -a^* & \frac{1}{2} & \frac{1}{2} & -a & -a^* & -a^* & -a^* & 1 & -i & -a & -a & i & 1 \\ 0 & -a & 0 & 0 & 0 & -a & 0 & 0 & 0 & 1 & 0 & 0 & 0 & i & 0 & 0 \\ -a^* & 0 & 0 & 0 & -a^* & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -i & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & -a^* & -a & \frac{1}{2} & \frac{1}{2} & -a^* & -a & -a & -a & 1 & i & -a^* & -a^* & -i & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -a & -a & 1 & i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{-i}{2} & \frac{-i}{2} & -a^* & a & \frac{-i}{2} & \frac{-i}{2} & -a^* & a & -a^* & -a^* & 1 & -i & a & a & -i & -1 \\ 0 & -a^* & 0 & 0 & 0 & -a^* & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -i & 0 & 0 \\ 0 & 0 & 0 & 0 & -a^* & -a^* & 1 & -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \rho^{(HH)} \\ \rho^{(HV)} \\ \rho^{(HD)} \\ \rho^{(HR)} \\ \rho^{(VH)} \\ \rho^{(VV)} \\ \rho^{(VD)} \\ \rho^{(VR)} \\ \rho^{(DH)} \\ \rho^{(DV)} \\ \rho^{(DD)} \\ \rho^{(DR)} \\ \rho^{(RH)} \\ \rho^{(RV)} \\ \rho^{(RD)} \\ \rho^{(RR)} \end{pmatrix} \tag{10}$$

where $a = \frac{1+i}{2}$. For example: $\varepsilon(\rho^{(12)}) = -a\varepsilon(\rho^{(HH)}) - a\varepsilon(\rho^{(HV)}) + \varepsilon(\rho^{(HD)}) + i\varepsilon(\rho^{(HR)})$, such that the letter combination denotes a tensor product of two of the input states $\rho^{(H)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ $\rho^{(V)} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ $\rho^{(D)} = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ $\rho^{(R)} = \frac{1}{2}\begin{bmatrix} 1 & -i \\ i & 1 \end{bmatrix}$, e.g. $\rho^{(HR)} = \rho^{(H)} \otimes \rho^{(R)}$ [4].

D. The Julia Programming Language

Released in 2012, Julia is a programming language designed for speed and efficiency. In a 2012 blog post, the language developers discuss how the many different languages currently available are all excellent at certain aspects of their work while being absolutely terrible at others. They sum up their motivation for developing Julia as, "We are greedy: we want more" [6].

The design of Julia addresses the language as being a solution to what they have identified as some notions previously held as law in the numerical computing landscape: that because high level languages are slow, one must prototype in one language and deploy in another, and that there are parts of the language better left untouched [7]. Benchmarks show that Julia does indeed address the first issue by offering good execution performance [2], [8], [9].

There are several features of the language that provide exceptional performance: multiple dispatch, Just-In-Time Compilation, and the use of LLVM. Multiple dispatch is a key characteristic of the language whereby the type information of a function's arguments are used to select a method implementation at run time [10]. Many times, this process is done automatically for the programmer, such as in the trivial example of adding two numbers together. The compiler is able to optimize based on this type information. A unique implementation in the Julia Programming Language is that this dispatch is symmetric [11]. This feature allows Julia to place equal emphasis on the types of all arguments during method selection [12].

A vital part of leveraging Julia's multiple dispatch capabilities is ensuring so called "type stability." The key concept is that return types only depend on the types of the inputs and not on their actual values. This practice allows for greater specialization and optimization in the compilation process [11]. Types are pivotal in ensuring the greatest performance of code in Julia, which is why all types are considered "first-class," meaning both native and user-defined types are equally important during compilation [13]. Many of the performance tips in Julia revolve around the proper use of types [14].

Furthermore, code in Julia is actually compiled as opposed to being interpreted despite its high level syntax and functionality. Similar to Java, Julia uses Just-In-Time compilation to convert code to native machine code at runtime, also meaning that programs will run slower the first time it is executed [15]. This is why when benchmarking code in Julia, it should be ran twice - once to compile it, and the second time to get an accurate runtime measurement [2].

To further optimize code, Julia also makes use of LLVM, an open-source compiler framework used by several other languages such as Clang and Swift [16]. This framework provides its own Intermediate Representation (IR) upon which multiple passes are performed to further optimize execution performance. Overall, there are three main categories of passes: Analysis, Transform, and Utility Passes. The output of this is an IR similar to Assembly [17]. A quick overview of the entire Julia Just-In-Time (JIT) compilation process is as follows:

1) The Julia source code is parsed into an abstract syntax tree.
2) The resulting trees are then transformed into a Julia-unique intermediate representation for optimization.
3) This IR is then translated into the LLVM IR whereby further optimizations take place to generate native machine code.
4) The executable code is generated [11].

The second issue, the notion that a developer must have a prototype language and deployment language, is a concept the developers refer to as "the two language problem" [10], [16]. Developers write code in a higher level language to prototype functionality, then rewrite performance critical sections in a higher performance, lower level language. Julia aims to solve this by allowing developers to write in a high level Python-

like syntax, yet execute it in a manner that is optimized for performance. This ability is critical when a developer leverages GPUs to perform computation. As an example, Julia natively supports kernel development using CUDA without having to switch to C or C++ [18]. Julia's CUDA libraries directly integrate into the JIT compilation process, allowing for the code to be compiled to GPU assembly. This lets the programmer develop code almost the same way as one would do for the CPU, offering the same productivity of the language itself alongside the performance enhancements of GPUs [19]. The more recent release of CUDA.jl introduces more features and enhancements to CUDA development in Julia. Key improvements include support for CUDA 11.1 as well as CUDA's simplified stream programming model which simplify concurrency [20], [21]. Performance also does not suffer by using Julia's CUDA implementation. The Julia team has ported portions of Rodinia, a benchmark suite for general purpose computing, to Julia, and the results are almost identical to statically compiled CUDA C++ [22], [23].

We feel that Julia is a useful language and ecosystem for numerical computation and high performance computing domain. There exists an active community with a growing list of research, pushing it to prominence in these areas [24]. With performance and productivity at its core, Julia seems well aligned for computational research.

III. Methodology

This section discusses the parallelization of the process above and relates it to the platform it will be executed on.

A. Parallel Algorithm Model

The description is based on the Master-Slave model whereby a process (known as the master) manages a pool of available tasks and distributes them to other processes (referred to as the slaves) [1]. The procedure we have previously described for quantum process tomography can be decomposed into a series of 4 stages:

1) Compute the tensor product combinations $\rho^{(\alpha\beta)}$ where $\alpha, \beta \in \{H, V, D, R\}$.
2) Compute the respective $\varepsilon(\rho)$ of the above.
3) Compute the elements of the center matrix of the χ matrix shown in Equation 9.
4) Compute the complete χ matrix.

It is noted that each time a system is measured, some level of noise or interference could affect the accuracy of our measurement, leading this to be a stochastic process. We would thus like to perform quantum process tomography a variable number of times n for our given system and report the average measurement. In the case of the CNOT gate, $\varepsilon(\rho)$ can now be represented as:

$$\varepsilon(\rho) = pCNOT\rho CNOT + (1-p)\rho, \text{where p represents noise} \tag{11}$$

The value of p can ranges from 0 to 1, where 1 represents an ideal CNOT gate [4]. This noise can be simulated by generating a random value of p each time the tomography is performed. Each instance of the tomography can run independently, and every instance needs to perform the same set of identical tasks previously described with the added task of first generating a value for p.

Stages 1-3 have sixteen tasks. Stage 1, computes the tensor product combinations and their respective $\varepsilon(\rho)$ in Stage 2. Stage 3 computes the elements of the center matrix of χ - each of which is a 4x4 matrix, resulting in a 16x16 matrix. The final stage performs two matrix multiplications, where each matrix is 16x16. The task-dependency graph of the first four tasks in each of Stages 1-3 in Figure 1.

With each task in Stages 1-3 computing a 4x4 matrix, this mapping is described as having a fine-grained granularity with a degree of concurrency of 16. For n computations of the tomography, the degree of concurrency is 16n. The task-interaction of Stage 2 is developed by examining the matrix in (10), indicating these interactions are static and read-only. The task-interaction graph for the first six tasks in Stage 2 is shown in Figure 2. These are thus irregular interactions due to the lack of pattern in the matrix.

All tasks are known a priori for a given number of computations n; therefore, tasks are statically generated. Tasks in the same stage are uniform. Across stages, however, tasks are non-uniform. Knowledge of task size varies across the stages as well. In Stage 1, for example, computing the tensor product combinations is known, but calculating $\varepsilon(\rho)$ varies with the quantum operation being performed. The size of the data being

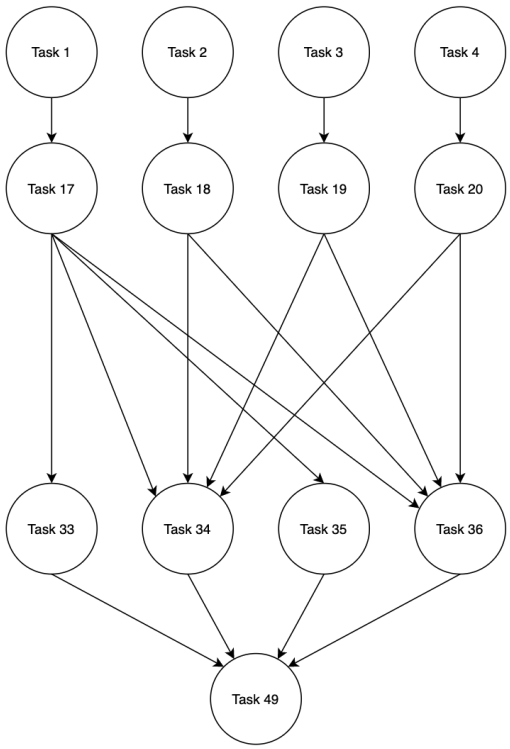


Fig. 1. Task-Dependency Graph for the first 4 tasks in Stages 1-3.

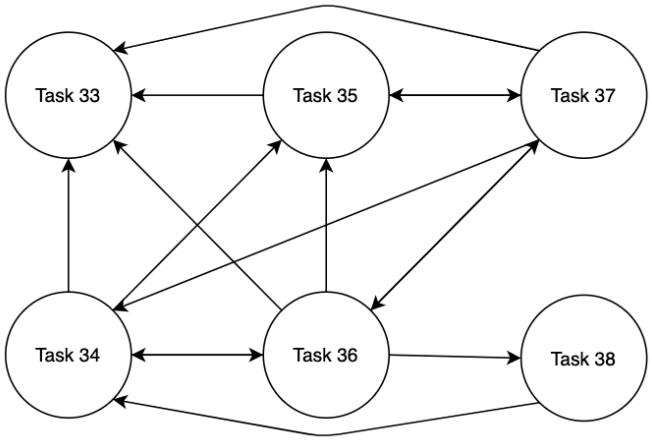


Fig. 2. Task-Interaction Graph for the first 6 tasks in Stage 2. The direction of the arrow indicates the task data is retrieved from.

computed in each task is ultimately a 4x4 matrix.

To avoid an imbalanced workload due to not having full knowledge of each task size, a dynamic centralized scheme is used. There is a central master process keeping track of the tasks that have not yet been performed. A source of the overhead associated with Master-Slave models is the contention of accessing a shared common data source [1]. Every worker process interacts with the master process in order to get more tasks. This is partly resolved through overlapping computation with interaction.

B. Mustang and Julia

Code is executed using the Department of Defense Supercomputing Resource Center's fastest machine, Mustang, with a peak performance of 4,777.6 TFlop/s. It has 12 login nodes, 1,1228 stand memory nodes, 24 large memory nodes, and 24 GPU accelerated nodes [25]. Mustang, an SMP architecture, shares memory among all the cores of a given node. This memory is not shared throughout the cluster; nodes communicate via message-passing [26].

Additionally, Mustang uses the Intel Omni-Path interconnect in a Non-Blocking Fat Tree [26]. With p equal to the number of standard memory nodes, the first switch in the tree has a total of p communication links, with $p/2$ going to each half [1]. Therefore, it has a bisection width of $1,128/2 = 564$. A fat tree will also have p links at every layer of the network down to the leaves. With p leaves, this gives a cost of $p\lceil \log_2 p \rceil = 1,128\lceil \log_2 1,128 \rceil \approx 12,408$. Similarly, the diameter can be found to be $2\lceil \log_2 p \rceil = 2\lceil \log_2 1,128 \rceil \approx 22$ with an arc connectivity of one.

The software implementation uses Julia's thread library. At the time of writing, these features are an experimental interface and types, macros, and functions are still evolving [27]. Julia v1.4.2 is used to leverage the latest features of the threads library available at the time of writing. Since version 1.3.0, Julia has added the ability to multi-thread nested for loops (previously only the outermost loop was possible) [28].

Additionally, support for thread-safe condition variables and locks have been added and developed [28]. This threading system is able to adapt to the available number of cores by dynamically scheduling work. Refined work on current features, in addition to support for more features, have continued to the current version as the developers work on maturing a composable set of parallel libraries for Julia [29].

To test our code, we increase the problem size for a given number of available threads. We will start with one thread, which is the default in the Julia environment. Each of problems sizes n = 1, 2, 4, 8, 16, 32, 64, and 96 will be executed on thread counts of 1, 2, 4, 8, 16, 32, 64, and 96 respectively for a total number of 64 tests. This is discussed further in the next section.

IV. Analysis

Defining the problem size as the number of times quantum process tomography is performed on a given system, the serial runtime is defined as $T_S = W = O(n)$. Following the derivation of the isoefficiency function, $W = KT_o(W, p)$:

$$W = Kp \tag{12}$$

The isoefficiency function of this mapping is thus $\theta(p)$. This function indicates that whatever ratio the problem size is increased by, the number of processing elements will also have to be increased by the same ratio in order to keep the efficiency constant [1]. We report the results of our experiment in Table 3.

The diagonal of the table results in both a doubling in the problem size and in the number of threads. Based on the isoefficiency function, a roughly equivalent run-time along these diagonals should be expected, and this is approximately the case. For example, starting in the top left corner of the table (thread count = problem size = 1) and going down its diagonal results in an increase in run-time of approximately 0.1 ms with every doubling. The diagonals of other problem sizes within the single thread count column act similarly. This additional overhead appears to approximately constant in each instance.

Another thing of note is that for most instances with 32 or more threads, we start seeing increasing run-times. This is to be expected for a problem size of 1 - its degree of concurrency is only 16, and thus additional threads over 16 would not be employed effectively [1]. However, for 32 threads, a decrease in run-time for a given problem size is not seen until a problem size of 32.

Further research into the issue reveals that Julia's garbage collector is not concurrent and could thus be a source of this overhead for a large number of threads [30]. Measurements were made using the *BenchmarkTools.jl* package, which provides some insight into this issue [31]. Using this package to examine the percentage of time the code was in garbage-collection for larger numbers of threads does show a considerable increase. A quick inspection showed that for numbers of 32 or more, average percent time in garbage collection ranged from 60% to over 80%.

Problem size \ Thread Count	1	2	4	8	16	32	64	96
1	1.6	1.4	1.2	1.1	1.1	1.2	1.5	1.7
2	3.1	1.7	1.4	1.3	1.2	1.3	1.8	2.9
4	6.1	3.4	1.8	1.4	1.3	1.6	2.1	3.4
8	12.1	6.8	3.6	1.9	1.6	1.9	2.7	3.0
16	24.5	13.8	7.4	3.9	2.1	2.5	2.7	4.2
32	52.6	27.7	14.9	8.2	4.9	4.4	4.7	6.0
64	116.5	61.3	35.9	19.5	11.1	8.8	9.7	9.7
96	181.8	96.8	70.6	30.3	16.8	14.7	15.9	18.6

Table 3: QPT Run-time in milliseconds: Program run-times for increasing problem sizes and thread counts.

V. Conclusion

This experiment provided useful insights into the Julia Programming Language and gave a better understanding of quantum process tomography and quantum computations in general. The exercise of first developing the parallel model also helped expose the inherent parallelism of quantum computations. The work presented here will therefore be highly beneficial for future efforts.

Though relatively robust, Julia's parallel computation libraries are still experimental. Despite increasing run-time with a considerably higher number of threads, results up to a respectable amount were approximately in the range of theoretical performance. Additional work should be done to develop the algorithm to handle the tomography of more qubits, therefore generalizing the procedure and providing another means by which to measure performance. It would also be interesting to do further research into Julia's garbage collector. The lack of concurrency there could be a bottleneck to increased performance and better speedups for a large number of threads.

References

[1] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing, 2nd Ed.* Harlow, England: Pearson Education, 2003.

[2] A. Sengupta, *Julia High Performance Second Edition*. Birmingham, UK: Packt Publishing, 2019.

[3] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information 10th Anniversary Ed.* Cambridge, UK: Cambridge University Press, 2010.

[4] A. G. White, A. Gilchrist, G. J. Pryde, J. L. O'Brien, M. J. Bremner, and N. K. Langford, "Measuring Two-Qubit Gates," *Journal of the Optical Society of America B*, vol. 24, no. 2, p. 172, Jan 2007. [Online]. Available: http://dx.doi.org/10.1364/JOSAB.24.000172

[5] W. Kozlowski *et al.* (2020) Architectural principles for a quantum internet. Internet draft. [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-qirg-principles/

[6] J. Bezanson, S. Karpinksi, V. B. Shah, and A. Edelman. (2012, Feb) Why We Created Julia. [Online]. Available: https://julialang.org/blog/2012/02/why-we-created-julia/

[7] A. Edelman, "Julia: A Fresh Approach to Parallel Programming," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 517–517.

[8] (2019, Aug) Julia 1.3 documentation. The Julia Project. [Online]. Available: https://docs.julialang.org/en/v1/

[9] S. Hunold and S. Steiner, "Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?" in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 20–25.

[10] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A Fast Dynamic Language for Technical Computing," 2012.

[11] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, and L. Zoubritzky, "Julia: Dynamism and Performance Reconciled by Design," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276490

[12] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," *SIGPLAN Not.*, vol. 21, no. 11, p. 17–29, Jun. 1986. [Online]. Available: https://doi-org.afit.idm.oclc.org/10.1145/960112.28700

[13] The Julia Programming Language. Types. [Online]. Available: https://docs.julialang.org/en/v1/manual/types/

[14] (2019, Aug) Performance Tips. The Julia Project. [Online]. Available: https://docs.julialang.org/en/v1/manual/performance-tips/

[15] (2021) The JIT compiler. IBM. [Online]. Available: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler

[16] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

[17] (2020, Mar) LLVM Documentation. LLVM Project. [Online]. Available: https://llvm.org/docs/

[18] T. Besard, C. Foket, and B. De Sutter, "Effective Extensible Programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019.

[19] T. Besard, P. Verstraete, and B. D. Sutter, "High-level GPU programming in Julia," 2016.

[20] M. Harris. (2015) GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. [Online]. Available: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[21] T. Besard. (2020) Cuda.jl 2.0. [Online]. Available: https://juliagpu.org/post/2020-10-02-cuda_2.0/index.html

[22] (2017, Oct) High-Performance GPU Computing in the Julia Programming Language. [Online]. Available: https://devblogs.nvidia.com/gpu-computing-julia-programming-language/

[23] G. Honan, S. Shivakumar, and A. Siraman, "Rodinia Benchmark Suite," Univ. of Pennsylvania, PA, Tech. Rep., 2017.

[24] (2021) Research. The Julia Project. [Online]. Available: https://julialang.org/research/

[25] (2020) Top500 Supercomputer Sites. [Online]. Available: https://www.top500.org/

[26] (2020) HPE SGI 8600 (Mustang) User Guide. [Online]. Available: https://www.afrl.hpc.mil/mustangUserGuide.html

[27] The Julia Programming Language. Multi-Threading. [Online]. Available: https://docs.julialang.org/en/v1/base/multi-threading/

[28] J. Bezanson, J. Nash, and K. Pamnany. (2019, July) Announcing composable multi-threaded parallelism in Julia. [Online]. Available: https://julialang.org/blog/2019/07/multithreading/

[29] Intel Corporation. (2020, Feb) New Threading Capabilities in Julia v1.3. [Online]. Available: https://www.codeproject.com/Articles/5260037/New-Threading-Capabilities-in-Julia-v1-3
[30] J. Bezanson. (2019, June) Allocation-heavy code is slower with Threads than Distributed. [Online]. Available: https://github.com/JuliaLang/julia/issues/32304
[31] J. Revels *et al.* (2020, April) BenchmarkTools.jl. [Online]. Available: https://github.com/JuliaCI/BenchmarkTools.jl

IV. Execution Performance of a Julia-based Hong-Ou-Mandel Simulation

The following paper, "Execution Performance of a Julia-based Hong-Ou-Mandel Simulation," is planned for submission to the Journal of Defense Modeling and Simulation.

Execution Performance of a Julia-based Hong-Ou-Mandel Simulation

Joseph Tippit
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
joseph.tippit@afit.edu

Douglas Hodson, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
douglas.hodson@afit.edu

Michael Grimaila, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
michael.grimaila@afit.edu

Richard Brewster, PhD
University of Maryland
College Park, MD, 20742, USA
rbrew1@umd.edu

Yanne Chembo, PhD
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA
ykchembo@umd.edu

Gerry Baumgartner, PhD
Laboratory for Telecommunication Sciences
College Park, MD, 20740, USA
gbaumgartner@ltsnet.net

***Abstract*—Interest in quantum networking has elicited research into quantum optics and its applications. Due to the significant time and resources required to conduct quantum optics experiments, the capability to create accurate models and execute simulations is therefore important. Existing simulation libraries, however, tend to include models oriented for either accuracy or performance.**

In this paper, we examine the features offered by the Julia programming language as a means to orient our models for both goals. This effort focuses on developing a quantum network simulation library which exploits the features of the Julia Programming language by re-implementing a MATLAB-based simulation of the Hong-Ou-Mandel experiment. The results show that exploiting Julia's unique features improves the simulation's execution performance.

***Index Terms*—high performance computing, Hong-Ou-Mandel, Julia, simulation, quantum**

I. Introduction

Performant software is of high interest when the underlying code is performing dense complex calculations. This situation arises when simulating large scale quantum optical systems which provide the foundation for modern quantum networking architectures. We examine the modeling and simulation of the Hong-Ou-Mandel (HOM) interference experiment using both MATLAB-based code and Julia-based code. The comparative results explore the impact that programming languages have on the performance of the simulation of quantum optical elements.

The HOM interference experiment is a fundamental quantum optics experiment which demonstrates the concept of photon indistinguishability [3]. Use of the HOM experiment is of particular interest because it includes effects that cannot be properly represented by classical physics, exercises basic quantum optics mathematics, and represents a fundamental building block in quantum optics systems. The HOM experiment simulation serves as the vehicle to compare the performance between a MATLAB-based and Julia-based simulation.

The main research objective is to develop a modular, scalable quantum network simulation support library. The hypothesis is that the Julia language provides the high level, object-oriented features needed to create better (low-level like) simulations in terms of execution.

The existing simulation of the HOM experiment is currently written in the scientific computing language MATLAB due to our subject matter experts' familiarity with the language. For this reason, the initial baseline performance of the simulation is written in MATLAB. We then rewrite the code in the Julia programming language taking advantage of its features to determine if it improved.

II. Background

Two main topics are important to describe before further detailing the research effort. The first is to provide a high level explanation of Hong-Ou-Mandel interference and how it will further additional research. The second is to introduce the Julia programming language and the rationale for using it.

A. Hong-Ou-Mandel Interference

Quantum interference is a fundamental concept in many applications such as quantum repeaters, technology important in the development of quantum networks [1], [2]. Verified in 1987 by Chung Ki Hong, Zhe Yu Ou, and Leonard Mandel, HOM interference is the result of two photons interfering on a 50/50 beam splitter [3], [4]. The idea behind the experiment is to fire two photons through a 50/50 beam splitter in which they interfere and then measure the coincidence probability of the photons. This probability is defined as the likelihood of detecting one photon at each of the beam splitter's two outputs [4]. Ultimately, as the probability drops near to zero, the more indistinguishable the photons become, resulting in a dip when plotted. Fig. 1 shows the four possible results of the photons interfering.

Distinguishability of the photons can be determined in multiple ways. For example, in the case of polarization, if one photon was horizontally polarized and the other vertically, then the two would be completely distinguishable. One could also measure the distinguishability of the photons as the time

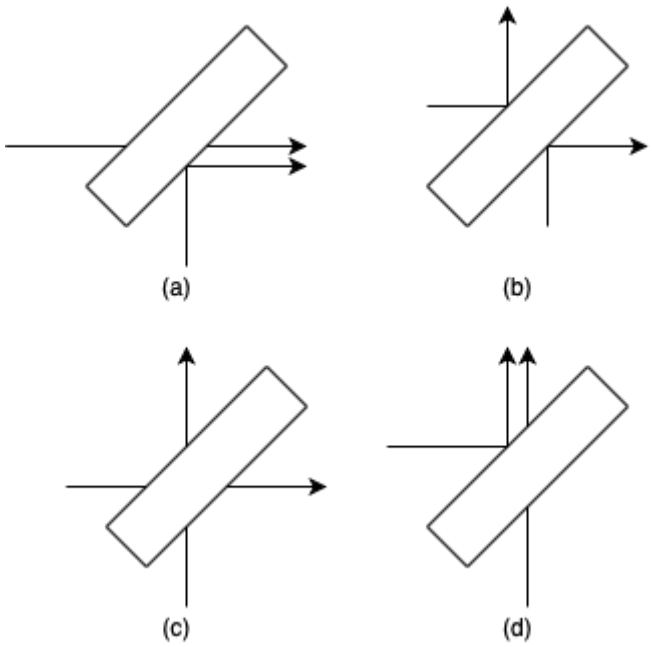


Fig. 1. The four possible interactions of the photons at the beam splitter.

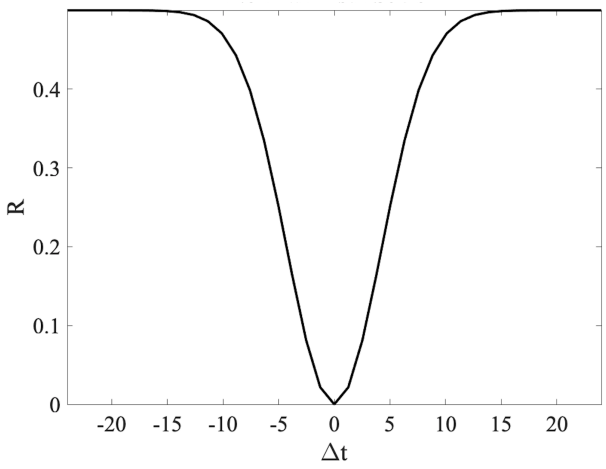


Fig. 2. HOM dip for a Single-Photon Pair with a normal Gaussian distribution.

difference in which the photon pulses where formed. If formed at exactly the same instance, then the photons would be considered indistinguishable and would become more distinguishable the further apart in time they were generated [4]. Fig. 2 shows a plot of the temporal creation time against the coincidence probability R; as can be seen, the dip occurs when $\Delta t = 0$, i.e., the photons are indistinguishable. At this point, the photons will always exit the same output of the beam splitter.

Through providing an efficient and general means for simulating HOM interference, it is hoped to be able to "build up" the means for simulating larger systems that contain more components. As an example, simulation of both the photon and the beam splitter are two pervasive features in many quantum networks.

B. The Julia Programming Language

Julia is a relatively new programming language designed with execution performance as a priority. Introduced in 2012, the developers proposed the language as a solution to what they refer to as "the two language problem," whereby logic is initially implemented in a higher level language for productivity and then re-implemented in a lower level language (such as C or Fortran) for performance [5], [6]. The Julia language is designed to solve what programmers previously saw as "the laws of nature" - that high-level languages must be slow, the two language problem is fact, and that there are parts of the language that must be left untouched [5]. Benchmarks show that Julia does indeed offer performance improvements over other higher-level languages (and in many circumstances, performance comparable to C) and is a serious competitor for high performance computing [5]–[11].

There are several aspects of the language that the developers exploit to improve execution performance: multiple dispatch, code specialization, and the Just-In-Time (JIT) compiler leveraging LLVM [6]. JIT compilation is a technique whereby high-level languages are converted into machine code executable directly on the CPU when the code executes [9]. Julia compiles code the first time it is executed, specializing to function type information during method selection. The results of this method specialization are then cached. The next time the code is executed, Julia is able to re-use the pre-compiled and specialized results for faster performance. This does mean there will be an overhead of compilation the first time code is executed, but subsequent runs will be able to leverage pre-compiled code existing in the cache [11].

During compilation, Julia also takes advantage of the optimizations provided by LLVM, a compiler framework that provides an intermediate representation (IR) used by several other languages as well [5], [10]. LLVM implements these optimizations as passes. There are three different categories of passes: Analysis, Transform, and Utility Passes. The transformation process results in an IR that is a lower-level language similar to Assembly for the x86 architecture [12].

Multiple dispatch is viewed as one of the most significant features in Julia. A method is a specific instance of a function for a specific type(s). Julia's code specialization is based on the idea of symmetric multiple dispatch - multiple dispatch is the ability to choose the right methods for the right types of arguments, and symmetric means that all argument types are equally important in this selection [6], [11].

Many of Julia's speedups and performance enhancements are largely due to multiple dispatch, and central to it is type stability. Type stability means that the return value type solely depends on the input types. This relation allows the compiler to specialize to those types and cache the result for subsequent executions. Furthermore, the developer can reuse function names for different code paths: polymorphism [5]. This programming feature is used in implementations such as mathematics, where operations can mean different things depending on the inputs. The developers also note that in many

circumstances, code is type stable by default, so programmers do not need to focus on the type annotations in their code [6].

Additionally, the use of GPUs is a focal point in our research. It has been shown that Julia enables "new and dynamic approaches for GPU programming" [13]. The introduction of CUDAnative.jl, as well as the more recent follow-on package CUDA.jl, make Julia a good fit for developing performant GPU programs at a higher level of abstraction than traditional C/C++ code [13], [14]. Furthermore, we note Julia's GPU performance compared to C [13]. Combined with the same productivity and efficiency of the language itself, Julia is a natural selection for this research.

III. Methodology

This section discusses the initial development of the HOM simulation MATLAB code. An overview of the re-implementation in Julia is also provided, alongside the techniques used to improve and benchmark the code.

A. Initial Development

The original program used was developed in MATLAB and simulates three different state types with two different distributions. The states include single-photon pairs, weak coherent pulses, and two-mode squeezed vacuum. The available distributions are Gaussian and Sinc. We attempt to compute the coincidence rate as a function of the temporal delay between the generation of two photon pulses.

For each state type, we work with Fock states (or number states) represented mathematically as a 3x1 matrix. The necessary constants are computed once a state and distribution are selected, which includes the vacuum and annihilation operators of appropriate dimensionality formed through the use of Kronecker products. Time is divided into a series of time bins, denoted by the variable nt, and the total number of bins is chosen to be odd because there is a central one needed for symmetry. These time bins represent the times in which a photon could be created. A pulse width is defined such that the time bin range is calculated, and from this and nt, dt (the length of each time bin) is computed.

Once in the main body of the code, the simulation only ever considers a single pair of time bins at any given application of the beam splitter evolution operator and the coincidence measurement. This occurs through the use of a nested *for* loop to calculate the pulse multiplicative factors and then the state through a series of matrix multiplications and exponentation-s/sinc functions, depending on the type of distribution used. Over the course of the series of *for* loops, the code initializes the temporal distribution and applies either the single-mode or two-mode beam splitter unitary operator through matrix multiplication. The outcome of each step of the loop is either a 9x1 or 81x1 matrix, depending on whether the time bins are equal (which corresponds to the photons being created at the same time).

Lastly, the coincidence rate is calculated by once again looping over each pair of time bins. At each time step, the contribution is calculated through application of the appropriately sized annihilation operator and then added to an overall total. This process is repeated for every point to plot along with every pair of time bins. In order to generate a plot containing 39 points using 51 time bins, the main body of the code executes this series of calculations 51,714 times. Once this is completed, we are able to plot the results and generate the HOM dip shown in Fig. 2.

B. Re-implementation in Julia and Benchmarking

Implementing CUDA.jl's workflow, the code was first implemented on the CPU [15]. Initially, the first step translates the code from MATLAB code to Julia. A notable change is wrapping the code into a function so it can be benchmarked. Wrapping also avoids the issue of global variables in Julia while allowing us to execute the code twice - once to compile it and once to benchmark it. The benchmarks of execution performance are measured using the Gaussian distribution for all three state types.

After initial benchmarks, the Performance Tips section of the Julia documentation is reviewed and implemented as appropriate [16]. At first glance, some observations are made that help contribute to reducing the code's runtime. This includes the fact that loops in the code use predefined arrays, which allows for bounds checking to be disabled. Other notable features are the ability to pull some computation out of nested loops and merge other loops. The direction of the loops through the main data structure is also changed to be column-major, since this is the system Julia uses.

Type information into the code is also introduced in the code. For example, type information is implemented to provide the compiler with information on what type of simulation was being performed (Sinc vs. Gaussian and Single Photon, Weak Coherent, or Two Mode) as well as functions that operate on these types. This takes advantage of Julia's mature application of multiple dispatch and offloads work to the compiler to choose the correct method while also ridding the code of its reliance on *if-else* statements. This is a benefit over MATLAB, as it simultaneously makes the code more readable.

Additional research shows that the code can be further improved by changing the basic linear algebra subprograms (BLAS) implementation that Julia uses [17]. The use of Intel CPUs implies that further improvements may be made using Intel's BLAS, MKL. Additional benchmarks using Julia's implementation of MKL show significant improvement [18]. All further benchmarks are made using MKL.jl. Furthermore, MATLAB makes use of a multi-threaded BLAS, so further benchmarks are made explicitly limiting MATLAB to a single thread in order to more accurately compare it to Julia. All benchmarks are made on a computer with dual 2.40GHz Xeon 4210R processors and 192GB of RAM using Julia v1.5.3 and Matlab R2020b.

IV. Results

Initial benchmarks of the MATLAB code yielded runtimes under 18 seconds for single-photon pairs, under one minute

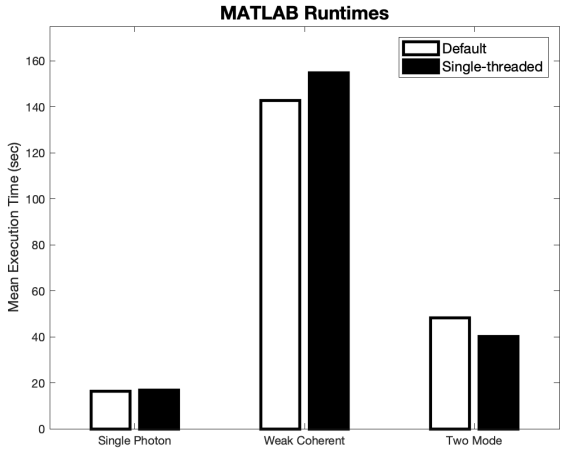


Fig. 3. Mean execution times for both the default MATLAB multi-threaded BLAS and with a single thread of execution. Averages are based on the runtimes of 50 executions.

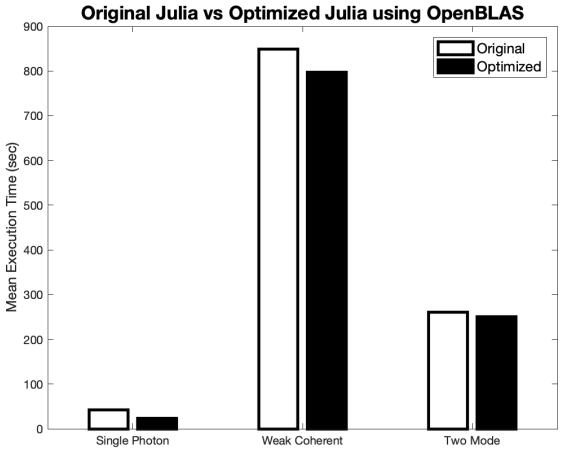


Fig. 4. Mean execution times of the original implementation of the Julia code compared to the version after improvements were made. These averages are based off 5 repeated executions due to considerably larger execution times.

for two-mode squeezed vacuum, and just over two minutes in the case of weak coherent. When benchmarks were generated for MATLAB using only a single thread of execution, the mean execution time for single photon stayed roughly the same as for the multi-thread version. Based on 50 runs, the mean execution time of weak coherent saw approximately an 8% increase while two-mode saw roughly a 17% decrease. T-tests confirm the benchmark value differences observed - it indicates differences exist with less than a 1 percent chance due to purely random effects ($p < 0.01$). The single-threaded two mode had a standard deviation of 0.68 seconds and a maximum runtime of 42.01 seconds. The multi-threaded version had a larger standard deviation of 1.57 seconds and a minimum runtime of 46.06 seconds. Fig. 3 displays these results.

Execution of the initial conversion to Julia took significantly longer. For example, mean execution time of 5 executions for weak coherent was over 14 minutes, and times were within 5 seconds of the average in for each of the three cases. Examining memory usage explains these results somewhat, as our computations are extremely heavy in linear algebra. Since this version of the Julia code did not take into consideration pre-allocating ouput, we were able to target this in our optimization. The largest performance gain was seen by switching the version of BLAS used from OpenBLAS to MKL.

After this transition, our improved version of the Julia code executed faster on average than both the default and single-threaded versions of MATLAB for the single photon case. Minimum runtimes for the default and single-threaded MATLAB versions, however, were 15.33 and 16.41 seconds respectively while the optimized Julia version had a maximum runtime of 16.46 seconds. The single-threaded MATLAB version had the smallest standard deviation of 0.24 seconds while the mutli-threaded MATLAB version and optimized Julia version had standard deviations of 0.56 seconds and 0.46 seconds respectively.

In terms of weak coherent, the average runtime for the optimized Julia version was approximately 4 seconds faster as compared to the single-threaded MATLAB version. It remained approximately 8 seconds slower than the default MATLAB version, however. The two mode version of Julia saw the least improvement, remaining just over 30 seconds slower on average than the default MATLAB version. The standard deviations for these cases were less than four seconds for weak coherent and under two seconds for the MATLAB two mode versions. The optimized Julia version for two mode was just under two seconds. T-tests were performed between both versions of the Julia MKL code as well as between the optimized Julia MKL and both versions of the MATLAB code. These results were confirmed with $p < 0.01$. Summaries of the collected statistics of execution times for each version and state type of the HOM simulation experiment cases are provided in Tables I-III.

TABLE I
SINGLE PHOTON SUMMARY STATISTICS: OVERALL SUMMARY STATISTICS FOR EACH VERSION OF THE SINGLE PHOTON CASE OF THE HOM SIMULATION EXPERIMENT.

Statistic	N	Mean (s)	St. Dev. (s)	Min (s)	Max (s)
MATLAB Single-Thread	50	16.92	0.24	16.41	17.46
MATLAB Multi-Thread	50	16.25	0.56	15.34	18.60
Original Julia - MKL	50	24.62	0.37	24.22	26.05
Optimized Julia - MKL	50	15.39	0.46	14.87	16.46

Memory allocation and usage remained the same between MKL and OpenBLAS, and there was also no noticeable difference between runs. For example, the single photon case for the original Julia version always reported the same memory usage and allocation. This implies that MKL had no effect on memory. After our improvements to the code were made, however, notable decreases in memory usage were seen - 25% for weak coherent and 50% for single photon. Two mode was

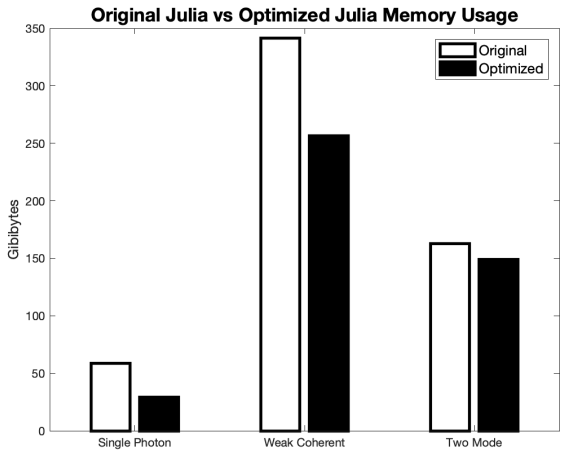


Fig. 5. Memory usage of the Julia code before and after the improvements were made. The optimized version saw a significant reduction in memory usage due to the care taken in pre-allocating memory and utilizing in-place operations.

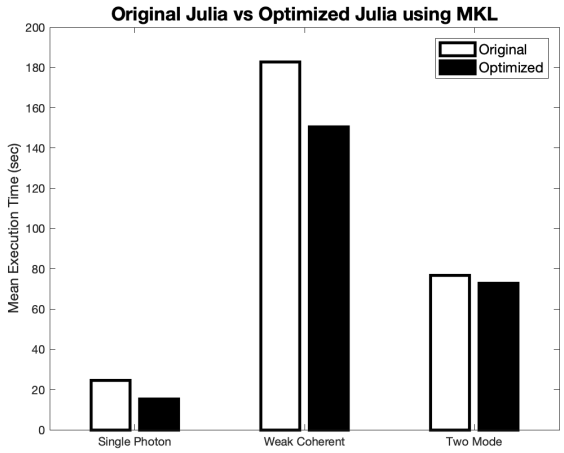


Fig. 6. Mean execution times of the original implementation of the Julia code compared to the version after improvements were made. These were done after changing the BLAS to MKL and are based on 50 runs.

TABLE II
WEAK COHERENT SUMMARY STATISTICS : OVERALL SUMMARY STATISTICS FOR EACH VERSION OF THE WEAK COHERENT CASE OF THE HOM SIMULATION EXPERIMENT.

Statistic	N	Mean (s)	St. Dev. (s)	Min (s)	Max (s)
MATLAB Single-Thread	50	154.83	2.47	152.45	160.41
MATLAB Multi-Thread	50	142.82	3.68	138.66	154.90
Original Julia - MKL	50	182.81	2.01	179.18	186.56
Optimized Juila - MKL	50	150.40	3.97	145.72	159.99

once again not as well improved, but still saw an almost 10% decrease. Allocations also saw improvement, but were not as significant. Julia is prone to allocating new arrays every time an array operation is made, which leads us to believe we still have plenty of room for improvement here. However, this may not be of much concern once the code is ported to GPUs.

TABLE III
TWO MODE SUMMARY STATISTICS : OVERALL SUMMARY STATISTICS FOR EACH VERSION OF THE TWO MODE CASE OF THE HOM SIMULATION EXPERIMENT.

Statistic	N	Mean (s)	St. Dev. (s)	Min (s)	Max (s)
MATLAB Single-Thread	50	40.09	0.68	39.22	42.01
MATLAB Multi-Thread	50	48.19	1.57	46.06	52.10
Original Julia - MKL	50	76.59	0.94	74.35	80.90
Optimized Juila - MKL	50	72.86	2.72	70.65	87.82

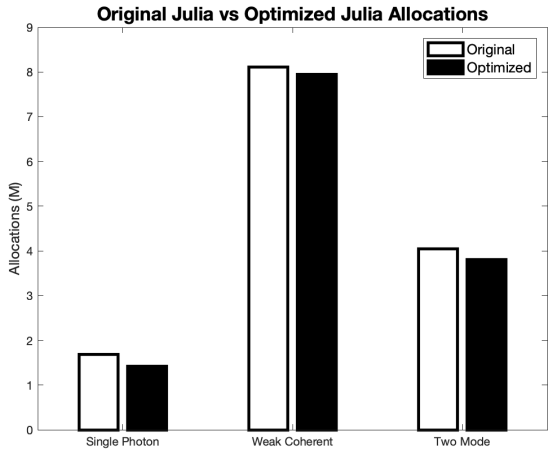


Fig. 7. Memory allocations of the Julia code before and after improvements were made.

These results are shown in Fig. 7.

V. CONCLUSION

Despite Julia being able to outperform MATLAB in the single photon instance, it was not a significant difference. While the memory allocations show that there is still room for improvement, it appears unlikely Julia will be able to perform better than MATLAB for these simulations without use of multi-threading or GPUs. MKL offered a significant runtime reduction over the original Julia implementation, however, it is noted that we are using Intel CPUs.

The benchmarks shown here are an important first step in developing future simulations with improved execution performance. There is now a better understanding of how a type representation hierarchy to leverage Julia's multiple dispatch abilities could be developed. These results also provide a baseline on which future work can be compared. Being cognizant of Julia's array allocations and pre-allocating arrays where possible are areas of further improvement in our work. However, those problems may be resolved by transitioning the code to utilize GPUs, the objective of future efforts.

REFERENCES

[1] W. Kozlowski, S. Wehner, R. Van Meter, B. Rijsman, A. S. Cacciapuoti, and C. Caleffi, "Architectural Principles for a Quantum Internet," no. 3, 2020.

[2] S. Wang, C. X. Liu, J. Li, and Q. Wang, "Research on the Hong-Ou-Mandel interference with two independent sources," Sci. Rep., vol. 9, no. 1, pp. 1–7, 2019, doi: 10.1038/s41598-019-40720-5.

[3] L. Mandel, Z. Y. Ou, and C. K. Hong, “Measurement of Subpicosecond Time Intervals between Two Photons by Interference,” Phys. Rev. Lett., vol. 59, 1987, doi: 10.1210/endo-79-5-927.
[4] A. M. Brańczyk, “Hong-Ou-Mandel Interference,” arXiv, pp. 1–17, 2017.
[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” SIAM Rev., vol. 59, no. 1, pp. 65–98, 2017, doi: 10.1137/141000671.
[6] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, “Julia: A Fast Dynamic Language for Technical Computing,” pp. 1–27, 2012, [Online]. Available: http://arxiv.org/abs/1209.5145.
[7] T. Driscoll, “Matlab vs. Julia vs. Python,” 28-Jun-2019. [Online]. Available: https://tobydriscoll.net/blog/matlab-vs.-julia-vs.-python/. [Accessed: 25-Jan-2021].
[8] S. Hunold and S. Steiner, “Benchmarking Julia’s Communication Performance: Is Julia HPC Ready or Full HPC?,” Sc20, pp. 20–25, 2020, doi: 10.1109/PMBS51919.2020.00008.
[9] A. Sengupta, *Julia High Performance Computing*, 2nd ed. Birmingham, UK: Packt Publishing, 2019.
[10] R. Lakhanpal and A. Joshi, *Learning Julia*, Birmingham, UK: Packt Publishing, 2017.
[11] J. Bezanson et al., “Julia: dynamism and performance reconciled by design,” Proc. ACM Program. Lang., vol. 2, no. OOPSLA, pp. 1–23, 2018, doi: 10.1145/3276490.
[12] LLVM Project, *LLVM Documentation*, Mar. 2020. [Online]. Available: https://llvm.org/docs/. [Accessed Mar. 19, 2020]
[13] Besard, T., Foket, C., & De Sutter, B. (2019). Effective Extensible Programming: Unleashing Julia on GPUs. IEEE Transactions on Parallel and Distributed Systems, 30(4), 827–841. https://doi.org/10.1109/TPDS.2018.2872064
[14] T. Besard, “CUDA.jl 2.0,” CUDA.jl 2.0 · JuliaGPU. [Online]. Available: https://juliagpu.org/2020-10-02-cuda_2.0/. [Accessed: 30-Jan-2021].
[15] T. Besard, “Workflow,” Workflow · CUDA.jl. [Online]. Available: https://juliagpu.gitlab.io/CUDA.jl/usage/workflow/. [Accessed: 02-Feb-2021].
[16] The Julia Project, Performance Tips, Aug. 2019. [Online]. Avail-able: https://docs.julialang.org/en/v1/manual/performance-tips/. [Accessed Feb. 02, 2021].
[17] A. Yahyaabadi, “juliamatlab/Julia-Matlab-Benchmark,” GitHub, 13-Oct-2019. [Online]. Available: https://github.com/juliamatlab/Julia-Matlab-Benchmark/blob/master/README-Julia-openBLAS-vs-Julia-MKL.md. [Accessed: 02-Feb-2021].
[18] V. Shah, “JuliaLinearAlgebra/MKL.jl,” GitHub. [Online]. Available: https://github.com/JuliaLinearAlgebra/MKL.jl. [Accessed: 02-Feb-2021].

V. Conclusions

The research presented here was used to develop a prototype software written in the Julia Programming Language. Using the lessons learned from previous experiments and research, we were able to develop our software with performance and use of graphics processing units (GPUs) in mind. While not yet released as an installable package, this two-module library is the first ever quantum optics simulation software designed from the beginning with GPU-acceleration.

This is an important milestone for further research in quantum network simulation. While other libraries and frameworks are currently developing support for GPUs, ours introduced it from the start. This allows one to immediately begin seeing the benefits of leveraging GPUs to perform the costly calculations involved in quantum simulation.

Along the way, we were also able to make contributions back to several open source projects. In one instance, we found a bug in the Julia source code which prevented enabling more than one profiler during installation [15]. As another example, we designed and proposed a kernel to enable CUDA.jl support for kronecker products [16].

5.1 Overview

The resulting library has been designed to be a two-module simulation software - one providing the base functionality and the other extending with GPU support. For clarity, we call the former QuMSim.jl and the latter CUDAQu.jl. This was done as a matter of convenience to allow scientists and researchers the ability to use the codebase in a consistent syntax regardless of whether they have a CUDA-compatible GPU. In the few instances where syntax consistency between the GPU and central processing unit (CPU) code is broken, we have opted for a similar styling convention

used by CUDA.jl. That is, we place `CUDAQU.` in front of the function name.

QuMSim.jl was developed to be modular, allowing its users to choose their desired level of abstraction. Higher level types, such as `Qubit`, are provided. However, through a technique called iterated dispatch, all objects and functions dispatch down to a lower level `AbstractQuantumRepresentation` type. These types are all made available to users. In addition, QuMSim.jl leverages metaprogramming to automatically generate methods for most types in a sort of polymorphic behavior. This provides developers a means to easily extend or even implement their own types or underlying mathematical representations.

We were also able to achieve our objectives of using this framework to simulate both the Hong-Ou-Mandel (HOM) Simulation Experiment and the Mach-Zehnder Interferometer Experiment. As will be discussed in Appendix C, we saw noticeable improvements in the execution times of our experiment simulations. These have been provided in convenient functions along with all the necessary documentation in order to replicate the results. A more in depth conversation of these modules is provided in Appendices A to C.

5.2 Future Work

There are four main areas of future work that would be beneficial to this research. The first is adding support for basis information in the codebase to improve mathematical consistency. Another area is providing comparative benchmarks of this software against other libraries. One area that is currently in progress is writing a standalone paper providing detailed documentation on the software. The last area is releasing the software as an installable Julia package. Further information for each of these areas is provided in the sections below.

5.2.1 Add Support for Basis Information

Further additions to the QuMSim.jl framework could include adding support for basis information within the types. This would provide users with a convenient fall-back mechanism to ensure consistency within all mathematical operations. Once included, it would be simple to leverage Julia's multiple dispatch system to either raise a warning if multiple bases are detected or, instead, follow a predetermined type promotion system. For instance, if an operation was performed with two differing bases, one would be converted to the other basis before performing the operation. A similar system was incorporated in QuantumOptics.jl, and it is noted as a viable advantage to the framework we have developed [12].

5.2.2 Detailed Benchmarks Against Other Libraries

One series of tests that was unable to be completed were detailed benchmarks against other libraries. It is noted, however, that there are currently no robust GPU-accelerated quantum libraries with the same objectives as ours. For a meaningful comparison to be made, it would be recommended to only use the base QuMSim.jl library.

5.2.3 Publish a Standalone Paper on the Software

A separate published paper providing an in-depth review of the software developed was not included as a part of this thesis. While the appendices do provide an overview, we wish to further highlight the software in a standalone paper to offer a more comprehensive assessment and review of the work performed in developing the software.

5.2.4 Release as a Julia Package

Lastly, one final step that was not taken here was making QuMSim.jl and CUDAQu.jl publicly available. The codebase used throughout this research was built and installed as a development package, meaning it would be simple to build and release in the General Registry [17]. However, it would be recommended to mature the framework before public release.

Appendix A. Software Documentation

The software developed as a result of this research was implemented in two modules. The first, QuMSim.jl, implements the base functionality and necessary types in Julia. CUDAQu.jl, the second module, extends QuMSim.jl to provide CUDA support. This approach allows developers to use the software with or without GPU support, which provides a more scalable approach to development.

In this appendix, the design choices in both modules will be discussed. Starting with QuMSim.jl, the base file structure and type hierarchy will be covered along with the design methodology used; implementation of operations and mathematics will be included. From there, an overview CUDAQu.jl and how it extends QuMSim.jl is provided.

1.1 QuMSim.jl

A goal of QuMSim.jl design was to minimize keep package dependencies; only two external libraries are required. In the current state, only two external libraries are listed as dependencies: Distributions.jl and LinearAlgebra.jl from the Julia base library [1, 18]. All files within the module, along with the module declaration, are included in the main file, QuMSim.jl. The module's basic functionality is defined within base.jl.

1.1.1 base.jl and Type Hierarchy

The most important file in the module, base.jl implements all the necessary functionality for the rest of the module to work properly. This file defines an abstract type hierarchy that is used throughout the rest of the code base and exports the composite types at the bottom of the type tree.

Three main abstract types are defined: `AbstractParticle`, `AbstractOperator`, and `AbstractQuantumRepresentation`. The structure of these types detaches the concept of the high level object one wishes to represent (such as a qubit, photon, or quantum gate) from the lower level mathematical representation of the object. This allows a modeler or researcher to choose the level of abstraction, whether in terms of concrete particles or in mathematical representations.

Additionally, the module leverages Julia's use of multiple dispatch to provide for a seamless and consistent development approach regardless of the underlying representation used. In order to switch from one representation to the next, or to create a unique representation, one merely just needs to change the underlying representation used in the initial declaration of a type; all further method calls will be the same. This convenience is because all high level objects in QuMSim.jl act only as a wrapper for their underlying representations. Through a process called iterated dispatch, all methods dispatch down to the representation, which is where the actual data being operated on is contained. The idea is to dispatch down, and build back up to the abstraction level being used. Figures 1 and 2 provide a summary of the type hierarchy for `AbstractQuantumRepresentation` and `AbstractParticle` .

Directly under `AbstractParticle` are three concrete types: `Qubit`, `FockState`, `Photon`. These are the currently supported high level objects, each taking a representation type as input. `Qubit` and `FockState` also support directly passing vectors as a shorthand method to instantiate them.

`AbstractQuantumRepresentation` is further broken down into two abstract types: `AbstractParticleRepresentation` and `AbstractOperatorRepresentation`, each for types representing `AbstractParticle` and `AbstractOperator` types respectively. `AbstractParticleRepresentation` contains two types. The first, `Pulse`, requires a function modeling the desired pulse in addition to the type of distribution desired.

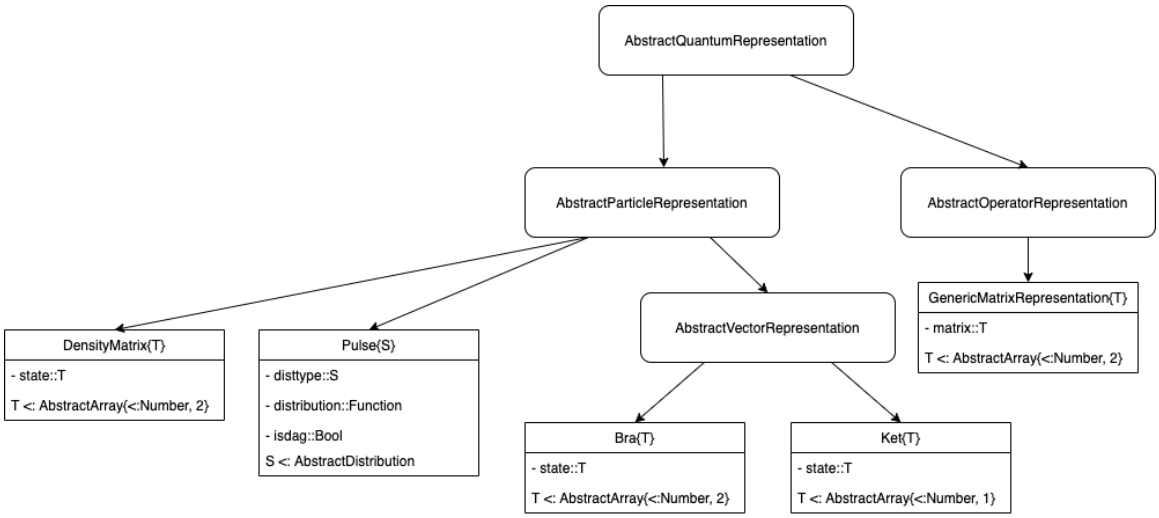


Figure 1: `AbstractQuantumRepresentation` Type Hierarchy

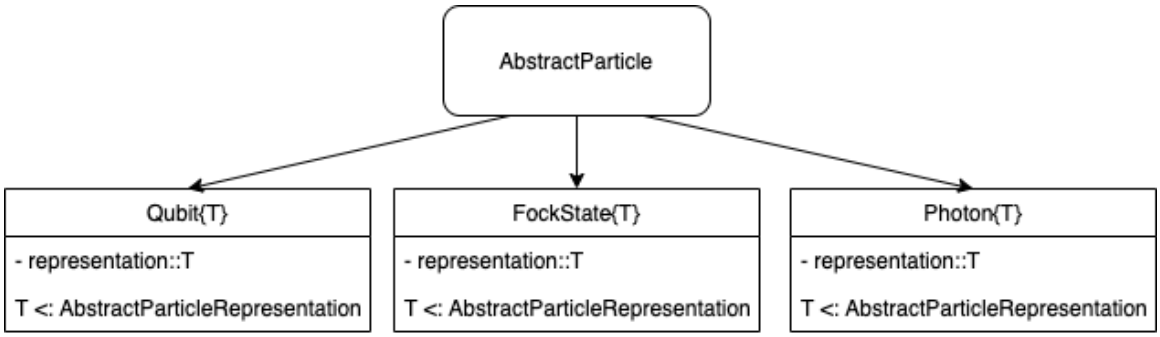


Figure 2: `AbstractParticle` Type Hierarchy

Currently, both `TemporalDistribution` and `SpectralDistribution` exist as empty types to fill this need. The second type is `DensityMatrix`, which takes a two dimensional array as input.

`AbstractParticleRepresentation` is also broken into another subtype named `AbstractVectorRepresentation`. The two structs here are `Ket` and `Bra`, which take vectors as inputs. `AbstractOperator` and `AbstractOperatorRepresentation` is covered in the next section.

1.1.2 Operators.jl

`Operators.jl` implements all of the quantum operators in QuMSim.jl. The types exported by this file all inherent from the `AbstractOperator` as well as the `AbstractOperatorRepresentation` types from base.jl. The highest level objects are `Operator` and `QuantumGate`, both falling directly under `AbstractOperator` and taking a representation of type `AbstractOperatorRepresentation` as input. Similar to `Qubit` and `FockState`, they can also take a two dimensional matrix as input as well. `GenericMatrixRepresentation` is currently the only implementation of `AbstractOperatorRepresentation` in QuMSim.jl. Its input is a two dimensional matrix.

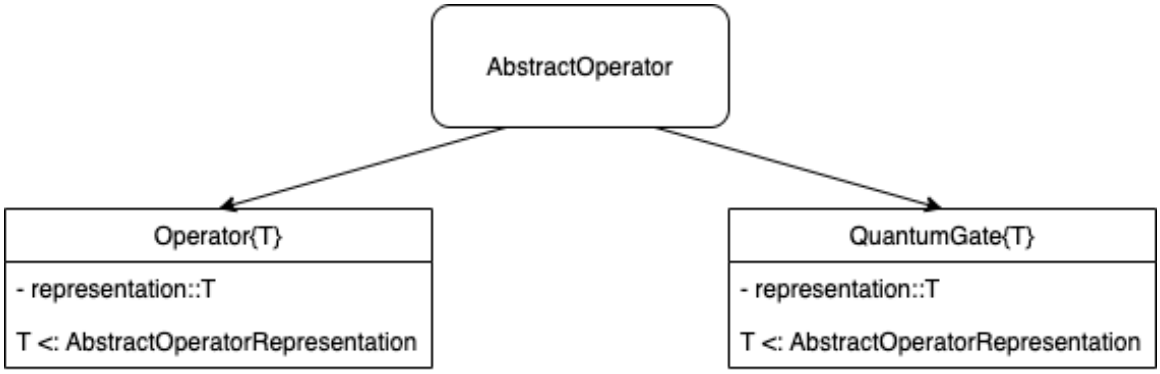


Figure 3: `AbstractOperator` Type Hierarchy

Operators.jl also defines a few functions to simplify the creation of certain common operators. These functions: `AnnihiliationOperator`, `CreationOperator`, and `IdentityOperator` each returns an `Operator` with a matrix representation of the respective operator contained within a `GenericMatrixRepresentation`. Inputs to these functions are merely the datatype of the number desired (e.g. Float32 or Float64) as well as the maximum number of photons in the system. For a system containing n photons, an $(n+1)x(n+1)$ dimension operator is returned.

Lastly, the `Ket` and `AbstractParticle` types are provided the `measure!` function.

Use of this function will permanently collapse the state of the object to one of the possible outcomes. The exclamation point denotes that the input will be directly modified versus creating a new particle.

1.1.3 QuMath.jl and other Functionality

QuMath.jl contains QuMSim.jl's math implementations, and this is where the iterated dispatch methodology becomes prevalent. Each method call on a higher level object (such as those under `AbstractParticle` or `AbstractOperator`) simply recalls the method on the underlying representation (an `AbstractParticleRepresentation` or `AbstractOperatorRepresentation`). Since almost all representations contain normal types in Julia, these types are able to dispatch the same method down to an already defined function within Julia. When a new representation is added, the methods operating on that representation are the only additions needed for a high level object to utilize it. For example, `Photon` does not care how it is represented, only that it has a representation. Multiple dispatch handles the rest.

Julia also contains robust support for metaprogramming, allowing ease of maintenance and scalability for new type additions. In many cases, new high level types can simply be added to one of the arrays at the top of the file, and Julia will generate the appropriate methods for them automatically. For example, an addition to `AbstractionVectorRepresentation` or `AbstractionParticleRepresentation` could simply be added to the correct array in QuMath.jl.

```
VectorTypes = (:Ket, :Bra);
ParticleTypes = (:Qubit, :FockState, :Photon)
```

The code contained further down in QuMath.jl would then generate the correct method for the `kron` function without any additional action.

```
# Defines kronecker products for all same type
```

```
# AbstractVectorRepresentation i.e. kron(::Ket,::Ket),
# kron(::Bra,::Bra)
for v in VectorTypes
    @eval kron(A::$v, B::$v) = $v(kron(A.state, B.state))
end

# Defines kronecker products for all same type
# AbstractParticle i.e. kron(::Qubit,::Qubit),
# kron(::FockState,::FockState), kron(::Photon,::Photon)
for p in ParticleTypes
    @eval kron(A::$p, B::$p) = $p(kron(A.representation, B
       .representation))
end
```

1.2 CUDAQu.jl

As previously stated, CUDAQu.jl extends QuMSim.jl by including support for CUDA, thus GPU capabilities. It is not a standalone library and requires QuMSim.jl to be included in the namespace to work. All includes for CUDAQu.jl occur in the file of the same name. The only external libraries required are CUDA.jl and LinearAlgebra.jl [1, 19, 20].

CUDAQu.jl's base.jl file is very simple compared to QuMSim.jl's. This file only contains the code to allow CUDAQu.jl to create `Qubit` and `FockState` types from CUDA.jl's `CuArray` without needing to declare a representation first. CuOps.jl likewise does a similar task for the `Operator` and `QuantumGate` types. Additionally, CuOps.jl redefines the functions for building the common operators in order to leverage the GPU.

Lastly, cuKron.jl defines a kernel for performing Kronecker product calculations on the `CuArray` type as discussed in [16]. All code usage is identical to QuMSim.jl with few exceptions, which will be discussed further in Appendix B.

Appendix B. Code Examples

The base code is started the normal way in Julia with the using statement.

```
using QuMSim
```

A natural first step would be to create a state vector, or ket. These can be created in QuMSim.jl by calling `Ket` with a vector as an argument.

```
A = Ket([1;0])
```

```
Quantum State Vector:
    [1, 0]
```

Operations on `Ket` types are also permitted. For instance, the adjoint of a ket in quantum mechanics is a bra, and this holds true for QuMSim.jl as well.

```
B = Ket([0;1])
C = adjoint(B)
```

```
bra: [0 1]
```

As would be expected, the adjoint of a `Bra` returns a `Ket`.

```
D = adjoint(C)
```

```
Quantum State Vector:
    [0, 1]
```

Operations between `Ket` types are also available, such as the creation of density matrices.

```
A*C
```

```
Quantum Density Matrix:
    [0 1; 0 0]
```

Kronecker products can also be performed on QuMSim.jl types as they normally would in the Julia base language.

```
kron(A,B)
```

```
Quantum State Vector:
    [0, 1, 0, 0]
```

QuMSim.jl supports other state types as well, such as Fock (or number) states. These are created the same way as `Ket` by calling `FockState`.

`Operator` types can also be created in QuMSim.jl, however, functions to generate both Creation and Annihilation Operators are provided. They have two arguments: the type of data to store (e.g. Float32 vs Float64) and the max number of photons of the state. For example, a Float32 Creation Operator for a maximum of three photons would be generated with the following:

```
a_dag = CreationOperator(Float32,3)
```

```
Operator:
    Float32[0.0 0.0 0.0 0.0; 1.0 0.0 0.0 0.0; 0.0 1.4142135 0.0 0.0; 0.0
0.0 1.7320508 0.0]
```

Similarly, Annihilation Operators are generated with the following:

```
a = AnnihilationOperator(Float32,3)
```

```
Operator:
    Float32[0.0 1.0 0.0 0.0; 0.0 0.0 1.4142135 0.0; 0.0 0.0 0.0 1.732050
8; 0.0 0.0 0.0 0.0]
```

Both will act on `FockState` types through the natural means of multiplication.

```
c = a_dag*b
```

```
Fock State with
Ket{Array{Float32,1}}(Float32[0.0, 1.0, 0.0, 0.0])
```

```
d = a*c
```

```
Fock State with
Ket{Array{Float32,1}}(Float32[1.0, 0.0, 0.0, 0.0])
```

In order to work at a higher level of abstraction, `Qubit` types are provided. They can be created by passing a `Ket` as an argument.

```
A = Ket([1;0])
Q1 = Qubit(A)
```

```
Qubit with
Ket{Array{Int64,1}}([1, 0])
```

However, `Qubit` types can also be created more succinctly by just passing a vector instead.

```
Q2 = Qubit([0,1])
```

```
Qubit with
Ket{Array{Int64,1}}([0, 1])
```

Like `Ket`, `Qubit` supports Kronecker products.

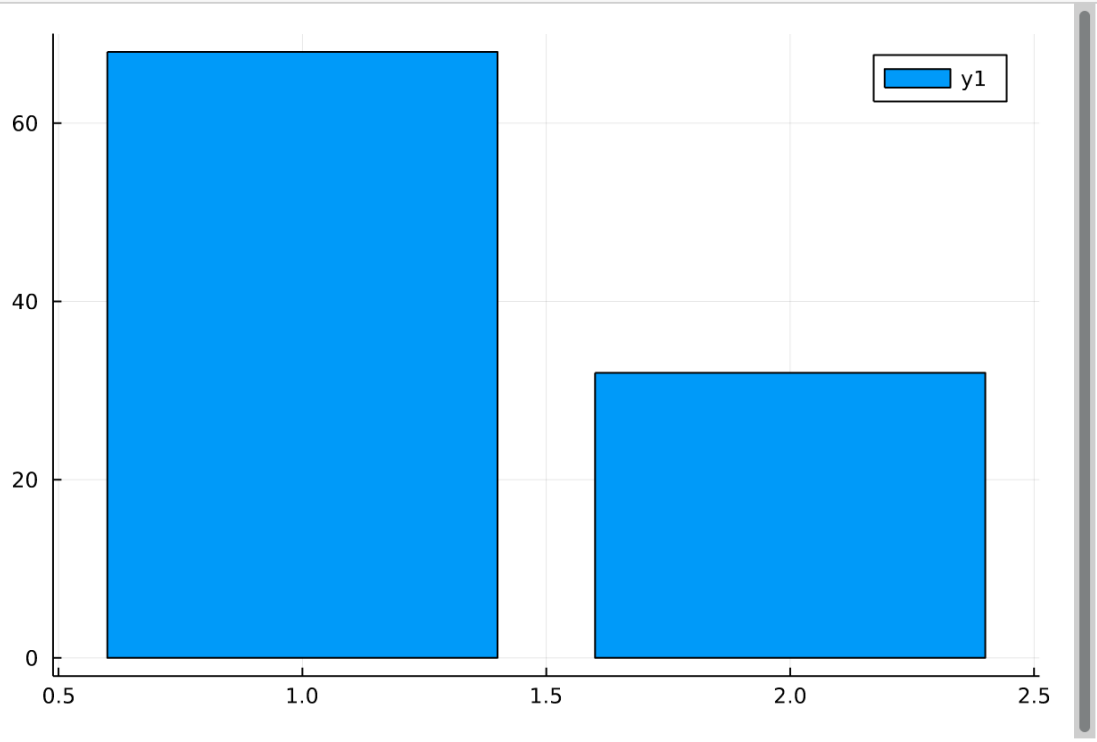
```
kron(Q1,Q2)
```

```
Qubit with
Ket{Array{Int64,1}}([0, 1, 0, 0])
```

Additionally, when a measurement is performed on a qubit in quantum mechanics, the state collapses to one of two states dependent probabilistically on the squares of its magnitudes. This is nonetheless true in QuMSim.jl, and we demonstrate it with the following code snippet.

In this example, a `Qubit` is created 100 times such that it is in state 0 with a probability of 0.75 and state 1 with probability 0.25. In each instance, the state is measured and the outcome is recorded. The distribution of the resulting states is then plotted in a bar graph below. Since each measurement is independent of the last, the resulting probability is not exactly 75/25, which would also be true in a real measurement.

```
state0 = 0;
state1 = 0;
for i=1:100
    A = Qubit([sqrt(0.75); sqrt(0.25)])
    measure!(A)
    if A.representation.state == [1.0; 0.0]
        state0+=1;
    else
        state1+=1;
    end
end

bar([state0,state1])
```



`Pulse` types are created by passing both the type of distribution desired as well as a function modeling its pulse. Currently, temporal and spectral distribution types are supported.

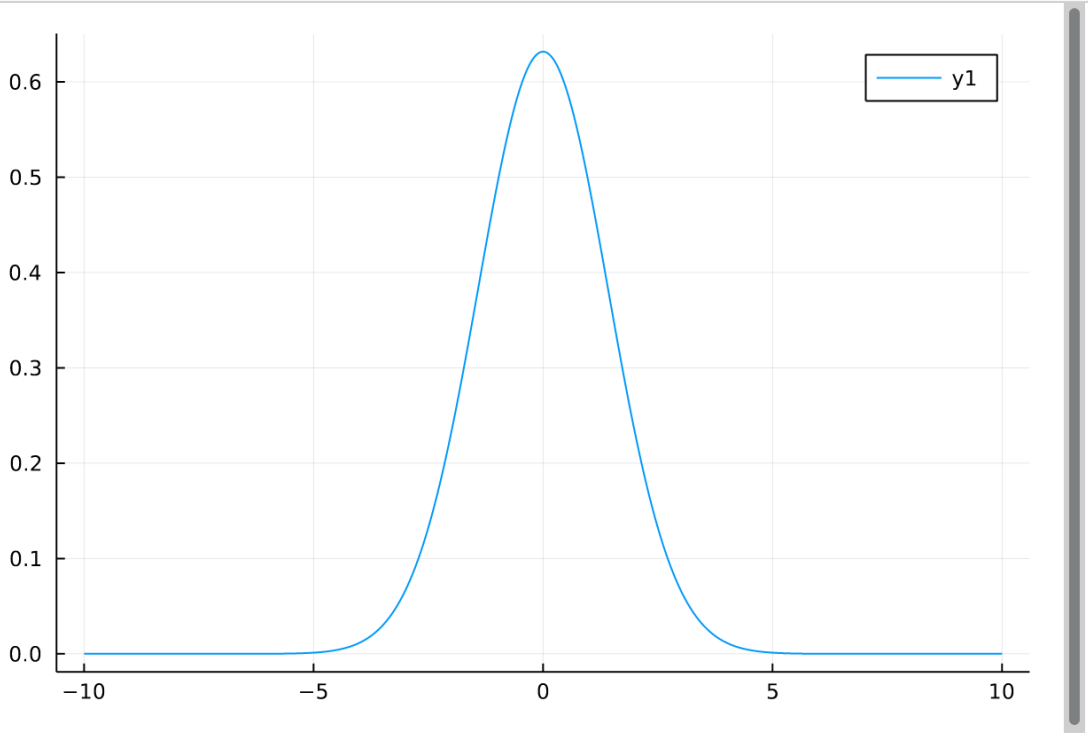
```
myPulse = x-> exp.(-x.^2 ./ 4) ./ ((2*pi)^(1/4));
Pulse1 = Pulse(TemporalDistribution(),myPulse)
```

```
Temporal Quantum Pulse with Distribution
#1
```

This `Pulse` can then in turn be used to created a `Photon`. Once created, both

types can be called and evaluated like a normal function.

```
using Plots
A = Photon(Pulse1)

t = -10:0.01:10
Waveform = A(t)

plot(t,Waveform)
```



CUDAQu.jl extends QuMSim.jl with support for CUDA, thus GPU programming. All previously mentioned functions and types are used the exact same way with `CuArray` arguments instead.

```
using CUDAQu
A = Qubit(CuArray{Float32}([1,0]))
```

```
Qubit with
Ket{CuArray{Float32,1}}(Float32[1.0, 0.0])
```

The shorthand creation of `Qubit` types also works in CUDAQu.jl, and Kronecker products between types can take advantage of the GPU as well.

```
B_ket = Ket(CuArray{Float32}([0,1]))
B = Qubit(B_ket)
```

```
Qubit with
Ket{CuArray{Float32,1}}(Float32[0.0, 1.0])
```

```
kron(A,B)
```

```
Qubit with
Ket{CuArray{Float32,1}}(Float32[0.0, 1.0, 0.0, 0.0])
```

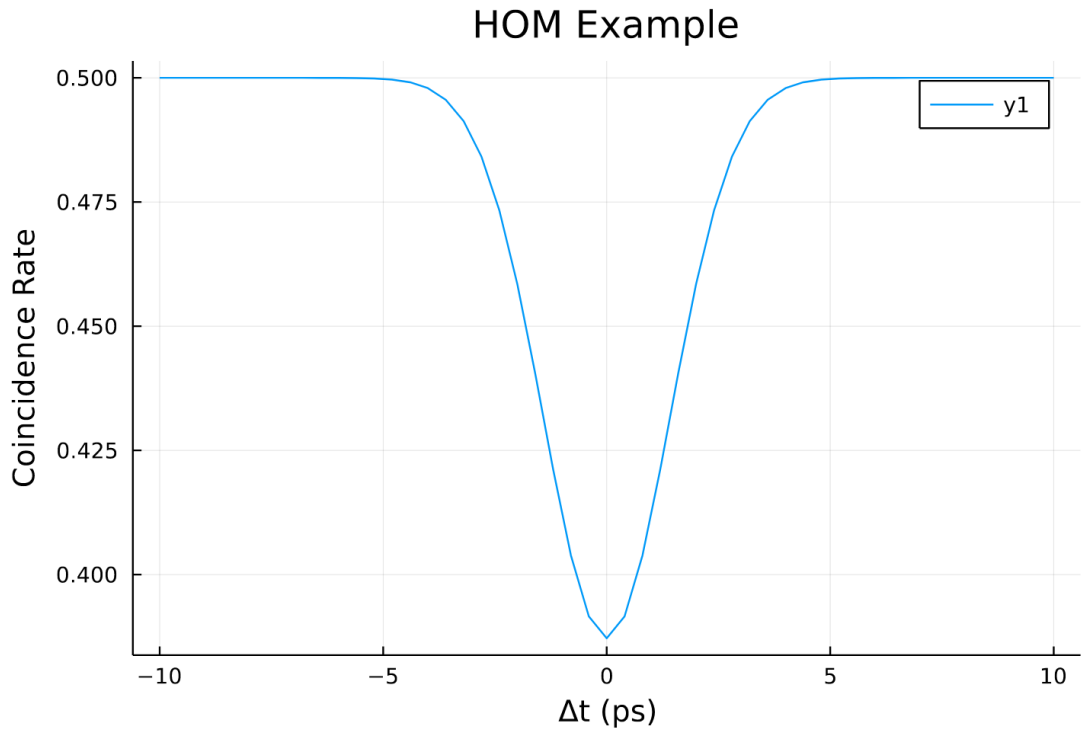
Consistent with how CUDA.jl handles the Julia base language, functions with the same name in both CUDAQu.jl and QuMSim.jl that cannot be determined by context can be called by appending CUDAQu to the function name.

```
a = CUDAQu.AnnihilationOperator(Float32, 3)
b = FockState(CuArray{Float32}([0,0,1,0]));

a*b
```

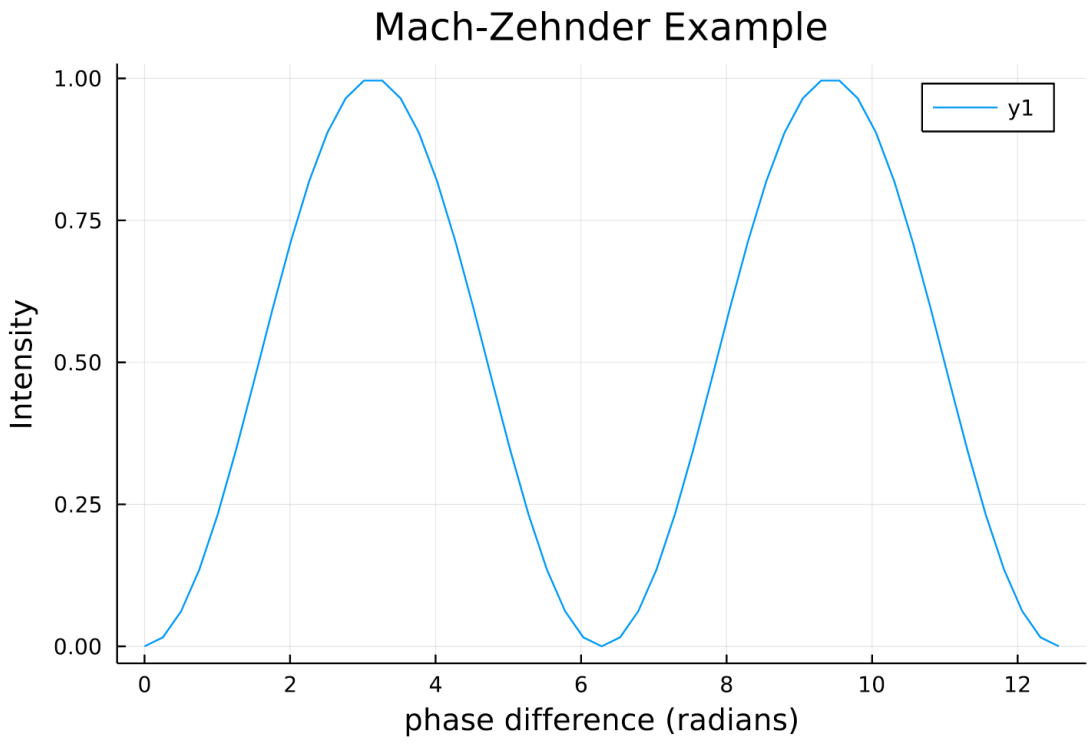
```
Fock State with
Ket{CuArray{Float32,1}}(Float32[0.0, 1.4142135, 0.0, 0.0])
```

As a matter of convenience, functions for both the HOM Interference and Mach-Zehnder experiments are also provided. In the case of the `HOM` function, two `Pulse` types are required as input in addition to the width of the time plot, the maximum plot point, the number of integration points, as well as the number of plot points. In the example below, the HOM function is called such that there are 51 integration points, 51 plot points, and a width of 20 centered at zero. The `MachZehnder` function is called similarly to the `HOM` function but with only one `Pulse` as input.

```
# Hong-Ou-Mandel Experiment - GPU

tmax = 20;
pmax = 10;
nt = 51;
pp = 51;
sigma_A = 1;
sigma_B = 1;

PulseA = Pulse(TemporalDistribution(),
    (t) -> (1/(2*pi*sigma_A^2)^(1/4)) .* exp.(-(t).^2 ./ ( 4*sigma_A^2)))
PulseB = Pulse(TemporalDistribution(),
    (t) -> (1/(2*pi*sigma_B^2)^(1/4)) .* exp.(-(t).^2 ./ ( 4*sigma_B^2)))

(plot_points, r_points) = CUDAQu.HOM(PulseA, PulseB, tmax, pmax, nt, pp)

plot(plot_points, r_points, title="HOM Example")
xlabel!("Δt (ps)"); ylabel!("Coincidence Rate")
```



```
# Mach-Zehnder Interferometer - GPU

sigma_A = 1;

tmax = 20;
pmax = 4*pi;

nt = 51;
pp = 51;

PulseA = Pulse(TemporalDistribution(),
    (t) -> (1/(2*pi*sigma_A^2)^(1/4)) .* exp.(-(t).^2 ./ ( 4*sigma_A^2)))

(plot_points, r_points) = CUDAQu.MachZehnder(PulseA, tmax, pmax, nt, pp)

plot(plot_points, r_points, title="Mach-Zehnder Example")
xlabel!("phase difference (radians)"); ylabel!("Intensity")
```



Appendix C. Simulation Benchmarks

Benchmarks of the software using the BenchmarkTools.jl library provides a means to measure performance [21]. For example, comparing the kron function included in CUDAQu.jl to that of the one in the base library reveals a 417x reduction in runtime, a result not overly surprising considering the difference in CPU and GPU execution. All benchmarks are made on a computer with dual 2.40GHz Xeon 4210R processors and 192GB of RAM using Julia v1.5.4.

```
A = randn(Float32,100,100)
B = randn(Float32,100,100)
@btime kron($A,$B)
```

```
  205.439 ms (2 allocations: 381.47 MiB)
```

```
A = CUDA.randn(Float32,100,100)
B = CUDA.randn(Float32,100,100)
@btime kron($A,$B)
```

```
  492.413 µs (55 allocations: 1.78 KiB)
```

A quick test to measure the overhead of using the software as compared to performing the calculations without shows only shows only a 2.4% increase in runtime. Since all calculations dispatch down to the predefined methods of the underlying data, the overhead is solely in the time it takes for the dispatch to occur.

```
A = [1,0,0,0]
B = [0,1,0,0]
@btime kron($A,$B)
```

```
  197.750 ns (7 allocations: 480 bytes)
```

```
A = Qubit([1,0,0,0])
B = Qubit([0,1,0,0])
@btime kron($A,$B)
```

```
  202.545 ns (8 allocations: 496 bytes)
```

A more interesting examination is in how the runtime is affected by increasing the number of integration bins used in the calculation. CUDAQu.jl tends to increase

approximately linear to the number of bins used while QuMSim.jl increases at a more exponential rate. Examining how this in turn impacts the accuracy of the simulation to the theoretical results, we see diminishing returns for both libraries after 23 integration bins, when error drops below one percent. Memory usage and allocations are also shown as the number of integration bins are increased in Figures 6 and 7.

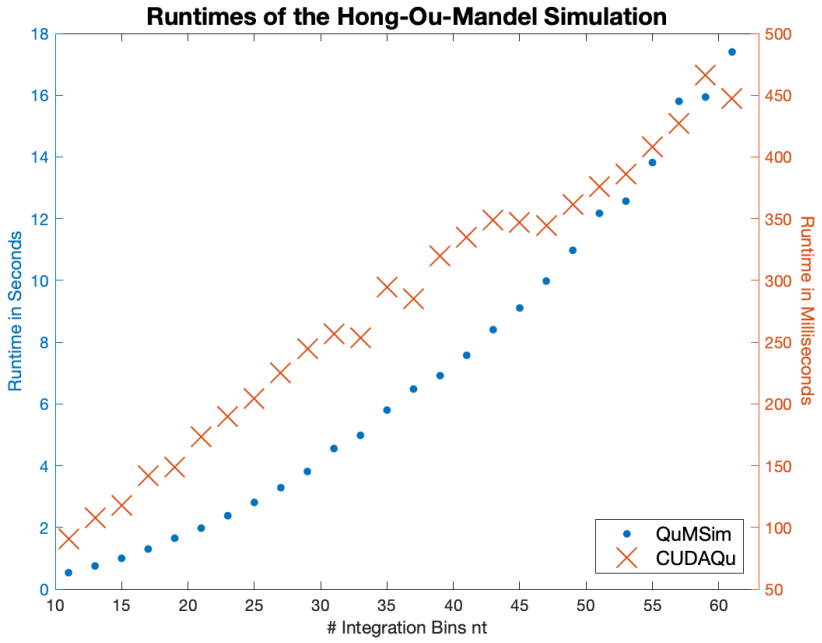


Figure 4: Runtimes of the HOM Simulation as the number of integration bins are increased.

Performing the same experiment with the Mach-Zehnder Interferometer Simulation shows an approximately linear increase in runtime for QuMSim.jl but no clear pattern for CUDAQu.jl. In fact, the longest runtime was at 15 integration bins. The error plot for this simulation is very simular to that of the HOM Simulation with the error dropping below one percent at 23 integration bins. Memory usage is interesting, as it stays constant for several bin increases before stepping to a slightly higher about of memory. As was done for the HOM Simulation, the number of allocations versus

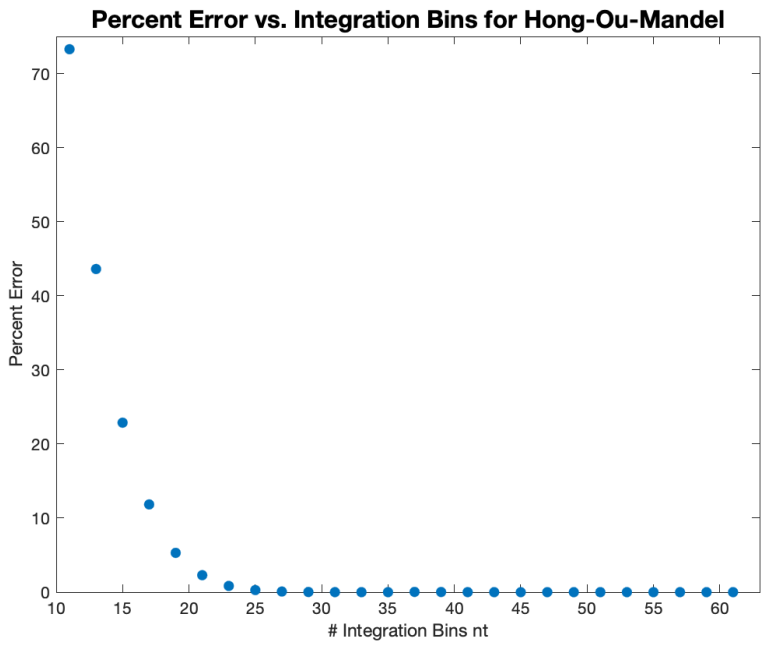


Figure 5: The error in the HOM Simulation in percent as the number of integration bins are increased. The simulation drops to less than one percent error after 23 integration bins are used.

the number of integration bins are provided in Figure 11 as well.

To get a more accurate estimation of the simulations' runtime, an average is taken over 100 executions for each. The point where the simulations' error drops under one percent (23 integration bins for all cases) is selected, and the results are plotted in Figures 12 and 13. Runtimes of QuMSim.jl's HOM and Mach-Zehnder Interferometer Simulations were comparable, at 2.3107 and 2.3292 seconds respectively. There was much larger discrepancy in the runtimes from CUDAQu.jl, however. The Mach-Zehnder Interferometer Simulation was about 3.8x longer than the HOM Simulation's despite being simpler in terms of computation. This result is attributed to the matrix exponentiation performed in the main loop of the Mach-Zehnder code. Summary statistics are provide in Table 1.

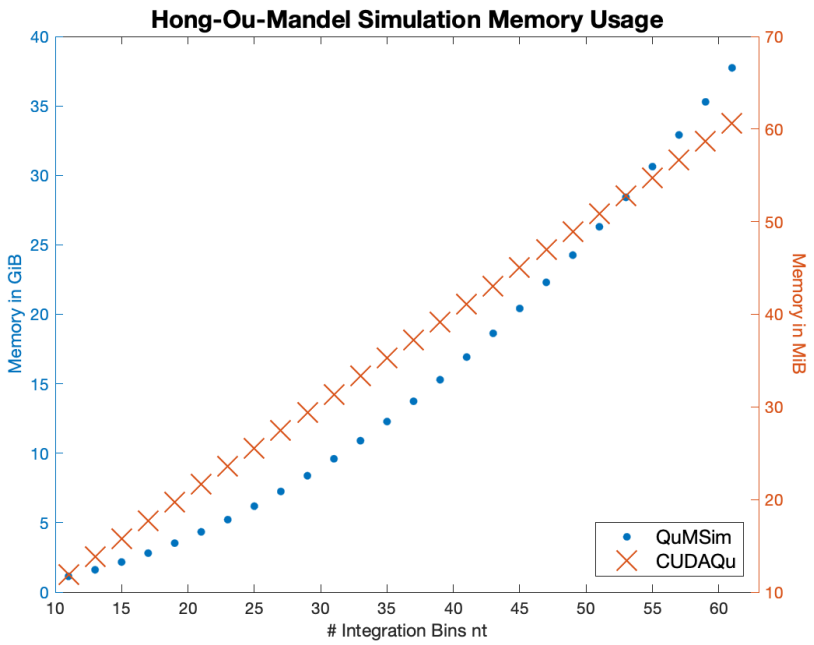


Figure 6: Memory usage of CUDAQu.jl and QuMSim.jl in performing the HOM Simulation as the number of integration bins are increased.

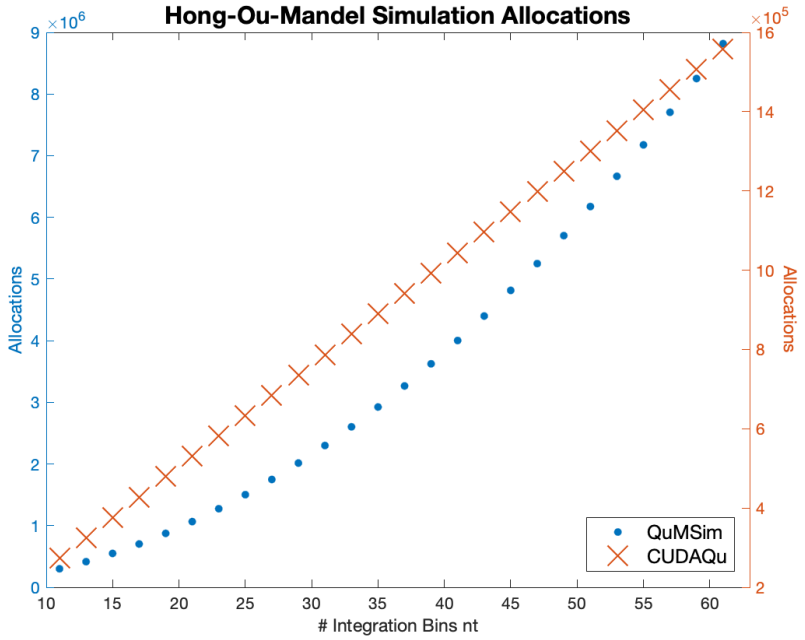


Figure 7: The number of allocations of CUDAQu.jl's and QuMSim.jl's HOM Simulation as the number of integration bins are increased.

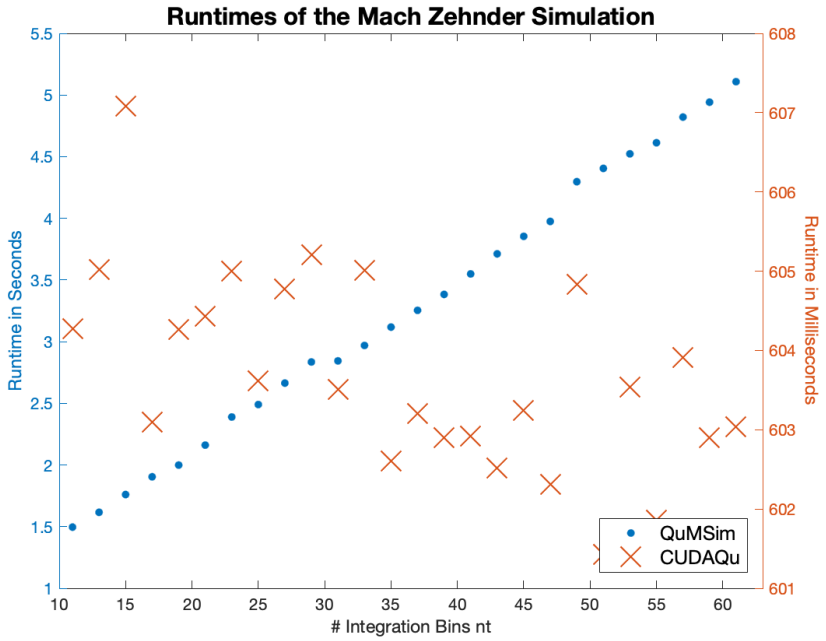


Figure 8: Runtimes of the Mach-Zehnder Interferometer Simulation as the number of integration bins are increased.

Table 1: Simulation Summary: Overall summary statistics for each simulation performed.

Statistic	N	Mean (s)	St. Dev. (s)	Min (s)	Max (s)
QuMSim HOM	100	2.31	0.07	2.22	2.62
CUDAQu HOM	100	0.17	< 0.01	0.16	0.17
QuMSim Mach-Zehnder	100	2.33	0.01	2.31	2.40
CUDAQu Mach-Zehnder	100	0.64	< 0.01	0.63	0.64

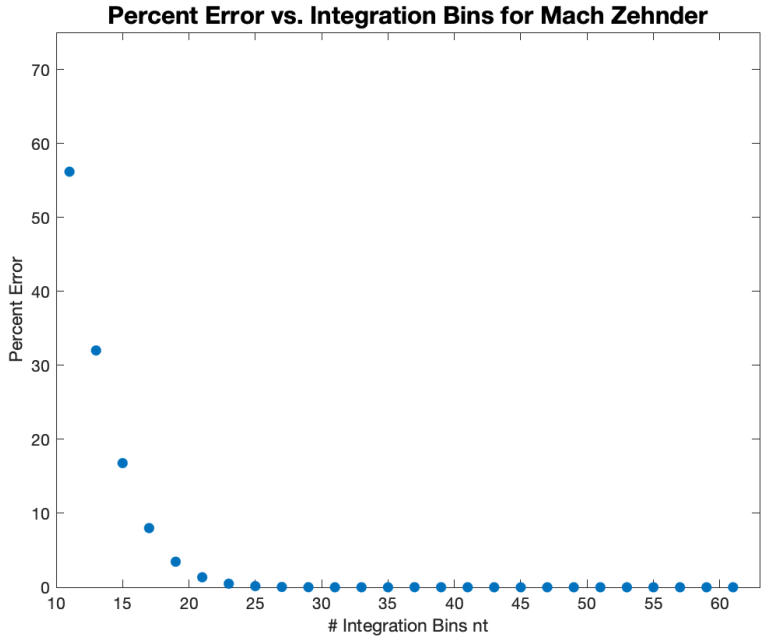


Figure 9: The error in the Mach-Zehnder Interferometer Simulation in percent as the number of integration bins are increased. The simulation also drops to less than one percent error after 23 integration bins are used.

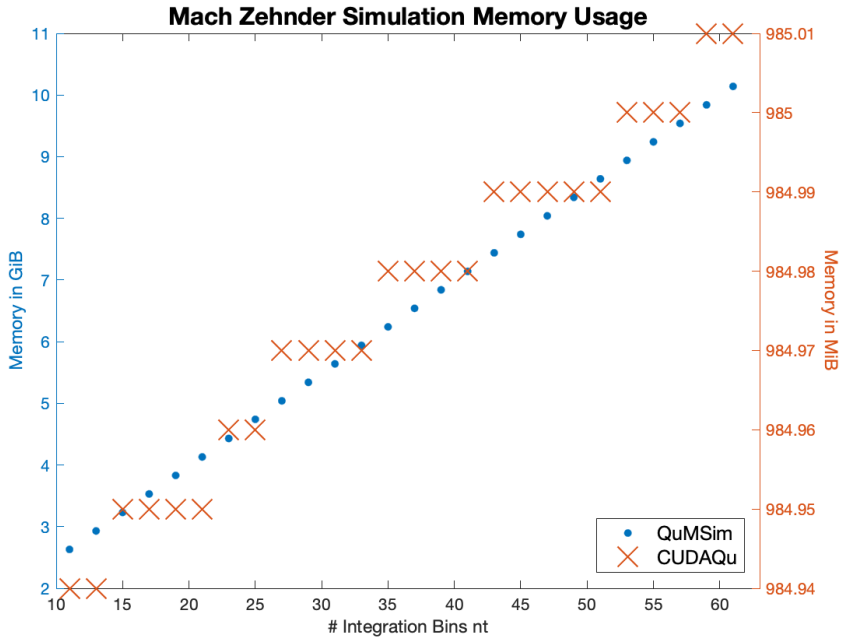


Figure 10: Memory usage of CUDAQu.jl and QuMSim.jl in performing the Mach-Zehnder Interferometer Simulation as the number of integration bins are increased.

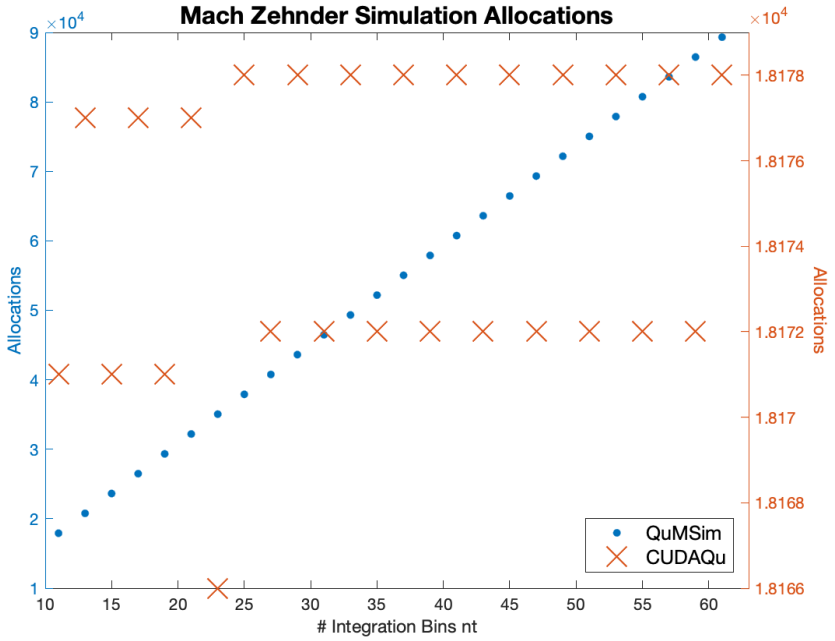


Figure 11: The number of allocations of CUDAQu.jl's and QuMSim.jl's Mach-Zehnder Interferometer Simulation as the number of integration bins are increased.

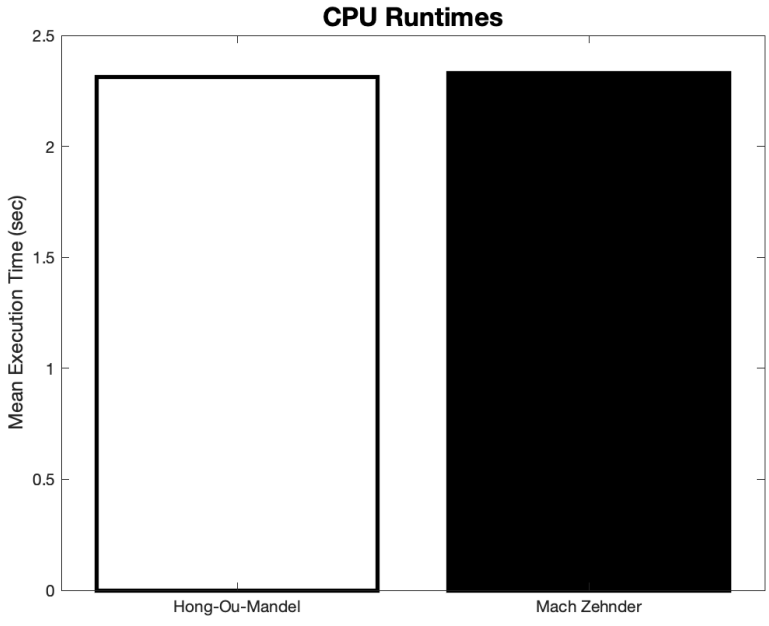


Figure 12: Mean runtimes in seconds of the CPU simulation experiments.

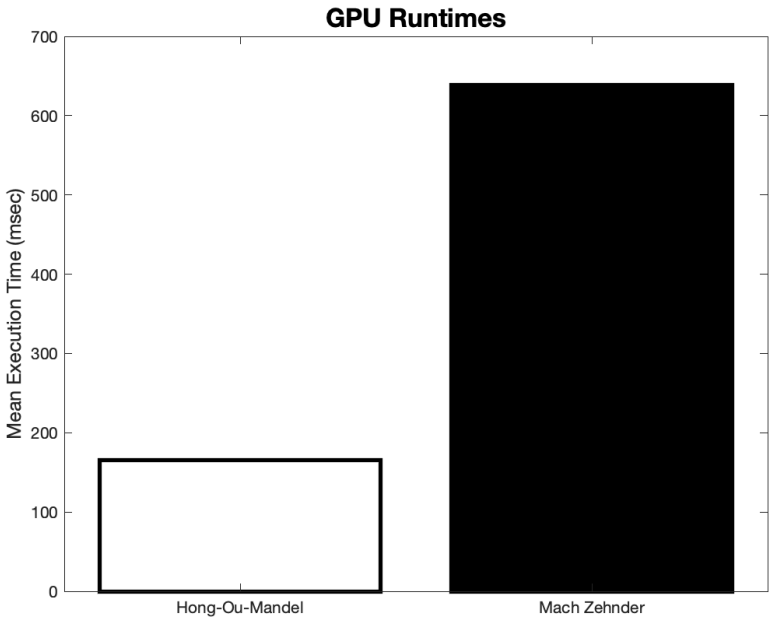


Figure 13: Mean runtimes in milliseconds of the GPU simulation experiments.

Bibliography

1. J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

2. D. Vergun. (2021, February) Quantum Science to Deliver Cutting-Edge Technology to Warfighters, Official Says. [Online]. Available: https://www.defense.gov/Explore/News/Article/Article/2509192/quantum-science-to-deliver-cutting-edge-technology-to-warfighters-official-says/

3. ——. (2021, March) DOD Officials Discuss Quantum Science, 5G and Directed Energy. [Online]. Available: https://www.defense.gov/Explore/News/Article/Article/2530494/dod-officials-discuss-quantum-science-5g-and-directed-energy/

4. C. H. Bennett and G. Brassard, "Quantum cryptography: Public Key Distribution and Coin Tossing," *Theoretical Computer Science*, vol. 560, p. 7–11, Dec 2014. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2014.05.025

5. P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, Oct 1997. [Online]. Available: http://dx.doi.org/10.1137/S0097539795293172

6. D. Franklin and F. T. Chong, *Challenges in Reliable Quantum Computing*. Boston, MA: Springer US, 2004, pp. 247–266. [Online]. Available: https://doi.org/10.1007/1-4020-8068-9_8

7. A. Dahlberg and S. Wehner, "SimulaQron—a simulator for developing quantum internet software," *Quantum Science and Technology*, vol. 4, no. 1, p. 015001, Sep 2018. [Online]. Available: http://dx.doi.org/10.1088/2058-9565/aad56e

8. T. Coopmans, R. Knegjens, A. Dahlberg, D. Maier, L. Nijsten, J. de Oliveira Filho, M. Papendrecht, J. Rabbie, F. Rozpedek, M. Skrzypczyk, L. Wubben, W. de Jong, D. Podareanu, A. Torres-Knoop, D. Elkouss, and S. Wehner, "NetSquid, a discrete-event simulation platform for quantum networks," *Commun Phys 4, 164*, 2021.

9. J. Johansson, P. Nation, and F. Nori, "QuTiP: An open-source Python framework for the dynamics of open quantum systems," *Computer Physics Communications*, vol. 183, no. 8, p. 1760–1772, Aug 2012. [Online]. Available: http://dx.doi.org/10.1016/j.cpc.2012.02.021

10. ——, "QuTiP 2: A Python framework for the dynamics of open quantum systems," *Computer Physics Communications*, vol. 184, no. 4, p. 1234–1240, Apr 2013. [Online]. Available: http://dx.doi.org/10.1016/j.cpc.2012.11.019

11. P. Gawron, D. Kurzyk, and Ł. Pawela, "QuantumInformation.jl—A Julia package for numerical computation in quantum information theory," *PLOS ONE*, vol. 13, no. 12, p. e0209358, dec 2018. [Online]. Available: https://doi.org/10.1371/journal.pone.0209358

12. S. Krämer, D. Plankensteiner, L. Ostermann, and H. Ritsch, "QuantumOptics.jl: A Julia framework for simulating open quantum systems," *Computer Physics Communications*, vol. 227, pp. 109–116, 2018.

13. J. Tippit, D. Hodson, and M. Grimaila, "Julia and Singularity for High Performance Computing," in *Advances in Parallel & Distributed Processing, and Applications*. Springer, Nov 2021.

14. *Advances in Parallel & Distributed Processing, and Applications*, ser. Transactions on Computational Science and Computational Intelligence. Boston, MA: Springer, 2021.

15. J. Tippit. (2021, Apr) Allowed enabling multiple external profilers. [Online]. Available: https://github.com/JuliaLang/julia/pull/38741

16. ——. (2021, Apr) Added support for kron. [Online]. Available: https://github.com/JuliaGPU/CUDA.jl/pull/814

17. T. J. Project. (2017, Aug) JuliaRegistries/General. [Online]. Available: https://github.com/JuliaLang/julia/pull/38741

18. D. Lin, J. M. White, S. Byrne, D. Bates, A. Noack, A. Arslan, K. Squire, D. Anthoff, T. Papamarkou, M. Besançon, J. Drugowitsch, M. Schauer, and other contributors, "JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions," July 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2647458

19. T. Besard, C. Foket, and B. De Sutter, "Effective Extensible Programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

20. T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29–46, 2019.

21. J. Revels *et al.* (2020, April) BenchmarkTools.jl. [Online]. Available: https://github.com/JuliaCI/BenchmarkTools.jl

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–12–2021	Master's Thesis	Jan 2019 — December 2021

4. TITLE AND SUBTITLE

Approaches to Improve the Execution Time of a Quantum Network Simulation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Tippit, Joseph A, 1st Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-21-D-013

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Laboratory for Telecommunication Sciences
Gerald B. Baumgartner, Ph.D.
8080 Greenmead Dr.
College Park, MD 20740
Email: gbaumgartner@ltsnet.net

10. SPONSOR/MONITOR'S ACRONYM(S)

LTS

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A:
Approved for Public Release; Distribution Unlimited.

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Evaluating quantum networks is an expensive and time-consuming task that benefits from simulation. A potential improvement is to utilize GPUs, namely by leveraging NVIDIA's programming framework, CUDA. To avoid performance pitfalls of higher level languages and programming models such as the so called "two language problem," the Julia Programming Language provides the basis for the development effort. This research develops a two module prototype quantum network simulation framework using GPUs and Julia. Performance of the software is measured and compared against other languages such as MATLAB.

15. SUBJECT TERMS

CUDA, Hong-Ou-Mandel, Julia Programming Language, quantum networks, simulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Douglas D. Hodson, AFIT/ENG
U	U	U	UU	65	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, ext 4719