COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

CSCI 370 Final Report

WLG!

Xander Lewis
Amelia Bell
Ethan Miles
Hiram Despain

Revised May 23, 2023

CSCI 370 Summer 2026

Dr. Iris Bahar

Table of Contents

I. Introduction 2
II. Functional Requirements 2
III. Non-Functional Requirements 2
IV. Risks 3
V. Definition of Done 4
VI. System Architecture 5
Introduction 5
Parser 6
Intermediate Representation 6
Target Representations 6
Tools and Applications 7
VII. Software Test and Quality 7
VIII. Project Ethical Considerations 8
ACM/IEEE Principles 8
Michael Davis Tests 8
Ethical Considerations 8
IX. Project Completion Status 9
X. Future Work 9
XI. Lessons Learned 10
XII. Acknowledgments 10
XIII. Team Profile 10
Appendix A – Key Terms 11

I. Introduction

The principal goal of our project is to create a tool which takes an arbitrary quantum circuit coded in Python with CUDA-Q, NVIDIA's library for simulated quantum circuits, and output equivalent representations of the circuit such as a DAG (directed acyclic graph) or a tensor product.

Our client is Will Buziak, a computer science PhD student in Bahar Lab. He has a B.S. in mechanical engineering from the University of Tennessee, Knoxville. He is currently doing research on secure memory architecture design and recently began work on the applications of Machine learning algorithms for the development of Quantum Algorithms, with which we are helping.

One of the more convenient ways to represent a quantum circuit is in a DAG. This representation of quantum circuits is often used in papers and publications in the quantum computing field. This graphical representation of the circuit lends itself well to parsing, manipulation, and modification. Due to this, it serves well as an input for machine learning algorithms. To create a DAG, an intermediate representation is used so the circuit can easily be converted to a different data representation, such as a tensor network. These other methods of representation can be used for a myriad of applications; one such application is the optimal hardware-aware compilation of quantum circuits, a very difficult problem in the space. Unlike many other Quantum SDKs, CUDA-Q does not have a way to convert a quantum circuit into a DAG. Therein lies the motivation for this project: CUDA-Q lacks functionality that is integral to Will's research, and we need to bridge that gap.

This tool will be used by Will over the course of his research to compile workable data sets for machine learning training and, eventually, for other purposes. We hope that this tool would also be of use to the CUDA-Q community, as currently, such a tool does not appear to exist. As of completion, we have handed the work over to Will, and he will maintain and/or update the tool as he sees fit. Additionally, we ensured thorough documentation for future developments so that Will can understand the current utility of the tool.

II. Functional Requirements

There are a number of functional requirements for our tool, some of which are current requirements, and others are requirements that may be added depending on the rate at which we progress through this project. The immediate functional requirements are as follows:

- Design an intermediate representation (IR) of a Quantum Circuit that fully encodes the necessary information
- Create an algorithm that converts the IR into a DAG (without losing information)
- Create an algorithm that converts the IR into a tensor representation (without losing information)
- Parse either CUDA-Q or Quake, isolate the quantum circuits (Clifford-T gate set), and convert them into the IR

The functional requirements that were considered to be reach goals:

- Allow for the parsing of custom gate sets
- Parsing of Qiskit or Pennylane circuits into the IR
- Creation of a data set for machine learning applications
- Conversion to other representations, perhaps more graphs

That second set of requirements, as previously mentioned, may be added in the case that the scope of the project widens. The possible widening of scope is something that has been established by the client, as such we felt it reasonable to include that second set of requirements within this section.

III. Non-Functional Requirements

The main non-functional requirement for this project is that the parser must be exhaustive and scalable, i.e., given any input, the resulting IR or DAG must be correct OR, the project must easily be scaled for more complex circuits that wouldn't originally have a correct output. Performance is not necessarily a requirement as the client understands that an implementation may not run quickly and due to the nature of the project and our implementation, runtime was still quick.

In terms of convenience, our intermediate representation should be intuitive, and our code should be well documented. This is especially important for any future development of our code. If someone were to want to convert a CUDA-Q circuit into some other specific data structure, it is important that our intermediate representation has an intuitive collection of qubits, gates, connections, and the gate orders so that a different target representation could be made. Therefore, it is also important that our code is well-documented since extending our code requires understanding its functionality.

IV. Risks

A major risk of creating a parser is if it is not rigorous or exhaustive. If the parser is not rigorous enough, a valid circuit written with CUDA-Q may lead to an incorrect DAG of the circuit or any number of functional flaws in the interpretation of the code. If the parser is not exhaustive enough, then the parser will fail as a useful tool needed for implementation of DAGs from quantum circuits. A non-exhaustive parser may throw syntax errors at valid code which will limit its utility as a tool for making DAGs from CUDA-Q (since not all circuits will be convertible to a DAG).

There are multiple stages of the pipeline, and we are only creating one target representation. This creates a risk in losing essential information for the target representation. This can be mitigated by ensuring that our intermediate representation has all the information that is essential to reconstruct the circuit. Since the data of the IR is important for creating various target representations, it is important that if someone wanted to expand our work with another target representation, they could do so easily.

Risk Table:

Asset	Threat	Vulnerability	Likelihood (1-5)	Impact (1-5)	Risk (1-25) Max:10	Treatment	Control
Quantum Circuit and DAG are equivalent	Outputted DAG is assumed to be equivalent to quantum circuit, but isn't	Non-rigorous parser	Low (2)	Serious (3)	6	Avoidance	In-depth testing of the parser/DAG conversion tool with numerous

							quantum circuits to ensure rigor
Quantum Circuit Representation	Potential invalid inputs or outputs are never verified	The tool that uses the parsers uses invalid circuits	Likely (4)	Low (1)	4	Reduce likelihood	The outputted DAG or even the code itself should be verified so it is never assumed valid.
Holistic usability of the parser	Valid CUDA-Q code cannot be converted into a DAG	Non-exhaustive parser	Medium (3)	Low (1)	3	Reduce Likelihood	Use the proper ASTs so all valid syntax can be parsed
Information in the target representation	Information is lost between some representation.	The circuit representation is changed multiple times	Low (2)	Serious (3)	6	Avoidance	Ensure the IR has all gates, qubits, connections, and order of gates.
Accessibility of the intermediate representation (IR).	Our parser is inconvenient to make other circuit representations	There is no standard IR for a quantum circuit	Medium (3)	Concerning (2)	6	Reduce likelihood	Create an intuitive and well commented IR

V. Definition of Done

The definition of done for our project would be being able to parse any given quantum circuit in CUDA-Q that meets the scope of our implementation (up to but not including custom gates and classical control) and turn it into a DAG. This would include verifying that we have created a valid and expected DAG. This involves exhaustive testing of every step of our parser to ensure that they match what is expected.

Another requirement is that it is well documented so that it can easily be extended (or modified) in the future if our tool is integrated into a larger workflow. This would include documenting our algorithms' functionality and creating succinct but verbose tests. This would require that we have passing tests for basic CUDA-Q features (such as a GHZ circuit), and that we have failing tests for features that could be implemented in the future (such as custom gates).

VI. System Architecture

Introduction

Our system acts as a tool to convert from a circuit written in CUDA-Q Python to target representations of the circuit. At its most basic, it is able to parse quantum circuits in CUDA-Q and output a variety of useful representations. Our intermediate representation is designed so that these different target representations could be added in the future.

We have decided to split our product into four broad areas in order to organize the architecture of our system, as seen in figure 1. These sections are as follows: first, the parser, which will handle parsing the CUDA-Q; second, the intermediate representation (IR), which will store the circuit once parsed; third, the other representations, this includes all valid circuit representations that we are converting to; and finally, the tools and applications. These four sections work in unison to create a tool chain for the conversion of CUDA-Q circuits into much more useful representations.

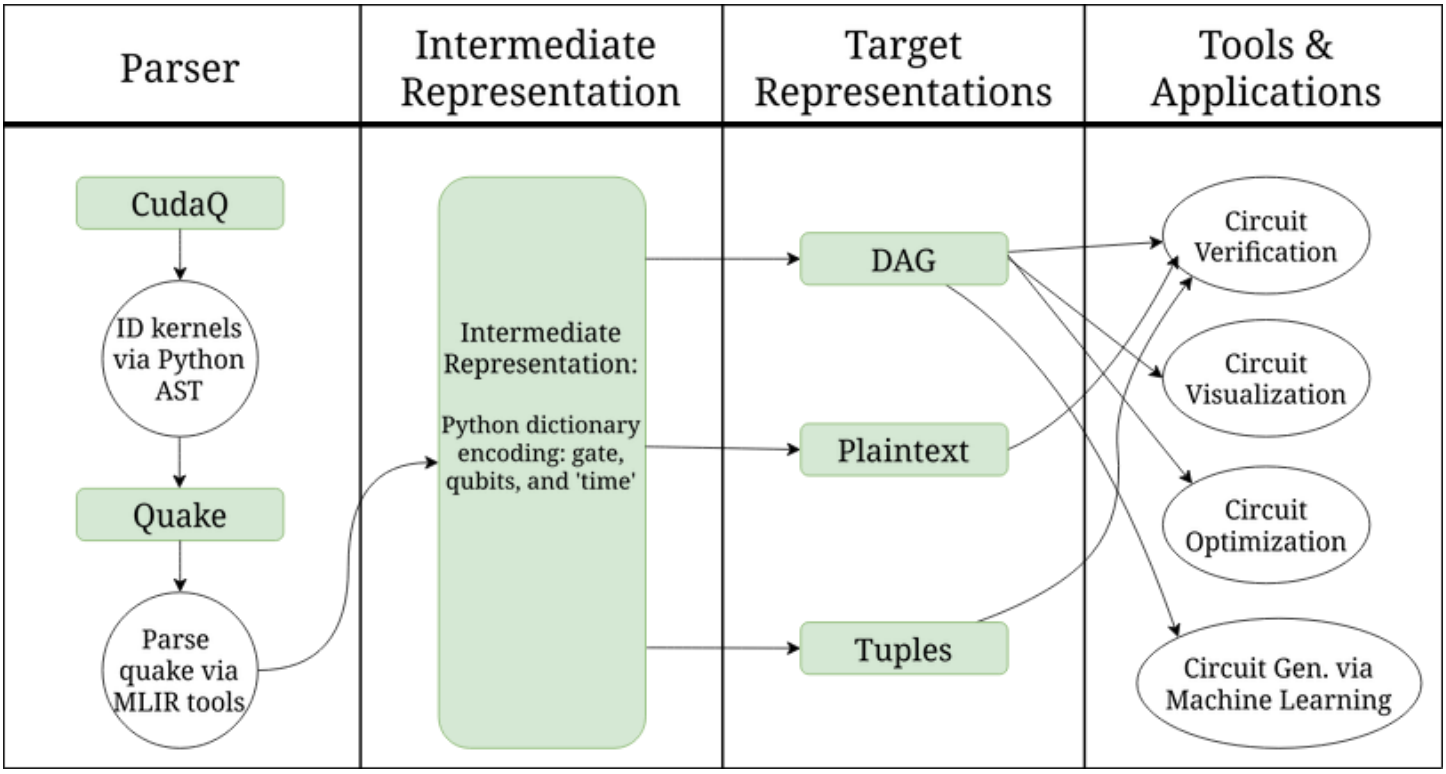


Figure 1

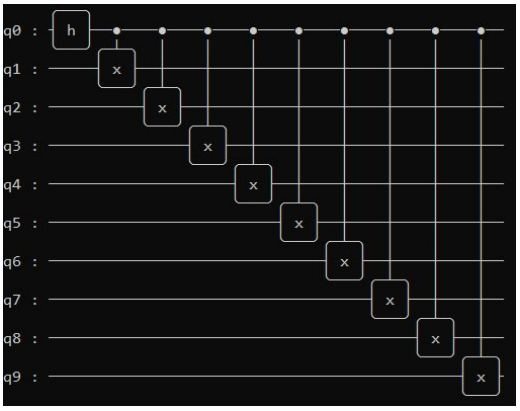


Figure 2

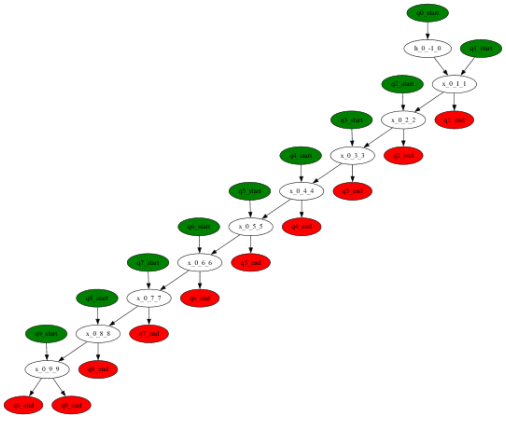


Figure 3

An example of this process is shown with the quantum circuit in figure 2 being converted to the DAG in figure 3. Figure 2 is a diagram of a quantum circuit printed out with CUDA-Q; the traditional way to notate a quantum circuit instead of code. To convert this circuit, the following tool chain processes are used.

Parser

As seen in Figure 1, the parser is the first part of our system. Its job, put simply, is to parse the CUDA-Q code and return an intermediate representation that is fully representative of the circuit. Unfortunately, this is something that could have been accomplished in a number of different ways. Aiming to create a good product, we have explored various methods of parsing CUDA-Q. Briefly explaining these forays into other methods will provide the motivation for our current design choices as they pertain to the parser. Our first approach to the problem of parsing CUDA-Q was to create a custom parser to parse the code. We played around with the idea of parsing Quake (a dialect of MLIR for quantum circuits developed by Nvidia), but found that to be unwieldy, especially when it came to the complexities of hybrid quantum-classical kernels. We also briefly considered directly parsing the Python code without support; Python is, unsurprisingly, much harder to parse than Quake. As such, we decided to move away from creating a custom parser and turned to some of the preexisting parsers for MLIR.

While researching premade parsers, we discovered the cudaq.translate() method that will transpile CUDA-Q kernels into other standard quantum languages, such as QIR and OpenQasm. To our great dismay, we found that translation to OpenQasm is only partially supported and essentially useless for our purposes. Additionally, the method only translated to a now outdated version of QIR. We decided that basing our whole functionality on a dubiously maintained method would perhaps not be the smartest option. Quake, as aforementioned, is a dialect of MLIR; therefore, it can be parsed with the MLIR toolchain. This is a very promising avenue, but it appears that we will need to include some dynamically linked libraries so that the MLIR parser can correctly identify and parse the Quake; Quake isn't an official dialect, complicating the parsing.

Something that has gone unmentioned until now is that, even if we were to be able to correctly parse the Quake, QIR, OpenQasm, or Python, we would still need to be able to correctly identify the location of the CUDA-Q kernels and collect their information. Due to the questionably lenient standards of Python, the creation of a CUDA-Q kernel could be hidden beneath many layers of functions and classes. Consequently, the only reliable way to identify kernels is via a traversal of the AST representation of the Python code, searching for the PyKernel and PyKernelDecorator types. Once those are found, they can be pulled out, converted to Quake and parsed with the MLIR tools and converted to the intermediate representation.

In the end, we chose to pull the CUDA-Q kernels out of the Python AST(s) using the built-in Python AST library. This was then converted to Quake and parsed using the MLIR library where the data was stored in an array of objects for further use.

Intermediate Representation

Our intermediate representation was a data structure that represents the flow of quantum gates from the starting qubits to their final state. We used a Python dictionary to accomplish this with the input(s) to any given gate being the key and the output(s) (with labeling as to what the gate was) being the value. This allowed easy tracing to simplify the implementation of our tools. This was the data that is actually stored for further use and was an equivalent, but simpler (as in more human-readable/easily interpreted by an algorithm) representation of a quantum circuit. This allowed for easy use in applications/visual representations such as a DAG. We ended up implementing this with gate objects as the key/value pairs and strings to represent the start and end qubits.

Target Representations

The circuit representations we converted into were chosen with the potential application to machine learning in mind. Our primary goal for this to start was a directed acyclic graph for the reasons listed below as well as DAGs being a common (if not standardized) representation of quantum circuits. In a DAG representation, the root nodes and leaf nodes represented the qubits, and the branch nodes represented the gates, with the edges representing the sequence

in which gates were applied to the qubits. We also implemented a tuple representation containing the gates in order with their name, qubits, and time step, as well as a plaintext representation of the gates.

Tools and Applications

A convenient representation of a quantum circuit has the potential to be used for numerous tools. Assuming there are no errors in the parsing and representation, then a directed acyclic graph of the circuit could be used to verify the validity of a circuit by considering in and out degrees of nodes, as well as timing factors with how qubits operate in a quantum circuit. A circuit being represented as a DAG certainly makes it easier to change that representation to something more valuable for a different application. For example, a circuit could be represented as matrices.

These data structures can be used for simulation and as a tool for understanding. A static depiction of a quantum circuit holds value for its structure, but a dynamic model of a quantum circuit may be useful in understanding the functionality of a quantum circuit. Beyond just simulation, a representation of a quantum circuit can be used with machine learning to produce circuits that may be tedious to construct otherwise. Quantum circuits can be used in chemistry by using qubits as representations of molecular interactions. Accounting for precise molecular interactions is a problem that scales exponentially, and thus classical computing is not as optimal. An effective ML model may be able to construct quantum circuits that are representations of molecules and their interactions efficiently. Another application for quantum circuits is cryptography. Quantum mechanics are unpredictable and truly random. A model of a quantum circuit can be used to understand or create effective means of encryption. While encryption might not necessarily entail the use of machine learning, it is possible it could pose a risk. Quantum circuits can compute difficult problems with extreme efficiency, and this involves computing large prime factors used in many encryption methods such as with RSA. Shor's algorithm can find prime factors in polynomial time, posing a threat to modern security. When paired with machine learning, this cybersecurity threat could be optimized and streamlined. Besides these potential applications, the use of generating quantum circuits is still a new field, and it currently only finds value over classical computing in niche scenarios. This makes the application of machine learning and quantum circuits expansive, but it is still nebulous.

VII. Software Test and Quality

We wrote our own testing software to ensure that our implementation retained high quality and meant all of the requirements laid out by our client. To do this we wrote code that passed every gate to both parsers. This ensures that every gate is handled as if the parsers can parse them once, no matter the actual circuit that is passed as an input, it can parse every gate every time. We also made code to track how many qubits, gates, and what types of gates there were in the original circuit input, then verified that both our intermediate representation and target representation(s) qubits/gates matched the original circuit.

Throughout the project, we have also collaborated to ensure quality. We have reviewed each other's code to spot any bugs that may have been inadvertently implemented during the development phase. We oftentimes fixed them before they become harder to track down as our codebase expanded, especially when we started to add tools and more representations. Finally, we have communicated with our client to make sure that the features we add align with his idea of what the output should look like. The user interface testing wasn't that big of a concern as there wasn't a GUI to go along with our tool, rather we documented our code with a "how to use it" to make sure that any user understands the prerequisite libraries they must install along with how to pass inputs and get the outputs they want. Finally, we ran every change by our client, explaining exactly how it works and what it does to ensure that everything we implemented met what our client wanted.

Due to the nature of our project and given that it is just a tool that does not have to interface with any other systems, there really wasn't any necessary integration testing that we'll have to do; our testing revolved around ensuring correctness and ease of use.

VIII. Project Ethical Considerations

ACM/IEEE Principles

The ethical considerations most relevant to our project are found within the Product (3) and the Profession (6) sections of the IEEE Software Engineering Code of Ethics. Given the limited time frame of our project, it is even more essential that we effectively consider the standards and quality of our final product. It is specifically important that we adhere to 3.02 and 3.10 since realistic goals and proof of quality are essential for a five-week time frame. It is also important to follow 3.08 and 3.11 for the nature of our project to be a tool that should be expanded upon. If it is difficult to understand why our code is doing what it is, then it will be a less effective tool. This is also why Profession is important. It is important that our code is credible and professional. As a tool, principle 6.02 and 6.04 are particularly important so that our code is beneficial to the professionals in software engineering.

As a group, we must stay aware of how closely we are following the Product principles throughout the project. Logistically, if we do not set a realistic schedule or expectations (3.01/3.02), we will not create a complete product for our client. If we fail to adequately test our product (3.10), we may also fail to create an entirely functional and acceptable product. We are also at risk of failing principle 3.07 since we are working as a group, and thus we might not all understand each other's code. This could lead to a design that is inefficient or ineffective if our design isn't synchronized.

Michael Davis Tests

Legality test: Would this choice violate a law or policy of my employer?

As this project is something that may be used in a future paper, there are a number of things to consider concerning the ownership and publication of the code. To begin, the parser does utilize code from the CUDA-Q codebase (specifically the Quake definition files) these files are all under Apache 2.0. As such, if we distribute these files, they must keep their copyright notices intact. We would also need to keep a full copy of the Apache 2.0 License in the repo. If we modify anything, it would need to be stated. Second, as this code is being made for research purposes and within the scope of a class, I believe that we would need to speak with the school to sort out ownership and, afterwards, publication and licensing of the code. We should figure out the specifics on that before handing the code over to the client so that we can correctly disclose that.

Mirror test: Would I feel proud of myself when I look in the mirror?

Yes, we have absolutely no qualms about this project. This project will be used for research regarding quantum algorithms. We suppose those algorithms could be applied in ethically concerning ways (eventually), but that is (at minimum) two degrees of separation away from what we are doing and likely a decade away. This is about as ethically concerning as being a farmer who sells food to a middleman who then, unknowingly, sells it to ethically questionable people.

Ethical Considerations

If our quality plan is not implemented properly or is not comprehensive enough, we will have several ethical considerations. Not properly setting reasonable goals for the timeframe will likely lead to an unfinished product. The lack of adequate testing will lead to a product that does not work and could cause bigger issues down the line if it is assumed that it does work. Documentation is critical as our product will likely need to be expanded and enhanced if our client wants to continue using it over a long period of time, even more so if he wishes to distribute it for public use. Documentation is also highly important to understand each other's code and work together efficiently, and to make sure that it can be used by others easily. If we don't communicate properly with our client, we will very likely have issues regarding the actual functionality and the expected functionality differing as well as issues regarding the feasibility of certain aspects. Not properly checking our own and each other's work can lead to bugs that are difficult to find and fix. Not being diligent about copyright and licensing can cause difficult legal issues with regard to the product and could cause problems for us and the client.

IX. Project Completion Status

Among the most important parts of our project is the ability to correctly parse quantum circuits as defined in CUDA-Q. The most natural way to accomplish this is with two parts: first, a CUDA-Q/Python parser and, second, a Quake Parser. The Python parser can currently handle both Jupyter notebooks and normal Python files. It begins by taking in a file and scans the AST for CUDA-Q kernels. Once it has identified all statically defined kernels, it will execute the code found in that file in order to retrieve the live Python object required to output the Quake representation of the Quantum Circuits. That being said, there are two very large caveats here. The first is that if a quantum circuit is defined within a function, we cannot retrieve the live object associated with it. Second, as previously mentioned, this process requires that the file is executed, something which can be very time consuming.

Now, moving on to the Quake parser. This has all of the functionality that was originally requested by the client. It can parse the most common gates (and a number of uncommon gates) with ease and output the intermediate representation without losing any information. It does not currently handle custom gates, nor does it currently handle classical control flow.

Our intermediate representation got fully fleshed out using objects to represent each gate. It links a qubit through each gate that is applied to it from start to finish; it equivalently represents the original quantum circuit in a way that eases conversion into other representations.

For each target representation, there is a function that traces through the intermediate representation to gather the information and output the final representation. There is currently functionality for directly printing the gates as they are stored in the intermediate representation in order, two types of DAGs, as well as a tuple representation with the format (gate, [qubits], time). These have been tested on a variety of circuits and checked with visual inspection looking for the correct number of starting and ending qubits, the correct number of qubits going in and out of gates, and the correct number of gates. Since each gate is represented by a node, as well as start and end qubits, simply counting these nodes can verify the correctness of the ou Visual inspection was used because the outputs are primarily visually based.

There are a number of reach goals that we were unable to reach. First, the parsing of custom CUDA-Q gates. This would require much more exploration of the Quake MLIR dialect and CUDA-Q itself, in a direction unrelated to most of the other requirements, making it hard to justify a foray into that area. Second, the handling of classical control within the CUDA-Q kernel definitions. As it stands, there are ~50 Quake operations (of which we handle around 20). If we were to consider the operations introduced by the classical controls, it would easily double the number of operations to handle and complicate the parsing. Third, there was a reach goal involving machine learning. Due to the complexity of the parsing of CUDA-Q and the creation of data representations, we had very little time and there is absolutely no way there would've been sufficient time to pivot away from parsing and into machine learning.

X. Future Work

Future work for our project would involve not only ensuring that our tool remains working correctly on future versions of CUDA-Q and/or more complex circuits but also extending its use for other applications. As CUDA-Q continues to be updated by Nvidia, we may have to update our tool so that it continues to work correctly. Of course, our output can be used for numerous applications such as machine learning or circuit optimization, but if a new application emerges that cannot use our output directly, we would like to update it to fit these needs.

A lot of the future work would also be the completion of the reach goals and extension of the tool and parser. It would be a very natural extension of the project to add many, many more target representations as the needs appear. Second, it would be reasonable to extend the parser to be more comprehensive. As it stands, there are still numerous Quake operations that cannot be parsed; the majority of which do not appear often in circuits. Notwithstanding, being able to handle those operations would make the user experience smoother and less frustrating. Another direction for future work would be overhauling the intermediate representation to be a graph. This seems to be a much more natural

way of storing the data pulled from Quake (especially now that we've found a way to traverse the Quake AST as a graph).

XI. Lessons Learned

Over the course of five weeks, we have learned technical and soft skills through working collaboratively on our project. In terms of knowledge, we have learned the basics of quantum circuits and got time to put knowledge from classes into practice (such as parsers and graphs). We have also worked on our skills to maintain an effective team workflow with GitHub and with the importance of good documentation. Since our program is a tool for CUDA-Q, we have realized the importance of making it near-trivial to use our circuit representations for any relevant application.

Apart from technical skills, we have learned how to effectively work and communicate with a client. While our meetings were unguided at the start of field sessions, we have learned how to prepare ourselves with what we need to discuss and how we will move forward. We have developed skills to work well with each other, and how to set goals and expectations for our team. These are both skills that are foundational to working in any industry.

XII. Acknowledgments

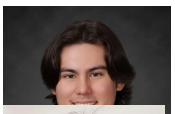
We would like to thank our client, Will Buziak, for being so understanding with not only the scope of the project, but what he wanted us to accomplish. He knew that we would likely not be able to implement custom gates and classical control due to the timeframe of field session and the complexity to implement either of these features.

We would also like to thank our faculty advisor, Iris Bahar, for asking us questions during our meetings that actually made us think about our implementation and thus improve its usability and scalability.

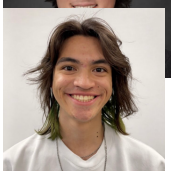
XIII. Team Profile

HIRAM DESPAIN – This coming academic year will be my final one as a computer science student. After that, I will begin on my master's degree in quantum engineering (a program to which I have already been accepted). I hope to complete the thesis software track. I am highly interested in quantum compiler theory and have been reading papers in that area for some time now. After the end of the advanced software engineering course, I will do research with a CU Boulder professor on quantum circuit cutting (gate and wire cutting) and gate/qubit teleportation (normally via EPR pairs and flying qubits). This project slots quite nicely into my interest areas and for that I am rather grateful.

XANDER LEWIS– I am a Junior in Computer Science and I was interested in the project because I find quantum mechanics and related fields to be very interesting.



ETHAN MILES – Starting next semester, I will be a senior in computer science, then I will go on to do a master's degree in the same field.



AMELIA BELL – I am a Junior on CS + Space track. This upcoming Fall, I plan on studying abroad at Temple University in Tokyo for machine learning and humanities, arts, and social sciences.

Appendix A – Key Terms

Include descriptions of technical terms, abbreviations and acronyms

Term	Definition
DAG	*Directed Acyclic Graph*
Qubit	*A quantum bit: the lowest representation of data in any quantum circuit*
CUDA-Q	*Nvidia's quantum library*
Quake	*A dialect of MLIR; CUDA-Q has a built-in method to compile a kernel to it*