4. Bridging Quantum and Classical: A framework to convert a CUDA Quantum code to Qiskit

4.1. Introduction

The explosive growth of quantum computing has seen the emergence of various programming frameworks, each focused on particular hardware and computational models. Yet this heterogeneity brings with it a serious challenge: that of interoperability. In particular, the split between CUDA-Q, which focuses on hybrid quantum-classical flows and GPU acceleration, and Qiskit, which targets gate-based quantum computation in IBM's hardware, becomes an obstacle to developers wanting to take advantage of the best features of both worlds. This project fills this gap by providing a translation library, based on Abstract Syntax Trees (ASTs), that addresses the translations between CUDA-Q programs and Qiskit code seamlessly. This project has the vision of democratizing IBM's quantum hardware access to users of CUDA-Q, without requiring heavy code rewriting, and accelerating a more cohesive and efficient quantum computing framework. Through this cross-platform capability, we aspire to speed up quantum algorithm development, increase the productivity of developers, and facilitate an open environment in which the advantages of various quantum computing frameworks can be easily mixed and matched.

4.2. Problem formulation

The need for this CUDA-Q to Qiskit translation initiative is motivated by a number of key considerations in the emergent quantum computing ecosystem. First, an explosive growth in quantum hardware and software has created a splintered marketplace where developers encounter enormous challenges in code porting among disparate platforms. This interoperability deficit creates vendor lock-in and constrains the capacity to take full advantage of the special capabilities of each quantum computing model. Second, the growing sophistication of quantum algorithms requires combining classical and quantum computing resources, as CUDA-Q's hybrid strategy illustrates. But the lack of straightforward deployment of these hybrid algorithms on various quantum hardware, such as IBM's, limits their real-world application and hinders research momentum. Third, the enormous time and resource investment in creating CUDA-Q-based quantum applications means there is a strong demand for a solution that enables these applications to be reused on other platforms without major rewriting. Compounding these difficulties, the simulation of quantum systems on classical computers has an exponential memory requirement: for $'n'$ qubits, the RAM needed grows as $8 * 2^n$ bytes. This exponential increase in size makes it computationally costly and often infeasible to simulate larger quantum circuits, making access to real quantum hardware, like IBM's, even more valuable and further underscoring the value of a smooth translation tool to take advantage of these resources. Without such a translating tool, there is pressure to either rewrite one's code or be locked down to one provider of hardware, stifling innovation and the ability to advance towards quantum computing usage. In the end, this project is essential to democratizing access to quantum resources, facilitating a more open and productive development community, and speeding the arrival of practical applications of quantum computing, particularly as the memory requirements of simulating more sophisticated quantum circuits become intractable.

Number of Qubits	Expected Ram for Simulation
10	8 KB
20	8 MB
40	8 GB
60	8192 PB

4.3. Abstract Syntax Tree (AST) Overview

An **Abstract Syntax Tree (AST)** is a hierarchical, tree-like data structure used to represent the syntactic structure of source code in a form that is easier for computers to analyze and manipulate. Rather than dealing with raw text, an AST breaks down code into structured components such as expressions, statements, and function calls. Each node in the tree corresponds to a specific syntactic element, with branches representing the relationships between these elements. This abstraction allows compilers, interpreters, and transformation tools to understand the underlying logic of a program in a consistent and reliable manner.

The construction of an AST begins with parsing the source code using a compiler frontend or parser library. This step verifies the syntactic correctness of the code and constructs the tree from the root (which typically represents the entire program) down to its leaf nodes, such as variable names and literals. For example, in a CUDA-Q program, a node may represent the definition of a quantum kernel, while its children represent operations inside that kernel, such as qubit allocation and gate applications. The tree can then be traversed programmatically to analyze or transform specific portions of the code.

In the context of this project, the AST serves as an essential intermediate representation for converting CUDA-Q programs into their Qiskit equivalents. It provides the necessary structural insight to extract quantum-specific components like kernel decorators (e.g., `@cudaq.kernel`), qubit initializations (e.g., `cudaq.qubit()`), and gate operations (e.g., `h(qubit), y(qubit)`). The AST not only allows us to detect these operations but also to transform them with precision, maintaining the logic and order of quantum instructions during translation.

Moreover, the AST makes it possible to support code transformation at scale. Because the tree representation is abstracted away from syntax-specific quirks, it can accommodate a wide variety of code patterns and structures. This flexibility is crucial for ensuring that even complex CUDA-Q programs are handled effectively, with accurate reproduction of the original quantum logic in Qiskit.

Ultimately, the AST enables a systematic, programmatic transformation of CUDA-Q code into Qiskit by:

- Allowing detailed inspection of the program's structure through node-based traversal.
- Supporting robust identification and replacement of quantum operations with Qiskit equivalents.
- Ensuring semantic consistency between the source (CUDA-Q) and target (Qiskit) code.
- Enabling extensibility for future additions, such as multi-qubit gates or controlled operations.

This foundational layer ensures that subsequent steps in the methodology, such as gate mapping, circuit reconstruction, and execution on real hardware, are accurate, reliable, and scalable.

4.4. Methodology

This project employs a systematic approach to translate CUDA-Q programs into executable Qiskit code, ensuring accuracy and usability. The methodology comprises the following key steps:

- Parsing CUDA-Q Code: Initially, the CUDA-Q source file is processed by the library. This involves constructing an Abstract Syntax Tree (AST), which provides a structured representation of the program's code, enabling detailed analysis of its components.
- Extracting Quantum Operations: The generated AST is then traversed to identify and extract crucial quantum operations. This includes recognizing quantum gates, qubit allocations, and measurement instructions, effectively capturing the quantum logic of the CUDA-Q program.

- Mapping CUDA-Q Gates to Qiskit Gates: A predefined mapping dictionary serves as a crucial bridge, translating CUDA-Q gate names to their corresponding equivalents within the Qiskit framework. This ensures accurate representation of quantum operations in the target environment.

```
        GATE_MAPPING = {"h": "h",
                        "x": "x",
                        "rx": "rx",
                        "ry": "ry",
                        "rz": "rz",
                        "cx": "cx",
                        "mz": "measure",
                        "t": "t",
                        "z": "z",
                        "y": "y",
                        "s": "s",  }

```

- Generating Qiskit Circuit: With the extracted and mapped operations, a Qiskit QuantumCircuit object is instantiated. The identified quantum operations are then applied to this circuit using Qiskit's specific syntax, effectively reconstructing the CUDA-Q quantum algorithm within the Qiskit environment. This Qiskit QuantumCircuit is formatted and saved as a Python script. The produced script is set up for immediate execution in the Qiskit environment so that users can execute the translated quantum algorithms directly.

4.4.1 Pseudo Code :

Algorithm 1 Conversion of CUDA-Q Code to Qiskit

```
1: Input: Path to CUDA-Q source file
2: Output: Qiskit Python Code
3: BEGIN
4: Step 1: Take Input
5: Print "Enter the path of the CUDA-Q source file: "
6: Read $cudaq\_file\_path$
7: if file does not exist then
8:     Print "Error: File not found!"
9:     Exit
10: end if
11: Step 2: Read CUDA-Q Source Code
12: $cudaq\_source\_code \leftarrow$ Read File($cudaq\_file\_path$)
13: Step 3: Parse the Code using AST
14: $parsed\_ast \leftarrow$ Parse AST($cudaq\_source\_code$)
15: $operations \leftarrow$ Empty List
16: Step 4: Extract Operations from AST
17: for each node in $parsed\_ast$ do
18:     if node is a quantum gate then
19:         $gate\_name \leftarrow$ Map to Qiskit Gate(node)
20:         $params \leftarrow$ Extract Parameters(node)
21:         $qubit \leftarrow$ Extract Qubit(node)
22:         Append ($gate\_name, params, qubit$) to $operations$
23:     end if
24: end for
25: Step 5: Determine Qubits Used
26: $unique\_qubits \leftarrow$ Sorted(Set of qubits from operations)
27: $qubit\_indices \leftarrow$ Assign Index to Each $unique\_qubit$
28: Step 6: Generate Qiskit Circuit
29: $num\_qubits \leftarrow$ Length($unique\_qubits$)
30: $qc \leftarrow$ New QuantumCircuit($num\_qubits$, $num\_qubits$)
31: Step 7: Apply Operations to Qiskit Circuit
32: for each ($gate, params, qubit$) in $operations$ do
33:     $q\_idx \leftarrow qubit\_indices[qubit]$
34:     if $gate$ is "measure" then
35:         Apply Measure($qc$, $q\_idx$)
36:     else if params exist then
37:         Apply Gate with Parameters($qc$, $gate$, $params$, $q\_idx$)
38:     else
39:         Apply Gate($qc$, $gate$, $q\_idx$)
40:     end if
41: end for
42: Step 8: Generate Python Code
43: $qiskit\_code \leftarrow$ Construct Python Code($qc$)
44: Step 9: Save Output
45: Write $qiskit\_code$ to "qiskit_output.py"
46: Print "Qiskit Python Code saved to qiskit_output.py"
47: END
```

4.4.2 Flow chart :

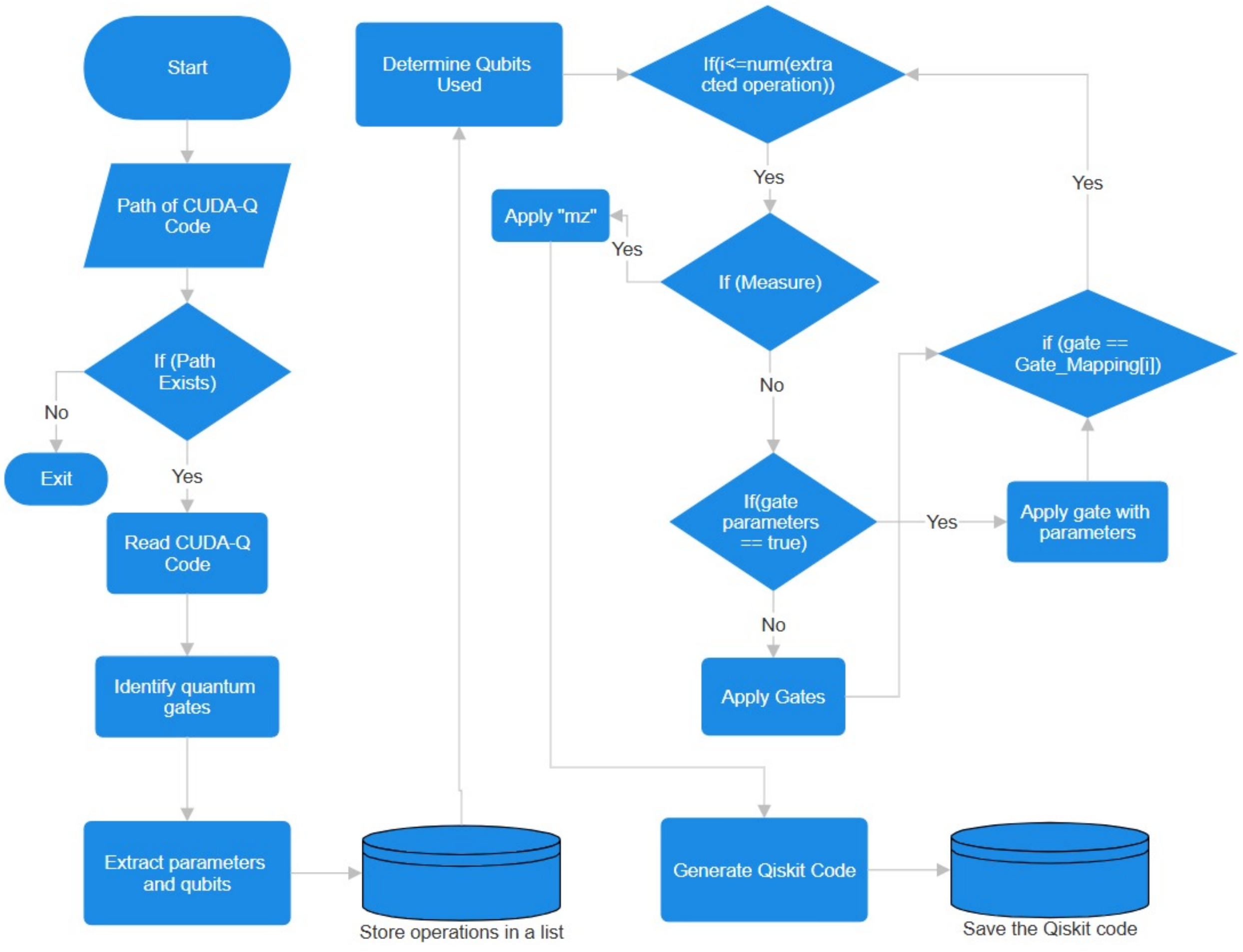

Figure 21: Flow chart

4.5. User Guide

The CUDA-Quantum to Qiskit Converter module is a Python-based library that facilitates the translation of CUDA-Q quantum programs into equivalent Qiskit implementations. This guide provides step-by-step instructions on how to install and use the module effectively.

Clone the repo

```
$ git clone https://github.com/NishankaDas/cudaq-qiskit-converter.git

```

Navigate to the directory

```
$ cd cudaq-qiskit-converter

```

Install the module using pip

```
$ pip install .

```

4.5.1 Code for conversion

This script is designed to convert quantum programs written in CUDA-Q into Qiskit code. CUDA-Q is an NVIDIA-supported quantum programming framework, while Qiskit is an open-source quantum computing framework developed by IBM. The conversion process allows CUDA-Q users to run their quantum circuits on Qiskit-compatible quantum simulators and hardware. **Importing Required Modules**

```
from cudaq_to_qiskit import convert_cudaq_source_to_qiskit
import os
```

- The $convert_{c}udaq_{s}ource_{t}o_{q}iskit$ function is imported from the $cudaq_{t}o_{q}iskit$ library, which is responsible for translating CUDA-Q code into Qiskit code.
- The os module is imported to handle file operations.

Getting the CUDA-Q File Path from User

```
cudaq_file_path = input("Enter the path of the CUDA-Q source file: ")
```

- The user is prompted to enter the file path of the CUDA-Q source code.

Reading the CUDA-Q Source Code

```
if not os.path.exists(cudaq_file_path):
    print("Error: File not found!")
    exit()
```

- The script first checks whether the specified file exists. If not, it prints an error message and terminates execution.

```
with open(cudaq_file_path, "r") as file:
    cudaq_source_code = file.read()
```

- If the file exists, it is opened in read mode, and its contents are stored in the $cudaq_source_code$ variable.

Converting CUDA-Q Code to Qiskit

```
circuit, qiskit_code = convert_cudaq_source_to_qiskit(
    cudaq_source_code)
```

- The function $convert_cudaq_source_to_qiskit()$ is called with the CUDA-Q source code as input.
- It returns: circuit: A Qiskit representation of the quantum circuit. $qiskit_code$: The equivalent Qiskit Python code as a string.

Displaying the Converted Code

```
print("Qiskit Circuit:")
print(circuit)

print("\nQiskit Code:")
print(qiskit_code)
```

- The converted Qiskit circuit and Python code are printed to the console.

Saving the Converted Code to a File

```
output_file = input("Enter File Name: ")
with open(output_file, "w") as f:
    f.write(qiskit_code)

print(f"\nQiskit Python Code saved to: {output_file}")
```

- The user is prompted to enter the name of the output file.
- The Qiskit code is written to this file.
- A confirmation message is displayed, indicating where the converted code has been saved.

4.5.2 Results after Conversion

Code in CUDA Quantum file

```python
import cudaq
@cudaq.kernel
def kernel():
    qubit = cudaq.qubit()
    h(qubit)
    y(qubit)
    z(qubit)
    t(qubit)
    s(qubit)
    mz(qubit)
result = cudaq.sample(kernel)
print(result)
```

Output Code File in Qiskit

```python
from qiskit import QuantumCircuit
from qiskit.primitives import Sampler
sampler = Sampler()
qc = QuantumCircuit(1, 1)
qc.h(0)
qc.y(0)
qc.z(0)
qc.t(0)
qc.s(0)
qc.measure(0, 0)
res = sampler.run(qc)
print(res.result())
```

4.6. Conclusion

The CUDA-Q to Qiskit Converter project offers a seamless way to translate quantum programs expressed in CUDA-Q to Qiskit, which facilitates interoperability among various quantum computing platforms. Automating the conversion eliminates human effort, the possibility of error, and provides easier accessibility to researchers and developers working on quantum circuits.

With the application of Abstract Syntax Tree (AST) parsing, gate mapping, and Qiskit circuit generation, the module provides a precise and optimal conversion of quantum operations. The project is also offered with a well-organized and user-friendly interface that can easily be applied to existing quantum computing environments.

This converter promises great possibilities in cross-platform quantum computing since, while one is allowed to run quantum algorithms for developing and debugging using CUDA-Q, one also uses Qiskit's immense simulation and hardware execution potential. This bridging between two different frameworks ensures more quantum algorithm development flexibility.

Future developments can consist of the implementation of multi-qubit quantum gates, optimization methods, and support for other quantum computing platforms. Overall, this project is a critical instrument in furthering the study and implementation of quantum computing, advancing towards a more standardized and efficient quantum development platform.