QASM-to-HLS: A Framework for Accelerating Quantum Circuit Emulation on High-Performance Reconfigurable Computers

Anshul Maurya
Florida Institute of Technology, amaurya2022@my.fit.edu

Follow this and additional works at: https://repository.fit.edu/etd

Part of the Computer and Systems Architecture Commons, Computer Sciences Commons, Hardware Systems Commons, Other Computer Engineering Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

Recommended Citation
https://repository.fit.edu/etd/1385

This Thesis is brought to you for free and open access by Scholarship Repository @ Florida Tech. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholarship Repository @ Florida Tech. For more information, please contact kheifner@fit.edu.
QASM-to-HLS: A Framework for Accelerating Quantum Circuit Emulation on High-Performance Reconfigurable Computers

by

Anshul Maurya

Bachelor of Technology
Electronics and Communication Engineering
Babasaheb Bhimrao Ambedkar University
2018

A thesis
submitted to the College of Engineering and Science
at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Engineering

Melbourne, Florida
December, 2023
We the undersigned committee
hereby approve the attached thesis

QASM-to-HLS: A Framework for Accelerating Quantum Circuit Emulation on High-Performance Reconfigurable Computers by Anshul Maurya

Naveed Mahmud, Ph.D.
Assistant Professor
Electrical Engineering and Computer Science
Major Advisor

Siddhartha Bhattacharyya, Ph.D.
Associate Professor
Electrical Engineering and Computer Science

Carlos E. Otero, Ph.D.
Professor
Electrical Engineering and Computer Science

Brian Lail, Ph.D.
Professor and Department Head
Electrical Engineering and Computer Science
Abstract

Title:
QASM-to-HLS: A Framework for Accelerating Quantum Circuit Emulation on High-Performance Reconfigurable Computers

Author:
Anshul Maurya

Major Advisor:
Naveed Mahmud, Ph.D.

High-performance reconfigurable computers (HPRCs) make use of Field-Programmable Gate Arrays (FPGAs) for efficient emulation of quantum algorithms. Generally, algorithm specific architectures are implemented on the FPGAs and there is very little flexibility. Moreover, mapping a quantum algorithm onto its equivalent FPGA emulation architecture is challenging. In this work, we present an automation framework for converting quantum circuits to their equivalent FPGA emulation architectures. The framework processes quantum circuits represented in Quantum Assembly Language (QASM) and derives high-level descriptions of the hardware emulation architectures for High-Level Synthesis (HLS) on HPRCs. The framework generates the code for a heterogeneous architecture consisting of a microprocessor (host) and FPGA (kernel). Space-time tradeoffs were investigated for the different architectures. We also explored methods for concurrent kernel execution to improve the circuit execution time.
# Table of Contents

Abstract ........................................................................................................ iii

List of Figures ............................................................................................... vii

List of Tables .................................................................................................. ix

Acknowledgments .......................................................................................... x

Dedication ......................................................................................................... xi

1 Introduction ................................................................................................. 1
  1.1 Prospect of Quantum Computing .......................................................... 1
  1.2 Challenges and Motivation ................................................................... 2
  1.3 Problem Statement ............................................................................... 3
  1.4 Research Goals and Approaches .......................................................... 3

2 Background and Related Work .................................................................. 6
  2.1 Quantum Computing Fundamentals ...................................................... 6
    2.1.1 Qubits, Superposition, and Entanglement ................................. 6
    2.1.2 Quantum Gates and Bloch Sphere Representation ................. 7
      2.1.2.1 Hadamard Gate: ............................................................... 8
      2.1.2.2 X-Gate: ............................................................................ 8
      2.1.2.3 Rotation Operators: ....................................................... 9
2.1.2.4 CNOT gate: ......................................................... 9
2.1.2.5 U-Gate: ............................................................ 10
2.2 Grover’s Search Algorithms ............................................. 10
2.3 CPU/GPU Based quantum computers ................................... 11
2.4 FPGA-based Hardware Emulation ....................................... 13
2.5 High Level Synthesis ..................................................... 14
2.6 OpenQASM ................................................................. 15
   2.6.1 Compilation .......................................................... 16
   2.6.2 Circuit generation ................................................... 16
   2.6.3 Execution ............................................................. 16
   2.6.4 Post-processing ...................................................... 16

3 OpenQASM to High-Level Synthesis ....................................... 18
   3.1 Proposed Method ...................................................... 18
   3.2 QASM-to-HLS Software Framework .................................. 19
      3.2.1 Class QASMProcessing() ....................................... 19
         3.2.1.1 class QASMProcessing.stringProcessing() ............... 20
         3.2.1.2 class QASMProcessing.qasmToList() .................. 20
      3.2.2 Class CircuitListToMatrix() .................................. 21
         3.2.2.1 Class CircuitListToMatrix.genMat ..................... 21
         3.2.2.2 Class CircuitListToMatrix.cnotLayerMat(CNot_gate_list) 22
         3.2.2.3 Class CircuitListToMatrix.toGateMatrix(Gate_Name) . 24
         3.2.2.4 Class CircuitListToMatrix.is_unitary(numpy.matrix) . 24
   3.3 Derived HLS Architectures .......................................... 25
      3.3.1 FPGA Application - Host Part ................................ 25
      3.3.2 FPGA Application - Programmable Logic Part ............... 27
List of Figures

2.1 Bloch Sphere ................................................. 8
2.2 Grover’s Search .............................................. 11
3.1 Workflow of proposed framework ................................. 19
3.2 Circuit Layering .............................................. 20
3.3 Circuit Layering In QASM Code ................................. 21
3.4 Identity Matrix To Form C-NOT Matrix ......................... 23
3.5 C-NOT Gate With Two Control and a Target Qubit ............ 23
3.6 C-NOT Gate With Two Control and a Target Qubit ............ 24
3.7 Complete Software Stack ...................................... 26
3.8 OpenCL buffer creation Example Code ......................... 27
3.9 OpenCL buffer userspace mapping Example Code .............. 27
3.10 Pseudocode for Host Application .............................. 28
3.11 Type-2 Design ............................................... 29
3.12 Matrix-Vector Kernel Function Code Snippet ................. 30
4.1 General Concurrent Kernel Flow .............................. 31
4.2 Task Divided Between Concurrent Kernels ...................... 33
4.3 Modified Matrix-Vector multiplication Kernel Code Snippet .... 33
4.4 Type-3 Design ............................................... 35
4.5 Concurrent computation of matrices ........................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6 Matrix-Matrix Function Code Snippet</td>
<td>36</td>
</tr>
<tr>
<td>4.7 Out-of-Order &amp; Event Queue Creation Code Snippet</td>
<td>36</td>
</tr>
<tr>
<td>4.8 Enqueuing Tasks in OOO Queue</td>
<td>37</td>
</tr>
<tr>
<td>4.9 Complete OOO Flow for Single Iteration</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Platform Diagram of Alveo U-200</td>
<td>40</td>
</tr>
<tr>
<td>5.2 Grover’s Multi-Pattern Search Circuit Used in Experiment</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Timeline Trace for Type-1 Design</td>
<td>43</td>
</tr>
<tr>
<td>5.4 Zoom In of Figure 5.3 for Single Kernel Run</td>
<td>43</td>
</tr>
<tr>
<td>5.5 Timeline Trace for Type-2 Design</td>
<td>44</td>
</tr>
<tr>
<td>5.6 Timeline Trace For Type-1 Concurrent Kernel Design</td>
<td>45</td>
</tr>
<tr>
<td>5.7 Implemented Design Diagram Generated by Xilinx VITIS IDE</td>
<td>47</td>
</tr>
<tr>
<td>5.8 Timeline Trace for Type-3 Design, K=8</td>
<td>47</td>
</tr>
<tr>
<td>5.9 Simulation Time Comparison Graph</td>
<td>48</td>
</tr>
<tr>
<td>5.10 Speedup Comparison Graph</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Tables

3.1 C-NOT Gate Transition Table ............................................. 24

5.1 Hardware Result of Type-1 Design ................................. 42
5.2 Hardware Result of Type-2 Design, $K = 5$ .................... 44
5.3 Hardware Result of Type-2 Design, $K = 4$ .................... 46
5.4 Hardware Result of Type-3 Design, $K = 8$ .................... 46
Acknowledgements

I extend my sincere appreciation to the chairs of my thesis defense, Dr. Naveed Mahmud, Dr. Siddhartha Bhattacharyya, and Dr. Carlos E. Otero for their insightful and thoughtful evaluation.

I am grateful to the Florida Institute of Technology for providing the conducive setup and scholarly environment that facilitated the completion of this work. The support and resources offered by the institution have been instrumental in my academic journey.

A special note of thanks goes to the dedicated librarians who assisted me in formatting and proofreading. Their attention to detail and commitment to excellence ensured the clarity and professionalism of this thesis.

This accomplishment would not have been possible without the collective support of these individuals and the institutional backing from the Florida Institute of Technology.
Dedication

I dedicate this thesis to Dr. Naveed Mahmud, my advisor, whose guidance and instruction were pivotal in completing this work. His scholarly insights and unwavering support greatly shaped my academic development. I am grateful for his invaluable contributions to this research. His commitment to excellence and passion for a thorough understanding of the subject matter served as an inspiration throughout this endeavor.
Chapter 1

Introduction

1.1 Prospect of Quantum Computing

Quantum computing stands as a paradigm that promises to redefine computational capabilities, presenting unparalleled opportunities in certain problem domains. Leveraging the principles of quantum mechanics, quantum computers have demonstrated their potential to outpace classical counterparts in various applications, such as factorization, optimization, quantum system simulation or simulation of nature, machine learning enhancement, and cryptographic innovation through quantum key distribution. These feats arise because quantum computers are fundamentally very different than classical computers in processing the data and have intrinsic parallelism. The fundamental computation unit in quantum computers is called a qubit, Unlike classical bits that can represent either 0 or 1, qubits can exist in a superposition of both 0 and 1 states simultaneously with associated probabilities.

Quantum computing represents a groundbreaking approach to computation, leveraging the unique properties of atoms. This paradigm has the potential to fundamentally transform the way we process information, enabling us to simulate natural phenom-
ena with unprecedented accuracy, conduct advanced chemical simulations to expedite
drug discovery, and deepen our comprehension of the natural world. Beyond these
remarkable prospects, the intrinsic parallelism of quantum computers stands to rev-
olutionize fields such as optimization, aerospace, and logistics. Quantum computers
have the capacity to simultaneously explore numerous possibilities, enabling the rapid
identification of efficient routes and solutions without the need for iterative processes.

1.2 Challenges and Motivation

The simulation of quantum circuits on classical platforms has been made necessary due
to the noisy nature and intermediate scale of current quantum devices. Many quantum
algorithms have the potential advantages to revolutionize the computing field. For
instance, the field of quantum machine learning (QML) has the potential to improve
machine learning algorithms in terms of improvement of model accuracy and faster
training time for large datasets is one such example. However, due to unavailability of
the reliable quantum hardware researchers exploring this area are heavily dependent
on the simulations of the quantum circuits.

The use of Field-Programmable Gate Arrays (FPGAs) to accelerate the simulation
of quantum circuits (also known as FPGA-based emulation) has been investigated for
some time. Existing FPGA-based emulation methods [9][12] exhibit limited flexibil-
ity in emulating a variety of quantum algorithms due to their fixed algorithm-specific
architectures. Additionally, the computational load of generating transformation ma-
trices of quantum data generally falls on the CPU, thereby restricting the extent to
which FPGAs can be leveraged for accelerating quantum algorithms. Mapping a quan-
tum algorithm to its corresponding FPGA architecture for emulation is challenging,
particularly for algorithm developers with limited FPGA design experience. Mapping
quantum algorithms to FPGA-based emulators can accelerate the research field of quantum algorithm development.

1.3 Problem Statement

Mapping different quantum algorithms to the FPGA hardware for emulation is a challenging and time-consuming task, and generalization of the emulators for a variety of quantum algorithms with a common, reusable architecture is not yet explored. Mapping of the quantum algorithm on hardware needs an understanding of the associated resource constraints because the quantum data grows exponentially with increasing number of qubits. The main challenge associated with the emulators is their simulation time, Achieving faster and more space-efficient simulations is a key objective, as it can have a significant impact on the practicality and utility of FPGA-based quantum emulation for a variety of applications.

1.4 Research Goals and Approaches

This work aims to develop an automation framework for converting quantum circuits represented in Quantum Assembly Language (QASM) into space and time efficient architectures for the emulation on FPGA backends. The utilization of QASM code makes the emulator generalized to a variety of quantum algorithms and is user-friendly for quantum algorithm development. Through a literature review of emulators (hardware/software) and an understanding of the high classical resource demands of quantum algorithms due to their inherent parallelism, our research goals can be summarized in three key points:

1. Resource Constraints: As the number of qubits in a quantum circuit increases,
classical data grows exponentially at a rate of $2^n$, where $n$ represents the number of qubits in the system. This exponential growth poses a significant challenge in terms of on-chip resources, such as FPGA’s LUTs/BRAMs, making quantum algorithms computationally expensive.

In our design, we addressed this challenge by avoiding the use of on-chip resources (LUTs/BRAMs) to store data or intermediate results. Instead, our architecture reads and writes directly from the DDR memory located closer to the chip. Additionally, we adopted a strategy of dividing tasks into multiple small kernels, aiming to make efficient use of resources for computation. These approaches were instrumental in optimizing FPGA resources primarily for the emulation of quantum circuits.

2. Efficient Mapping: Our software framework adopts a strategy of breaking down the quantum circuit into multiple layers, focusing on computing layer matrices instead of the entire circuit matrix for emulation. This division of work effectively offloads CPU processing to the implemented FPGA architecture, allowing for the mapping of a significant portion of the quantum algorithm onto the FPGA.

To ensure precise emulation results, we employ the double-precision complex data type, with each real and imaginary part occupying 8 Bytes, resulting in a total of 16 Bytes per data element.

3. Emulation Time: Achieving faster emulation time is a primary motivation for leveraging FPGA as an accelerator, and the discussed approaches contribute to enhanced simulation speed. In our hardware design approach, we strategically incorporated multiple AXI-Ports for diverse inputs, enabling parallel execution of multiple memory operations. The pipelining of our kernels ensures that the initiation interval aligns with a single clock cycle.
Given the absence of data dependencies in the emulation computation, we optimized performance by unrolling the loops to a certain extent. Moreover, to further boost overall simulation time efficiency, we partitioned the emulation computation task into multiple concurrent kernels. This segmentation allowed us to accomplish more work in the same time frame, resulting in a notable improvement in simulation time.
Chapter 2

Background and Related Work

2.1 Quantum Computing Fundamentals

2.1.1 Qubits, Superposition, and Entanglement

Qubits are the quantum analogue of classical bits, which are the basic units of information in classical computing. While classical bits can represent either a 0 or a 1, qubits can exist in a superposition of both 0 and 1 states simultaneously. Qubits are typically realized using physical systems with two distinct quantum states, such as the spin of an electron or the polarization of a photon. Unlike classical bits, which are always in one of two definite states, qubits can be in a linear combination of these states, described mathematically as $|0\rangle$ and $|1\rangle$. This is what gives rise to the concept of superposition.

Superposition is a fundamental principle of quantum mechanics that allows qubits to exist in a combination of multiple states at the same time. In other words, a qubit can be in a superposition of $|0\rangle$ and $|1\rangle$ states, represented as $\alpha |0\rangle + \beta |1\rangle$, where $\alpha$ and $\beta$ are complex numbers called probability amplitudes. The probabilities of measuring the qubit in the $|0\rangle$ or $|1\rangle$ state are determined by the square magnitudes of the probability
amplitudes: $|\alpha|^2$ and $|\beta|^2$. These probabilities add up to 1, meaning that the qubit will collapse to one of the two states upon measurement. Superposition enables quantum computers to perform certain calculations much faster than classical computers, as they can explore multiple possibilities simultaneously.

Entanglement is a unique quantum phenomenon in which two or more qubits become correlated in such a way that their individual states cannot be described independently. Measuring one entangled qubit instantly provides information about the others, regardless of the physical distance separating them. This phenomenon was famously described by Albert Einstein as "spooky action at a distance." Entanglement occurs due to the strong correlations that can exist between qubits, which go beyond classical correlations. Entanglement is a powerful resource in quantum computing and quantum communication.

2.1.2 Quantum Gates and Bloch Sphere Representation

Quantum gates are fundamental building blocks of quantum circuits and are analogous to classical logic gates in classical digital circuits. They are the basic unitary operators that manipulate and transform qubits in a quantum computer or quantum information processing system. To visualize the change in qubits, the Bloch sphere is a useful tool. Quantum gate operation can be visualized as rotations or reflections on the Bloch sphere, see figure 2.1.

Quantum gates are used to perform various quantum operations, including creating superpositions, entangling qubits, and executing quantum algorithms. A gate that acts on $n$ qubits will be represented by a $2^n \times 2^n$ unitary matrix, some notable quantum gates are described below.
2.1.2.1 Hadamard Gate:

This is a single-qubit gate that maps the basis state $|0\rangle$ to $\frac{|0\rangle + |1\rangle}{\sqrt{2}}$ and $|1\rangle$ to $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ thus creating an equal superposition of the two basis states. The unitary description of this gate is shown in (2.1).

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$ (2.1)

2.1.2.2 X-Gate:

Also known as the Pauli-X gate, this gate is analogous to the classical not gate. X-gate rotates the state to $\pi$ radian around X-axis i.e. it flips the $|0\rangle$ state to $|1\rangle$ and vice-versa.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$ (2.2)
2.1.2.3 Rotation Operators:

These unitary operators rotate the state vector of a qubit around a given axis by a given angle. The most common quantum rotation operators are the Pauli X (2.2), Y (2.3), and Z (2.4) gates which rotates the state vector by $\pi$ radian with respect to $x$, $y$, and $z$ axis.

\[
Y = \begin{bmatrix} 0 & -1i \\ 1i & 0 \end{bmatrix} \quad (2.3)
\]

\[
Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (2.4)
\]

2.1.2.4 CNOT gate:

The CNOT gate, also known as the controlled-NOT gate, is a universal two-qubit quantum gate that flips the state of the second qubit, target qubit if and only if the first, control qubit is in the state $|1\rangle$. A CNOT gate is used to entangle two qubits. Any quantum computation can be performed using only CNOT gates and single-qubit gates. CNOT gate matrix can be derived from a unitary matrix by flipping the target states based on the control states. The eq. (2.5) refers the two qubit CNOT gate matrix, when the $0^{th}$ qubit and the $1st$ qubit are the control and target qubits respectively.

\[
CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad (2.5)
\]
2.1.2.5 U-Gate:

The U gate is a single-qubit universal quantum gate that can be used to perform any single-qubit unitary operation. It is a powerful tool for quantum computing, and it is used to implement a wide range of quantum gates and operations, including the Hadamard gate, the Pauli gates, and rotations around arbitrary axes.

The U gate is represented by the following matrix (2.6):

\[
U(\theta, \phi, \lambda) = \begin{pmatrix}
\cos \frac{\theta}{2} & -e^{i\lambda} \sin \frac{\theta}{2} \\
e^{i\phi} \sin \frac{\theta}{2} & e^{i(\lambda+\phi)} \cos \frac{\theta}{2}
\end{pmatrix}
\]  

(2.6)

By utilizing the above quantum gates quantum algorithms get implemented in quantum computers.

2.2 Grover’s Search Algorithms

Grover’s algorithm [8] works by using a sequence of quantum operations to perform a search on an unsorted list with the algorithm complexity of \(O(\sqrt{N})\). The basic idea behind the algorithm is to use quantum mechanics to amplify the probability of finding the desired item in the list. The algorithm first initializes the qubits to a state of equal superposition by applying Hadamard gates across all qubits.

The algorithm works in two phases: the oracle phase and the diffusion phase, refer 2.2. The oracle phase is responsible for inverting the phase of the target state. Oracle can be designed to search for multiple searches, called multipattern search, or can be used to search for the solution of a problem. This reduces the mean of all the state amplitudes. Consequently, the diffusion phase performs an inversion about the mean, which amplifies the target state and reduces the amplitudes of the other states. After a number of iterations of these two phases, the target state’s amplitude will be amplified.
and it will have a much higher probability of being measured than any other state.

2.3 CPU/GPU Based quantum computers

To simulate quantum computers, software-based solutions are widely favored for conducting small-scale experiments. These emulators primarily function by modeling qubits as software entities and subsequently executing quantum operations through mathematical computations. Given that quantum computing heavily relies on extensive matrix and vector mathematics, software libraries incorporate GPU support to enhance computational speed. GPUs, with their SIMD (Single Instruction, Multiple Data) architecture and numerous parallel cores, significantly boost the simulation speed in comparison to CPUs.

- Cirq: Cirq is an open-source quantum programming framework developed by Google. It’s designed for writing quantum algorithms and running them on quantum computers or simulators. Cirq provides a high-level interface for working with quantum circuits and offers a variety of tools and libraries to facilitate quantum programming and research.

- Qulacs: Qulacs is a quantum computing simulator developed by the University of Tokyo.
• **Qiskit:** Qiskit is the most popular quantum computing software development framework developed by IBM. It is designed to facilitate quantum programming, research, and application development. Qiskit provides a comprehensive suite of tools and libraries for working with quantum computers, quantum algorithms, and quantum circuits.

Key components of Qiskit include:

1. **Terra:** This component is focused on the low-level quantum circuit description and optimization. It provides tools for creating, manipulating, and optimizing quantum circuits, as well as interfacing with quantum hardware.

2. **Aer:** Qiskit Aer is a high-performance simulator that allows researchers to simulate quantum circuits on classical computers. It is useful for testing quantum algorithms and circuits before running them on actual quantum hardware.

3. **Ignis:** Ignis focuses on quantum error correction and mitigation. It provides tools for characterizing and mitigating noise in quantum devices, which is crucial for improving the reliability of quantum computations.

4. **Aqua:** Qiskit Aqua is a library for developing quantum applications in fields such as chemistry, optimization, and machine learning. It offers a range of pre-built algorithms for various quantum computing applications.

5. **IBM Quantum Experience:** This is an online platform that allows users to access and run quantum experiments on IBM’s cloud-based quantum computers. It’s integrated with Qiskit to make it easier for users to experiment with real quantum hardware.

**CPU/GPU quantum computer emulators** are an essential tool for the development
of quantum computing. They allow researchers and developers to explore the potential of quantum computing without the need for access to physical quantum computers.

### 2.4 FPGA-based Hardware Emulation

FPGAs (Field-Programmable Gate Arrays) have attracted attention in quantum computing simulations due to their parallel processing capabilities and adaptability. They are well-suited for simulating quantum algorithms and circuits, involving complex mathematical operations. The inherent advantages of FPGAs, such as parallel processing, offer a notable speedup compared to traditional software-based simulations.

In the exploration of FPGA-based quantum simulation, Noda et al. [4] focused on mapping quantum algorithms onto FPGA resources using register-based techniques. Their approach involved manipulating quantum data stored in registers based on the utilized gates. However, designing the hardware requires manipulation in HDL code for different quantum algorithms.

In some studies, researchers have employed emulation algorithms tailored to specific quantum algorithms, resulting in faster implementations but with limitations confined to a single algorithm. For instance, in the work by Khalil et al. [11], the authors implemented the quantum Fourier transform on an FPGA. In this particular study, the authors attempted to obtain the layer matrix of the circuit and saved in 16-bit registers.

In a recent study, In the work by Fujishima [7], the author introduces a logic quantum processor and a quantum index processor implemented on an FPGA. The state vector probabilities were encoded in binary form to minimize memory space usage, and in the index processor, only the non-zero index states were stored. The primary focus of this work was on space-saving techniques to enhance scalability, limited to
search algorithms. Khalid et al. [10] introduced a hardware abstraction layer-based FPGA emulator, aiming to simplify the intricacies of FPGA hardware. This approach allows users to input quantum data into a register file. Nevertheless, these methodologies lack the conventional approach to quantum circuit creation, and the need for hardware synthesis persists. A complete emulation of quantum emulation on reconfigurable computers have been explored by Mahmud et al. [5], showing the emulation of encoding classical data to quantum system, execution of algorithm and decoding the measurement back to classical domain.

2.5 High Level Synthesis

High-level synthesis (HLS) is a design process used in digital integrated circuit design and electronic system-level (ESL) design. It is a process that allows designers to create complex digital hardware systems by specifying their functionality at a high level of abstraction, typically using high-level programming languages or design languages like C, C++, or SystemC, rather than specifying them in a low-level hardware description language like VHDL or Verilog.

In HLS, designers work at a higher level of abstraction, focusing on the system’s algorithm and functionality. This makes the design process faster and more accessible to software developers and other engineers who may not be well-versed in hardware description languages. Designers start by specifying the desired behavior of the hardware system using a high-level programming language, and HLS tools then automatically transform this high-level description into RTL (Register-Transfer Level) code, which represents the hardware components, their interconnections, and their functionality. HLS tools can improve the hardware design’s performance, power efficiency, and area utilization by applying various techniques such as pipelining, resource sharing, and
scheduling. Designers can make trade-offs between these factors during the HLS process to meet specific design goals.

The generated RTL code from HLS can be synthesized for specific target platforms, such as FPGAs (Field-Programmable Gate Arrays) or ASICs (Application-Specific Integrated Circuits), based on the intended application and deployment requirements.

2.6 OpenQASM

Open Quantum Assembly Language (OpenQASM) serves as an imperative programming language designed for framing quantum circuits and algorithms destined for quantum computer execution. OpenQASM links with gate-based quantum computing paradigms, akin to how classical CPU systems communicate with assembly instructions. Its core purpose lies in functioning as an intermediary representation, facilitating interaction between higher-level quantum compilers and the quantum hardware. OpenQASM helps developers to communicate with quantum and classical hardware in a feed-forward flow based on the measurement outcomes. A particular feature of OpenQASM is its adaptability, enabling various representations of the same program during the transformation from a high-level conceptual description to pulse-level representations. OpenQASM program execution takes into account qubit’s coherent time and executes the quantum program in pieces by using distinct quantum circuits whose results can’t be passed to another circuit. In the middle of these distinct circuits execution classical computation takes place. A quantum program executes in phases with a classical system in the loop to complete the execution and produce results. The different phases of execution of quantum program are as following:
2.6.1 Compilation

In this step, which happens on a classical computer and doesn’t involve the quantum computer, the input will be the source code and the compile time parameters like the supported basis gates for the target quantum processor, and the output will be the high-level description of the quantum-classical program.

2.6.2 Circuit generation

Quantum devices come in different circuit topologies mainly showing the entanglement scheme between two qubits. This phase takes account of such topology and produces the collection of quantum circuits, or quantum basic blocks, together with associated classical control instructions and classical object code needed at run-time.

2.6.3 Execution

Quantum circuits and associated classical run time control statements will be processed into the stream of real-time instructions on a high-level controller which converts these instructions to the corresponding physical operations applied onto the qubits. A corresponding result stream takes the measurement data back to the high-level controller when requested. The output of this phase is a collection of data from measurements to a high-level controller.

2.6.4 Post-processing

At last, we process all the measurements on a classical system to get the final output of the quantum program or an intermediate result for the utilization of further quantum circuits.
OpenQASM offers a valuable abstraction layer that enhances portability and provides a user-friendly approach for expressing quantum circuits. In this work, we used OpenQASM however ongoing research is exploring other high-level languages that offer a closer connection to quantum hardware when describing quantum circuits. Some of these languages include Quipper, Scaffold, and Quil.
Chapter 3

OpenQASM to High-Level Synthesis

3.1 Proposed Method

In this work, we present a framework that automatically derives hardware emulation architectures described in high-level languages for High-Level Synthesis (HLS). HLS is a fast and convenient entry point into the hardware design process. This work implements an interface between the QASM representation of quantum circuits and the HLS design process for facilitating the emulation of quantum circuits on FPGA hardware. By processing the QASM description of the quantum circuit, our automation framework generates an FPGA application. The host component of the application handles the quantum data and communicates with the programmable logic (PL) within the OpenCL [2] framework for quantum data processing. The proposed methodology is shown in Figure 3.1.
3.2 QASM-to-HLS Software Framework

In order to derive the FPGA application code from QASM, we have developed a software package, hereby named QASM-to-HLS. This package takes QASM code as input and generates a high-level language-based application code for FPGA.

The processing of the QASM code to extract the quantum data will be explained as we continue our discussion with the QASM-to-HLS software framework. The software is designed in two classes \texttt{QASMProcessing} and \texttt{CircuitListToMatrix}, as evident from the name first class will process the QASM code to get the quantum circuit’s implementation level detail and will produce a list of gates based on their applied order on the quantum circuit, and \texttt{CircuitListToMatrix} class takes the produced circuit list as an input and generates the matrices for the computation.

3.2.1 Class QASMProcessing()

\texttt{[Class QASMProcessing (qiskit.circuit.QuantumCircuit, Transpiler = False)]}

This class supports the \texttt{Qiskit’s circuit object} or a QASM code file as input to produce the list of applied gates. This class divides the circuit into layers, where a single group of gates that can be executed together, are called layers, see Figure 3.2, and then based on the order of applied gates produces a list describing the circuit. Sometimes the QASM code can get complex and the programmer might add custom gates
or upgrades in the language that can introduce new gates which is not known by our framework, to tackle such runtime behavior we can turn on the transpiler which will deduce the circuit to basis gates, described in the background section.

### 3.2.1.1 class QASMProcessing.stringProcessing()

QASM code syntax to describe the entangled and non-entangled gates are as below:

**Non-Entangled Gate:**

```
Gate_Name(Param_1, Param_2, Param_3) q[Qubit_Number]
```

**Entangled Gate:**

```
Gate_Name(Param_1, Param_2, Param_3) q[Ctrl_Qubit_Number][Target_Qubit_Number]
```

In this function, based on the above syntax we process the QASM code to gather the basic details of the circuit like the number of qubits, gate name with qubit position, and the gate parameters if any. The assembled information will be appended to a list and returned.

### 3.2.1.2 class QASMProcessing.qasmToList()

This function produces the final list of gates after the formation of layers, internally this function calls the above function, `class QASMProcessing.stringProcessing()`. In this function, we make sure that gate ordering is correct and if there is any missing
gate in between that will be covered by an identity gate see figure 3.2 and 3.3.

3.2.2 Class CircuitListToMatrix()

[Class CircuitListToMatrix (Circuit_List, Number_of_qubits, check = True]

This class defines a getter function that calls two other class-internal functions to produce HLS code. Results from the Class QASMProcessing() will be utilized in this class.

3.2.2.1 Class CircuitListToMatrix.genMat

This function returns the layer matrices and final circuit matrix, and if the check argument is True, it produces the circuit debugging information such as whether each layer produces the unitary matrix and the statevector after the execution of each layer. Finally, we write the computed layer matrices into a file with the combination of the HLS code.

To compute the layer matrix we make the sub-list form the Circuit_List where the
length of the list will be \( n \), the number of the qubits. Based on each gate in the sublist we extract the respective gate matrix and then perform the kronecker product to form the final layer matrix. The below example shows the complete computation of a layer-based approach to get the final circuit result.

**Example 1:**

Referring to Figure 3.3, let’s focus on the initial two layers, denoted as L1 and L2. These layers are of length 4, given that \( n = 4 \). In this context, the leftmost gate within each layer corresponds to qubit0, while the rightmost gate pertains to qubit3.

For L1: \([h, h, h, h]\) the layer matrix:

\[
M_{L1_{16 \times 16}} = M_{h_{2 \times 2}} \otimes M_{h_{2 \times 2}} \otimes M_{h_{2 \times 2}} \otimes M_{h_{2 \times 2}}
\]

\[
S_{out1_{16 \times 1}} = M_{L1_{16 \times 16}} \times S_{in_{16 \times 1}}
\]

For L2: \([x, h, I, h]\) the layer matrix:

\[
M_{L2_{16 \times 16}} = M_{X_{2 \times 2}} \otimes M_{h_{2 \times 2}} \otimes M_{I_{2 \times 2}} \otimes M_{h_{2 \times 2}}
\]

\[
S_{out2_{16 \times 1}} = M_{L2_{16 \times 16}} \times S_{out1_{16 \times 1}}
\]

where \( M_h, M_X, M_I \) is Hadamard gate (2.1), Not gate (2.2) and Identity gate matrix. Following the computation of the layer matrices, we proceed to multiply them with the statevector from the previous layer. For instance, in the above example, \( M_{L1} \) is multiplied with the initial statevector, denoted as \( S_{in} \) since it’s the first layer in the circuit. This operation results in \( S_{out1} \). Subsequently, \( M_{L2} \) is multiplied with \( S_{out1} \), generating \( S_{out2} \). This recursive process is iteratively applied across the layers to ultimately produce the final statevector of the entire circuit.

### 3.2.2.2 Class CircuitListToMatrix.cnotLayerMat(CNot_gate_list)

The getter function of this class, every time it encounters of C-NOT gate calls this function to retrieve the corresponding C-NOT gate matrix. It’s important to note
that, unlike single-qubit gates, the matrices associated with entangled gates are not constant. They dynamically change based on the specific arrangements of entanglement in the quantum circuit.

To form the C-Not matrix of a $m$ qubit gate, where there are $m - 1$ control qubits and a single target qubit, we will take an Identity matrix of size $2^m \times 2^m$. The row and column of this matrix will be indexed from 0 to $m - 1$. The indexing value will be changed in the binary and the place value of binary digits will correspond to the value of the qubit, numbered the same as the place value, see fig. 3.4.

Let a C-NOT gate where $q_0, q_1$ are the control qubits and $q_2$ is the target qubit as shown in figure 3.5. When the $q_0, q_1$ are 1, the $q_2$ will be flipped. This will produce a list of column-index pairs where the bits are going to be flipped, table 3.1.

From the table 3.1 we can see that only row indexes 110 and 111 are affected as in these two binary strings the $q_0$, and $q_1$ are 1 which flips the row index. The updated C-NOT matrix is shown in Fig. 3.6.
Figure 3.6: C-NOT Gate With Two Control and a Target Qubit

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\]

Table 3.1: C-NOT Gate Transition Table

<table>
<thead>
<tr>
<th>Row Index</th>
<th>Column Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>110</td>
</tr>
</tbody>
</table>

3.2.2.3 Class CircuitListToMatrix.toGateMatrix(Gate_Name)

This function serves as a matrix generator for non-entangled gates. It operates by taking a string as input, which encompasses the gate name and its associated parameters. This function is versatile and capable of computing matrices for a broad spectrum of quantum gates, including but not limited to H, X, I, Z, Sx, T, R, and U gates. Most of the gates have fixed matrices like H, X, Z, and Sx but rotation gates and U-gate matrices can be computed as discussed in the background section from equation (2.6).

3.2.2.4 Class CircuitListToMatrix.is_unitary(numpy.matrix)

To extract the debug information of the circuit this static method can be utilized. This function takes a numpy matrix object and returns True, if the argument passed, is a unitary matrix.
3.3 Derived HLS Architectures

In this work based on the space-time trade-offs, we derived three types of architectures in an FPGA application design. This application is in two parts as mentioned before, the host part and the controlled FPGA architecture.

3.3.1 FPGA Application - Host Part

The host part of the application is divided into three layers, the top of the layer stack is our software which communicates to XRT (Xilinx Runtime) [14], responsible for interaction between the host and FPGA. Lastly, a layer of communication driver, XOCL drivers (as in this work we are using PCIe-based accelerator), transfers the data between the host and the FPGA.

At the top of the stack, our software processes the matrix file generated by the software package. A file pointer reads the file and a complex number parser combines the read strings from the file and returns a complex number. This application uses a double-precision complex type to represent the quantum data. After getting the data from the parser we pass the data from the host to our FPGA architecture for the computation. This data transfer and hardware management will be done by using the OpenCl runtime XRT APIs. XRT has many features but its primary usage can be seen as programming the FPGA hardware, memory allocation, and hardware management.

Typically, when creating memory, we resort to the C or C++ new operator, which lacks strict alignment constraints and, in practice, can lead to significant memory fragmentation. Another approach that provides the memory with strict alignment is by utilizing the posix_memalign API. While this method ensures strict alignment like fixed page size (typically 4KB), the underlying issue of memory fragmentation persists. Utilizing posix_memalign offers a speed advantage over the new keyword due to its
inherent compatibility with DMA operations, as DMA devices often require memory to be aligned to a certain boundary to perform transfers efficiently. When data is not aligned with the page size, DMA must align the data to make our buffer’s content in order, which introduces additional overhead.

In our application, we used XRT for memory allocation, using `cl::buffer` constructor, see figure 3.8, where the context is a command queue contains device specific details, creates buffer at the device end, later we map user space pointers to these buffers for the application usage. The OpenCL API `clEnqueueMapBuffer` does this for us as shown in figure 3.9. For the $n$ qubit system, the buffer size will be $2^{2n} \times 16$ for the matrices, and $2^n \times 16$ for the statevector buffer, where 16 is the size of a double precision complex number.

While the creation of these buffers is somewhat costlier than the method discussed earlier, it significantly boosts the speed of read and write operations on these buffers. Given that our application necessitates frequent transfers of matrices to the device kernels, the initial allocation time for these buffers is not a major concern. The advantage of XRT-allocated buffers is their provision of pinned memory, where virtual addresses are directly mapped to physical addresses. This reduction in page address translation
time and DMA overhead results in faster read and write operations.

In the initial stage of our host code execution, it traverses through the list of connected platforms until it locates the Xilinx accelerator device. Once identified, it proceeds to load the specified kernel code onto the FPGA. It’s important to note that this initial configuration step can be relatively time-consuming. Following the creation of buffers, we set these buffers as kernel arguments and initialize our user space pointers. The next step involves transferring data from the host to the FPGA’s memory, and then we instruct the device to execute our kernel. After the kernel completes its execution, we reverse the process, migrating the data back from the FPGA to the host. For a more comprehensive understanding, please refer to the pseudocode provided in Figure 3.10.

### 3.3.2 FPGA Application - Programmable Logic Part

In HLS-based designs, the hardware synthesis phase can be time-consuming. Xilinx tools may take a significant amount of time, varying from minutes to hours, to convert our high-level language code into a hardware design. To address this, we’ve structured our kernel design to be highly adaptable. This means we can modify our application...
### Host Pseudocode:

```plaintext
for (j=0; j < pow(2, numberOfQubits))  //Creating ket 0 statevector
  if (j==0)
    statevectorBufferPointer[j++] = 1+0i
  else
    statevectorBufferPointer[j++] = 0+0i
while(FilePtr):
  mychar = myfile.get();
  complxStr += mychar;
  if (mychar == ',')  // Matrix file is comma separated between two complex numbers
    inputBufferPtr[i++] ← parseComplex()
    complxStr = " "
  if (i == matrixSize)
    i = 0
  enqueueMigrateMemObjects({InputbufferObjects, 0})  //From Host to Kernel
  enqueueTask(kernelObject)  //From Kernel to Host mapped
  statevectorBufferPointer ← enqueueMigrateMemObjects({OutputbufferObjects})
  CheckCommandFinishStatus()
```

Figure 3.10: Pseudocode for Host Application

for different scenarios, such as changing the number of qubits or switching between various quantum circuits, without the need for a lengthy synthesis process.

### 3.3.2.1 TYPE-1 Design

The design involves consecutive multiplication of the initial quantum state vector $S_{i/p}$ with each circuit layer matrix $M_{Li}$, to produce the final output state vector, $S_{o/p}$. This is performed by a single kernel that takes a layer matrix and state vector as inputs and performs complex matrix-vector computation, see Figure 3.11, where $K = 1$. The expression for the total execution time of this architecture is $(t_{avg} + t_{avg}^c) \times L$, where $t_{avg}$ is the average time of data transfer between host and kernel, $t_{avg}^c$ is the average computation time of kernel which will be the order of $O(N^2/2)$, where $N$ is the number of elements in the layer matrix, and $L$ is the total number of circuit layers. The space complexity of the architecture using 64-bit floating-point precision is $2^{n+5} + 2^{2n+4}$,
3.3.2.2 TYPE-2 Design

The implemented kernel in this design, see Figure 3.11, takes $K$ matrices as input, reducing the number of data transfers from host to FPGA. This design is optimal for the emulation of small circuits and utilizes the FPGA resources more efficiently. The data transfer time and overall application time are also reduced compared to the Type-1 design. For $K$ parallel matrix inputs, the number of kernel calls will be $r = L/K$. The time expression of this design is $r \times t_{\text{avg}} + L \times t_{\text{avg}}^c$ and space complexity will be $2^n + 5 + K \cdot 2^{2n+4}$.

The above-mentioned designs are implemented with AXI4 interfaces, allowing them to read directly from the device’s global memory. In our approach, we’ve created dedicated ports for each function input using the pragma HLS interface. This setup enables the kernel to read from the global memory using multiple AXI ports. There are also HLS::Stream<> based designs, which simplify memory management. However, these designs are less suitable for our purposes. Since matrix-matrix and matrix-vector multiplications involve iterative operations, using HLS::Stream<> isn’t practical. Saving data from these streams can increase the utilization of Block RAM (BRAM) and impact the design’s scalability.

After defining the ports and dedicated AXI channels, we’ve made our design pipelined
Figure 3.12: Matrix-Vector Kernel Function Code Snippet

with an initiation interval of 1. This ensures that for each clock cycle of the FPGA, we execute a single iteration of the loop. You can refer to Figure 3.12 for the code snippet of the Matrix-Vector kernel function. In our implementation, we’ve made an effort to make efficient use of available resources. We’ve designed the algorithm to load two-row elements of the matrix, $R_{i,j}$ and $R_{i+1,j}$ and the respective two column elements from the vector $C_i$ and $C_{i+1}$, where $i$ and $j$ represents the row and column position, and compute the result in parallel. Quantum matrices are square matrices with even orders, so, for the sake of generality, we have considered two-element operations at each clock cycle. However, the design can be expanded based on the number of qubits and available on-chip resources.

There is still scope to exploit the matrix-vector multiplication by splitting the computation into different concurrent kernels. We will discuss these kernel optimizations in the next chapter. Additionally, the next chapter will cover the implementation of the Type-3 design also.
Chapter 4

Kernel Optimization and
Concurrent Kernel Execution

In the previous chapter, our primary focus was on addressing memory scalability issues, streamlining DMA transfers, and implementing basic kernel optimizations through pragmas. We structured the kernels to enable pipelining of operations and at initiation interval of each clock cycle. In this chapter, we will split our kernels into multiple concurrent instances or computation unit (CU), allowing more processing to happen

Figure 4.1: General Concurrent Kernel Flow
in parallel. General concurrent flow of multiple parallel CU is shown in figure 4.1 flow.

4.1 Designing Concurrent Kernel

To introduce concurrency into our computation kernels, we need to partition the computation in a manner that allows us to create multiple computation units (CUs). Given that our architectures rely on matrix-vector multiplication, we’ll strategically divide this operation by handling calculations for both the upper half and lower half of the resultant vector (i.e., the statevector after the layer). This division will pave the way for concurrent processing and improved performance in Type-1 design. As we divide the overall computational task into two equal segments, we will maintain our existing strategy of computing two values at each clock tick. With this approach, our time complexity will further reduce to the order of $O(N^2/4)$.

4.1.1 Modified TYPE-1 Design

Now in the modified design, we will have two kernels, $krnl_vMmul_{Upper}$, for the computation of first $2^{n-1}$ elements and $krnl_vMmul_{Lower}$ for the rest $2^{n-1} + 1$ to $2^n$ elements as shown in figure 4.2. In a later section, we will see how to call these kernels in concurrent or parallel execution. The HLS code for implementing this kernel modification is illustrated in figure 4.3. It closely resembles the code presented in the previous chapter for the Type-1 design, with the primary change being the loop bound, which is now set to half of the previous implementation. The HLS code for the lower part of the computation remains the same, but the data mapped from the host will vary for each of these concurrent kernels.
Figure 4.2: Task Divided Between Concurrent Kernels

```
1- void krnl_vMmul_Upper(complex_t* in1, complex_t* in2, complex_t* out, long long int constSize) {
2- #pragma HLS INTERFACE m_axi port = in1 bundle = gmem0
3- #pragma HLS INTERFACE m_axi port = in2 bundle = gmem1
4- #pragma HLS INTERFACE m_axi port = out bundle = gmem2
5-
6-   complex_t temp1 = 0;
7-   complex_t tempBuff1 = 0, tempBuff2 = 0, tempBuff3 = 0, tempBuff4 = 0;
8-   int index = 0;
9-
10-   for (long long int i = 0, count = 0; i < constSize*constSize/2; i++) {
11-     #pragma HLS pipeline II=1
12-     READ: tempBuff1 = in1[i];
13-     tempBuff2 = in2[count];
14-     tempBuff3 = in1[i+1];
15-     tempBuff4 = in2[count+1];
16-
17-     COMP: temp1 = temp1 + (tempBuff1 * tempBuff2) + (tempBuff3 * tempBuff4);
18-     if (count == (constSize-2)) {
19-       write: out[index++] = temp1;
20-       count = 0;
21-       temp1 = 0;
22-     } else {
23-       count += 2;
24-     }
25-   }
26- }
```

Figure 4.3: Modified Matrix-Vector multiplication Kernel Code Snippet
4.1.2 Type-3 Design

In a quantum circuit, no two layers are dependent on each other, for example, the layer $L_l$ is independent of all other Layers, we can exploit this fact and multiply the group of two adjacent layer matrices in concurrence to form a single layer. This means the layer $L_l$ can get combined with $L_{l-1}$ or $L_{l+1}$. The approach involves passing $K$ layer matrices into kernel buffers. Matrix-Matrix multiplication kernel stores the result in intermediate buffers and then these intermediate buffers get multiplied later to produce a final matrix representing the full quantum circuit. The resultant final matrix is used in matrix-vector computation to obtain the output state vector. Figure 4.4 shows the total architecture design with two kernels, one for matrix-matrix multiplication with time $t^c$, which will be the order of $O(N^3/2)$, and another for matrix-vector multiplication with time $t^{c'}$. The total time and space complexity, $T$ and $S$ respectively, of this design, is given by (4.1).

\[
T = (t^c_{avg} \cdot \log_2 K + t_{avg}) \times r + t^{c'}
\]

\[
S = K \cdot \left(\frac{k}{2} + 1\right) \cdot 2^{2n+4}
\]

(4.1)

Let’s take an example where the number of input ports, $K$ in the Matrix-Matrix multiplication kernel is 4. Then the kernel execution flow will be as shown in figure 4.5. At each time epoch, we parallelly compute $\log_2^K$ matrices and continue the same for $k/2$ times.

For the Matrix-Matrix multiplication similar kernel design was implemented as discussed in the previous chapter for the other types, AXI protocol is used to read and write data from the DDR memory of the device shown in figure 4.6. As discussed earlier, this design also computes two resulting values for the final matrix at each clock cycle.
4.2 Executing Multiple Compute Units

XRT APIs uses the command queue object to communicate with the FPGA kernels. By default, this command queue processes operations in a sequential order. Tasks are inserted into the queue via `clEnqueue*` operations, and the device processes them in the order they are added to the queue. However, it’s possible to change this behavior and execute operations in a different order by specifying a special flag when creating the command queue. Inside the FPGA all the kernels are implemented in dedicated hardware regions. These hardware regions can execute two different tasks in parallel without compromising the performance or resource allocation of other concurrent
Our application takes advantage of this feature by using the out-of-order command queue. These out-of-order (OOO) queues allow for more flexibility in launching multiple operations simultaneously, including both memory transfers and kernel calls as soon as respective hardware become available. Some operations demand synchronous operation like data transfer from the kernel to host should start after the execution of the kernel. These sync between commands can be maintained using the `cl::event` objects, see figure 4.7. Figure 4.8 presents a code snippet from our host application. This code snippet illustrates how we enqueue tasks in the OOO queue and effectively
use event objects to achieve synchronization. Specifically, we can observe how we set the wait status for the read task by providing the relevant wait event while enqueuing it. This mechanism ensures certain order in the OOO task execution.

Figure 4.9 shows the complete execution of the aforementioned code snippet (figure 4.8). XRT utilizes a FIFO command queue, and in an out-of-order execution, the scheduler continually checks for available resources. When the kernels are idle, it submits the kernel command jobs for execution. Subsequently, based on the next
command, which is a read operation with a dependency on kernel execution completion, the responsible hardware is the DMA. If the DMA is available and the respective kernel execution has finished, it pushes the read task to the DMA.

Since our design is symmetric, the kernels execute their tasks nearly simultaneously. They become available at the same time and commence execution at time $t_1$, ending at time around $t_2$. This synchronized execution allows them to run concurrently. The DMA operation waits for the kernel execution to complete and begins its operation at time $t_3$.

With this, we conclude our implementation of concurrent kernels. In our experimental results, we will further analyze and compare the performance of our design on real hardware to validate its effectiveness.
Chapter 5

Experimental Results And Analysis

5.1 Experimental Platform

To implement our design on physical FPGA hardware, we employed the Xilinx Alveo U-200 accelerator card [1]. For the development of test circuits and software-based simulations, we relied on the Qiskit[3] Software Development Kit (SDK) provided by IBM Quantum. The software-based simulations serve a dual purpose: firstly, they enable us to benchmark the performance acceleration relative to CPU-based simulations, and secondly, they serve as a validation tool to verify the accuracy of our results.

5.1.1 Xilinx Alveo

The Alveo U200 is equipped with a Xilinx UltraScale+ [6] FPGA. It features the XCU200 package, which provides various computational resources and high-speed transceivers. The FPGA in the Alveo U200 offers a variety of resources, including programmable logic cells, block RAM, DSP slices, and more. The platform diagram of the system is shown in the figure. 5.1. The card includes 4 DDR4 memory with a total size of 64GB
(16GB each) and 3 PLRAM each with a size of 128KB. The Alveo U200 connects to the host system via a PCIe Gen3 x16 interface. This high-speed interface allows for efficient data transfer between the host and the FPGA.

5.1.2 Initial Setup

For experimental purposes, we selected Grover’s multi-pattern search algorithm and applied it to search for two distinct states: \(|00\ldots0\rangle\) and \(|11\ldots1\rangle\) for varying numbers of qubits. The circuit depth or a number of layers of this circuit will be 16, the figure 5.2 shows an example 5 qubit circuit. To generate the quantum circuits and their QASM code, we utilized the qiskit circuit library. Using our proposed framework we processed the QASM file and derived the HLS code for all three types of architecture.

To establish a baseline for comparing our work with state-of-the-art software-based simulators, we employed IBM’s software-based statevector simulator[13]. We used
the \texttt{\%\%timeit} inline magic function to benchmark the execution timing of the used simulator. In the development of our HLS-based designs, we utilize the Vitis Software Platform Development Environment provided by Xilinx. This environment facilitates the integration of the code generated by our \textit{QASM-to-HLS} software framework into the corresponding VITIS HLS host and kernel files. Vitis software offers a Vitis analyzer tool that we used for profiling and performance analysis of our designs during our experiments.

Our experimental simulations are currently limited to 11 qubits due to constraints in our experimental environment. The theoretical limitations for simulating larger qubit systems are determined by the available DDR memory on the FPGA device. According to the space complexity equations, the Type-1 design could theoretically simulate up to 16 qubits, the Type-2 design with K=4 up to 15 qubits, and the Type-3 design with k=8 up to 13 qubits.

However, practical challenges have limited our simulations to 11 qubits. One challenge is that the XRT doesn’t support buffers equal to or larger than 4GB, and for 64-bit floating-point precision, the size of a matrix for 13 qubits would exactly be 4GB, which XRT cannot manage. Although this limitation could be addressed by breaking the matrix into smaller chunks, it remains a potential avenue for future optimization. Another limitation arises from the file size of large matrices. The Vitis HLS code editor struggles to process very large text files, and the File I/O system crashes after the 11-qubit threshold.
5.2 Evaluation of Implemented Designs

5.2.1 Type-1 design

Table 5.1 presents the hardware emulation results for the Type-1 design. The table provides details on the kernel execution time, total data transfers between the kernel and the host, and resource utilization, like LUTs, Registers, BRAM, and DSP. From the table, our design achieves a significantly lower total time of simulation and is almost $43 \times$ faster than the CPU when compared to Qiskit-based software simulations. These results prove that we succeeded in beating the CPU. Figure 5.3 and figure 5.4 display the timing traces captured from Vitis Analyzer, showcasing the execution of our Type-1 simulator. The use of separate ports for the matrix and statevector enables parallelization of the write operation to the DDR memory of our FPGA. Following the completion of the writing operation, the kernel execution occurs, and the results are written back. This process is iterative and sequential in terms of FPGA operations, for each layer of our Grover’s circuit.

5.2.2 Type-2 design

In our experiments, we implemented the Type-2 design with $K=5$, as depicted in figure 3.11. This design features four individual AXI ports directly reading the layer matrices.
Figure 5.3: Timeline Trace for Type-1 Design

Figure 5.4: Zoom In of Figure 5.3 for Single Kernel Run
from the DDR memory, resulting in a reduction in the total data transfer time compared to the Type-1 design, as shown in Table 5.2. Notably, the kernel execution time is comparable to that of the Type-1 design. The decreased data transfer time leads to a further reduction in the total execution time of the FPGA-based design, significantly improving the speedup achieved by our hardware-accelerated approach. In figure 5.5, which illustrates the timeline trace for the Type-2 design, we can observe 3 parallel write operations in the pair of two. Five writes for matrices ($K = 5$), and a short burst is from statevector. This parallelization is a key factor contributing to the reduction in the total data transfer time. We were expecting a total of 6 write operations in parallel, 5 matrix ports, and single statevector write, but the HLS compiler optimized our design for 2 AXI ports.
5.2.3 Concurrent Kernel Design: Type-1

As discussed in chapter-5, the modified version of the Type-1 design with two concurrent kernels was implemented and tested, similarly as we performed experiments on the Type-1 and 2 designs. Table 5.3 shows the results performed on the hardware of the Type-1 design’s concurrent implementation. From the table, the kernel execution time of this concurrent design almost went down to half of the previous type-1 design implementation. Figure 5.6 shows the timeline trace of kernels execution. From the figure, we can see that our design is running in full parallel way from reading the data and kernel execution to writing the result back.

5.2.4 Concurrent Kernel Design: Type-3

Type-3 design is the slowest design in comparison to our other implementations. While matrix-vector multiplication kernels have a worst-case complexity $O(N^2)$ (without any
optimization), this design has a complexity of $O(N^3)$ which makes this model slower in comparison to others. In our implementation, 4 CU with a total of 8 ports was implemented, refer to figure 5.7. From the table 5.4 we can see that only for 5 qubits, the design shows the speedup but for the rest of the experiments it performed poorly, also the design is expensive in terms of resource utilization. The concurrent execution of the design is shown in the timeline trace figure 5.8. In figure 5.8, at the beginning of the execution, four kernels run concurrently, producing two intermediate resultant matrices. These matrices are then multiplied concurrently in the next stage to generate the final matrix. Finally, the matrix is multiplied by the initial statevector. Despite making the design concurrent, the matrix multiplication involves two sequential stages, impacting the overall performance. This observation suggests a potential area for improvement, where optimizing the matrix multiplication process could further enhance the efficiency of the Type-3 design.
Figure 5.7: Implemented Design Diagram Generated by Xilinx VITIS IDE

Figure 5.8: Timeline Trace for Type-3 Design, K=8
5.3 Analysis of Results

The experimental results from our simulations lead to the conclusion that the concurrent version of Type-1 is the fastest among the various implementations. The timing comparison graph in figure 5.9 illustrates the timing performance of all designs relative to the software timing. The Type-3 design is not included in this graph as its performance is not comparable to the other designs. From the graph, the timing of the Type-1 concurrent kernel design exhibits a steady behavior and increases at a slower rate than the others. The speedup graph in figure 5.10 further supports the conclusion that the performance of the Type-2 and Type-1 concurrent designs is superior to the other implementations. This emphasizes the efficiency gains achieved through concurrent processing in these designs.
Figure 5.10: Speedup Comparison Graph
Chapter 6

Conclusion & Future Work

FPGAs can be used for efficient emulation of quantum algorithms, however, mapping quantum circuits to FPGA emulation architectures is challenging. The proposed automation framework facilitates the automatic mapping of quantum circuits to efficient FPGA emulation architectures. QASM-based quantum circuit description to HLS abstracts the Hardware complexity. Our heterogeneous application code and derived architectures outperformed the software-based simulation and showed $\times47$ speedup for the 11 qubits emulation.

Future work involves further optimizing the kernel to address the observed decline in speedup with an increasing number of qubits. The limitation imposed by the XRT can be mitigated by utilizing multiple small buffers, each with a size less than 4GB. Incorporating the latest revisions in OpenQASM can enhance the design for future emulations. Additionally, porting the proposed framework to HLS native languages like C/C++ has the potential to create a monolithic and portable software system, potentially improving overall emulator performance. These future enhancements aim to refine and advance the capabilities of the emulator.
Bibliography


