Moawad, Youssef (2025) *Architectures and optimisations for FPGA-based simulation of quantum circuits.* PhD thesis.

https://theses.gla.ac.uk/84894/

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

ARCHITECTURES AND OPTIMISATIONS FOR FPGA-BASED SIMULATION OF QUANTUM CIRCUITS

YOUSSEF MOAWAD

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

OCTOBER 2024

© YOUSSEF MOAWAD

Abstract

The increasing complexity and scale of quantum algorithms, coupled with the current limitations of physical quantum hardware, have led to a growing need for efficient quantum circuit simulation techniques. While CPUs and GPUs have traditionally been used for simulating quantum circuits, their energy consumption and scalability issues have prompted exploration into alternative platforms. Field-Programmable Gate Arrays present a promising alternative, offering the potential for customisable parallelism, energy efficiency, and flexible hardware configurations. This thesis investigates the use of FPGA architectures for Full State Vector Quantum Circuit Simulation, evaluating their performance, scalability, and energy efficiency relative to traditional CPU and GPU platforms.

The core aim of this work is to explore whether scalable FPGA architectures can be designed for quantum circuit simulation, and to assess their comparative performance and energy efficiency against established CPU and GPU solutions. The work was guided by several research questions: Can FPGA architectures be optimised for quantum circuit simulation? How does the performance of FPGA architectures scale with hardware utilisation? What types of circuits benefit most from FPGA-based simulation? Is there a performance-per-Watt advantage to using FPGAs over GPUs and CPUs?

To answer these questions, a variety of FPGA-based architectures were designed and evaluated. The architectural approaches investigated include Direct Iteration Processing, Buffered Architectures, and Gate Fusion Architectures. Each of these architectures was tested on benchmark quantum circuits, including Quantum Fourier Transform, and Grover's search algorithm, representing a range of qubit counts and gate complexities. These architectures were compared in terms of scalability, execution time, and energy consumption, with their performance assessed against CPU and GPU implementations. One of the key contributions of this thesis is a controlled gate scheduling optimisation designed to improve performance for control-heavy circuits (i.e. circuits with a high number of controlled and multi-controlled gates). This architecture demonstrated substantial performance improvements for some circuits, where it was up to $5\times$ faster than the baseline architecture. While the GPU still outperformed the FPGA in raw speed, the optimised architecture showed a significant energy

advantage, consuming $2.6\times$ less energy than the GPU for circuits with a high density of controlled gates. This highlights the potential of FPGA architectures to outperform traditional platforms in energy-constrained environments.

This work also presents a set of circuit width reduction techniques aimed at improving the scalability and efficiency of quantum circuit simulations on FPGA hardware. These techniques reduce the number of qubits required by identifying and transforming portions of the circuit that can be simplified without affecting the overall computation. Initially developed for circuits defining algorithms employing computational basis data encoding, the techniques were extended to handle circuits implementing algorithms employing the more widely-used amplitude-based data encoding approach, demonstrating their versatility. These optimisations were applied to circuits for computational fluid dynamics and quantum arithmetic, leading to more efficient use of FPGA memory and computational resources.

The introduced FPGA-based quantum circuit simulation platform is, to our knowledge, the first of its kind capable of simulating general-purpose quantum circuits, rather than being limited to specific algorithms or gate sets. Unlike many existing FPGA simulators that are specialised for particular quantum algorithms, such as Grover's search or the Quantum Fourier Transform, this platform is designed to simulate any quantum circuit regardless of its structure or gate complexity, at high numbers of qubits (> 25). We simulate general-purpose quantum circuits of up to 29 qubits in this work, but in theory, the platform can scale up to any number of qubits, given sufficient memory resources. This level of flexibility, combined with the ability to handle larger quantum systems, positions this platform as a significant step forward in FPGA-based quantum circuit simulation, making it a versatile and scalable tool for both research and practical quantum computing applications.

Overall, this thesis demonstrates that while FPGAs may not match the raw execution speed of GPUs, they offer significant advantages in terms of energy efficiency for quantum circuit simulation, particularly for control-heavy circuits. The control scheduling optimisation and buffering strategies were found to significantly improve performance, especially for circuits with high controlled-gate density. However, challenges remain in terms of scalability, with High-Level Synthesis limitations posing barriers to further performance gains. The use of multi-FPGA clusters and further advancements in High-Level Synthesis tools could address these limitations and enable FPGAs to handle larger quantum circuits more efficiently.

Acknowledgements

First and foremost, to my esteemed supervisors, Prof. Wim Vanderbauwhede and Dr. René Steijl, I would like to express my deepest gratitude for your invaluable guidance, encouragement, and unwavering support throughout the course of my research. Your expertise and insight have been instrumental in shaping this thesis, and I am truly fortunate to have had the opportunity to learn from you. We have been through a lot together during this journey, and you have supported me through the highs and the lows. I cannot thank you enough for your endless patience throughout these years.

I am deeply thankful to Dr. Syed Waqar Nabi, who supported me since my undergraduate studies, and without whom I would have had a much harder time getting this project off the ground. From the very beginning, your mentorship, insightful advice, and technical expertise were invaluable in shaping the direction of my research.

I am particularly grateful to my fellow PhD candidates and friends, Robert Szafarczyk and Andrew Brown, for all our coffees and lunches, during which we had invaluable discussions and shared insights, ideas, and even frustrations about our research journeys. Your companionship made the challenging times more bearable, and our conversations often sparked new ideas that contributed to the progress of my work. Thank you for being such great colleagues and friends throughout this journey.

I would like to extend my heartfelt thanks to my parents, for their unconditional love, and belief in me. Their constant encouragement has been my foundation and has carried me through the toughest moments of this journey. Mama, Papa, I am forever grateful for your support, without which none of this would have been possible. I can never thank you enough!

To my cousin, Abdelrahman, and my sister, Yomna, who were always there to offer me comfort when I needed it, I cannot thank you enough! Whether it was through late-night conversations, words of encouragement, or simply being there to listen, you both provided me with the emotional strength to keep going during the most stressful times. Your constant presence reminded me that I was never alone in this journey, and for that, I am truly grateful. I am very thankful to have such caring family members by my side.

To my close friends Yasmin, Peter, Dasha, Josh, Zoe, Iain, Freyja, Cal, and Chloe, thank you for being there to offer distractions when I needed them and for listening to my endless research updates (and, yes, complaints). Your support, both emotional and practical, gave me the balance I needed to get through this journey. Better friends have never and will never exist; and I consider myself extremely lucky to call each and every one of you my friend!

Lastly, I want to express my immense gratitude to my partner, Jenny, for her unyielding love and patience. Your constant encouragement and belief in me have been a source of strength, and your presence made the last critical steps of this journey possible. Thank you for standing by my side, and for believing in me, even as I struggled to believe in myself. Te quiero mucho!

To all of you, I owe a debt of gratitude that I cannot put into words. This accomplishment is as much yours as it is mine.

Dedicated to my parents, Mohamed and Naglaa, for their constant love and endless support.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Youssef Moawad

University of Glasgow

College of Science and Engineering

Statement of Originality to Accompany Thesis Submission

Name: Youssef Moawad

Registration Number: xxxxxxx

I certify that the thesis presented here for examination for a PhD degree of the University of Glasgow is solely my own work other than where I have clearly indicated that it is the work of others (in which case the extent of any work carried out jointly by me and any other person is clearly identified in it) and that the thesis has not been edited by a third party beyond what is permitted by the University's PGR Code of Practice.

The copyright of this thesis rests with the author. No quotation from it is permitted without full acknowledgement.

I declare that the thesis does not include work forming part of a thesis presented successfully for another degree.

I declare that this thesis has been produced in accordance with the University of Glasgow's Code of Good Practice in Research.

I acknowledge that if any issues are raised regarding good research practice based on review of the thesis, the examination may be postponed pending the outcome of any investigation of the issues.

Signature: Youssef Moawad

Date: 01/10/2024

Published Journal Papers

Moawad, Y., Vanderbauwhede, W. and Steijl, R. (2022) Investigating hardware acceleration for simulation of CFD quantum circuits. Frontiers in Mechanical Engineering, 8, 925637. (doi: 10.3389/fmech.2022.925637)

Moawad, Y., Vanderbauwhede, W. and Steijl, R. (2023) Quantum circuit-width reduction through parameterisation and specialisation. Algorithms, 16(5), 241. (doi: 10.3390/a16050241)

Conference Contributions

Moawad, Y., Vanderbauwhede, W. and Steijl, R. (2022) Transformations for accelerator-based quantum circuit simulation in Haskell. (doi: arXiv:2210.12703) - Presented as a poster at PLanQC Workshop at the International Conference on Functional Programming in 2022.

Table of Contents

Abstract

Acknowledgments

Education Use Consent ii

1 Introduction 1

1.1 Fundamentals 2

1.1.1 End of Moore's Law 2

1.1.2 Quantum Phenomena: Superposition and Entanglement 2

1.2 Quantum Computing 6

1.2.1 State-of-the-art in Quantum Computing Hardware 6

1.2.2 Qubits 9

1.2.3 Systems of Qubits 10

1.2.4 Quantum Circuit Model 11

1.2.5 Quantum Information Encoding 15

1.3 Quantum Algorithms 17

1.3.1 Deutsch and Deutsch-Josza Algorithms 17

1.3.2 Quantum Fourier Transform 17

1.3.3 Shor's Algorithm 18

1.3.4 Grover's Search Algorithm 20

1.3.5 Quantum Arithmetic 20

1.3.6 Computational Fluid Dynamics Applications 23

1.4 Field-Programmable Gate Arrays 26

1.4.1 FPGA Components 26
1.4.2 Dynamic Random Access Memory 27
1.4.3 Programming Models for FPGAs 28
1.4.4 HLS/OpenCL Compilation Flow 30
1.4.5 Optimisations for FPGAs 31

2 Literature Survey **33**

2.1 Quantum Computing Simulation 33
2.1.1 Full-state vector simulation 33
2.1.2 Single Amplitude Computer Simulation 35
2.1.3 Tensor Networks 36
2.1.4 Decision Diagrams 37
2.2 State-of-the art CPU/GPU and cluster-based simulators 40
2.2.1 CPU-based Simulators 41
2.2.2 GPU-based Simulators 44
2.3 FPGA Acceleration of Scientific Computing 45
2.3.1 Overview 46
2.3.2 Search 48
2.3.3 Graph Computing 49
2.3.4 Linear Algebra Applications 50
2.4 FPGA-based simulators 52
2.4.1 Quantum Circuit Emulation on FPGAs 53
2.4.2 Single-Amplitude Simulation 54
2.4.3 Reusable architectures 54
2.4.4 High qubit count architectures 55
2.4.5 Summary 57
2.5 Conclusion 57

3 Full State Vector Quantum Circuit Simulation 60
3.1 Direct Iteration Processing . . . 60
3.1.1 Examples . . . 62
3.1.2 Controlled gates . . . 65
3.1.3 Optimising Controlled Gates Scheduling . . . 66
3.2 Group-based Processing . . . 71
3.2.1 Examples . . . 73
3.3 Adding a buffer: Iterating over Buffer Passes . . . 75
3.3.1 Motivating different buffer cases . . . 76
3.3.2 Case 1 $t < l$: Single burst access per buffer pass . . . 79
3.3.3 Case 2 $t \geq l$: Two burst accesses per group pass . . . 82
3.4 Controlled gates in a buffered architecture . . . 84
3.4.1 Controlled Buffered Cases breakdown . . . 85
3.4.2 Examples . . . 86
3.4.3 Summary . . . 88
3.5 Gate Fusion . . . 90
3.5.1 Grover's diffusion operator example . . . 95
3.5.2 Buffer Case 2 for Gate Fusion . . . 97

4 OpenCL FPGA Implementation of Full State Vector QCS 99
4.1 Architectures Overview . . . 99
4.1.1 Quantum Circuit Representation and Execution Flow on FPGAs . . 99
4.2 Quantum Circuit Description to FPGA IR . . . 105
4.2.1 Core . . . 105
4.2.2 Testing . . . 106
4.2.3 Compiler . . . 106
4.2.4 Circuit Qubit Reduction . . . 107
4.3 OpenCL . . . 107
4.3.1 OpenCL Programming Model . . . 108
4.3.2 OpenCL for Intel-based FPGAs . . . 111
4.3.3 FPGA OpenCL Model . . . 111

4.4 Architectures Implementation Overview 114
4.4.1 Host Overview 114
4.4.2 Device Overview 115
4.4.3 Integrating the overall system 118
4.4.4 Summary of Architectures 119
4.5 Baseline NDRange Architecture 120
4.5.1 Host Code Description 120
4.5.2 Device Code (Kernel) Description 121
4.6 Unrolled Loops Architecture 123
4.6.1 Non-parallel version 123
4.6.2 Parallelism through unrolled loops 124
4.7 OnBoard Circuit Execution 125
4.8 Parallel Execution of Different Circuits 127
4.9 Scheduling Optimisation for Controlled Gates 128
4.10 Buffered Architecture 130
4.10.1 Case 1: Single Memory Access per Pass 130
4.10.2 Case 2: Two Memory Accesses per Pass 132
4.10.3 Double-Buffering 133
4.11 Gate Fusion Architecture 134
4.11.1 Case 1: Single Memory Access per Pass 135
4.11.2 Case 2: Two Memory Accesses per Pass 136
4.11.3 Double buffering for Gate Fusion 137
4.12 Summary 140

5 Circuit Transformations and Width Reduction Techniques 141
5.1 Contributions 141
5.2 Motivation 142
5.3 Background 143
5.4 Circuit Width Reduction by Qubit Specialisation 144
5.4.1 QFT-Based Adder 144
5.4.2 Step-by-step reduction of the Cuccaro Modulo Adder 145

5.5 Context and Published Works 152
5.5.1 Contributions of the published works 152
5.6 The D1Q3 Circuit 154
5.6.1 D1Q3 Part 1 155
5.6.2 Shift-and-add Squaring 156
5.6.3 D1Q3 Part 2 156
5.7 Reducing the D1Q3 Circuit 157
5.7.1 Further reduction 158
5.7.2 Reduction of the u^2 computation 158
5.7.3 Reduction including the modulo-adder 166
5.8 Complexity Analysis and Evaluation of Reduced D1Q3 170
5.8.1 Full circuit - before reduction 171
5.8.2 Quantum circuit after reduction steps 172
5.8.3 Examples of D1Q3 quantum circuit reduced to 25 qubits 173
5.8.4 Conclusion of D1Q3 Reduction 175
5.9 Circuits with Amplitude-Basis Encoding 176
5.9.1 Applying superposition to the reduced qubit 178
5.9.2 Summary 181

6 Evaluation of Developed Architectures 183

6.1 Evaluation Setup 183
6.2 Evaluation Circuits 184
6.3 Direct Iteration Processing Architectures Evaluation 185
6.3.1 BaselineNDRange 185
6.3.2 UnrolledLoops 189
6.3.3 OnBoardUnrolledLoops 190
6.3.4 TwoCircuitNDRange 192
6.4 Evaluating the Controls Scheduling Optimisation 194
6.4.1 QFT Experiments 194
6.4.2 Reduced D1Q3 Circuits 195
6.4.3 Grovers and Streaming Circuits Experiments 195

6.5 Buffered Architectures Evaluation 197
6.5.1 Single Buffering 198
6.5.2 Double Buffering 200
6.6 Gate Fusion Evaluation 201
6.6.1 Single-Buffered Gate Fusion 202
6.6.2 Double-Buffered Gate Fusion 203
6.7 FPGA vs CPU vs GPU 205
6.7.1 Controls Scheduling Optimisation on CPU and GPU 208
6.8 Energy Consumption Analysis 210
6.8.1 Architectures without Controls Scheduling Optimisation 210
6.8.2 OptContNDRange FPGA vs CPU and GPU 211
6.8.3 Relative benefit of Optimised Controls Scheduling 212
6.9 Summary 216

7 Conclusions **218**
7.1 Contributions to the Field 218
7.1.1 Novel Universal FPGA-based QC Simulators 218
7.1.2 Gate Fusion FPGA Architectures 219
7.1.3 Controls Scheduling Optimisation 220
7.1.4 Circuit Width Reduction 220
7.2 Summary of Results 221
7.2.1 Direct Iteration Processing 221
7.2.2 Buffered and Gate Fusion Architectures 221
7.3 Discussion 222
7.3.1 Research Questions 222
7.3.2 Limitations 225
7.4 Future Work 226
7.4.1 Multi-FPGA Clusters and Distributed Architectures 227
7.4.2 Real-Time Power Monitoring 227
7.4.3 Addressing HLS Limitations 227
7.4.4 Evaluating Other Quantum Circuits and Applications 228

7.4.5 Improved Memory Management and utilising HBM 229
7.4.6 Improving Gate Fusion and Further Evaluation 229
7.5 Closing Remarks 229

A Biased Quantum Floating Point Representation 231

Bibliography 235

List of Tables

5.1 Required number of qubits for original and transformed quantum circuits ($N_E = 3$). 173

5.2 Input and output of examples for 25-qubit example circuits. Mantissa is shown without 'hidden qubit'. 174

6.1 Total available resources on the Intel Stratix V GS D5 FPGA. 184

6.2 FPGA resource utilisation for a BaselineNDRange architecture with up to 8 Compute Units and 1 maximum control per gate; and average circuit runtimes for high qubit count QFT circuits on these architectures. 187

6.3 FPGA resource utilisation for a BaselineNDRange architecture with varying with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture. 188

6.4 FPGA resource utilisation for a BaselineNDRange architecture with up to 8 Compute Units and 29 maximum controls per gate; and average circuit runtimes for high qubit count Grover's circuits on these architectures. . . . 190

6.5 FPGA resource utilisation for a UnrolledLoops architecture with up to 8 Compute Units and 1 maximum control per gate. 191

6.6 FPGA resource utilisation for the OnBoardUnrolledLoops architecture with up to 8 Compute Units and 1 maximum control per gate. 192

6.7 FPGA resource utilisation for a TCNDRange architecture with up to 8 Compute Units and 1 maximum control per gate. 193

6.8 FPGA resource utilisation for the OptContNDRange architecture with up to 8 Compute Units and 1 maximum number of controls per gate. 195

6.9 FPGA resource utilisation for a OptContNDRange architecture with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture. 195

6.10 FPGA resource utilisation for the OptContNDRange architecture with up to 8 Compute Units and 29 maximum number of controls per gate. 198

6.11 FPGA resource utilisation for a UnrolledSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate. 199

6.12 FPGA resource utilisation for a UnrolledSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture. 200

6.13 FPGA resource utilisation for a UnrolledDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate. 200

6.14 FPGA resource utilisation for a UnrolledDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture. 201

6.15 FPGA resource utilisation for a GateFusionSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate. Also shown is the QFT circuits' performance for this architecture. 203

6.16 FPGA resource utilisation for a GateFusionSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture. 203

6.17 FPGA resource utilisation for a GateFusionDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate. Also shown is the QFT circuits' performance for this architecture. 204

6.18 FPGA resource utilisation for a GateFusionDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture. 205

6.19 Best-performing FPGA architectures without the optimised controls scheduling optimisation against the NDRange kernels on CPU and GPU. All runtimes are in seconds. For the FPGA architectures, the percentage improvement shown in parenthesis compared to the FPGA BaselineNDRange 8NCU architecture. For CPU and GPU, the performance improvement is shown compared to the best available FPGA architecture (SingleBuffered GateFusion 3BQS in the case of the QFT circuits). 207

6.20 Controls scheduling optimised NDRange architectures runtime comparison (in seconds) across different platforms. 209

6.21 Energy consumption comparison for the best performing architectures without the controls scheduling optimisation. 212

6.22 Energy consumption comparison for the NDRange architectures with the controls scheduling optimisation (OptContNDRange). Results shown in Joules. 215

6.23 Comparison of the performance improvements from the controls scheduling optimisation (OptContNDRange) across FPGA, CPU, and GPU platforms. The table shows the runtimes for each platform before and after applying the optimisation, with percentage improvements indicated in parentheses. In general, the FPGA demonstrates the largest relative benefit, particularly for control-heavy circuits like the streaming circuits. 215

A.1 Sub-normal numbers for $N_M = 4$ and $N_E = 3$. Leading qubit acts as 'sign' qubit. In 2's complement 'hidden qubit' is represented. 233

A.2 Example normalised numbers for $N_M = 4$ and $N_E = 3$. Leading qubit acts as 'sign' qubit. In 2's complement 'hidden qubit' is represented. 234

List of Figures

1.1 Examples of common quantum gates. 12

1.2 Alternative quantum circuit representation of the X (or NOT) gate. 12

1.3 Controlled-gate (in particular $CNOT$) circuit example. 12

1.4 Examples of multi-controlled gates. 13

1.5 Anti-controlled gate example. The right-hand side shows the equivalent circuit using normal controls. 13

1.6 Different possible controlled quantum gates mixing normal-controlled and anti-controlled gates. 14

1.7 Example quantum circuit which generates an entangled pair of qubits after execution. The resulting quantum register is in the entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. 14

1.8 General n-qubit QFT circuit. Circuit diagram from Nielsen and Chuang (2010) [1]. 19

1.9 Grover's oracle quantum circuit for a four qubit search register desired to be in state $|0011\rangle$. The top four qubits are the search register and the bottom qubit is the oracle. 21

1.10 Grover's diffusion quantum circuit for a four qubit search register. The bottom oracle qubit is not used in this operator. 21

1.11 Visualisation of Grover's search from [1, p. 253]. Here, the operator O is the Grover oracle and G is Grover's diffusion operator. 22

1.12 MAJ (Majority) and UMA (UnMajority and Add) operators used as building blocks for the Cuccaro adder. 22

1.13 A 4-bit input Cuccaro full adder circuit [2]. 23

1.14 4-bit input Cuccaro modulo adder circuit [2]. 23

1.15 $+1$ (left) and -1 (right) 1D streaming quantum circuits. 25

2.1 QuIDDs of different 2-qubit state vectors. Diagram from Viamontes (2007) [3]. 38

3.1 Direct iteration processing demonstration of applying a gate, G, to the first qubit ($t = 0$) of a two-qubit system. The arrows demonstrate the flow of data (the probability amplitudes) from the memory system of the simulation device (could be DDR, HBM, or another type of memory). The $t = 0$ case is the simplest in terms of memory access pattern, as the state vector is simply processed in pairs of contiguous amplitudes. 63

3.2 Direct iteration processing demonstration of applying a gate, G, to the second qubit ($t = 1$) of a two-qubit system. In this case, the state vector is processed in pairs, with an element pair stride of $2^t = 2$. 64

3.3 Direct iteration processing demonstration of applying a gate, G, to the second qubit ($t = 1$) of a three-qubit system. Amplitudes paired together for the same iteration are grouped by colour. 64

3.4 Direct iteration processing demonstration of applying a gate, G, to the third qubit ($t = 2$) of a three-qubit system. 65

3.5 Direct iteration processing demonstration of applying a gate, G, to the first qubit ($t = 0$) with a single control on the second qubit ($c_0 = 1$). The first iteration is skipped due to the conditions imposed by the control qubit. 66

3.6 Direct iteration processing applying a gate, G, to the second qubit ($t = 1$) of a three-qubit system with a single control on the first qubit ($c_0 = 0$). The red boxes indicate skipped iterations due to the control qubit. 67

3.7 Direct iteration processing applying a gate, G, to the third qubit ($t = 2$) of a three-qubit system with the other qubits acting as controls ($c_0 = 0$ and $c_1 = 1$). The red boxes indicate skipped iterations due to the controls. Since each control halves the number of iterations which update the state vector, the number of required iterations here is a fourth of the original. 67

3.8 Access patterns for gate applications for a 3-qubit register. The numbers on the state vector represent the index of the complex probability amplitude. The table shows how the amplitudes are accessed depending on different target qubits. Like-coloured boxes are accessed and computed on together. . . 69

3.9 Access patterns for controlled gate applications for a 3-qubit register. For demonstration, the boxes representing the amplitudes are rearranged such that required pairs are contiguous, in contrast to Figure 3.8. Crossed out pairs indicate skipped iterations due to the controls imposed on the gate. The pattern of control skips is the same when controls are arranged in ascending order, as shown. 70

3.10 Group processing application of gate G, to the first qubit ($t = 0$). 73

3.11 Group processing application of gate G, to the second qubit ($t = 1$). 74

3.12 Group processing application of gate G, to the second qubit ($t = 1$) with a control on the first qubit ($c0 = 0$). 74

3.13 Group processing application of gate G, to the third qubit ($t = 2$). 75

3.14 Buffered gate application of gate G, to the first qubit ($t = 0$) for with buffer size 4 ($l = 2$). 81

3.15 Buffered gate application of gate G, to the second qubit ($t = 1$) for with buffer size 4 ($l = 2$). 82

3.16 Buffered gate application of gate G, to the third qubit ($t = 2$) for with buffer size 4 ($l = 2$). 84

3.17 Buffered gate application of a controlled gate G, to the second qubit ($t = 1$) for buffer size 4 ($l = 2$) with a control on the first qubit ($c0 = 0$). 87

3.18 Buffered gate application of a controlled gate G, to the first qubit ($t = 0$) for buffer size 4 ($l = 2$) with a control on the second qubit ($c0 = 1$). 87

3.19 Buffered gate application of a controlled gate G, to the second qubit ($t = 1$) for buffer size 4 ($l = 2$) with a control on the last qubit ($c0 = 2$). 88

3.20 Buffered gate application of a controlled gate G, to the third qubit ($t = 2$) for buffer size 4 ($l = 2$) with a control on the first qubit ($c0 = 0$). 89

3.21 Buffered gate application of a controlled gate G, to the third qubit ($t = 2$) for buffer size 4 ($l = 2$) with a control on the second qubit ($c0 = 1$). 89

3.22 Buffered gate application of a controlled gate G, to the second qubit ($t = 1$) for buffer size 2 ($l = 1$) with a control on the last qubit ($c0 = 2$). 90

3.23 Demonstrating application of two quantum gates, with gate fusion, to a 3-qubit quantum register. The targets of both gates are such that they both are Case 1 for a buffered architecture, allowing them to be fused. For a buffer with $l = 2$, the 3-qubit state vector is divided into two buffer passes. Each buffer pass executes two iterations, for each of the fused gates. As before, the red line separates sequentially executing parts of the simulation (there is only one buffer). For each gate, the buffer is read with the same strategy as described for the Case 1 buffered architecture in Section 3.3.2. This allows for both gates to be simulated while performing the same number of memory accesses as needed for one gate. 94

3.24 Grover's diffusion quantum circuit for a four qubit search register with gate fusion for a buffer qubit size $l = 3$. The blue boxes indicated fused quantum gates. 95

3.25 Alternative gate fusion ($l = 3$) scheme for Grover's diffusion circuit. 96

3.26 With $l = 4$, and no restriction on the number of gates which can be fused, the entire diffusion operator can be fused into a single gate block. 96

3.27 Fused gate blocks for $l = 4$ with a restrictive gate block buffer, with a maximum gate count of four per block. 96

3.28 Application of two buffer case 2 quantum gates using gate fusion. The condition for fusing case 2 gates is that all gate target qubits must be the same; in this case both gates target the third qubit, $t = 2$. There are still two buffer passes, however each pass now requires two contiguous memory accesses instead of one. 98

4.1 QProblem to FPGA flow for a gate-by-gate architecture. At each gate, the QP instruction is sent to the compute units over the host-FPGA PCIe connection. 102

4.2 3-qubit QFT circuit used for demonstrating the IR representation for the FPGA platform. 103

4.3 eDSL to FPGA intermediate representation. 106

4.4 OpenCL platform model. From [4]. 108

4.5 OpenCL execution model [4]. 109

4.6 OpenCL NDRange kernels work-items indexing and mapping example. From [4]. 109

4.7 OpenCL memory model [4], showing the differences between, global, constant, local, and private memory regions. 110

4.8 Intel SDK for FPGAs programming flow. From [5]. 112

5.1 Example specialisations of the Draper QFT-based adder [6], for input a values of $+4, +5$, and $+7$. Note that most significant bits of the input integers are shown at the bottom of the integer's quantum register (e.g. for $a = +4$, we have $a2 = 1, a1 = 0, a0 = 0$). 145

5.2 Original 3-qubit Cuccaro Modulo adder unpacking the MAJ and $UMAJ$ blocks, as opposed to Figure 1.14. 146

5.3 Step 1 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$. 146

5.4 Step 2 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$. 147

5.5 Step 3 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$. 148

5.6 Step 4 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$. 149

5.7 Step 5 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$. 149

5.8 Step 2 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |010\rangle$. 150

5.9 Step 3 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |010\rangle$. 150

5.10 Step 4 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |010\rangle$. 151

5.11 Step 3 of reducing 3-qubit Cuccaro Modulo Adder for $|a2|a1|a0\rangle = |011\rangle$. . 151

5.12 Step 4 of reducing 3-qubit Cuccaro Modulo Adder for $|a2|a1|a0\rangle = |011\rangle$. . 152

5.13 Step 5 of reducing 3-qubit Cuccaro Modulo Adder for $|a2|a1|a0\rangle = |011\rangle$. . 152

5.14 Quantum circuit design for evaluation of $\vec{g}^{eq}$ in modified D1Q3 model. Velocity u and distribution functions defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$. Most-significant qubits at top of circuit. Part 1. 160

5.15 Quantum circuit design for evaluation of $\vec{g}^{eq}$ in modified D1Q3 model. Velocity u and distribution functions defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$. Most-significant qubits at top of circuit. Part 2. 161

5.16 Quantum circuit defining $SQ4$ with 4-qubit mantissa qubits ($N_M = 4$, using hidden qubit approach) and a 3-qubit exponent ($N_E = 3$). 162

5.17 Quantum circuit defining $ISQ4$ with 4-qubit mantissa qubits ($N_M = 4$, using hidden qubit approach) and a 3-qubit exponent ($N_E = 3$). 162

5.18 Quantum circuit-implementations of $SgnA5$ and $SgnB4$ used to introduced sign change in 5-qubit modulo addition. 163

5.19 Reduced circuit with 26 qubits: $|dv1|dv0\rangle = |01\rangle$ or $|10\rangle$, $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$, $|101\rangle$ or $|110\rangle$ (and therefore $|cut\rangle = |0\rangle$). 163

5.20 CC_{u2} for reduced circuit with 26 qubits. Squared velocity u^2 defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$. Label indicates for which $|eu2|eu1|eu0\rangle$ circuit was derived. 164

5.21 Reduced circuit with 25 qubits: $|dv1|dv0\rangle = |01\rangle$ or $|10\rangle$, $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$, $|101\rangle$ or $|110\rangle$ (and therefore $|cut\rangle = |0\rangle$). 164

5.22 Operator 0110 for reduced circuit with 25 or 26 qubits. Squared velocity u^2 defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$. . . . 165

5.23 Definition of $SQ4$ and $ISQ4$ for reduced quantum circuit (26 qubit) for $|mu2\rangle = |0\rangle$ ($|qu3|qu2\rangle = |10\rangle$ for normalised input). $CAdd^{10}$ and $CRmv^{10}$ are defined as Add^{10} and Rmv^{10} with $|anc\rangle$ acting as control qubit. 166

5.24 Definition of $SQ4$ and $ISQ4$ for reduced quantum circuit (25 qubit) for $|mu2|mu1\rangle = |00\rangle$ ($|qu3|qu2|qu1\rangle = |100\rangle$ for normalised input). $CAdd^{100}$ and $CRmv^{100}$ are defined as Add^{100} and Rmv^{100} with $|anc\rangle$ acting as control qubit. 167

5.25 Definition of $SQ4$ and $ISQ4$ for reduced quantum circuit (24 qubit) for $|mu2|mu1|mu0\rangle = |001\rangle$ ($|qu3|qu2|qu1|qu0\rangle = |1001\rangle$ for normalised input). Controlled add/remove units not needed for the further reduction by 4 qubits (all mantissa qubits for $N_M = 4$). 168

5.26 Shift operators used for left- and right-shifting of results register in reduced circuit with 25 qubits 169

5.27 Reduced circuit with 25 qubits: $|dv1|dv0\rangle = |00\rangle$ or $|11\rangle$, $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$, $|101\rangle$ or $|110\rangle$ (and therefore $|cut\rangle = |0\rangle$). 170

5.28 Re-arranged quantum circuit (without further reduction on workspace qubits). Quantum circuit shows setting of $-u$ (for $|dv1|dv0\rangle = |00\rangle$) or u (for $|dv1|dv0\rangle = |11\rangle$) into 'a' register of modulo-5 adder. Velocity u and squared velocity u^2 are defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$. . 171

5.29 Quantum circuits for $MA5$ used in reduced circuit with 26 qubits and with 25 qubits for $|dv1|dv0\rangle = |00\rangle$. 172

5.30 Memory requirements for the original circuit and the two types of reduced circuits as a function of N_M, for $N_E = 3$ 173

5.31 Proposed quantum circuit for demonstration of application of circuit reduction techniques to circuits utilising amplitude-basis encoding. 176

5.32 Split circuits based on the applying the circuit reduction techniques to $|q_0\rangle$ in the circuit shown in Figure 5.31. The circuit on the left corresponds to $|q_0\rangle = |1\rangle$, and the one on the right is for $|q_0\rangle = |0\rangle$. 177

5.33 Alternate version of the circuit in Figure 5.31, adding an H gate to apply superposition to the qubit to be reduced. 178

6.1 Total circuit runtimes for the QFT circuit of with different qubit counts, for the BaselineNDRange architecture parameterised with 1 maximum controls per gate and for different numbers of compute units (NCU). 187

6.2 Total circuit runtimes for the Grover's circuit with different qubit counts, for the BaselineNDRange architecture parameterised with 29 maximum controls per gate and for different numbers of compute units (NCU). 189

6.3 QFT circuit runtimes for the UnrolledLoops architecture with varying numbers of compute units, compared to the best-performing BaselineNDRange architecture with 8 NCUs. 189

6.4 Performance comparison between the OnBoardUnrolledLoops and BaselineNDRange architectures for QFT circuits with varying qubit counts for different numbers of compute units. 191

6.5 Performance comparison of the TwoCircuitNDRange architecture for different numbers of compute units, against the 8 NCU BaselineNDRange architecture for QFT circuits. 193

6.6 Timing results of the Controls Scheduling Optimisation for the NDRange kernel with varying numbers of compute units, compared to the 8 NCU BaselineNDRange architecture for QFT circuits. 194

6.7 Total circuit runtimes for the Grover's circuit with different qubit counts, for the OptContNDRange architecture parameterised with 29 maximum controls per gate and for different numbers of compute units (NCU), shown against the BaselineNDRange with 8NCU. 196

6.8 Total circuit runtimes for the Streaming circuits with different qubit counts, for the OptContNDRange architecture parameterised with 29 maximum controls per gate and for different numbers of compute units, shown against the BaselineNDRange with 8NCUs. 197

6.9 Performance comparison of the UnrolledSingleBuffer architecture with varying buffer sizes (BQS) and the 8 NCU BaselineNDRange architecture for QFT circuits. 199

6.10 Performance comparison between the UnrolledDoubleBuffer architecture and the 3 BQS single-buffered architecture for QFT circuits. 201
6.11 Performance of GateFusionSingleBuffer architectures with different buffer qubit sizes (BQS) compared to the 8 NCU BaselineNDRange architecture and the 3 BQS SingleBuffered architecture for QFT circuits. 202
6.12 Performance of the DoubleBuffered Gate Fusion architecture compared to its single-buffer counterpart for QFT circuits. 204
6.13 Performance comparison between the best-performing FPGA architectures from each class and the NDRange kernels on the CPU and GPU for QFT circuits. 206
6.14 Performance comparison between the best-performing FPGA architecture (BaselineNDRange) and CPU/GPU for Grover's algorithm circuits. 206
6.15 Performance comparison between the best-performing FPGA architecture (BaselineNDRange) and CPU/GPU for streaming circuits. 207
6.16 Timing performance of QFT circuits across FPGA, CPU, and GPU platforms using the OptContNDRange architecture. 208
6.17 Timing performance of Grover's algorithm circuits across FPGA, CPU, and GPU platforms using the OptContNDRange architecture. 208
6.18 Timing performance of streaming circuits across FPGA, CPU, and GPU platforms using the OptContNDRange architecture. 209
6.19 Energy consumption comparison for QFT circuits using the best-performing FPGA architecture (GateFusionSingleBuffer 3BQS) compared to CPU and GPU platforms. 210
6.20 Energy consumption comparison for Grover's algorithm circuits using the best-performing FPGA architecture compared to CPU and GPU platforms. . 211
6.21 Energy consumption comparison for streaming circuits using the best-performing FPGA architecture compared to CPU and GPU platforms. 211
6.22 Energy consumption comparison for QFT circuits using the OptContNDRange architecture on FPGA, CPU, and GPU platforms. 213
6.23 Energy consumption comparison for Grover's algorithm circuits using the OptContNDRange architecture on FPGA, CPU, and GPU platforms. 213
6.24 Energy consumption comparison for streaming circuits using the OptContNDRange architecture on FPGA, CPU, and GPU platforms. 214

Chapter 1

Introduction

Quantum Computing is a rapidly evolving field encompassing several topics, including the study of quantum computational models, quantum algorithm development, hardware implementations of quantum computers, quantum error correction, simulation on classical systems, and more [1]. Despite being relatively young, with the seminal paper by Richard Feynman [7] emerging about 40 years ago, it has grown very quickly. For a more thorough treatment of recent advances in Quantum Computing and the challenges the field faces, we refer to the survey papers, [8] and [9].

The past two decades saw rapidly growing interest in quantum computing. Developments in quantum algorithms demonstrate the potential improvement in complexity offered, which ranges from quadratic to exponential. However, the cost of building and maintaining such a quantum computer is still relatively high and only a few operable ones currently exist. The technology and hardware needed to implement functional quantum computers also remain relatively young and in-development. Current quantum hardware is highly error-prone and has short coherence times (meaning quantum states, which is how information is stored in quantum computers, do not remain stable for long enough to allow for algorithms to run).

Quantum computing simulators come in as a fix for these issues. Access to functional and efficient simulators allows quantum algorithm researchers to test and verify their algorithms without access to a physical quantum computer. They allow for experimenting with novel quantum algorithms with perfect quantum behaviour (in preparation for fault-tolerant quantum hardware in the future), or to simulate how such algorithms would operate under noisy conditions such as with current quantum hardware. Typically, simulating quantum hardware can itself be quite expensive, since the complexity benefit gained by using such a computational model can be up to exponential compared to existing computational models, and thus requires exponential compute and memory resources to simulate.

In this chapter, a short primer on the theoretical background behind quantum computing is provided. For a more thorough treatment, we cite the quintessential reference on quan-

tum computing and quantum circuits, *Quantum computation and quantum information* by Nielsen and Chuang (2010) [1].

1.1 Fundamentals

In this section, the underlying quantum principles behind quantum computing are introduced and briefly explained.

1.1.1 End of Moore's Law

Moore's law is an observation made by Intel co-founder Gordon Moore, which states that the number of transistors that can be fit onto a computing chip doubles every roughly two years, by virtue of our ability to create ever smaller transistors. However, as this continues we eventually start reaching limits set forth by nature on how small we can create transistors. Since electrons have to flow through these transistors, the smaller the transistors are, the more likely quantum mechanical effects will cause errors in computation. In fact, this predicts that Moore's law must come to an end [10] in the next two decades.

1.1.2 Quantum Phenomena: Superposition and Entanglement

In the early 1900s, physics was faced with several crises that arose because the physical theories at the time were predicting absurdities such as the so-called ultraviolet catastrophe. In the early 1920s, the theory of quantum mechanics was developed to resolve these absurdities. A full treatment of quantum mechanics can be found in [11]. The rest of the section will describe the two important quantum mechanical phenomena relevant to discussing quantum computing.

Quantum theory, developed between 1900 and 1925, began with Max Planck's introduction of the quantum hypothesis to explain black-body radiation [12], followed by significant contributions from Albert Einstein and Niels Bohr, who extended these ideas to phenomena like the photoelectric effect [13] and atomic structure. Quantum mechanics, formalised in the mid-1920s by Werner Heisenberg, Erwin Schrödinger, and Paul Dirac, provides the mathematical framework to describe the behavior of particles at the quantum level using wave functions and operators [14]. While quantum theory refers to the broader conceptual foundation, quantum mechanics focuses on the precise mathematical models governing particle interactions. The primary difference lies in their scope: quantum theory encompasses the overarching principles, while quantum mechanics deals with specific, calculable predictions based on those principles. In the following prelude to quantum computing, we take quantum

mechanics as a starting point and briefly introduce concepts to provide an intuition of the ideas that enable quantum computing.

Superposition

A key feature of quantum mechanics is that particles, like electrons, atoms, or photons, do not have a definite position or momentum until they are observed (i.e. probed or measured in some way); instead existing in a **superposition** of all possible states. Such particles (or systems of multiple particles) constitute what the theory refers to as **quantum systems**. The Schrödinger equation (Eq. 1.1) lies at the heart of quantum mechanics, describing the evolution of a quantum system's **wave function**, $\psi(x, t)$. The wave function represents the state of a quantum system and encapsulates all possible information about the system, including the probability distribution of measurable properties such as position and momentum.

$$i\hbar \frac{\partial \psi(x,t)}{\partial t} = \hat{H}\psi(x,t) \tag{1.1}$$

This is the primary difference between **classical states** and quantum states: a classical state (e.g. a bit) exists as a single definite predictable value (0 or 1 in the case of a bit), whereas a quantum state (e.g. a quantum bit, explained further below) exists as a probability distribution until it is observed.

The Schrödinger equation describes the evolution of the wave function in terms of the system's Hamiltonian, $\hat{H}$, which contains information about the potential and kinetic energies of the particle(s) which constitute(s) the system. In its continuous form, the wave function can take on a range of values over a continuous Hilbert space [11] (with infinite degrees of freedom) and is usually used to describe systems like particles moving through space or oscillating in potential wells.

Quantisation One of the most important conceptual steps towards the description of quantum computing is the move to quantised discrete quantum states. Certain physical systems exhibit naturally **quantised states**, meaning that only specific, discrete measurable energy levels are allowed. Such systems include the electrons in an atom, where solutions to the Schrödinger equation constrain the energy levels of the electron to specific values, and the quantum harmonic oscillator, which, rather than having a continuum of energies, can only exist in discrete energy levels.

Ket notation The conventional way of representing quantum states is in ket notation, introduced by Paul Dirac [15]. Ket notation represents a quantum state as a vector in a complex vector space. A quantum state is denoted as a ket: $|\psi\rangle$, representing a column

vector containing the **probability amplitudes** of the probability distribution that represents the quantum system.

For example, consider an atom which can exist in some number, N, of energy levels, represented by $|\psi_0\rangle, |\psi_1\rangle, |\psi_2\rangle, ..., |\psi_{N-1}\rangle$. When observed, the atom takes on the state of one particular energy level. Until it is observed, however, no function can describe with certainty what state the atom will take on when it is observed. The only mathematical description of the energy level of the atom is the atom's wavefunction, which can be written as:

$$|\Psi\rangle = \alpha_0 |\psi_0\rangle + \alpha_1 |\psi_1\rangle + ... + \alpha_{N-1} |\psi_{N-1}\rangle = \sum_{k=0}^{N-1} \alpha_k |\psi_k\rangle$$

where $\alpha_k \in \mathbb{C}$ are complex coefficients of the observable states of the system, each representing the probability amplitude of its corresponding state. This wavefunction represents a system in **superposition**, and the probability amplitudes allow us to determine the probability of measuring the system in each of its observable states:

$$P(|\psi_k\rangle) = |\alpha_k|^2. \tag{1.2}$$

Note that because the squares of the probability amplitudes represent physical probabilities, they must add up to 1, imposing the condition:

$$\sum_{k=0}^{N-1} P(|\psi_k\rangle) = \sum_{k=0}^{N-1} |\alpha_k|^2 = 1,$$

known as the **normalisation condition**. This means that all vectors describing any quantum state must be unit vectors, and that any operation which mutates this vector (changing the state) must preserve this unitarity (mathematically, the operators on this vector space must be unitary). This also implies that any computational model that utilises such operators on these systems must be reversible (as unitary operators must have an inverse).

Often, a wavefunction is represented as an N-dimensional state vector containing the probability amplitudes that describe the system: $|\Psi\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ ... \\ \alpha_{N-1} \end{pmatrix}$.

Entanglement

The second important quantum mechanical phenomenon is **entanglement**. While there is no comprehensive theory explaining entanglement, it can be observed experimentally, and

in fact it is a fundamental pillar for the computational complexity improvements offered by quantum computing. Dubbed by Einstein as "spooky action at a distance", entanglement refers to the persistent correlation that emerges between quantum systems when they interact. As an example to demonstrate entanglement, consider two 2-level quantum systems, $|\Psi_1\rangle$ and $|\Psi_2\rangle$. Their wavefunctions can be described as $|\Psi_1\rangle = \alpha_0 |\psi_0\rangle + \alpha_1 |\psi_1\rangle$ and $|\Psi_2\rangle = \beta_0 |\psi_0\rangle + \beta_1 |\psi_1\rangle$. The systems can be combined into a single quantum system by multiplying their wavefunctions (or, equivalently, taking the tensor product of their state vectors):

$$|\Psi_1\Psi_2\rangle = |\Psi_1\rangle \otimes |\Psi_2\rangle = \alpha_0\beta_0 |\psi_0\psi_0\rangle + \alpha_0\beta_1 |\psi_0\psi_1\rangle + \alpha_1\beta_0 |\psi_1\psi_0\rangle + \alpha_1\beta_1 |\psi_1\psi_1\rangle .$$

Note that we can combine quantum states into a single ket representation $|\psi_0\rangle \otimes |\psi_0\rangle = |\psi_0\psi_0\rangle$ and that this operation is not commutative ($|\psi_0\psi_1\rangle \neq |\psi_1\psi_0\rangle$). When this is done, the result is a single 4-dimensional quantum system with observable states: $|\psi_0\psi_0\rangle$, $|\psi_0\psi_1\rangle$, $|\psi_1\psi_0\rangle$, and $|\psi_1\psi_1\rangle$. At this point, the system can still be "factorised" (or decomposed) back into two quantum systems with their own independent wavefunctions. However, imagine the independent systems are combined in such a way that $\alpha_0\beta_1 = \alpha_1\beta_0 = 0$ (more on how this is accomplished in the Quantum Circuit Model section later), i.e. the "mixed" states, $|\psi_0\psi_1\rangle$ and $|\psi_1\psi_0\rangle$, can no longer be observed. Then the wavefunction of the combined system takes the form $|\Psi_1\Psi_2\rangle = \gamma_0 |\psi_0\psi_0\rangle + \gamma_1 |\psi_1\psi_1\rangle$. γ_0 and γ_1 are used here because the probability amplitudes of this system would be different to simply the products of the amplitudes of the original systems. Note now that this combined system cannot be factorised down to two independent wavefunctions. However, the systems, $|\psi_0\rangle$ and $|\psi_1\rangle$, still exist as different physical entities and can in fact be separated by any distance, and more importantly, measured independently; it is only their mathematical description before measurement which can only be expressed as one function. The special property of this system now is that once one of the physical subsystems is measured into one of the observable states $|\psi_0\rangle$ or $|\psi_1\rangle$, the other subsystem's state is instantly determined and known; because the combined system can only exist either as $|\psi_0\psi_0\rangle$ or $|\psi_1\psi_1\rangle$, implying that after measurement, the individual subsystems' observed states must be equal. This happens regardless of the physical distance separating the two subsystems after entanglement.

Entanglement allows for quantum algorithms to be developed which have far more efficient complexity than can be developed classically and so has very tangible applications.

1.2 Quantum Computing

1.2.1 State-of-the-art in Quantum Computing Hardware

Quantum computing, once a largely theoretical field, has seen significant advancements in recent years, particularly in the development of quantum computing hardware. As of today, the field is characterised by a rapidly evolving landscape where multiple technologies and approaches are competing to achieve scalable, reliable quantum computation. These advancements are driven by both academic research and significant investments from industry leaders such as IBM, Google, Intel, and emerging quantum startups. For example, IBM predicts that by 2030, they will have developed error-corrected quantum computers with up to 200 qubits supporting up to 100 million gates [16].

At the forefront of quantum computing hardware are superconducting qubits, which have gained prominence due to their relatively advanced development and scalability. Superconducting qubits operate by exploiting the properties of superconducting circuits to create qubits that can exist in superpositions of states. The leading approach in superconducting circuits uses transmon qubits [17, 18], which are less susceptible to charge noise and offer coherence times long enough to perform meaningful quantum operations. Companies like IBM and Google have made significant strides with superconducting qubit-based quantum processors, with Google's Sycamore processor famously running a quantum circuit, performing a task that was thought to be infeasible for classical computers at the time [19] (although this was later shown to be possible to run efficiently on high performance classical hardware). The implementation of such systems still faces several challenges though, including short coherence times, as they are highly sensitive to environmental noise, and a high operational cost, as superconductivity requires ultra-low temperatures near absolute zero to be maintained (usually utilising liquid helium and requiring a vacuum).

Another promising avenue of quantum hardware development is trapped ion quantum computing [20]. This technology leverages individual ions trapped in electromagnetic fields, with quantum information encoded in their electronic states. Trapped ion qubits boast some of the longest coherence times and highest fidelity gate operations among current technologies, making them highly suitable for error-corrected quantum computing. However, challenges such as scalability and the complexity of the required laser systems still need to be overcome. Companies like IonQ and Honeywell are leading the way in this space, with recent demonstrations of high-fidelity quantum gates and small-scale quantum processors, with up to 32 entangled trapped ions [21].

Topological qubits [22, 23, 24], based on the manipulation of anyons and their non-abelian statistics, represent a more speculative but potentially revolutionary hardware approach. Topological qubits promise to be inherently resistant to local noise, which could greatly

reduce error rates and simplify the implementation of error correction protocols. While still in the early stages of experimental realisation, research efforts, particularly by Microsoft, are ongoing to make this approach viable.

Beyond these leading contenders, other qubit technologies such as quantum dots [25], photonic qubits [26, 27, 28], and neutral atom qubits [29] are also being actively explored. Each of these technologies brings its own set of advantages and challenges. Quantum dots, for instance, are appealing due to their potential for integration with existing semiconductor technologies, while photonic qubits offer the advantage of long-distance quantum communication.

In conclusion, the state-of-the-art in quantum computing hardware is marked by a diversity of approaches, each with varying degrees of maturity and potential. While no single technology has yet emerged as the definitive path to large-scale, fault-tolerant quantum computing, the progress across multiple fronts suggests that we are on the brink of significant breakthroughs. As research continues, the coming years are likely to see further refinements in these technologies, leading towards the ultimate goal of practical quantum computing.

Algorithm development is crucial for harnessing the potential of current quantum computers. As discussed, these machines are limited by their small qubit numbers and susceptibility to noise and errors, making classical error correction methods impractical. As a result, new quantum algorithms must be designed to operate within these constraints while delivering computational advantages. Developing algorithms that minimise gate operations, reduce error propagation, and are optimised for the specific architectures of existing quantum hardware is essential.

NISQ and the need for simulation

We are currently in what is known as the Noisy Intermediate-Scale Quantum (NISQ) era, a term coined by physicist John Preskill in 2018 [30]. This period in quantum computing is characterised by the existence of quantum processors with tens to a few hundred qubits, which are powerful enough to perform calculations that are beyond the reach of classical computers for specific tasks, yet still too small (in terms of number of qubits) and error-prone to achieve full-scale, fault-tolerant quantum computing. NISQ devices are capable of running quantum algorithms and experiments that offer a glimpse into the potential of quantum computing, but they are inherently limited by the noise and decoherence that affect quantum systems.

The primary challenges facing the development of quantum computers in the NISQ era revolve around the issues of noise and decoherence. Noise in quantum computing refers to unwanted interactions between qubits and their environment, which can lead to errors in

quantum computations. These errors arise from a variety of sources, including imperfections in qubit control, thermal fluctuations, and electromagnetic interference. Decoherence, on the other hand, is the process by which a quantum system loses its quantum properties due to interaction with its surroundings, causing qubits to revert to classical states. Both noise and decoherence severely limit the ability to perform long, complex quantum computations, as they introduce errors that accumulate over time, eventually rendering the results unreliable.

Addressing these challenges requires the development of error correction techniques and fault-tolerant quantum computing architectures. Quantum error correction involves encoding logical qubits into a larger number of physical qubits in such a way that errors can be detected and corrected without destroying the quantum information [31, 32, 33, 34, 35]. However, implementing error correction is resource-intensive, requiring a significant overhead in terms of the number of qubits and gate operations. This makes error-corrected quantum computing a distant goal, as current NISQ devices do not yet possess the necessary qubit counts or coherence times.

Quantum Computing Simulation

In light of these challenges, classical quantum computer simulators play a crucial role in the development and understanding of quantum algorithms and hardware. Classical simulators allow researchers to model and test quantum algorithms on conventional computers, providing valuable insights into how these algorithms might behave on real quantum devices. Since NISQ devices are still prone to errors and noise, simulators offer a controlled environment to study these effects in detail, helping to identify which algorithms are most robust against noise and which quantum error correction strategies might be most effective.

Moreover, classical simulators are essential for benchmarking and validating the performance of quantum computers. By simulating small quantum circuits and comparing the results with those obtained from actual quantum hardware, researchers can assess the accuracy and reliability of quantum computations. This is particularly important in the NISQ era, where understanding the limitations of current quantum processors is key to guiding future developments.

Finally, simulators enable the exploration of quantum algorithms that are too complex to run on current quantum hardware. As quantum devices grow in capability, simulators will continue to provide a bridge between theoretical research and experimental realisation, helping to pave the way for the transition from the NISQ era to the era of large-scale, fault-tolerant quantum computing. In this context, classical quantum computer simulators are not just a stopgap measure, but a vital tool in the ongoing quest to harness the full power of quantum computation.

As described in this section, multiple paradigms and architectures for quantum computing each with its own methodology for applying quantum gates and managing qubit topologies. These variations often depend on the physical constraints of specific quantum hardware, such as qubit-connectivity limitations, noise characteristics, and gate fidelity. In contrast, we highlight here that the focus of this work is on *hardware-agnostic quantum simulation*, where the developed simulation architectures operate independently of any specific quantum hardware or qubit topology. Our simulators assume idealised conditions in which every qubit in a system can interact freely with every other qubit, without the need for routing, error correction, or topology-aware gate decomposition. This abstraction allows for the faithful simulation of quantum algorithms as defined at a theoretical level, providing a pure representation of quantum computational behaviour. This approach is important for benchmarking the potential of quantum algorithms without conflating their performance with hardware-specific limitations. Furthermore, hardware-agnostic simulation serves as a foundational tool for exploring the scalability of quantum algorithms and evaluating their resource requirements, unencumbered by current hardware constraints. In addition, a hardware-specific noise model can be added to the simulator architecture to study how noise affects a particular quantum hardware configuration.

In the following sections, the fundamental concepts of quantum computing and the quantum circuit model are presented.

1.2.2 Qubits

The fundamental unit of information in quantum computing is the quantum bit, or **qubit**. Analogous to the classical digital bit, which can have a state of either 0 or 1, the qubit is a quantum system with observable states $|0\rangle$ and $|1\rangle$. Because the qubit is a quantum system, however, until it is measured, it can only be described mathematically by a wavefunction, from which probabilities for the qubit to be measured to exist in specific states can be computed; and until it is measured, it exists in a superposition of both observable states. This wavefunction is usually represented by a state vector containing the two complex probability amplitudes of the qubit. The probabilities of measuring each of the $|0\rangle$ and $|1\rangle$ states can be computed from these probability amplitudes by taking the square of their complex magnitude. We can write the quantum state of the qubit as a ket vector, $|q\rangle$:

$$|q\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix},$$

where α and β are the complex probability amplitudes of the states $|0\rangle$ and $|1\rangle$, respectively. This allows us to represent two states in superposition using one qubit. Note that the states

$|0\rangle$ and $|1\rangle$ can themselves be written as vectors: $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$; and that all qubit states are linear combinations of these two vectors, which form the basis states of this vector space, known as a **Hilbert space**.

As an example, consider the qubit state $|q\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle$. Taking the square of the magnitude of the probability amplitudes, the probabilities of observing each state is computed to be: $P(|q\rangle = |0\rangle) = P(|q\rangle = |1\rangle) = \left|\frac{1}{\sqrt{2}}\right|^2 = 0.5$. This implies that the qubit has an equal chance of being measured to be in either state. Note that because the probability amplitudes are complex numbers, there are an infinite number of states which give these probabilities; e.g. the state $|q\rangle = (\frac{1}{2} + \frac{1}{2}i) |0\rangle + (\frac{1}{2} - \frac{1}{2}i) |1\rangle$ is another which has an equal chance of being measured in either state: $(P(|q\rangle = |0\rangle) = P(|q\rangle = |1\rangle) = \left|\frac{1}{2} + \frac{1}{2}i\right|^2 = \left|\frac{1}{2} - \frac{1}{2}i\right|^2 = 0.5$.

1.2.3 Systems of Qubits

We can double the the allowed number of states by combining two qubits into one system: $|q_0\rangle = \alpha_0 |0\rangle + \beta_0 |1\rangle$ and $|q_1\rangle = \alpha_1 |0\rangle + \beta_1 |1\rangle$. With such a system, four states can be represented ($|00\rangle$, $|01\rangle$, $|10\rangle$, and$|11\rangle$) in superposition:

$$|q_0 q_1\rangle = |q_0\rangle \otimes |q_1\rangle = \alpha_0\alpha_1 |00\rangle + \alpha_0\beta_1 |01\rangle + \beta_0\alpha_1 |10\rangle + \beta_0\beta_1 |11\rangle = \begin{pmatrix} \alpha_0\alpha_1 \\ \alpha_0\beta_1 \\ \beta_0\alpha_1 \\ \beta_0\beta_1 \end{pmatrix}.$$

This is commonly thought of as the quantum system being *in all four states at the same time*. The number of states increases by a factor of 2 for each qubit, and we can write a full state vector for an n-qubit system by taking the tensor product of the individual qubit state vectors, forming a complex 2^n dimensional vector. This can written as a complex-weighted sum of all the possible states in the system, as in Eq. 1.3. Systems of qubits in superposition lets us take advantage of *quantum parallelism*, allowing us to perform quantum operations affecting several concurrent states (up to 2^n) using n qubits.

$$|\Psi\rangle = \sum_{k=0}^{2^n-1} C_k |k\rangle \tag{1.3}$$

Entanglement can be demonstrated on qubits by considering a system of two qubits prepared in a state such as $|q_0 q_1\rangle = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle$. Note that this state cannot be decomposed into two separate wavefunctions for each of the qubits, and that the only possible states in which

the system can be observed are $|00\rangle$ and $|11\rangle$. Like in the entanglement example above involving arbitrary quantum states, the qubits are still independent physical entities which can be separated by any space. However, once either qubit is measured (they each have uniform probability of being observed in either state), the state of the other qubit is immediately determined. This allows for advanced communication protocols such as quantum teleportation [36] to be realised.

1.2.4 Quantum Circuit Model

Several models exist for quantum computation, including adiabatic quantum computing [37], measurement-based quantum computing [38], the quantum Turing machine [39], and topological quantum computing [40]. However, the most common model for describing quantum computing and algorithms is the quantum circuit model [41]; and indeed all of these models have been shown to be equivalent and reducable to each other. This work focuses primarily on this model.

In the quantum circuit model, operators on qubits are represented as **unitary Hermitian matrices**, and are referred to as **quantum gates** [42]. Application is equivalent to matrix multiplication of the gate matrix by the state vector. Examples of single-qubit quantum gates include the X, or quantum NOT, gate (Eq. 1.4) which has the effect of swapping the probability amplitudes of the qubit (Eq. 1.5), and the H (Hadamard) gate (Eq. 1.6) which introduces a superposition in the qubit (Eq. 1.7).

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.4}$$

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \tag{1.5}$$

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.6}$$

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{1.7}$$

$$CNOT \equiv \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.8}$$

These gates are represented as 2×2 matrices; thus, in order to apply them to n-qubit systems (which have 2^n dimensional state vectors), an expanded $2^n \times 2^n$ matrix has to be computed, by taking repeated tensor products of the identity and the gate matrix according to Eq. 1.9.

$$G_t = \bigotimes_{i=1}^{n} \begin{cases} G, & i = t \\ I, & \text{otherwise} \end{cases} \tag{1.9}$$

Figure 1.1 shows examples of some quantum gates as they would be used in quantum circuits. The X gate is special in its representation as it can be represented as a box containing X, or (more commonly) as the $\oplus$ symbol as shown in Figure 1.2.

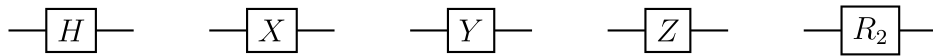


Figure 1.1: Examples of common quantum gates.

Figure 1.2: Alternative quantum circuit representation of the X (or NOT) gate.

Furthermore, a special type of many-qubit gates, known as controlled gates exist. These controlled gates allow us to introduce entanglement (discussed below) into the quantum systems. The most famous is the $CNOT$ gate, which has the effect of applying the NOT gate to the target qubit only if the value of the control qubit is $|1\rangle$ in the state vector. Other control gates exist like $CCNOT$ (a.k.a. $TOFF$ gates), which is a NOT gate with 2 control qubits, and CZ, controlled-Z, are common. Indeed, a controlled gate can be constructed using any number of controls and any quantum gate. Figure 1.4 shows examples of the $CNOT$ and $TOFF$ gates, and other multi-controlled gates.

In addition, an *anti*-controlled gate can be constructed, which has the effect of applying the target gate to the target qubit only in the cases where the control qubit's value is $|0\rangle$ (the opposite of a normal controlled-gate). This is represented by the control qubit having an unfilled circle as in Figure 1.5.

A quantum circuit is constructed by chaining several quantum gates on some number of qubits. Some restrictions are imposed on such circuits due to the nature of quantum mechanics. The No-Cloning property, proven in [43], prevents the copying of qubits and so no qubit

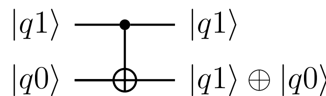


Figure 1.3: Controlled-gate (in particular $CNOT$) circuit example.

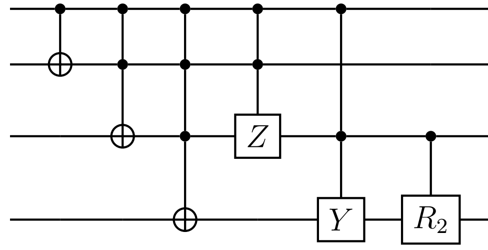


Figure 1.4: Examples of multi-controlled gates.

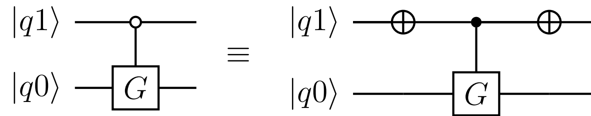


Figure 1.5: Anti-controlled gate example. The right-hand side shows the equivalent circuit using normal controls.

fanout or feedback is allowed. Such a circuit is shown below in Figure 1.7. This is a 2-qubit circuit, where the Hadamard gate is applied to the first qubit, introducing a superposition, and setting its state to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$. A controlled-NOT gate is then applied, where the control qubit is denoted by the black dot (the first qubit) connected to the target gate (X, or NOT) on the target qubit.

To execute this circuit, a quantum state is prepared as $|00\rangle$ and $H \otimes I$ followed by $CNOT$:

$$|\psi\rangle = |00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$|\psi'\rangle = (H \otimes I) \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}, |\psi''\rangle = CNOT\,|\psi'\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

This particular circuit results in an *entangled pair*, which represents a system where the only measurable states are $|00\rangle$ and $|11\rangle$. Here it is important to note that while our qubits may be part of one quantum system, they still exist an independent entities which we may be able to measure separately. However, because they are coupled, operations which affect one qubit will also affect the other. Particularly, when we perform a measurement on one qubit, we will immediately know the state of the other qubit (since if the first qubit is measured to be $|0\rangle$ e.g., the other qubit will necessarily have to also exist in the state $|0\rangle$, since there are no other states permissible by the system where the first qubit is $|0\rangle$). For the purpose of simulation, it

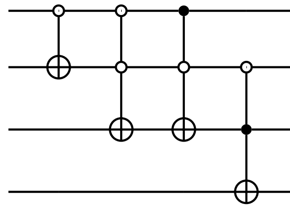


Figure 1.6: Different possible controlled quantum gates mixing normal-controlled and anti-controlled gates.

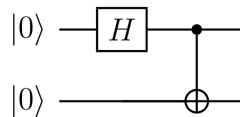


Figure 1.7: Example quantum circuit which generates an entangled pair of qubits after execution. The resulting quantum register is in the entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

is important to recognise that an entangled quantum system is one whose state vector **cannot** be written as the tensor product of smaller state vectors. For the above example, this means that while we can write the initial state vector, $|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, as a tensor product of the state vectors of the individual qubits, $|0\rangle\,|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, we cannot write the final entangled state, $\frac{|00\rangle+|11\rangle}{\sqrt{2}} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, as a similar tensor product involving the individual qubits.
This matrix multiplication is the main challenge in simulations of the quantum circuit model. Several optimisations can be applied to improve this (discussed in Section 2.1).

Remark about multi-controlled gates In idealised quantum circuits in the quantum circuit model, gate operations can include multi-controlled gates, such as the examples shown in Figures 1.4 and 1.6. However, on real quantum hardware, such multi-controlled gates do not exist as native operations. The fundamental hardware operations are typically single- and two-qubit gates, such as the $CNOT$ or controlled-Z gates, due to limitations in current quantum processors. To execute quantum circuits that involve multi-controlled gates on real quantum devices, transpilation is necessary. Transpilation translates abstract, high-level quantum circuits into an equivalent sequence of hardware-compatible operations.

For instance, in this work, we describe and use quantum circuits that utilise highly controlled gates. These gates, while convenient for theoretical descriptions, must be transformed into an implementable set of native gates (e.g., single- and two-qubit gates) specific to the hardware architecture. In the context of simulation, it may still be useful to use transpilers to convert a high-level circuit description into the native set of gates for some quantum hardware system; and simulate based only on those gate. On a real quantum system, there are also limitations on qubit connectivity due to the physical configuration of the system, whereas a general simulator assumes that all qubits are connected together. The transpilation can also take into account this configuration and can introduce additional gates to ensure that this is accounted for. This provides a closer-to-hardware simulation, as the gate sequences reflect the real qubit connectivity and limitations of physical qubits, ensuring that the simulation aligns more closely with how the circuit would behave on a quantum device.

Prevalence of the Quantum Circuit Model The quantum circuit model allows for the straightforward design and analysis of algorithms. These algorithms can be naturally expressed as circuits involving a combination of basic quantum gates. Quantum circuits can be mapped onto quantum hardware, where operations are implemented by applying physical gate operations to qubits (as long as the gate set is supported by the physical implementation, possibly through transpilation as described above). The model is hardware-agnostic, meaning it can be implemented on various quantum platforms, including superconducting qubits and trapped ion implementations. The quantum circuit model is universal, meaning that any quantum computation can be represented using a finite set of quantum gates. This allows for the efficient compilation of complex algorithms into simpler, hardware-executable sequences of quantum gates. Finally, The modular nature of quantum circuits makes them easier to scale and optimise, especially as quantum processors evolve to accommodate more qubits and gates.

1.2.5 Quantum Information Encoding

In quantum circuits, information encoding is crucial for the correct implementation of quantum algorithms. There are several methods to encode information in quantum states, with amplitude-basis encoding and computational-basis encoding being two primary approaches.

In **amplitude-basis encoding**, a vector of normalised complex data (of size up to $N = 2^n$) is encoded into the amplitudes, C_k in Eq. 1.3. This encoding technique is the most widely used encoding technique in quantum algorithms since it creates the most direct means of taking advantage of quantum parallelism (i.e., exponential growth of number of degrees of freedom with linear increase in the number of qubits). For this type of data encoding, the most widely used quantum circuit simulation approach on classical computers is **full state vector**

simulation, where the 2^n complex amplitudes are all stored and the gate operations in a considered circuit lead to step-by-step modifications of these amplitudes.

Alternatively, **computational-basis encoding** can be used. In this data encoding approach, the quantum algorithm is designed such that at initialisation only the complex amplitude of a single computational basis state has non-zero amplitude. After completion of the quantum algorithm the output is represented similarly by a single non-zero amplitude for one of the quantum basis states. For quantum algorithms employing the computational basis encoding a few important observations relevant to the present work can be made:

- The motivation for this type of encoding is typically performing quantum arithmetic operations;
- To maintain the property that only a single computational basis state has a non-zero amplitude throughout the computation, the gates in the Quantum Circuit model are limited to quantum equivalents of logic gates (e.g., X as equivalent of NOT and $TOFF$ as doubly-controlled NOT). By doing so, the quantum circuit can efficiently be simulated on a classical computer using a **logic-based simulator**. In such a simulator, n classical bits suffice to represent the state of n qubits. Then, the controlled logic gate operations conditionally flip states between 0 and 1;
- If quantum arithmetic operations are implemented in the quantum circuit model based on the Quantum Fourier Transform [6, 44], then efficient simulation using classical logic-based simulation is not possible, since the QFT in the case of quantum arithmetic circuits temporarily moves the encoding approach to amplitude-based encoding, before finally returning an output in computational basis encoding.

The key difference between these two methods lies in their flexibility and computational power. Amplitude-basis encoding leverages quantum superposition, allowing for more complex and parallel processing of information but requires careful manipulation of quantum states and is more challenging to measure directly. Computational-basis encoding is simpler to understand and implement, especially in the context of classical-quantum hybrid systems where measurements in the computational basis are standard, but it does not exploit the full potential of quantum parallelism as effectively as amplitude encoding.

In Chapter 5, we demonstrate how we can use the computational-basis encoding to specialise circuits for particular qubit initial values, reducing their qubit counts to achieve various goals including faster simulation times and lower memory requirements. We also demonstrate that this method does not limit the utilisation of the amplitude-basis encoding for the split circuits, allowing us to still benefit from quantum parallelism.

1.3 Quantum Algorithms

Algorithms which demonstrate the potential speed up offered by quantum computing include Shor's algorithm for factoring large numbers [45], Grover's search algorithm [46], and the Deutsch and Deutsch-Jozsa algorithms [47]. In addition, quantum-based algorithms have been devised with applications in Quantum Chemistry [48] [49], Quantum Physics, Machine Learning [50] [51], and other fields. Of particular relevance to this project is the work by Steijl et al. on quantum algorithms for Computation Fluid Dynamics [52] [53]. In this section, some of these algorithms are explored.

1.3.1 Deutsch and Deutsch-Josza Algorithms

The Deutsch algorithm, introduced by David Deutsch in 1985 [39], was the first quantum algorithm to demonstrate how a quantum computer could outperform a classical one. It solves a specific problem: determining whether a function is constant or balanced with only one query, whereas a classical computer would require two queries. This marked a major milestone in showing the power of quantum parallelism.

The Deutsch-Jozsa algorithm, an extension of the Deutsch algorithm developed by Deutsch and Richard Jozsa in 1992 [47], further demonstrated quantum speedup. It generalises the problem to functions with multiple inputs, providing a deterministic quantum solution in a single query, compared to the exponential number of queries required by classical algorithms. This algorithm was important in inspiring further research on quantum computation, leading to the development of more complex algorithms like Grover's and Shor's.

1.3.2 Quantum Fourier Transform

The Quantum Fourier Transform (QFT) [54] is one of the most fundamental operations in quantum computing, analogous to the classical discrete Fourier transform (DFT). The QFT is an essential tool for analysing quantum states in the frequency domain, which is critical in many quantum algorithms, particularly those dealing with periodicity and phase estimation.

The QFT operates on a quantum state by mapping an n-qubit input state $|x\rangle = \sum_{k=0}^{2^n-1} a_k |k\rangle$ into a new state $|y\rangle = \sum_{k=0}^{2^n-1} b_k |k\rangle$, where the coefficients b_k are determined by the Fourier transform of the original coefficients a_k. Mathematically, this is expressed as:

$$b_k = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} a_j \cdot e^{2\pi i \frac{jk}{2^n}}$$

The QFT leverages the superposition principle to perform this transformation across all basis states simultaneously, making it exponentially faster than classical DFT for large-scale problems. Implementing the QFT involves a series of Hadamard gates and controlled phase rotations. Specifically, the algorithm begins by applying a Hadamard gate to the first qubit, creating a superposition. It then applies controlled phase rotations, where each subsequent qubit is rotated by an angle that depends on the state of the previous qubits. This process is repeated iteratively across all qubits. Finally, the qubits are reversed in order to complete the transformation. This structure is shown in Figure 1.8.

One of the primary applications of the QFT is in Shor's algorithm (described below), where it is used to find the period of a function, which is a crucial step in integer factorisation. The QFT is also central to quantum phase estimation, which is used in quantum algorithms for solving problems such as eigenvalue determination in quantum chemistry and Hamiltonian simulation. The efficiency and speed of the QFT make it a cornerstone in the development of more complex quantum algorithms.

A key gate required for the QFT is the phase shift gate, $R(\theta)$, which applies a phase factor to the $|1\rangle$ component of the target qubit without affecting the probability of measure the qubit in the computational basis (i.e. in either the 0 or 1 state). This phase factor is parameterised by an angle θ, such that the application of $R(\theta)$ to a single-qubit quantum state, $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ can be expressed as:

$$R(\theta) |\psi\rangle = \alpha |0\rangle + \beta e^{i\theta} |1\rangle$$

In matrix form, the phase gate can be expressed as: $R(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$.

In the context of the QFT, the phase gate is parameterised by a discrete integer m, instead of a continuous angle θ. This discrete parameterisation corresponds to phase shifts that are powers of 2 in the denominator of the phase angle. Specifically, the integer-parameterised version is: $R_m = R(\frac{2\pi}{2^m}) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{2\pi}{2^m}} \end{bmatrix}$.

1.3.3 Shor's Algorithm

Shor's algorithm [55, 45] is one of the most significant breakthroughs in quantum computing, demonstrating a quantum computer's ability to solve certain problems exponentially faster than classical computers. Specifically, Shor's algorithm can factor large integers in polynomial time, a task that is infeasible for classical algorithms when the numbers involved are sufficiently large.

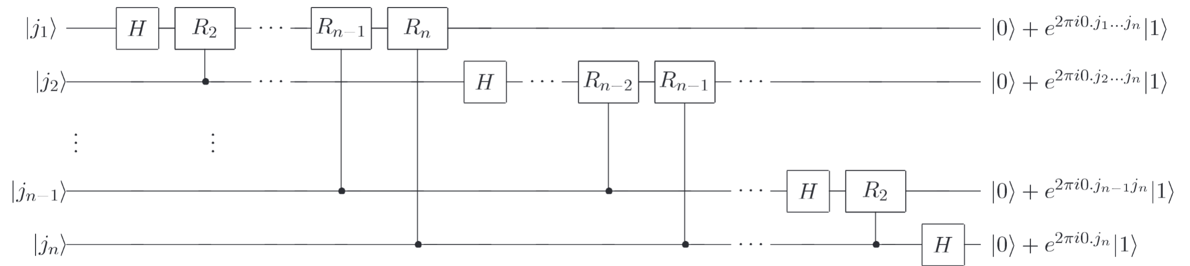


Figure 1.8: General n-qubit QFT circuit. Circuit diagram from Nielsen and Chuang (2010) [1].

The algorithm exploits the connection between integer factorisation and the problem of finding the order of an element in a multiplicative group modulo N. The order-finding problem, which involves determining the smallest integer r such that $x^r \equiv 1 \pmod{N}$, can be solved efficiently using quantum phase estimation (QPE). QPE is an important subroutine that provides the foundation for several quantum algorithms like Shor's algorithm and quantum algorithms for solving linear systems. The goal of quantum phase estimation is to determine the phase θ in the eigenvalue equation $U|u\rangle = e^{2\pi i\theta}|u\rangle$, where U is a unitary operator, and $|u\rangle$ is an eigenvector of U.

Shor's algorithm begins by choosing a random integer x and checking if it shares a non-trivial factor with N. If no such factor is found, the algorithm proceeds to find the order r of x modulo N using QPE. The quantum part of the algorithm involves preparing a superposition of states representing the possible outcomes of the modular exponentiation function $x^a \text{mod} N$. Quantum phase estimation is then used to extract the phase information corresponding to the order r, which is encoded in the quantum state's amplitudes.

Once the order r is determined, classical post-processing is used to find the greatest common divisor (GCD) of $\gcd(x^{r/2} \pm 1, N)$, which yields a non-trivial factor of N. This process is repeated until all prime factors of N are found.

Shor's algorithm has profound implications for cryptography, particularly for RSA encryption, which relies on the difficulty of factoring large integers. The ability of Shor's algorithm to factorise large numbers in polynomial time could theoretically pose a threat to current cryptographic systems in the long-term, highlighting the need for quantum-resistant cryptographic protocols. The emergence of this algorithm prompted the development of quantum-resistant encryption protocols giving rise to post-quantum cryptography.

1.3.4 Grover's Search Algorithm

Grover's search algorithm [46] represents a significant quantum advantage by providing a quadratic speedup for unstructured search problems. Classically, algorithms require $O(N)$ queries to search through an unsorted database of N items, but Grover's algorithm reduces this to $O(\sqrt{N})$ queries.

Grover's algorithm operates on a quantum register consisting of $n+1$ qubits, where n qubits constitute the **search register**, and a single ancilla qubit, known as the **oracle qubit**. The search register stores the superposition of all possible candidate solutions to the search problem. The algorithm starts by initialising the whole quantum register (including the oracle qubit) into an equal superposition of all possible states using Hadamard gates. Grover's algorithm then applies a sequence of operations known as the Grover iterate, which consists of two main steps: the oracle and the diffusion operator.

The oracle is a quantum subroutine that marks the correct solution by flipping the sign of its amplitude. This step does not alter the probability distribution but changes the phase of the correct solution, distinguishing it from the others. This is done using a multi-controlled NOT gate targeting the oracle qubit and using the search register's qubits as controls. Qubits desired to be in state $|0\rangle$ for the search are used as anti-controls, while qubits desired to be $|1\rangle$ are used as normal controls. An example Grover's search oracle for a 4-qubit search register desired to be in the state $|0011\rangle$ is shown in Figure 1.9.

The diffusion operator, often referred to as the "inversion about the mean," amplifies the amplitude of the correct solution while suppressing the amplitudes of the incorrect ones. This is achieved by reflecting all states about the average amplitude of the current state vector. Details of how this operator achieves this are omitted for brevity and can be found in the original paper [46] or in Nielsen and Chuang [1]. The diffusion operator only operates on the search register and does not utilise the ancilla qubit. The diffusion circuit for a 4-qubit search register is shown in Figure 1.10.

These steps are repeated $O(\sqrt{N})$ times (in particular $\frac{\pi}{4}\sqrt{N}$ times for optimal results), after which the correct solution's amplitude is significantly larger than the others. A final measurement on the search register collapses the quantum state, yielding the correct solution with high probability.

1.3.5 Quantum Arithmetic

In quantum computing, arithmetic operations form the building blocks of more complex algorithms. Cuccaro adders [2] are a specific type of quantum circuit designed to perform efficient binary addition on quantum computers. These adders are essential for implementing

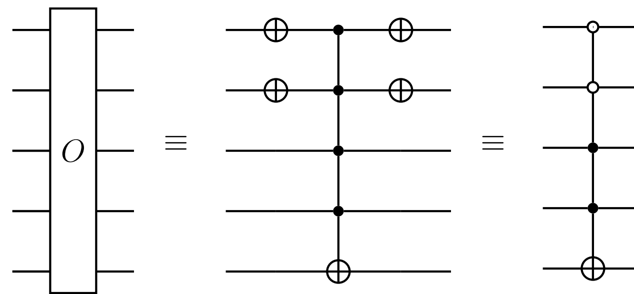


Figure 1.9: Grover's oracle quantum circuit for a four qubit search register desired to be in state $|0011\rangle$. The top four qubits are the search register and the bottom qubit is the oracle.

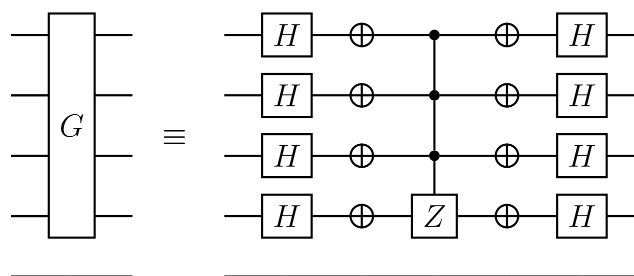


Figure 1.10: Grover's diffusion quantum circuit for a four qubit search register. The bottom oracle qubit is not used in this operator.

more complex arithmetic operations required in algorithms like Shor's algorithm, which relies on modular exponentiation and multiplication. [2] introduces two versions of the adder, a modular version with a demonstrably simple pattern of gates using MAJ (Majority) and UMA (UnMajority and Add) operators (shown in Figure 1.12), and a circuit depth optimised version with fewer overall number of quantum gates. A 4-bit input quantum circuit of the modular version is shown in Figure 1.13.

Cuccaro adders are constructed using a combination of controlled-NOT (CNOT) gates, TOFF gates, and NOT gates. The basic principle of the Cuccaro adder is to compute the sum and carry bits of two binary numbers in a reversible manner, which is a key requirement in quantum computing to preserve quantum coherence.

The addition process begins with the sum calculation, where each bit of the sum is computed using a series of $CNOT$ gates. These gates are applied to pairs of qubits representing the bits of the two numbers being added. The next step is carry propagation, where $TOFF$ gates are used to propagate the carry bits generated by the sum calculation. The carry bits ensure that the addition is performed correctly, especially when adding multi-bit numbers. In the modular version, this is accomplished using the MAJ operator.

The MAJ circuit is responsible for calculating the carry between successive bits. It takes three inputs: a (the first input bit), b (the second input bit), and c_{in} (the carry input from the

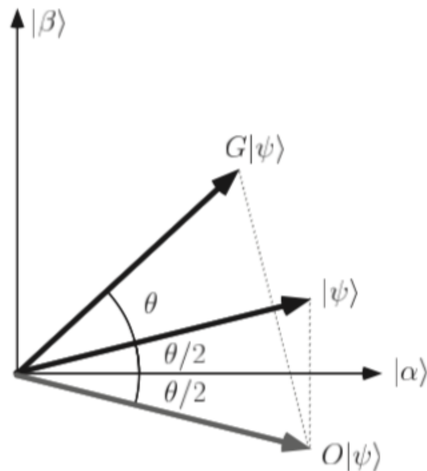


Figure 1.11: Visualisation of Grover's search from [1, p. 253]. Here, the operator O is the Grover oracle and G is Grover's diffusion operator.

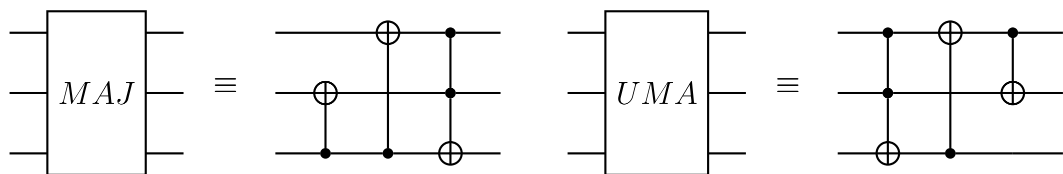


Figure 1.12: MAJ (Majority) and UMA (UnMajority and Add) operators used as building blocks for the Cuccaro adder.

previous bit). The circuit computes the new carry, c_{out}, as the majority function of the three inputs, meaning c_{out} is 1 if at least two of the inputs are 1.

After the carry is propagated, the UMA circuit comes into play to calculate the sum and reverse the carry propagation. The UMA operator takes as input the modified bit from the MAJ circuit, the second input bit, and the carry. It computes the sum as $a' \oplus b \oplus c_{in}$, using $CNOT$ gates to XOR the values. The UMA circuit then undoes the carry propagation to restore the original values of the qubits. This ensures that the circuit is reversible, as required in quantum computing, while calculating the final sum of the two binary numbers. Before the UMA operators calculate the sum, a single $CNOT$ is used to compute the final carry.

A modulo adder version of the circuit can be constructed by removing the ancillary qubit, z and the $CNOT$ operation that computes the final carry. The 4-bit version of this is shown in Figure 1.14.

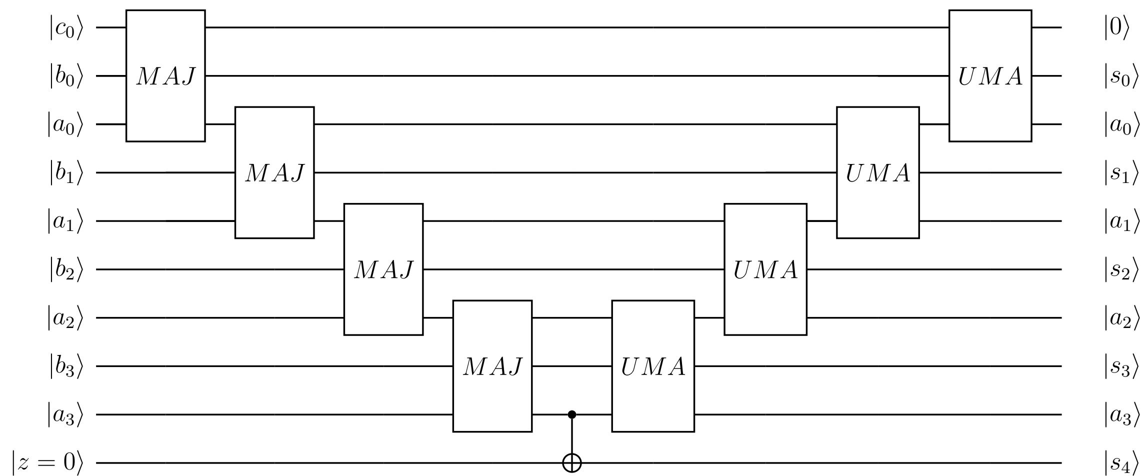


Figure 1.13: A 4-bit input Cuccaro full adder circuit [2].

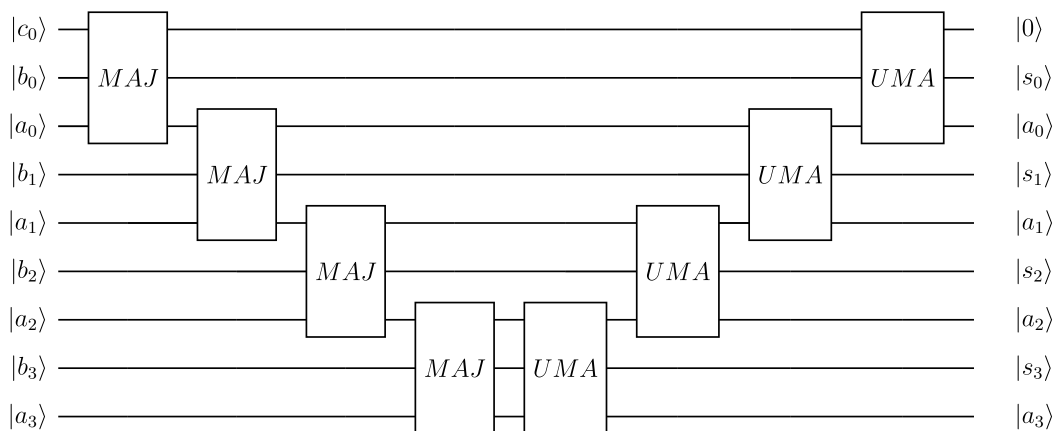


Figure 1.14: 4-bit input Cuccaro modulo adder circuit [2].

1.3.6 Computational Fluid Dynamics Applications

Computational Fluid Dynamics (CFD) involves the use of numerical methods and algorithms to solve and analyse problems involving fluid flows. CFD ssimulations are critical in a wide range of applications, including aerodynamics, weather prediction, and industrial processes. By solving the Navier-Stokes equations, which describe the motion of fluid substances, CFD enables the study and prediction of fluid behavior under various conditions. However, the complexity and non-linearity of these equations make them computationally intensive, especially for large-scale and high-fidelity simulations. Traditional CFD approaches often require substantial computational resources and time, which has driven the search for more efficient computational methods, including quantum computing approaches.

The Lattice Boltzmann Method (LBM) is a numerical approach used in CFD to simulate fluid dynamics. Unlike traditional methods that solve the Navier-Stokes equations directly, LBM operates on a mesoscopic scale, using a discrete lattice grid to model the fluid. It sim-

ulates the movement of fluid particles across this grid, capturing the macroscopic properties of the fluid through the collective behavior of these particles. LBM is particularly effective for simulating complex fluid flows and interactions at boundaries. It simplifies the treatment of complex geometries and boundary conditions, making it a powerful tool for various CFD applications. The method involves two primary steps: streaming, where particles move to neighboring lattice sites, and collision, where particles interact and redistribute their velocities.

The work [56], published during this PhD program in collaboration with the author's supervisors, aims to leverage quantum computing to enhance the efficiency of LBM simulations. In particular, it introduces quantum circuit implementations for lattice-based fluid dynamics models, specifically the D1Q3 model.

The D1Q3 model

A full treatment of the D1Q3 model can be found in the discussed work [56], here we provide a short summary and present the required equations for which the paper presents novel quantum circuits.

The quantum circuit described in the work is concerned with the computation of the equilibrium distribution function of a one-dimensional lattice model with three possible velocity directions (positive, negative, and zero), hence D1Q3. In terms of velocity, u, the equilibrium distribution function, $\vec{f}^{eq}$ is computed per Eq. 1.10. In this vector, the first element corresponds to the negative velocity direction, the second to the zero 'rest' velocity, and the third to the positive velocity direction.

$$\vec{f}^{eq} = \begin{pmatrix} \frac{1}{2}\left[\frac{1}{3} - u + u^2\right] \\ \frac{2}{3} - u^2 \\ \frac{1}{2}\left[\frac{1}{3} + u + u^2\right] \end{pmatrix} \tag{1.10}$$

To facilitate the implementation using the quantum circuit model, some modifications and re-scaling are introduced. Firstly, the original 3 direction vectors are replaced by 4, where the original single 'rest' velocity is replaced by two identical 'rest' velocities. This results in the following modified equilibrium distribution function (Eq. 1.11):

$$\vec{f}^{eq} = \begin{pmatrix} \frac{1}{2}\left[\frac{1}{3} - u + u^2\right] \\ \frac{1}{2}\left[\frac{2}{3} - u^2\right] \\ \frac{1}{2}\left[\frac{2}{3} - u^2\right] \\ \frac{1}{2}\left[\frac{1}{3} + u + u^2\right] \end{pmatrix} \tag{1.11}$$

The next modification step aims to remove the constant factors $1/3$ and $1/6$, by introducing a re-scaled equilibrium distribution function $\vec{g}^{eq}$ defined as the deviation away from the 'rest' state as,

$$\vec{g^{eq}} = \vec{f^{eq}} - \begin{pmatrix} \frac{1}{6} \\ \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{6} \end{pmatrix} = \begin{pmatrix} -\frac{u}{2} + \frac{u^2}{2} \\ -\frac{u^2}{2} \\ -\frac{u^2}{2} \\ \frac{u}{2} + \frac{u^2}{2} \end{pmatrix} \tag{1.12}$$

Eq. 5.1 is the fundamental equation for which a quantum circuit is presented in the work. This circuit is shown and explained in further detail in Chapter 5, which is based and builds on this work.

Streaming circuits

A specialised class of arithmetic circuits can be constructed which perform a specific operation on a quantum register encoding a single integer. One such operation is a simple "+1" addition, which represents a specialised version of the Cuccaro adder introduced above, shown in Figure 1.15 for a 6-bit integer input. Its inverse, implementing a "-1" operation, is also shown.

In the context of quantum circuit implementations of the Lattice-Boltzmann method, these circuits are known as streaming circuits, as they allow for representing the movement between neighbouring points in a uniformly-spaced lattice in a specific direction. The example shown here is for such movement in one direction. In [53], Todorova and Steijl introduce 2-dimensional versions of these circuits and utilise them in a quantum algorithm for solving the collisionless Boltzmann equation.

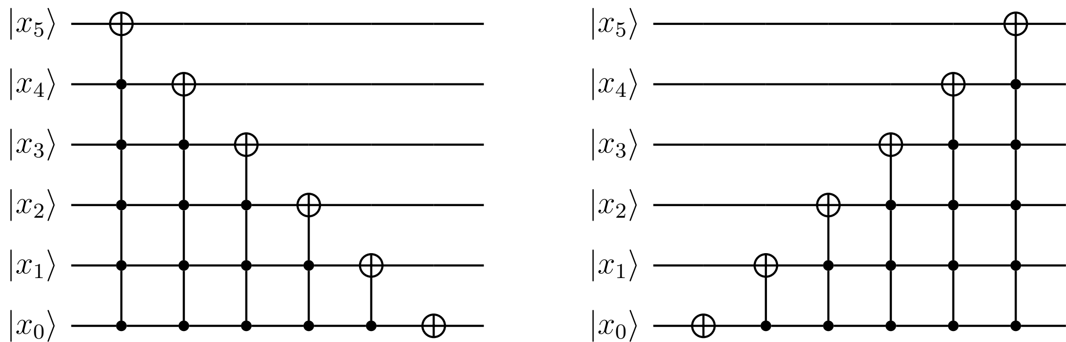


Figure 1.15: $+1$ (left) and -1 (right) 1D streaming quantum circuits.

These circuits are included here as they demonstrate a uniform variation of control counts in controlled quantum gates, and this property will allow us to showcase a novel optimisation for scheduling controlled gates in a quantum circuit simulator.

1.4 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are integrated circuits designed to be configured by the customer or designer after manufacturing—hence "field-programmable." FPGAs are composed of an array of programmable logic blocks and a hierarchy of reconfigurable interconnects, allowing blocks to be wired together similarly to that of a one-chip solution. These logic blocks can be configured to perform complex combinational functions, or simple logic gates like AND and XOR. In addition to logic blocks, FPGAs also include memory elements, which may be simple flip-flops or more complete blocks of memory.

The history of FPGAs dates back to the mid-1980s when Xilinx, a semiconductor company founded in 1984, introduced the first commercially viable FPGA. The XC2064, released in 1985, featured a modest 64 configurable logic blocks and a few hundred logic gates, which was revolutionary at the time for its flexibility and reprogrammability compared to Application-Specific Integrated Circuits (ASICs). This innovation laid the groundwork for a rapidly evolving field. Throughout the 1990s and 2000s, FPGAs grew exponentially in capacity and capability, with major players like Altera (acquired by Intel in 2015) and Xilinx (acquired by AMD in 2020) pushing the boundaries of performance and integration. Advances in semiconductor technology allowed FPGAs to incorporate millions of logic gates, sophisticated routing schemes, and hard blocks for specific functions like multipliers and memory interfaces.

1.4.1 FPGA Components

The core of an FPGA is made up of an array of configurable logic blocks (CLBs). These CLBs are responsible for implementing the logic functions required by the user's design. Each CLB typically contains a few basic logic elements (BLEs), which include lookup tables (LUTs), multiplexers, and flip-flops. LUTs are used to implement combinational logic functions, while flip-flops store the state information for sequential logic. The flexibility of these blocks allows them to be configured to perform a wide range of logic functions, making FPGAs suitable for diverse applications.

To connect the CLBs and other components, FPGAs utilise a network of programmable interconnects. These interconnects consist of wiring segments and programmable switches that can be configured to establish the necessary connections between different logic blocks. The interconnect architecture is hierarchical, allowing for both short-range and long-range connections. The programmability of the interconnects is what gives FPGAs their reconfigurable nature, enabling designers to create custom circuits and change them as needed.

Modern FPGAs include various types of embedded memory to store data and intermediate results. This memory can be distributed throughout the FPGA in the form of small blocks,

such as Block RAM (BRAM), which provides high-speed, on-chip storage. In addition to BRAM, some FPGAs include other types of memory, such as distributed RAM, which is implemented using LUTs, and larger embedded memory blocks that offer higher capacity. The availability of embedded memory allows FPGAs to handle data-intensive tasks efficiently.

1.4.2 Dynamic Random Access Memory

DRAM (Dynamic Random Access Memory) systems, including DDR (Double Data Rate) and HBM (High Bandwidth Memory), are widely used in modern computing for temporary data storage due to their high speed and large capacity. DDR memory, such as DDR4 and DDR5, provides a balance between speed and cost and is often used in general-purpose computing applications. HBM, on the other hand, is a newer technology that stacks memory vertically, enabling much higher data bandwidth while reducing power consumption and space. In FPGAs, both DDR and HBM can be integrated to enhance memory bandwidth, enabling the FPGA to handle more data-intensive tasks. DDR is typically used for applications where lower bandwidth is acceptable, while HBM is leveraged in high-performance applications like AI, machine learning, or high-frequency trading, where large data throughput is critical. These memory systems enable FPGAs to process larger datasets more efficiently, improving overall system performance.

FPGA memory interfaces are specialised modules that allow FPGAs to communicate efficiently with external memory systems like DDR and HBM banks. These interfaces handle the physical and protocol-level communication to ensure data is transferred correctly and at high speeds. DDR controllers manage the timing, signaling, and protocol for reading and writing data between the FPGA and the DDR memory, including handling row/column addressing, refresh cycles, and managing burst transfers to optimise bandwidth.

HBM interfaces are more complex because of the higher bandwidth and closer integration. HBM is stacked directly on top of the FPGA (often through a 2.5D integration using an interposer). The interface for HBM involves a memory controller that supports multiple channels (often 8 or more), enabling parallel data access. This allows the FPGA to access multiple memory banks simultaneously, significantly improving data throughput.

In both cases, memory controllers on the FPGA abstract the complexity of low-level operations, providing designers with easier access to memory while optimising performance based on application requirements. Some FPGAs offer pre-built, configurable memory IP blocks to further simplify integration and meet specific bandwidth and latency needs.

1.4.3 Programming Models for FPGAs

Programming FPGAs involves creating a hardware configuration that enables specific computational tasks, and various programming models cater to different levels of abstraction and control. Traditional **Hardware Description Languages** (HDLs) like **VHDL** and **Verilog** allow for detailed, low-level design, offering precise control over the hardware but at the cost of complexity. **High-Level Synthesis** (HLS) tools, which translate C/C++ code into HDL, reduce the complexity of FPGA programming by enabling developers to describe hardware using familiar high-level languages. **OpenCL** (Open Computing Language) [4] provides a framework for writing programs that execute across heterogeneous platforms, including FPGAs, offering a unified language for both software and hardware development. Additionally, **Domain-Specific Languages** (DSLs) like MaxJ and Halide offer tailored solutions for particular application domains, making FPGA development more accessible to domain experts.

Each programming model has its own compilation flow, transforming high-level descriptions into FPGA configurations:

1. HDL Compilation Flow:
   - Design Entry: The hardware design is described using VHDL or Verilog.
   - Synthesis: The HDL code is synthesised into a netlist, representing the logical elements and their connections.
   - Implementation: The netlist is mapped to the FPGA's physical resources through placement and routing.
   - Bitstream Generation: The final configuration is compiled into a bitstream file, which is used to program the FPGA.
2. HLS Compilation Flow:
   - Design Entry: The design is specified in C/C++ or OpenCL. In the case of OpenCL, this is split into host and device code.
   - HLS Synthesis: The high-level code is analysed and transformed into an RTL (Register Transfer Level) description. In the case of OpenCL, the device code is the transformation target.
   - HDL Compilation: The RTL code undergoes the traditional HDL compilation flow (synthesis, implementation, and bitstream generation).
3. DSL Compilation Flow:
   - High-Level Specification: The application is described using the DSL.

- DSL Compiler: The high-level code is translated into an intermediate representation or directly into RTL.
- HDL Compilation: Similar to HLS, the generated RTL code undergoes the traditional HDL compilation process to produce the final bitstream.

The versatility and adaptability of FPGAs in HPC stem from their reconfigurable nature and the diverse programming models and compilation flows available. From low-level HDLs to high-level abstractions like HLS, OpenCL, and DSLs, each approach offers unique advantages and trade-offs, enabling developers to tailor FPGA configurations for optimal performance and efficiency in various computational tasks. The following chapters will delve deeper into these programming models and compilation techniques, illustrating their application in real-world HPC scenarios.

OpenCL and High-Level Synthesis compilation flows share several similarities, particularly in how they abstract the hardware design process to make FPGA development more accessible to software developers. In fact, we can view the OpenCL flow as a special case of the HLS programming model. Both approaches allow developers to write code in high-level languages—C/C++ for HLS and OpenCL C for OpenCL—and then automatically translate this code into hardware descriptions that can be synthesised onto an FPGA. In both flows, the high-level code is first compiled into an intermediate representation, such as a Register Transfer Level (RTL) description, which is then synthesised, implemented, and finally translated into a bitstream that configures the FPGA. This abstraction simplifies the process of targeting FPGA hardware, allowing developers to focus more on algorithmic design rather than low-level hardware details.

In this thesis, we primarily choose the HLS/OpenCL approach. We justified this by the ability of this model to significantly reduce development time and complexity while still offering competitive performance. These high-level approaches allow developers who may not be experts in hardware design to effectively utilise FPGAs, leveraging their existing software development skills. Furthermore, the portability offered by OpenCL, which enables code to run across different types of hardware platforms including CPUs, GPUs, and FPGAs, is particularly advantageous in heterogeneous computing environments. HLS, on the other hand, allows for rapid prototyping and design space exploration, making it easier to iterate on designs and optimise them for specific performance or resource constraints. These advantages make OpenCL and HLS compelling choices for FPGA development. The next section illustrates more details about this approach.

1.4.4 HLS/OpenCL Compilation Flow

High-Level Synthesis for Field-Programmable Gate Arrays is a transformative approach that allows designers to use high-level programming languages like C, C++, or OpenCL to describe hardware functionality. This method abstracts the low-level hardware design process, enabling more rapid development and verification of complex systems.

High-Level Specification The HLS compilation flow for FPGAs begins with the high-level specification phase. In this phase, the algorithm to be implemented on the FPGA is initially designed using a high-level programming language. Designers identify parts of the algorithm that will benefit most from hardware acceleration and partition the design accordingly. Specific pragmas or directives are often used to guide the HLS tool on optimisation strategies such as loop unrolling, pipelining, and data flow control.

High-Level Synthesis Following the high-level specification, the high-level synthesis phase occurs. The high-level code is compiled using an HLS tool like Xilinx Vivado HLS or Intel HLS Compiler, translating the code into an intermediate representation. The HLS tool then schedules operations to optimise performance and resource utilisation and assigns these operations to specific hardware resources. Various optimisations, such as loop optimisations, data path optimisations, and memory access optimisations, are applied to enhance performance, resource utilisation, and power efficiency. The HLS tool ultimately generates RTL code, typically in VHDL or Verilog, representing the hardware description of the algorithm.

RTL Simulation and Verification The generated RTL code undergoes a crucial phase of RTL simulation and verification. Functional simulation ensures the correctness of the RTL code against the original high-level specification. Many HLS tools offer co-simulation capabilities, allowing the high-level and RTL code to be simulated together to ensure the synthesised hardware behaves as expected. Additional verification techniques, including formal verification and hardware-in-the-loop testing, can be employed to ensure the design's correctness and performance.

Synthesis Once verified, the RTL code proceeds to the synthesis phase, where it is fed into an FPGA synthesis tool like Xilinx Vivado or Intel Quartus Prime. This tool maps the RTL code to FPGA-specific primitives and applies various optimisations to reduce resource usage and improve performance.

Place and Route The place and route phase follows, involving the placement of the synthesised netlist on the FPGA fabric and routing the connections between placed elements. This ensures signal paths meet timing requirements and minimise delay. Post-placement and routing timing analysis is performed to ensure the design meets the required timing constraints.

Bitstream Generation The final steps involve bitstream generation and FPGA programming and testing. The place-and-route tool generates a bitstream file containing the configuration data for programming the FPGA. This bitstream is loaded onto the FPGA, configuring it to implement the designed hardware. The programd FPGA is then tested in the target system to validate its functionality, performance, and power consumption under real-world conditions.

In summary, the HLS compilation flow for FPGAs involves transitioning from high-level algorithm descriptions to low-level hardware implementations through a series of compilation, optimisation, synthesis, and verification steps. This flow leverages advanced tools to streamline the development process, making it more accessible and efficient to design high-performance hardware accelerators for high-performance computing (HPC) applications.

1.4.5 Optimisations for FPGAs

During the HLS step, compilation tools generally apply a variety of optimisations to enhance the performance and resource usage of the hardware design. In this section, these optimisations are presented.

Loop Optimisations

Several loop-based optimisations can be utilised to make optimal use of FPGA resources. **Loop Unrolling** involves replicating the loop body multiple times, reducing the overhead of loop control and allowing multiple iterations to be executed concurrently. Full unrolling removes the loop control logic entirely, while partial unrolling replicates the loop body a specific number of times. **Loop Pipelining**, also known as loop folding, allows multiple iterations of a loop to overlap in execution. By starting a new iteration before the previous one completes, loop pipelining increases throughput and reduces latency. **Loop fusion** combines adjacent loops with the same iteration space into a single loop, reducing loop overhead and improving data locality. **Loop fission**, or loop splitting, breaks a loop into multiple loops to expose parallelism or to fit hardware resource constraints better.

Data Path Optimisations

Operation Chaining links multiple operations together without intermediate storage, reducing latency. For instance, a multiply-accumulate operation can be chained to perform multiplication and addition in a single cycle. **Resource Sharing** aims to optimise resource utilisation, such that the same hardware resource can be used for multiple operations if they do not occur simultaneously. This is especially useful in designs with limited resources. **Retiming** is a technique that involves shifting operations across clock cycles to improve timing performance and achieve better clock frequency. Retiming can balance the critical path delays across different stages of the design.

Memory Access Optimisations

Burst Access groups multiple memory accesses into a single transaction reduces the overhead of individual memory transactions, thereby improving memory bandwidth utilisation. **Memory Partitioning** divides a large memory block into smaller, independently accessible partitions can increase parallel memory access, enhancing data throughput. This is particularly useful for arrays and large data structures. **Data Caching** involves the temporary storage of frequently accessed data in on-chip memory (e.g., BRAMs or LUTRAMs) with the aim of reducing the latency associated with off-chip memory accesses.

Data Flow Optimisations

Function Inlining involves replacing a function call with the function's body. Inlining reduces function call overhead and can expose additional optimisation opportunities at the cost of increased code size. **Data Dependency Analysis** can identify and minimise data dependencies between operations in order to expose parallelism. Techniques such as dependency breaking or reordering can help in optimising the data flow. **Pipelined Data Flow** invloves structuring the design to allow data to flow continuously through the pipeline stages without stalling, improving overall throughput. This involves careful management of data paths and control signals to avoid bottlenecks.

Chapter 2

Literature Survey

The literature surrounding quantum circuit simulation is expansive, covering a wide range of topics including classical simulation techniques, different theoretical models of quantum circuits, circuit compilation and representation, and suitability of quantum algorithms. In this chapter, the relevant literature is investigated.

We start by presenting the literature on quantum circuit simulation algorithms, followed by a study of the state-of-the-art CPU-based quantum circuit simulators. We then introduce FPGA programming methodologies, relevant FPGA-based applications, and finally existing FPGA-based quantum circuit simulators.

2.1 Quantum Computing Simulation

Quantum circuits can be simulated with a variety of different methods and techniques. In this section, these methods are presented and relevant works are discussed. We will also examine the limitations of classical simulators, particularly in handling circuits with a large number of qubits and complex gate structures

2.1.1 Full-state vector simulation

Quantum circuit gate operations can be represented mathematically as linear algebraic operations. Simulation which directly evaluates these operations is known as Schrödinger-style simulation, also known as full state vector simulation, since the full set of amplitudes specifying the state vector are stored and operated on.

To remove the requirement for computing and storing a $2^n \times 2^n$ matrix for each gate application in a quantum circuit, a technique called **qubit-wise multiplication** (QWM) is commonly used. Instead of computing the full expanded matrix, the gate application is performed as

repeated 2×2 matrix-vector multiplications of the gate matrix and permutations of pairs of amplitudes from the state vector. For a gate matrix G and target qubit t, we can write these updates as:

$$\begin{pmatrix} \alpha_{n_i} \\ \alpha_{n_i+2^t} \end{pmatrix} \mapsto G \begin{pmatrix} \alpha_{n_i} \\ \alpha_{n_i+2^t} \end{pmatrix},$$

where the substate vectors are constructed by: $n_i = \lfloor i/2^t \rfloor 2^{t+1} + (i \bmod 2^t)$ for $0 \leq i < 2^{n-1}$ [57].

QWM handles controlled gates in a special way. Controlled gates, though represented as multi-qubit gates, can be simplified. In QWM, they are treated as s-qubit gates, where s is the number of target qubits, and the gate matrix is applied only to the amplitudes of states that meet the control condition. For instance, a $CCNOT$ gate, typically a three-qubit gate with two controls and one target, is treated as a single-qubit gate in QWM, with the controls handled separately. As a result, single-control single-target gates affect only half of the states in the state vector, meaning only half the iterations update the state.

This also offers improvements in the structure of the allowed gates compared to direct simulation, which would require that qubits being operated on are always adjacent and in the right order. This requires direct simulation to utilise $SWAP$ gates to make sure qubits are in the right order for the operation being performed, which adds further significant overhead. However, because of the flexibility that qubit-wise multiplication offers, this is not required here.

Gate Fusion

Modern CPU caching architectures have evolved significantly to address the growing disparity between processor speeds and memory access times, commonly known as the "memory wall." Contemporary designs typically employ a multi-level cache hierarchy, with each level offering a trade-off between size, speed, and proximity to the processor core. Cache blocking, also known as loop tiling, is a crucial optimisation technique in high-performance computing (HPC) and scientific computing. This technique is particularly important for improving the performance of algorithms that operate on large datasets, especially in linear algebra and numerical simulations. Cache blocking aims to maximise data reuse within the cache by restructuring loops to process data in smaller chunks that fit well within the cache hierarchy. This approach reduces cache misses and improves overall memory access efficiency. The basic idea is to partition the problem space into smaller blocks that can reside in cache, allowing for multiple operations on the same data before it's evicted from cache.

Gate fusion [58] is a technique to allow for blocking the cache of a CPU by evaluating

several gates on the same slice of a state vector while it is still in the CPU's cache. This reduces the number of full state vector reads and writes required. In [57], the authors use the gate fusion optimisation to achieve cache blocking on GPU implementation of a full state vector simulation. The technique is described in detail in Chapter 3.

2.1.2 Single Amplitude Computer Simulation

An alternative to the Schrödinger-based method of simulation described in Section 2.1.1 is the Feynman Path Integral method [59, 60], which provides a framework for calculating quantum amplitudes by summing over all possible paths a particle could take between two points. In this formulation, rather than focusing on a single classical trajectory (as in classical mechanics), quantum behavior is described as a superposition of contributions from every conceivable path. The probability amplitude for a particle to transition from one point to another is given by integrating over all paths, each weighted by a complex exponential factor. This approach naturally captures the quantum phenomena of superposition and entanglement.

This formulation provides an alternative to maintaining the full state vector in memory and iterating over the entire state for each gate. Using this method means we only compute a single amplitude at the end of the entire circuit, without requiring to store the full state in global memory. It offers several opportunities for locally caching commonly used values. A quantum circuit simulator based on the Feynman path integral method would compute the evolution of a quantum system by evaluating all possible computational paths corresponding to different quantum states of the system.

In a quantum circuit, qubits evolve through a series of gates, with each gate applying a transformation to the quantum state. At each step in the circuit, the qubits can exist in a superposition of multiple states. The path integral formalism would treat each possible sequence of qubit states as a potential "path." Just as in the original path integral formulation, the simulator would sum the contributions of all possible quantum states the system can take throughout the circuit. For each possible sequence of state transitions, an amplitude would be computed based on the action of the circuit (analogous to classical action) along that path. The complex amplitude associated with each path reflects quantum interference, with different paths potentially interfering constructively or destructively. Simulators which use this method include the works described in [61, 62].

The main challenge in such a simulation is managing the exponentially large number of paths. As the number of qubits and gates increases, the number of possible paths becomes intractably large. Efficient simulation would require sophisticated methods to approximate the sum over paths without explicitly evaluating each one. Though this method in theory

aims to optimise memory use by not having to store the entire state vector; in practice, storing the weights contributed by the paths can still be exponential.

2.1.3 Tensor Networks

Tensor networks [63] are mathematical structures that generalise matrices to higher dimensions, allowing them to represent complex multi-linear relationships. In the context of quantum computing, tensor networks serve as a powerful tool for modeling and simulating quantum circuits. A quantum circuit can be viewed as a network of interconnected tensors, where each tensor represents a quantum gate or operation. The process of simulating the circuit involves contracting these tensors, which means systematically combining them to reduce the network to a single tensor representing the final output of the circuit.

Tensors can be graphically represented as nodes in a network, with the edges (or lines) representing the indices. Each tensor (node) can have multiple ”legs” (edges), where each leg corresponds to one of the indices of the tensor. When two tensors share an index, they are connected by an edge, which indicates a relationship or interaction between the data represented by the tensors. A tensor network is thus a collection of tensors interconnected by edges, representing how different parts of a system interact with one another. This graphical language is particularly useful in simplifying the visualisation and manipulation of complex multi-linear relationships in high-dimensional spaces.

The primary operation in tensor networks is contraction. Tensor contraction is a generalisation of matrix multiplication to higher dimensions. It involves summing over the shared indices (edges) between tensors, effectively reducing the dimensionality of the network. The result of a contraction between two tensors is a new tensor with fewer indices.

For example, if two tensors A and B share an index, contracting over this index involves summing over all possible values of this index, producing a new tensor C. This operation can be visualised as merging two nodes in a graph, where the shared edge is removed, and a new node representing the result is created.

One of the key challenges in simulating quantum circuits using tensor networks is determining the optimal order in which to contract the tensors, as the computational cost can vary significantly depending on this order. The complexity of tensor contraction is related to a graph-theoretical concept known as treewidth, which measures how ”tree-like” a graph is. The lower the treewidth, the more efficiently the corresponding quantum circuit can be simulated.

The work by Markov and Shi [64] leverages the relationship between treewidth and tensor network contraction to propose efficient methods for simulating quantum circuits. They show that for quantum circuits whose underlying graph has a small treewidth, the simulation

can be performed in polynomial time. This is particularly relevant for circuits with a regular structure, such as those that only involve local interactions between qubits.

Additionally, the authors extend their analysis to one-way quantum computation, a model where the computation is performed by measuring a highly entangled state known as a graph state. They demonstrate that if the underlying graph of the quantum state has a low treewidth, the one-way quantum computation can be efficiently simulated using tensor networks. This finding provides a crucial insight into the types of quantum circuits that are likely to offer computational advantages over classical ones.

Tensor networks were shown to be capable of simulating the QFT efficiently in [65].

2.1.4 Decision Diagrams

Highly-optimised approaches of the full state vector approach aimed at avoiding the exponential scaling of memory and computational cost with increasing number of qubits have been investigated for more than two decades. The primary idea behind such approaches is to exploit redundancies in the structure of the state vector, storing only non-zero amplitudes and storing repeated amplitudes only once. One possibility (e.g. Viamontes et al.[66] and Rosenbaum[67].) involves employing the Schrödinger wavefunction representation (as used in the full-state vector approach) along with compact representation of amplitudes using tree-based or decision-diagram based data structures. For a range of practically relevant quantum algorithms, significant memory and time savings were documented relative to the full-state vector approach. However, worst-case situations often occur where memory and time complexity are still exponential with number of qubits.

QuIDDs

In [66, 3], the author uses specialised binary decision diagrams (BDDs) called QuIDDs (*Quantum Information Decision Diagrams*) to represent the matrices and state vectors involved in a quantum circuit simulation. QuIDDs' main benefit comes from the compression it achieves in the memory required to store these matrices and vectors by exploiting the natural repetition in their structure. Some examples are shown in Figure 2.1.

BDDs were first described in [68] in the context of switching circuits. Later, ADDs (algebraic decision diagrams) [69] and MTBDDs (multi-terminal binary decision diagram) [70] demonstrated their viability in performing matrix operations including matrix multiplication and tensor products. Viamontes' work introduces QuIDDs as ADDs or MTBDDs which have terminal nodes that contain integer pointers to an array of complex numbers.

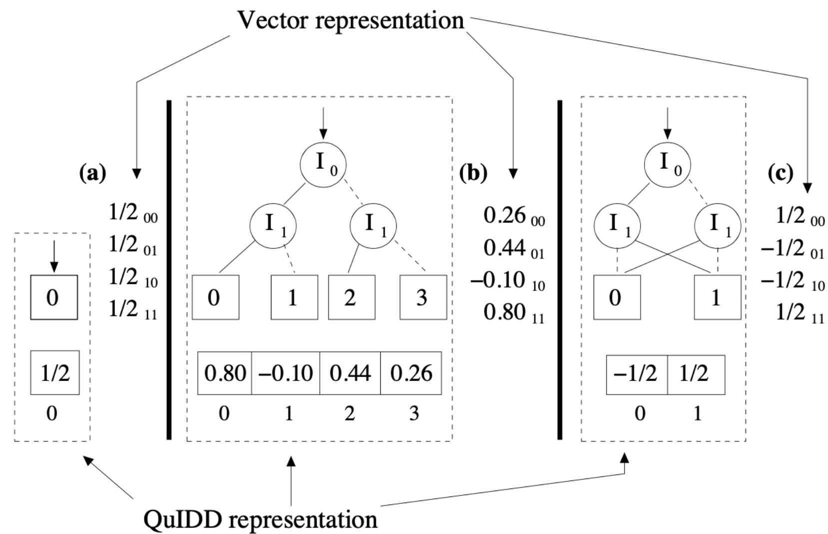


Figure 2.1: QuIDDs of different 2-qubit state vectors. Diagram from Viamontes (2007) [3].

A QuIDD represents quantum states and operations as a graph-based structure where nodes correspond to subfunctions or submatrices, and edges represent the relationships between these subfunctions. This structure allows for the sharing of common subcomponents, significantly reducing the memory required to store quantum states and operations. Specifically, QuIDDs leverage the fact that many quantum operations involve repeated patterns or values, which can be compactly encoded in the decision diagram. This encoding leads to a more scalable simulation process, where the time and memory complexities for certain operations become linear or polynomial in the number of qubits, rather than exponential.

In practice, QuIDDs are particularly effective for simulating quantum algorithms like Grover's search algorithm. In the author's QuIDD-based simulator QuIDDPro, they demonstrate the ability to simulate Grover's algorithm with runtime and memory complexities that are much closer to the ideal quantum computer, especially when compared to earlier simulation methods. The improved efficiency of QuIDDs allows for the simulation of quantum circuits involving a larger number of qubits than would be feasible with straightforward full state vector approaches.

For a quantum operator with an arbitrary number of qubits, their simulation relies on constructing a QuIDD for the expanded matrix, using repeated tensor products. However, they provide an optimisation to this for single qubit gates, removing this requirement.

In their evaluation, they present results for emulating Grover's database search algorithm in QuIDDPro, and compare it to simulator implementation based on qubit-wise multiplication, in terms of memory usage and run time. They show that the number of the nodes of the QuIDDs representing the algorithm matrix grows linearly in the number of qubits, resulting in linear growth in peak memory usage; compared to an exponential growth observed for the

qubit-wise multiplication based simulations. The runtime observed is still exponential in the number of qubits, but this is a result of the number of iterations of the Grover's oracle and diffusion operators which are required, per Boyer et al. [71].

The authors also discuss the Quantum Fourier Transform, arguing that it is not a suitable target for QuIDD-based simulations; a result of the QuIDD representing the QFT matrix growing exponentially with the number of qubits, and of the values in the state vector having an exponentially growing number of different values after application of the QFT.

QMDDs

The work by Wille et al.'s group on Quantum Multiple-Valued Decision Diagrams (QMDDs) ([72], [73], [74], [75]) carries on from their initial introduction in the late 2000s by Miller and Thornton et al. ([76], [77], [78]). In them, the properties and theorems on QMDDs are described.

A QMDD is a specialised data structure employed for the compact and efficient representation of quantum states and operations, which are inherently large and complex due to the exponential nature of quantum computation. QMDDs utilise a decomposition scheme that is particularly suited to the matrices and vectors arising in quantum computations, enabling the representation of quantum states and operations in a more compact form compared to traditional array-based methods.

Both QuIDDs and QMDDs exploit redundancies in quantum states and operations, but QMDDs go further by introducing additional mechanisms such as edge weights and normalisation. In QMDDs, common factors in quantum states are extracted and encoded as edge weights, allowing for even more compact representations compared to QuIDDs, which primarily rely on structural redundancy in the decision diagram.

The decomposition scheme used in QMDDs is more natural to the matrices and vectors arising in quantum computations, which allows for more efficient manipulation of these structures. For example, QMDDs can efficiently perform operations like Kronecker products and matrix-vector multiplications, which are more complex to implement in QuIDDs due to their less specialised decomposition approach.

In the context of quantum circuit simulation, QMDDs are used to represent the state of the quantum system and the operations applied to it. During simulation, quantum operations are performed by manipulating the QMDDs corresponding to the state vector and the unitary matrices. This manipulation involves operations such as matrix-vector multiplication, which is handled efficiently within the QMDD framework due to its natural decomposition scheme. The QMDD structure also supports efficient implementation of quantum measurements, which collapse quantum states into classical outcomes, by facilitating the calculation

of probabilities and updating the state vector accordingly.

QMDDs generally provide better performance in both time and memory compared to QuIDDs. For instance, while QuIDDs can represent certain quantum operations efficiently, they struggle with operations like the Quantum Fourier Transform (QFT) or complex oracles, where QMDDs maintain a linear or polynomial complexity. This improvement is partly due to the more advanced decomposition and redundancy-exploitation techniques used in QMDDs. QuIDDs have been shown to be particularly effective in simulating specific quantum algorithms, such as Grover's search, where the structure of the algorithm's operations matches well with the QuIDD representation. However, QMDDs offer a more general-purpose solution that can efficiently handle a broader range of quantum operations, including those that are challenging for QuIDDs.

2.2 State-of-the art CPU/GPU and cluster-based simulators

Quantum circuit simulation has been a critical tool for understanding quantum algorithms and emulating their behaviour on quantum hardware, especially as large-scale, fault-tolerant quantum computers are not yet widely available. Over the past two decades, significant advancements have been made in both CPU-based and GPU-based simulators, focusing on optimising performance, scalability, and hardware utilisation. Despite the exponential growth of quantum state spaces with increasing qubit numbers, modern classical hardware has pushed the boundaries of simulation capabilities for quantum circuits. Early simulators laid the groundwork for exploiting massively parallel architectures using state-vector representations, showcasing impressive parallel scaling on distributed systems. Since then, more advanced simulators have emerged, offering higher efficiency and flexibility across diverse computing environments.

This section surveys the state-of-the-art quantum circuit simulators, emphasising their architectural design and performance enhancements. The review covers early CPU and cluster-based simulators that use distributed memory architectures, as well as modern simulators optimised for multicore CPUs, GPUs, and hybrid systems. We explore how these simulators leverage vectorisation, parallelisation, and memory optimisations to simulate increasingly larger quantum circuits. It is important to note that the work done in this field is vast and that there exists a plethora of software-based quantum circuit simulators. The works chosen to survey here are a sample of these simulators which demonstrate the key optimisations and implementation methodologies.

While CPUs excel in handling sequential tasks and offer flexibility in algorithm design,

GPUs, with their massive parallel processing capabilities, can efficiently handle the highly parallelisable operations required for quantum simulations, such as matrix-vector multiplications. Cluster-based simulators provide an additional layer of scalability by distributing the simulation workload across multiple nodes, thereby pushing the qubit limit beyond that of standalone CPUs and GPUs.

Early works related to the development of massively parallel quantum computer simulators include DeRaedt et al.[79] and Tabakin and Julia-Diaz[80]. Both works involve highly optimised simulators based on the full state vector approach, implemented in Fortran using the MPI library to exploit parallelism on distributed-memory architectures. For a range of textbook examples, very good parallel scaling was observed for tests with up to 4096 processors documented in DeRaedt et al.[79]. The work on QCMPI reported by Tabakin and Julia-Diaz[80], was mainly motivated to facilitate numerical examination of not only how QC algorithms work, but also to include noise, decoherence, and attenuation effects and to evaluate the efficacy of error correction schemes.

Both works focus on evaluating quantum circuits, while earlier work by DeRaedt et al. [81] involved a Quantum Computer Emulator designed to simulate the physical realisation of the quantum computer and a graphical user interface to program and control the simulator.

Some more recent works utilise the Single Amplitude method to achieve far higher qubit counts when simulating on clusters (up to 200 qubits) including the work by Wang et al. [82] and Chen et al. [83].

2.2.1 CPU-based Simulators

Quantum circuit simulation on CPUs has been one of the foundational methods for testing and validating quantum algorithms. CPUs offer a well-understood, mature infrastructure for performing complex, sequential computations, making them a reliable platform for simulating small to moderate-sized quantum circuits. Their architecture allows for efficient handling of control flow and logic-intensive tasks, making them suitable for controlled and multi-controlled operations. However, as the qubit count grows, the computational demands quickly outpace the capabilities of even the most powerful CPUs. This section reviews two state-of-the-art CPU-based quantum simulators as illustrative examples.

Intel Quantum Simulator (formerly known as qHiPSTER)

The Intel Quantum Simulator (IQS) [58], formerly known as qHiPSTER [84], is a distributed implementation of a quantum simulator, built with MPI, that can simulate single-qubit gates and two-qubit controlled gates. Their MPI communication is based on the scheme defined in

[85] and they apply several optimisations to it; including vectorisation (SIMD), multithreading on the nodes, and cache blocking through gate fusion. We first present the earlier work by Smelyanskiy et al. [84], then the improvements presented by Guerreschi et al. in [58].

The original scheme is summed up here: Given 2^p nodes, to simulate n qubits, the 2^n state vector is divided such that each node holds $2^{n-p} = 2^m$ amplitudes. Qubits are indexed from 0 for the most significant qubit. For a target qubit k, where $k < m$, no inter-node communication is necessary as all required amplitudes reside within a single node. For $k \geq m$, the pairs of amplitudes are located on different nodes and MPI is used to facilitate communication. Each local state vector is partitioned to two and an additional 2^{m-1} memory buffer is allocated. When applying the gate, one nodes send its first half to the other, while the other node sends its second half, utilising the memory buffer. The gate is then applied on both nodes; the first node applying it on its first half and the received second half, and the second node applying it on its second half and the received first half. This means the first node updates the second half of the second node, while the second node updates the first half of the first node. The halves are then communicated back to their original node. Application of controlled gates is similar to this but involves more steps.

This scheme is improved upon by further dividing the communication into multiple steps, i.e., instead of reserving an additional 2^{m-1} amplitudes worth of memory, they reserve a smaller amount and perform more message passes during the gate application. For instance, with space for 2^{m-2} amplitudes in the buffer, one additional qubit can be simulated. They point out that as long as the data exchanged in each message is enough to saturate the network bandwidth, the overall run-time remains the same. In addition, the authors also apply gate fusion to extract maximum benefit from the CPU's Last Level Cache.

Using 1000 cores on the TACC supercomputer, Smelyanskiy et al. [84] were able to simulate the QFT for up to 40 qubits, consuming 32 TB of memory.

In [58], Guerreschi et al. present a new release of the simulator, rebranded as the Intel Quantum Simulator, and describe further optimisations and new features included. Importantly in this release, they extend the MPI implementation to allow multiple quantum circuits to be simulated in parallel. They present three operational modes of the simulator: the first mode uses all available HPC resources to simulate a single circuit, and hence is capable of simulating the highest possible number of qubits; the second mode divides the available resources into groups to be able to simulate several, smaller, quantum circuits; while the third mode utilises the functionality of the second mode to model noise and decoherence through a stochastic ensemble.

They demonstrate the first mode by running 42-qubit circuits on the SuperMUC-NG[3] system hosted at the Leibniz Supercomputing Center of the Bavarian Academy of Science (LRZ), and present strong and weak scaling analyses.

Relevance to FPGA Clusters This communication strategy can be relevant to FPGA-based implementations. As we discuss later in the Evaluation chapter, in order to be competitive with GPU and CPU-based simulators, it is necessary to move from a single-board system to FPGA clusters. In an FPGA cluster, each board would be analogous to an MPI node with such a communication scheme. Each FPGA would similarly hold a portion of the quantum state vector. The host would divide the state vector across the FPGAs, and the FPGAs would be responsible for applying quantum gates to the vector slices they manage. Just as in the MPI system, when a gate operates on a qubit target index such that the amplitude pairs would span multiple FPGAs, communication between the FPGAs becomes necessary. Communication between FPGAs can happen via a direct connection between them (e.g., over a high-speed interconnect like Ethernet, or specialised FPGA-to-FPGA links) or routed through the host. Similar to the MPI system, the FPGAs would need to exchange portions of their state vectors when applying gates that span amplitude pairs on different FPGAs. FPGAs could allocate local buffers for intermediate data storage and use pipelined transfers to manage data movement efficiently. Like the multi-step communication in MPI, FPGA communication could be broken into smaller chunks, reducing the need for large memory buffers and avoiding bandwidth saturation. FPGAs could transfer portions of the state vector incrementally and apply gates in steps, mimicking the strategy of using partial buffers in the MPI system.

QX Simulator

In [86], Khammassi et al. introduce QX, a universal quantum circuit simulator allowing users to specify their circuits in a QASM (Quantum Assembly) format similar to OpenQASM [87, 88]. QX takes advantage of vectorised complex arithmetic operations through modern processor instruction sets like SSE3 and AVX. These instruction sets support parallel processing of multiple data points within a single operation, enabling 2- and 4-way double precision arithmetic. Specifically, Fused Multiply-Add instructions allow combining multiplication and addition into a single operation, reducing the time spent on arithmetic operations. This improved QX's performance by 24% over the automatic vectorisation provided by the GNU compiler.

QX also utilises several gate-specific optimisations taking advantage of knowing the matrix structure of the supported quantum gates in advance to reduce the number of floating point operations required. These optimisations allowed for a reported 40% improvement in performance. QX employs thread-level parallelism through the XPU runtime parallel programming framework, allowing matrix-vector multiplications and state vector manipulations to be parallelised across available hardware cores automatically. Furthermore, they utilise a sparse-matrix representation to optimise memory usage; which is essential for working with

large quantum circuits.

The paper benchmarks QX against other simulators, demonstrating superior performance in simulating complex quantum algorithms like Grover's algorithm, Quantum Fourier Transform, and entanglement circuits. They show QX to achieve between $14 - 94\times$ speedup over Microsoft's LIQUi$| >$ simulator [89].

2.2.2 GPU-based Simulators

GPUs have emerged as a powerful alternative to CPUs in quantum circuit simulation due to their ability to perform thousands of parallel computations simultaneously. Quantum circuit simulations, which often require repetitive and independent operations such as applying quantum gates to large state vectors, can be highly optimised for GPU architectures. In this section, we discuss two example simulators that illustrate how GPU-based simulators make use of this parallelism to speed up quantum simulations, particularly for circuits with a high number of qubits.

QuEST

In [57], Jones et al. introduce QuEST, a versatile and high-performance quantum circuit simulator. QuEST is designed to efficiently simulate quantum circuits across various computing architectures, ranging from personal laptops to large supercomputers. This versatility is achieved through a series of optimisations that allow QuEST to take full advantage of the available hardware, whether it is a multi-core CPU, a distributed computing system, or a GPU-accelerated environment.

The authors demonstrate a simulator that can target single-threaded, multithreaded, and distributed architectures, and is also capable of GPU acceleration. Their GPU-accelerated distributed architecture simulated random circuits with 38 qubits over 2048 nodes with 24 cores/node. Like previously-described distributed architecture implementations, QuEST partitions the state vector equally between processes, minimising communication overhead during gate operations by using a pairwise process communication strategy. This approach ensures that the system efficiently scales for large qubit simulations on both multicore and distributed architectures, providing strong and weak scaling results.

QuEST employs NVIDIA's CUDA (Compute Unified Device Architecture) to parallelise the simulation tasks across thousands of GPU cores. CUDA enables efficient execution of parallel code, allowing significant speedup compared to traditional CPU-based computations. GPUs operate on the SIMT model, where a single instruction is executed across multiple threads simultaneously. QuEST utilises this model to perform the same quantum gate operation across many qubit states in parallel.

QCGPU

In [90], Kelly introduces an OpenCL-based quantum computer simulator designed to accelerate circuit simulations on GPUs. The tool's simulation technique focuses on the state-vector representation and employs the QWM method (described above in Section 2.1.1) to optimise gate application. The tool was shown to be 150 times faster than Qiskit [91] and 8 times faster than ProjectQ [92] for simulations up to 24 qubits.

A key contribution of this work is the state vector indexing methodology which facilitates the parallelisation of the QWM method through NDRange kernels. Kelly introduces the `nth_cleared(n,t)` function, shown in Listing 2.1, which returns the n-th integer whose t-th bit is zero using bitwise operations, where n in this case is an iteration index and t is a quantum gate's target qubit index in the quantum register. This is a key requirement for converting an NDRange global work item index to the corresponding indices of the state vector in the memory of the system. This is described further in Chapter 3.

```
1  static int nth_cleared(int n, int target)
2  {
3     int mask = (1 << target) - 1;
4     int not_mask = ~mask;
5
6     return (n & mask) | ((n & not_mask) << 1);
7  }
```

Listing 2.1: nth_cleared function introduced by Kelly [90].

2.3 FPGA Acceleration of Scientific Computing

As scientific computing problems grow in complexity and demand increasing computational power, Field-Programmable Gate Arrays have gained significant attention as a powerful tool for accelerating a wide range of applications. Unlike CPUs and GPUs, which have fixed architectures, FPGAs offer the flexibility of reconfigurable hardware, allowing for the customisation of data paths and parallel execution tailored to the specific requirements of scientific workloads. This adaptability makes FPGAs highly suitable for applications that involve intensive data processing, such as signal processing, molecular dynamics, and fluid dynamics simulations, where both performance and power efficiency are critical.

The use of FPGAs in scientific computing is driven by their ability to execute fine-grained parallelism, which enables tasks like matrix operations, FFT computations, and Monte Carlo simulations to be carried out much faster than on traditional processors. Additionally, their low power consumption and the ability to minimise overhead through customised hardware

pipelines make them an attractive option for high-performance computing centers focused on energy efficiency. Despite these advantages, the challenges of designing efficient FPGA-based solutions, such as the need for specialised hardware knowledge and the limitations of current design tools, continue to limit widespread adoption.

This section will explore the application of FPGAs in scientific computing, focusing on key areas where FPGAs have demonstrated substantial performance improvements over traditional architectures. We will review notable examples of FPGA-accelerated scientific workloads, discussing their reported performance and energy efficiency, and examine the challenges faced when integrating FPGAs into existing scientific computing frameworks. Through this analysis, we aim to highlight the growing role of FPGAs in solving computationally intensive scientific problems and the potential they hold for future developments in the field.

2.3.1 Overview

The book by Vanderbauwhede and Benkrid [93] provides an extensive overview of the state-of-the-art uses and applications of Field-Programmable Gate Arrays for High-Performance Computing.

Originally designed for hardware emulation and teaching computer design, FPGAs became integral to communication technologies and consumer electronics, though they were slow to penetrate HPC due to limitations compared to custom hardware in terms of power, speed, and area. However, after the early 2000s, with the slowing of Moore's Law in terms of processor speed scaling, FPGAs have gained attention due to their parallel processing flexibility, lower power consumption, and capability to emulate specialised computing machines. Three primary reasons are highlighted for FPGA's growing importance in HPC:

- The difficulty in scaling multicore CPU implementations for HPC applications, as the programming models that were developed up until that point were oriented towards sequential processing as opposed to the parallel processing which is typically required in such applications.
- Increasing transistor densities over the decade since Moore's law for frequency scaling reached an end; allowing for very large FPGA configurations.
- Flexibility of FPGAs in enabling designers to optimise computing systems for specific applications, making them suitable for large-scale, important HPC applications.

In the preface, the editors focus on how HPC had reached a technological turning point. By the mid-2000s, seamless exponential growth in computing power through increased clock

frequency had ended. Multicore processors provided a solution, but without the necessary software recoding to exploit parallelism, performance gains would remain only theoretical. The growing importance of parallel computing opened opportunities for niche parallel computing technologies, such as FPGAs, which provide custom hardware performance and low power consumption with the flexibility of general-purpose processors. This shift in hardware demands fostered the development of High-Performance Reconfigurable Computing (HPRC), a new field centered on the use of FPGAs for HPC.

This work is structured as three parts: the first being a comprehensive survey of HPRC applications including financial computing, bioinformatics, molecular dynamics, graph processing and search implemented on FPGAs; the second covering architectural developments in HPRC; and the final part presenting tools and methodologies in HPRC, which are necessary for making FPGAs an accessible economic option for HPC applications. The future potential of FPGAs in HPC is emphasised, especially in comparison to other technologies like GPUs and general-purpose processors, positioning FPGAs as a key technology for the future of computing.

The thesis by Zohouri [94] explores the potential of FPGAs as alternatives to GPUs for High-Performance Computing applications, particularly in the context of the impending limitations of Moore's Law and the need for more power-efficient computing solutions. The research focuses on the usability, productivity, and performance of FPGAs in various HPC workloads, using Intel's FPGA SDK for OpenCL to enable easier programming by software developers.

The study begins by evaluating the performance and power efficiency of FPGAs using a subset of the Rodinia benchmark suite, optimised for Intel FPGAs. It demonstrates that while direct ports of CPU and GPU kernels to FPGAs often result in poor performance, FPGA-specific optimisations can yield significant performance improvements—up to two orders of magnitude in some cases. The results show that FPGAs can outperform CPUs in all cases and are competitive with GPUs in most, with FPGAs having a clear advantage in power efficiency, achieving up to 16.7 times higher efficiency compared to CPUs and 5.6 times compared to GPUs.

Building on these findings, the thesis further explores the suitability of FPGAs for stencil computation, a critical pattern in HPC. Zohouri designs and implements an OpenCL-based template kernel for accelerating 2D and 3D stencils on FPGAs, incorporating various optimisations to maximise performance. The study concludes that FPGAs not only offer superior performance to CPUs and GPUs in 2D stencil computation but also maintain competitive performance in 3D stencils while providing significantly better power efficiency.

The contributions of this thesis are manifold, providing insights into optimising HPC applications on FPGAs using High-Level Synthesis and establishing FPGAs as viable and efficient

alternatives to traditional processors for certain HPC workloads. The research also offers valuable guidelines for optimising FPGA-based applications, which could be applicable beyond HPC to other domains.

2.3.2 Search

In [95], Vanderbauwhede et al. explore the development of a high-efficiency information retrieval system using Field-Programmable Gate Arrays. The motivation for this research is the growing need to reduce energy consumption in data centers, where the costs of power and cooling have become a dominant concern [96]. The authors present a novel use of FPGAs to accelerate document filtering tasks. They target the filtering of incoming documents against user-defined topic profiles, a process often required in applications like spam detection, patent monitoring, and news tracking. The research demonstrates that FPGA acceleration can achieve up to 20x speed improvements compared to CPU-based implementations. The key computational task — matching documents to profiles based on a probabilistic language model — is offloaded to the FPGA, which significantly reduces processing time and energy consumption. The document filtering application is implemented on an SGI Altix system with two Xilinx Virtex-4 FPGAs. The filtering algorithm, based on a relevance-based language model, is implemented using Mitrion-C, a high-level language designed for FPGA programming. The algorithm scores documents by comparing the terms in the document with those in the profile and assigning weights.

The paper concludes that FPGA-based systems offer a promising solution for accelerating information retrieval tasks, providing significant performance gains while using less power than traditional CPU-based systems. The authors also highlight future directions for improving performance and scaling the system for larger data centers.

The paper by Putnam et al. [97] presents an innovative approach to enhancing the performance of large-scale datacenter services through the deployment of a reconfigurable fabric, referred to as "Catapult," which leverages FPGAs. The paper addresses the growing need for high computational capabilities, flexibility, and power efficiency in datacenter environments, where traditional server designs are increasingly limited by power and performance constraints. By embedding FPGAs into servers and connecting them via a high-speed network, the authors propose a scalable and flexible architecture that significantly accelerates key datacenter workloads, particularly focusing on Microsoft Bing's search engine.

The Catapult fabric integrates medium-sized FPGAs into a half-rack of 48 servers, with each FPGA connected via PCIe and organised into a 6x8 2-D torus network. This setup allows the FPGAs to be composed into larger virtualised areas that can efficiently implement complex functions that exceed the capacity of a single FPGA. The authors highlight the

flexibility of this architecture, which can dynamically allocate FPGA resources based on the computational demands of the workload, thereby maximising resource utilisation and reducing power consumption.

One of the critical challenges addressed in the paper is the resilience and robustness of the FPGA fabric in a datacenter environment. The authors detail the engineering solutions implemented to ensure that the system can tolerate hardware failures, reboots, and updates to the ranking algorithms used in Bing's search engine. The failure handling protocol developed for the Catapult fabric can reconfigure FPGAs and remap services in response to failures, ensuring continuous operation and minimal disruption to the datacenter services.

The evaluation of the Catapult fabric is demonstrated through its application to the Bing search engine, where it offloads a significant portion of the document ranking process to the FPGAs. The results show a 95% improvement in throughput per server under high load conditions, with a 29% reduction in tail latency when maintaining equivalent throughput compared to a software-only implementation. This substantial performance gain highlights the potential of FPGAs to accelerate large-scale services while maintaining or even reducing latency.

In conclusion, this paper demonstrates the feasibility and benefits of deploying a reconfigurable FPGA fabric in datacenter environments. The Catapult fabric not only improves the performance and efficiency of critical workloads but also provides a flexible and scalable solution that can adapt to the rapidly changing demands of datacenter services. This research lays the groundwork for broader adoption of FPGA-based accelerators in datacenters, offering a promising path forward as traditional server performance improvements slow down.

2.3.3 Graph Computing

In [98], Melikoglu et al. introduce a high-throughput accelerator designed specifically for Binary Search Tree (BST) operations on FPGAs. The research addresses the growing need for efficient hardware accelerators capable of handling the computational demands of modern applications like databases, machine learning, and file systems, where BSTs play a critical role. Traditional implementations of BSTs on FPGAs have not fully exploited the inherent parallelism and pipelining capabilities of these devices. To overcome these limitations, the authors propose a novel architecture that maximises the use of FPGA BRAMs and introduces several innovative techniques to improve the performance of BST operations, particularly the Lookup operation.

The core of the proposed solution lies in its ability to parallelise and pipeline BST operations by efficiently partitioning the tree across the available BRAMs. The authors explore multiple partitioning strategies, including horizontal, duplicated, and hybrid (horizontal-vertical)

partitioning. One of the key contributions of this work is the introduction of a buffering mechanism to reduce stalling during search operations. Stalling occurs when multiple search requests target the same BRAM partition, causing delays. The authors propose two buffering techniques—direct mapping and queue mapping—to address this issue. Direct mapping assigns search requests to specific buffer slots, which can lead to stalling if the slots are occupied. Queue mapping, however, dynamically allocates slots based on availability, reducing stalling and improving overall throughput.

The experimental evaluation of the proposed accelerator was conducted on the Xilinx Virtex-7 VC709 platform. The results demonstrated a significant improvement in throughput, with the most optimised configuration achieving an 8X increase compared to a baseline fully-pipelined FPGA-based accelerator. The flexibility of the design is another notable aspect, as it allows for reconfiguration of system parameters such as buffer sizes and the number of tree partitions at compile time. This flexibility enables the design to be tailored for specific performance and resource utilisation requirements, making it suitable for a wide range of high-performance computing applications.

2.3.4 Linear Algebra Applications

In [99], the authors explore the potential of FPGAs in scientific and high-performance computing. The paper discusses the limitations of traditional computing paradigms, including CPUs, in handling computationally intensive scientific workloads, and propose FPGAs as a promising alternative due to their ability to exploit parallelism and offer high performance with lower power consumption. FPGAs are highlighted for their ability to perform double-precision floating-point operations, previously achievable only with custom ASICs, while offering flexibility and lower power consumption. The paper details the implementation of Basic Linear Algebra Subroutine (BLAS) routines on the SRC MAPstation, emphasizing the sustainable performance of FPGAs for large problem sizes. The DGEMM and SGEMM routines are used as case studies, demonstrating how FPGAs can maintain high performance even with minimal data reuse, which typically limits CPU performance. The authors discuss their efforts to develop a library of common scientific kernels optimised for FPGA implementation. This library aims to accelerate a wide range of scientific applications at ORNL, including molecular dynamics, climate modeling, and bioinformatics. The paper concludes by addressing the challenges of FPGA programming, particularly the need for high-level language tools to make FPGA development more accessible to scientists. In summary, the paper showcases the viability of FPGAs in scientific computing, particularly in overcoming the limitations of traditional CPUs in handling complex, data-intensive tasks.

In [100], the authors describe three types of solvers (dense linear equation solvers, sparse iterative linear equation solvers, and dense least square solvers) developed for the LAPACKrc

library, a family of FPGA-based linear algebra solvers designed to achieve significant computational speedups. Each solver is designed to exploit the parallelism offered by FPGAs, achieving speedups between 40x to 150x over conventional CPU implementations. Mixed-precision arithmetic and high-speed communication between FPGA cores further enhance performance. The results demonstrate that LAPACKrc, and the use of FPGAs, can significantly accelerate complex numerical tasks in HPC.

[101] highlights the inefficiency of existing CPU and GPU-based libraries for SpMxV, Sparse Matrix-Vector Multiplication, particularly due to the mismatch between memory access patterns of sparse matrices and traditional computing architectures. The paper discusses the benefits of using FPGAs, such as high floating-point performance, abundant on-chip memory, and flexible architecture, which allow for better adaptation to different problems compared to CPUs and GPUs. A scalable FPGA-based SpMxV kernel is proposed that enhances computational efficiency by transforming irregular memory access patterns into regular, sequential ones. This architecture utilises a compressed sparse column (CSC) format to optimise memory bandwidth usage. The FPGA-based kernel demonstrated significantly higher computational efficiency compared to CPUs and GPUs. Benchmarking on a Virtex-5 SX95T FPGA achieved a peak computational efficiency of 99.8%, with an average performance improvement of over 50x compared to Intel Core i7 processors and over 300x compared to NVIDIA GPUs. The proposed FPGA kernel also demonstrated a 38-50x improvement in energy efficiency over traditional CPU and GPU implementations, making it highly suitable for energy-constrained applications. The paper compares the proposed architecture with other FPGA-based SpMxV implementations, showing superior performance, particularly for matrices with varying sparsity. The paper concludes that the proposed FPGA-based SpMxV kernel not only outperforms traditional CPU and GPU implementations but also offers significant energy efficiency improvements, making it a viable solution for high-performance scientific computing.

[102] explored Gaxpy (Level 2 BLAS, specifically matrix-vector multiplication) and Level 1 BLAS implementations on an FPGA, and study how they compare against similar solutions on CPUs and GPUs. They demonstrate vector memory and matrix memory implementations on the FPGA which gave them efficient concurrent read/writes to and from the memory on the FPGA. The study highlights that FPGAs, CPUs, and GPUs all have advantages and trade-offs depending on the specific task. The researchers implemented custom solutions for BLAS Level 1 and Level 2 on the FPGA using the BEE3 platform [103] and standard libraries for the CPU (Intel Math Kernel Library) and GPU (Nvidia CUBLAS). Their experiments evaluated these platforms in terms of execution time, power consumption, and energy efficiency. The CPU's parallel MKL implementation outperformed both the FPGA and GPU in terms of raw execution speed for BLAS Level 2 (matrix-vector multiplication). However, the FPGA implementation showed competitive performance, achieving 3.1 GFLOPS.

The FPGA proved to be highly energy efficient, offering between 2.7x to 293x better energy efficiency than the CPU and GPU platforms, making FPGAs a compelling choice for energy-constrained environments.

FBLAS

DeMatteis et al. [104] introduced FBLAS, an open-source implementation of the Basic Linear Algebra Subroutines (BLAS) optimised for FPGAs using High-Level Synthesis tools. This work addresses the challenges of integrating spatial computing architectures like FPGAs into high-performance computing environments, where traditional load-store architectures face limitations in terms of energy efficiency and computational performance due to their inherent overheads in data movement and control logic.

The primary goal of FBLAS is to provide a reusable and customisable library of linear algebra routines that can be efficiently executed on FPGAs. By enabling these routines to be composed and integrated with existing software and hardware codes, FBLAS significantly lowers the barriers to entry for developers looking to leverage FPGAs for numerical computations. The library is designed to exploit the parallelism and streaming capabilities of modern FPGAs, allowing for the efficient use of on-chip resources and minimising the need for costly off-chip communication.

One of the significant contributions of the FBLAS library is its support for streaming on-chip communications, which allows different hardware modules to communicate directly through the FPGA fabric without involving off-chip memory. This feature is particularly beneficial for I/O-bound computations, where reducing the volume of off-chip communication can lead to substantial performance gains. The authors also offer guidelines for composing modules that can communicate efficiently within the FPGA, further enhancing the library's usability and effectiveness in HPC applications.

2.4 FPGA-based simulators

FPGAs have emerged as a promising alternative to traditional CPU and GPU-based quantum simulators, particularly for their ability to offer customisable parallelism and energy-efficient computation. Unlike fixed-architecture processors, FPGAs are reconfigurable, allowing designers to tailor the hardware specifically to the needs of quantum circuit simulation. This adaptability enables FPGAs to perform specific tasks, such as quantum gate applications and state vector updates, with greater efficiency by exploiting fine-grained parallelism and minimising unnecessary computation.

One of the key advantages of FPGAs is their low-power consumption, which makes them highly suitable for scaling quantum circuit simulations, particularly in power-constrained environments like HPC centers. Additionally, FPGAs offer the ability to implement advanced hardware-level optimisations such as pipelining, buffering, and control over gate fusion caching behaviour, further enhancing their performance. However, FPGAs also face several challenges, including resource limitations, scalability issues, and the complexity of designing and optimising hardware.

This section will explore the current state of FPGA-based quantum simulators, reviewing notable simulation frameworks that utilise these devices.

2.4.1 Quantum Circuit Emulation on FPGAs

The earliest work in simulating quantum circuits on FPGAs dates back to Khalid et al. [105]. This consists of a compiler which produces VHDL, a hardware description language, code that emulates the quantum circuit on the FPGA. It emulates quantum parallelism by constructing parallel data paths for the state vector amplitudes representing the qubits, i.e. implementing the whole quantum circuit in the FPGA fabric. State vector amplitudes are implemented by fixed point numbers to keep the size of the circuits manageable. Fixed point was also chosen since the probability amplitudes can only have a decimal part of 0 or 1. This approach emulates a full quantum circuit on the FPGA, requiring a full synthesis when changing the circuit.

The approach used in Aminian et al. [106] divides quantum circuit simulation into two circuit types based on gates used in the circuit. Like in the previous work, fixed-point representation is chosen for the complex probability amplitudes of the state vector. For circuits involving only X, Y, Z, and CNOT, they reduce the Logic Cell (LC) usage required for each type of gate to a handful (X: 2, Y: 6, Z: 2, CNOT: 4). They do this by adding extra information bits (basis, complexity, sign) and simply operating on them when applying any of these gates (however there is more basis bits in the case of CNOT). The second group is H, PS (phase shift), and CR (controlled rotation), which are implemented directly as adders and multipliers, requiring resources which increase with the number of mantissa bits. For circuits involving both groups of gates, they apply a different simulation policy than when just XYZC gates are used.

Pilch and Dlugopolski[107] proposed, designed and implemented an easily scalable universal quantum computer emulator, focused on reflecting natural quantum processes in hardware, while maintaining the time complexity of quantum algorithms. The underlying idea is to move the weight of complexity from time to hardware resources by using the inherent parallelism of FPGAs. As proof-of-concept, the authors created a hardware-software system capable of running and correctly interpreting results of the Deutsch quantum algorithm.

So far, only small circuits were considered, e.g. the Deutsch algorithm was emulated for a 2-qubit quantum computer.

2.4.2 Single-Amplitude Simulation

Frank et al. [108] describe an algorithm for simulating a quantum circuit based on Feynman's path integral formalism. In computational complexity theory, BQP (problems solvable in polynomial time on quantum computers with bounded errors) is a subset of PSPACE (problems solvable in polynomial classical memory) [109]. This forms the theoretical basis of their simulation algorithm, SEQCsim, which works by going through the gates one at a time, while only keeping track of the amplitude of one computational basis state, selected pseudo-randomly at each step. They prototype this algorithm on a CPU and compare its memory and CPU time use to another tool, QCAD. Both tools grow exponentially in CPU time (with QCAD performing better than SEQCsim at higher number of qubits, likely due to trading computation for memory). They provide some initial design concepts for an FPGA-based SEQCsim, and they estimate they would obtain about a 50× speedup. This would fall under the Single-Amplitude Computer approach described above.

2.4.3 Reusable architectures

Conceicao and Reis [110] address the issue of re-synthesis present in prior works. They present a reusable architecture for which synthesis is only rerun when the number of qubits or mantissa bits of the fixed point representation is required to be changed. In their design, a control unit holds an address of a an instruction in some instruction memory (list of gates) and a quantum ALU (Arithmetic Logic Unit) is fed a gate operational code (opcode), target qubit, and two control qubits at each gate and then communicates with a quantum register to perform the gate. They report their LC usage for a number of algorithms and benchmarks. In terms of LC usage, they are outperformed by [106], which they point out, but also observe that the ratio between their average usage of logic cells decreases in comparison when increasing the number of qubits from 3 to 8, leading them to believe their system would be better when scaled up.

Lee et al.[111] developed a serial-parallel architecture-based FPGA emulation framework for quantum computing and, for small numbers of qubits (up to 7), demonstrated significant speed-ups relative to CPU-based emulations. The framework proposed by the authors aims to address resource and scalability issues in FPGA-based quantum computing emulation. Two key quantum algorithms, the QFT and Grover's search algorithm, serve as case studies because they are core components in many quantum algorithms. The paper reports experiments

comparing the proposed architecture against prior pipeline-based FPGA designs. A linear reduction in resource utilisation is achieved, which enhances the scalability of the framework. Various fixed-point formats were tested to optimise resource usage and minimise precision errors.

2.4.4 High qubit count architectures

The works discussed so far ([105] [106] [111] [110] [107]) demonstrate the promise for emulating quantum circuits on FPGAs, albeit for low number of emulated qubits. Mahmud and El-Araby[112] focus on scalability, presenting two architectures for emulation. The first is a CMAC (complex multiply-and-accumulate) unit-based system, which for a given quantum circuit, relies on having the full algorithm matrix precomputed. An optimisation to this is to have a kernel which dynamically generates the values of the algorithm matrix, massively reducing the memory requirement. Using this architecture, Mahmud et al. [113] emulated a 20-qubit QFT, an increase in qubits compared to previous works in FPGA emulation of quantum circuits. This required the creation of a custom hardware architecture for generating the values of the QFT matrix. The second recognises that there may be algorithms which have sparse algorithm matrices which may not be suited for the CMAC-based architecture. Instead, this architecture requires a custom acceleration kernel to be developed from the quantum algorithm, which is then applied to the input state vector. Using this architecture, they emulated a 30-qubit Quantum Haar Transform. This required the extraction of a simplified kernel from the mathematical description of the QHT rather than from its quantum circuit description. This is a considerably higher number of qubits than those achieved in previous works. However, no method of automating the generation of the kernels from a quantum circuit model description is discussed. The authors extend this method to Grover's database search algorithm in [114].

Khalid et al.[115] describe a proposal for a resource-efficient FPGA-based abstraction of quantum circuits. A non-programmable embedded system capable of storing, measuring, and introducing a phase shift in qubits is implemented. The proposed single-input single-output architecture implements single-input gates with corresponding memory and measurement blocks. A fixed-point quantum gate representation is used, using 8 bits (2-bit integer, and 6-bit fraction). By increasing the number of bits used for qubit representation, the quantised superposition states of the of qubit increase, leading to enhanced accuracy of the emulation results. The main objective of the proposed abstraction was to provide an FPGA-based platform as the fundamental sub-block for the design of quantum circuits. The quantum key distribution algorithm BB84 was implemented using the proposed platform as a proof-of-concept. The proposed design exhibits two principal properties of quantum computing, i.e. parallelism and probabilistic measurement.

The concept of using exponentially-increasing resources (with problem size) on an FPGA to maintain the exponential time-complexity gain of quantum algorithms relative to their classical counterparts was also investigated by Bonny and Haq[116] who implemented the quantum k-means clustering algorithm on an FPGA emulator. Clustering is a technique involving the classification of unlabeled data into a number of categories, and is widely used in machine learning and data mining. The main computational work in the k-means clustering algorithm is the computation of the distance between points. Bonny and Haq model the points as n-dimensional vectors on the Bloch sphere and then use the inner product as an estimation of the distance between two vectors. In the example implementation 2D data was used. The work presented forms an example of a quantum-inspired algorithm allowing (exponential) speed-up relative to classical algorithms even when running on classical hardware, by trading time-complexity and resource use on the FPGA. Clearly, the rapid (exponential) increase of logic-gate resources with problems size limits this approach to relatively small problems. For quantum circuit emulation, this means that only a few qubits can be used when trying to main exponential time-complexity gain. This work follows on from previous works into such quantum-inspired algorithms including work on modified Grover's search algorithm [117] as well as quantum-inspired evolutionary algorithms for optimisation problems[118]. More recently, Fujitsu's quantum-annealing inspired optimiser as described by Aramon et al.[119] uses extensive hardware acceleration techniques to achieve time-complexity improvements.

High-Bandwidth Memory (HBM) is a type of memory designed to offer extremely high bandwidth while consuming less power compared to traditional memory technologies like DDR4. It is specifically engineered for applications requiring large amounts of data to be processed quickly, such as high-performance computing, artificial intelligence, and graphics processing units. It can provide bandwidths up to 410 GB/s per stack, which is significantly higher than traditional memory types like DDR4, by integrating multiple layers of DRAM in a vertical stack, connected to the processor through a silicon interposer. Each stack can have up to 8 layers. In [120], Waidyasooriya et al. present a specialised architecture to simulate the QFT using FPGAs equipped with HBM. The HBM memory system in the FPGA consists of multiple independent memory channels, each with its own address space. In the case of the Stratix 10 MX FPGA used in the study, the board contains 16 HBM memories, each with two channels, providing 32 pseudo-independent memory banks that can be accessed simultaneously. The authors describe a state vector banking strategy that efficiently allocates and accesses data across the multiple independent HBM memory banks, ensuring parallel memory access and minimising unnecessary data movement, allowing for scalable quantum circuit simulation both within a single FPGA and across multiple FPGAs. Their architecture is designed to be extendable to multiple FPGAs, where each FPGA manages a portion of the state vector. The state vector is distributed evenly across HBM modules. The system

supports inter-FPGA communication using high-speed links, achieving a significant speed-up when multiple FPGAs are used. They could run a 30-qubit QFT circuit across 2 FPGAs, achieving a $23.6 - 24.5\times$ speedup compared to an optimised 24-core CPU implementation. This is the largest FPGA-based QFT emulation reported to date.

2.4.5 Summary

This summary of works involving hardware acceleration of quantum computing simulation shows that there is a growing interest in simulating quantum circuits on FPGAs and their results show that there can be a considerable computational advantage to using an FPGA to simulate a quantum computer. However, most of the research so far only considered circuits with few (< 10) qubits and also did not consider circuit transformation techniques for reducing the number of qubits. The work which demonstrated the most promise for scaling to a high number of qubits recently is Mahmud et al.'s [113] [114] specialised kernel-based approach, using which they ran a 30-qubit QHT, and a 32-qubit Grover's search circuits on an FPGA. While these approaches reach the highest number of qubits simulated on an FPGA in the literature, of which we are aware, they is not very easily reusable. Using their more circuit-independent CMAC-based approach, they were able to simulate a 20-qubit QFT circuit. Our work is in line with their more generic approach because, as we discuss in the next section, reusability is one of our primary goals. Waidyasooriya et al. [120] also presented an architecture which reached a high number of qubits, simulating a 30-qubit QFT across two FPGAs with HBM.

2.5 Conclusion

In this chapter, we reviewed the various methods which can be used for Quantum Circuit Simulation such as full-state vector simulation, tensor networks, and decision diagrams. Each technique has its own strengths and limitations in handling certain types of circuits. Full-state vector simulations are the most common, but methods like tensor networks and decision diagrams offer advantages in certain cases.

The state-of-the-art CPU and GPU-based simulators have pushed the boundaries of quantum circuit simulation, supporting simulations with more qubits and optimising performance through parallelisation. Simulators like Intel Quantum Simulator [58] and QuEST [57] have demonstrated scalability with distributed systems, utilising massive parallelism on CPU and GPU clusters to accelerate simulation of circuits with large qubit counts.

We demonstrated how FPGAs offer a flexible and energy-efficient alternative for accelerating scientific computing tasks, including quantum circuit simulation. They enable fine-grained

parallelism and allow for custom hardware pipelines tailored to specific tasks, offering significant performance and power efficiency improvements over traditional CPU/GPU architectures for some scientific workloads.

The literature on FPGA-based quantum circuit simulation reveals a growing interest in utilising FPGA hardware to accelerate the simulation of quantum algorithms. Several approaches have been developed to efficiently simulate quantum circuits using FPGA technology. Early works such as those by Khalid et al. [105] and Aminian et al. [106] laid the foundation by utilising fixed-point arithmetic to represent quantum states, which enables efficient emulation of quantum circuits. These methods demonstrated significant speed-ups compared to classical CPU-based simulators, but their scalability was limited to small qubit circuits due to resource constraints on FPGAs.

Further advancements in single-amplitude simulation and reusable FPGA architectures addressed some of the limitations by optimising the resource usage and proposing more flexible architectures. For example, Frank et al. [108] introduced a simulation method based on Feynman's path integral formalism that could be accelerated using FPGAs, showing promising speed-ups while maintaining manageable resource requirements. Similarly, reusable architectures like those proposed by Conceicao and Reis [110] introduced quantum ALUs, enhancing the scalability and reusability of FPGA-based quantum circuit simulators.

The exploration of high-qubit-count architectures, particularly by Mahmud et al. [112, 113, 121, 114] and Waidyasooriya et al. [120], represents a significant leap in the field, enabling the emulation of up to 32 qubits with specialised hardware architectures. This work showcased the potential of FPGA systems to handle more complex quantum algorithms and simulations. However, for high-qubit counts, their described methods are quantum circuit specific and not demonstrably generalisable to any quantum circuit.

In conclusion, FPGA-based quantum circuit simulators provide a promising solution for accelerating quantum simulations by leveraging the parallelism and flexibility of FPGA hardware. While significant progress has been made in the emulation of small to intermediate qubit circuits, further research is required to address the scalability challenges associated with large qubit counts and to develop more generalisable simulation frameworks.

Based on our learnings from this survey of works, our primary goals for developing a quantum circuit simulation platform based on FPGAs are:

- **Universality:** The simulator should support a set of gates which enable universal quantum computation to be simulated.
- **Scalability:** The simulator should show improved performance with scaling compute resources.

- **Reusability:** The simulator should not need to be re-compiled to simulate different quantum circuits and different qubit counts.

In the next chapter, we discuss different designs and optimisations for Full State Vector Quantum Circuit Simulation architectures.

Chapter 3

Full State Vector Quantum Circuit Simulation

Full state vector simulation, also commonly referred to as Schrödinger-style simulation, evaluates the value of the state vector after every time-step (after every gate in the simplest case with no optimisations applied). The entire state vector is stored in memory for the duration of the circuit simulation. To process a quantum gate, pairs of amplitudes are read into the processor's registers from the memory, updated and then written back. Without control qubits, every element of the state vector has to be updated. The access pattern of the state vector for a particular gate depends on the gate's target qubit's index in the quantum register. Every added control halves the amount of memory which needs to be updated.

In this chapter, different implementations of full state vector quantum circuit simulation are described, along with optimisations developed to benefit FPGAs.

Note about endianness and addressing (or indexing) qubits. In this thesis, unless otherwise stated, we always assume to operate with little-endian addressing. In addition, unless stated otherwise, by default, quantum circuit diagrams presented in this work will have qubits with a smaller address (or index) at the top of the diagram, with qubits with a larger address going to the bottom.

3.1 Direct Iteration Processing

As stated in the introduction to this chapter, a Schrödinger-style simulator maintains the entire state vector representing the quantum system in memory and updates it every time step. Consider the case where no optimisations are applied and every time step processes a single gate. To process a single gate, the entire memory space containing the state vector has

to be updated. The access pattern of the memory depends on the index of the target qubit in the quantum register (referred to as the target going forward). For a gate represented as a 2×2 matrix, the state is updated in pairs. The stride between the pair elements grows exponentially with the target, t, as: 2^t; we define this as the **element pair stride**.

For the simplest case, referred to as Direct Iteration Processing (DIP) here, it is sufficient to realise that for a quantum register consisting of n qubits, there will be 2^{n-1} 2×2 matrix multiplication iterations, where each iteration processes a unique pair of amplitudes from the state vector stored in memory. Every iteration index needs to be mapped to the index of the first pair element in the memory space. The second pair element is then found by adding 2^t to the index of the first; this parameter is called the stride corresponding to the target t. A 2-dimensional vector is then formed from the pair and multiplied by the gate matrix to evolve the state by the quantum gate. The updated vector then forms the updated element pair and is written back to memory at the same indices. An iteration refers to a full flow of computing the indices of a pair of amplitudes, reading the amplitudes from memory, updating them through multiplication by a 2×2 matrix, and writing them back to memory at the same indices. This flow is shown in Listing 3.2.

A simple function that uses bit-wise operations is presented in [90] to map the iteration indices $\{0, 1, 2, ..., 2^{n-1}\}$ to their respective pair of amplitudes which need to be processed based on the target qubit t. This function is presented in Listing 3.1; it is named `ithCleared` as it returns the i-th integer whose t-th bit is zero, where i is the iteration index and t is the target qubit of the gate being processed. This is exactly the index of the first element of the pair of amplitudes required to process at a given iteration i. The index of the second element in the pair is then the one with the same bit pattern except the t-th bit is one; this can be found simply by anding (or adding) the first index with the target stride, 2^t.

```
1 int ithCleared(int i, int t) {
2  int mask = (1 << t) - 1;
3  int notMask = ~mask;
4
5  return (i & mask) | ((i & notMask) << 1);
6 }
```

Listing 3.1: ithCleared Function for mapping iteration indices to amplitude indices.

This function starts by computing a target bit mask; `(1 << t)` gives a binary number where the bit at position t is 1, and all the lower and upper bits are 0. Subtracting 1 gives us a mask where the lower t bits are set to 1, and all higher bits are 0. The inverse mask is then computed, using the $\sim$ operator, resulting in a mask where bits at position t and above are 1, and bits below t are 0. `(i & mask)` then extracts the lower t bits from i, isolating the bits that are less significant that the target qubit position; while `(i & notMask)` extracts the bits

from i at position t and above. Shifting the higher bits by 1, `((i & notMask)<< 1)`, then effectively inserts a zero bit at position t in the higher bits, accomodating that we want the t-th bit to be zero. Finally the bitwise OR, `(i & mask)| ((i & notMask)<< 1)` combines the lower bits and the higher shift bits to gives us the final result, *the i-th integer whose t-th bit is zero.*

```
1  for i = 0; i < 2^(n-1); i++ do
2     in0_index = ithCleared(i,t);
3     in1_index = in0_index + 2^t;
4     in0 = vec[in0_index];
5     in1 = vec[in1_index];
6     out0 = mat0 * in0 + mat1 * in1;
7     out1 = mat2 * in0 + mat3 * in1;
8     vec[in0_index] = out0;
9     vec[in1_index] = out1;
10 endfor
```

Listing 3.2: Iteration-based quantum gate application pseudocode. The loop iterates over 2^{n-1} iterations processing an iteration pair each time. Lines 2 and 3 use the ithCleared function to map the iteration index to the memory space indices of the required element pair, with a stride of 2^t. Lines 4 and 5 then perform two memory reads to read the amplitudes. Lines 6 and 7 perform the evolution of the amplitudes by applying the gate matrix multiplication; and finally lines 8 and 9 update the state vector in memory with the new values.

3.1.1 Examples

In the following diagrams, examples of applying a generic gate, $G = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ to different sized circuits with different targets are shown, emphasising the evolution of the state vector amplitudes and their required groupings.

Applying a gate G to the first qubit, i.e. with target qubit index $t = 0$, in a two-qubit system. Figure 3.1 shows the example of applying an arbitrary gate, G, with matrix elements a, b, c, d to the first qubit of a two qubit system. As mentioned above, the qubits are addressed with little-endianness, such that the first qubit, with $t = 0$, refers to the right-most qubit in the qubit register. In the left part of the diagram, the initial state vector, $\Psi = \begin{pmatrix} \psi(00) \\ \psi(01) \\ \psi(10) \\ \psi(11) \end{pmatrix}$ is shown as cells. They are colour-coded to show the iteration groupings

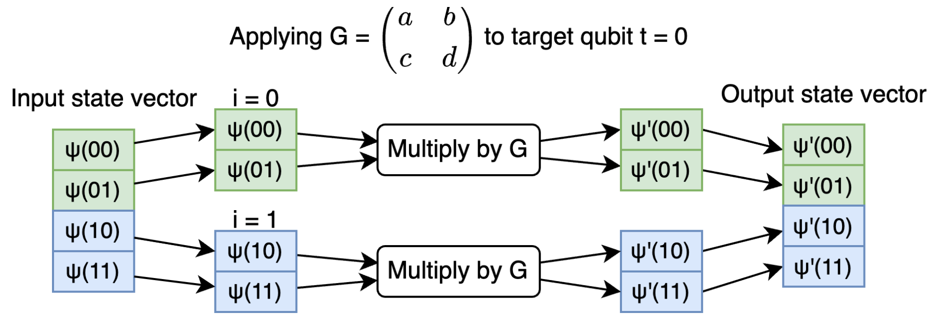


Figure 3.1: Direct iteration processing demonstration of applying a gate, G, to the first qubit ($t = 0$) of a two-qubit system. The arrows demonstrate the flow of data (the probability amplitudes) from the memory system of the simulation device (could be DDR, HBM, or another type of memory). The $t = 0$ case is the simplest in terms of memory access pattern, as the state vector is simply processed in pairs of contiguous amplitudes.

determined by $t = 0$; in this case, the first iteration processes the element pair $\begin{pmatrix} \psi(00) \\ \psi(01) \end{pmatrix}$, and the second contains $\begin{pmatrix} \psi(10) \\ \psi(11) \end{pmatrix}$. The second column demonstrates the iteration indices and the corresponding iteration element pairs. The element pairs are then evolved to their updated values through matrix multiplication by G: $\begin{pmatrix} \psi'(00) \\ \psi'(01) \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \psi(00) \\ \psi(01) \end{pmatrix}$ and $\begin{pmatrix} \psi'(10) \\ \psi'(11) \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \psi(10) \\ \psi(11) \end{pmatrix}$. After the matrix multiplication, the updated pairs are demonstrated in the figure and they are written back to memory at the same indices.

Applying a gate to the second qubit ($t = 1$) of a two-qubit system. Figure 3.2 shows the data flow for the same arbitrary gate, G, but applied to the second qubit of the two-qubit system. It is useful in this example to look at the return values of the `ithCleared(i,t)` function. For the first iteration, the `ithCleared(0,1)` function call determines that the index of the first amplitude is 0 (this is always the case when $i = 0$). The element pair stride (2^t) then determines the index of the second element as $0 + 2^1 = 2$. For the second iteration, `ithCleared(1,1)` returns 1 as the index of the first amplitude and $1 + 2^1 = 3$ is the index of the second amplitude of the pair. As in the previous example, the amplitude pairs are then each evolved through matrix multiplication with G, and the updated pairs are written back to memory in the same indices.

Applying a gate to the second qubit ($t = 1$) of a three-qubit system. Figure 3.3 demonstrates the last example extended to the $n = 3$ case. $t = 1$ implies the same element pair stride of $2^t = 1$, however there are now $2^{n-1} = 4$ iterations to consider ($i \in \{0, 1, 2, 3\}$). We can see that the results of the `ithCleared(i,1)` give indices for the first members of the amplitude pairs to be, 0, 1, 4, and 5. This essentially divides the state vector into two groups which can each be treated as the same two-qubit problem from the last example. More on

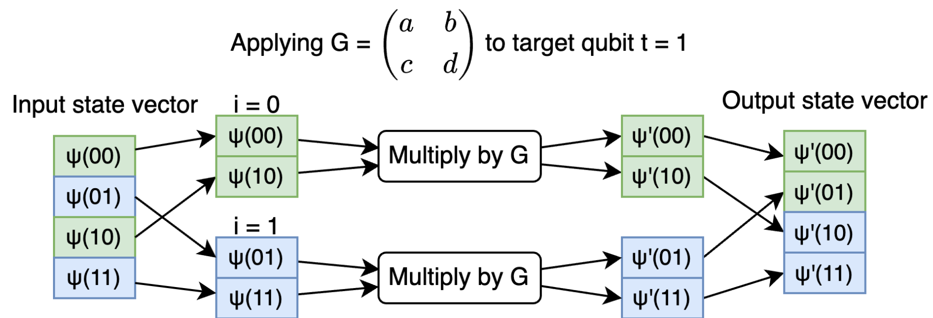


Figure 3.2: Direct iteration processing demonstration of applying a gate, G, to the second qubit ($t = 1$) of a two-qubit system. In this case, the state vector is processed in pairs, with an element pair stride of $2^t = 2$.

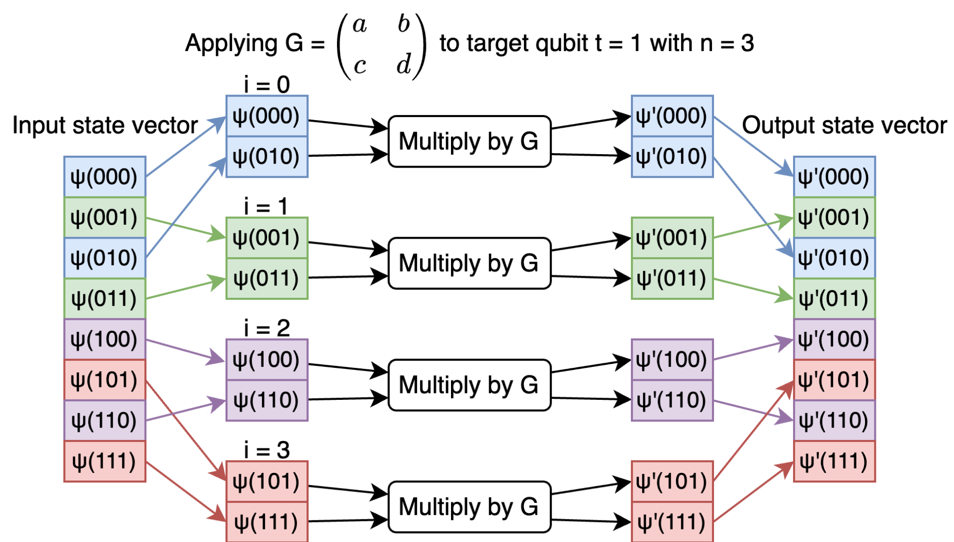


Figure 3.3: Direct iteration processing demonstration of applying a gate, G, to the second qubit ($t = 1$) of a three-qubit system. Amplitudes paired together for the same iteration are grouped by colour.

this grouping is explained in the next section. Similar to before, the element pair stride then determines the indices of the second elements of each pair to be 2, 3, 6, and 7, respectively.

Applying a gate to the third qubit ($t = 2$) of a three-qubit system. For completeness, we also demonstrate the most complex access pattern for the $n = 3$ case in Figure 3.4. In this case, no such grouping as in the $t = 1$ case is apparent (unless we consider the entire state vector as a single group). For each of the iterations, the `ithCleared(i,2)` function calls give indices for the first members of the amplitude pairs to be, 0, 1, 2, and 3; and again the element pair stride determines the indices of the second elements to be 4, 5, 6, and 7, respectively.

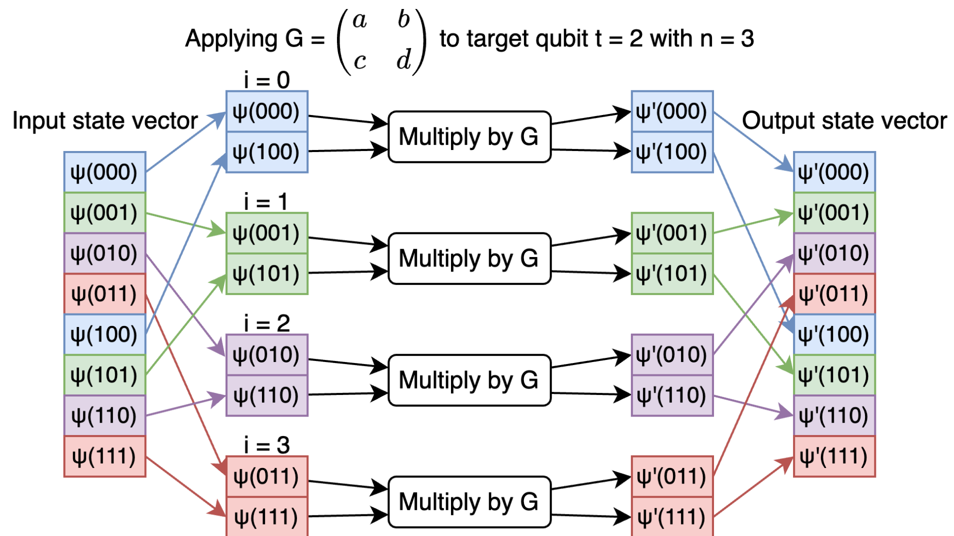


Figure 3.4: Direct iteration processing demonstration of applying a gate, G, to the third qubit ($t = 2$) of a three-qubit system.

3.1.2 Controlled gates

Every control imposed on a gate operation halves the number of amplitude pairs which need to be processed through the matrix multiplication. For a control qubit index c in the quantum register, the iterations which would operate on the amplitude pairs corresponding to the qubit c being zero are skipped for the case the gate operation is conditional on $|c\rangle = |1\rangle$. This is also the case for when there exists multiple control qubits; the iterations operating on amplitude pairs where **any** of the control qubits are zero are skipped. Corresponding to the anti-controls described in the Introduction, in Section 1.2.4, some circuits contain controls on a qubit c where the condition is $|c\rangle = |0\rangle$. The anti-controls on those gates are replaced with normal controls by adding NOT gates surrounding the anti-controlled qubits during the circuit compilation step performed by the pre-FPGA toolchain described in Section 4.2. Without the optimisation described in section 3.1.3, all the iterations are still scheduled and are checked on an iteration-by-iteration basis to determine whether the evolution of the iteration pair is necessary based on the control qubits. This is determined by computing the index of the first element in the amplitude pair, then checking if any of the control qubit indices are 0 in the index. A code snippet demonstrating how this check is computed is presented in Listing 3.3.

The first example we describe is for a 2-qubit system with a target qubit index 0, and a control qubit index 1. Without the control, to process the gate, we would need $2^{n-1} = 2$ iterations. However when the control qubit is imposed, the number of iterations is halved and so only one iteration is needed. This example is described in Figure 3.5.

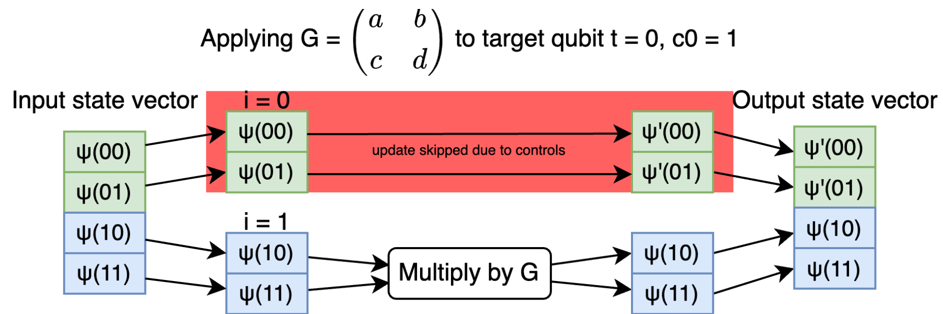


Figure 3.5: Direct iteration processing demonstration of applying a gate, G, to the first qubit $(t = 0)$ with a single control on the second qubit $(c_0 = 1)$. The first iteration is skipped due to the conditions imposed by the control qubit.

```
in0_index = ithCleared(i,t);

// check controls
bool perform = true;
for(c in controls) do
   perform &= ((1 << c) & in0_index) > 0;

if(perform) {
   // evolve the pair
   ...
}
```

Listing 3.3: Determining whether an iteration evolves the amplitude pair in a given iteration, i, based on an array of controls, and a target qubit index, t.

The second example in Figure 3.6 demonstrates a single control on a gate being applied to the second qubit ($t = 1$) of a 3 qubit system, where the first qubit is used as the control ($c = 0$). Here, 2 iterations are needed instead of 4, as the iterations which process the amplitude pairs $(000, 010)$, $(100, 110)$ are skipped (recall we use little-endian qubit addressing).

The final example in Figure 3.7 demonstrates two control qubits on a gate applied to the third qubit ($t = 2$) of a 3-qubit system. Since there are two control qubits, the number of iterations required is halved twice resulting in only the iteration which processes the amplitude pair $(011, 111)$ to be run.

3.1.3 Optimising Controlled Gates Scheduling

In this section, a novel optimisation is introduced which benefits the scheduling of iterations in a Direct Iteration Processing architecture. When controls are imposed on a gate, **each control halves the set of pairs that need to be updated in memory**. As described previously, the condition to apply a controlled gate to an iteration pair is that the value of the bit

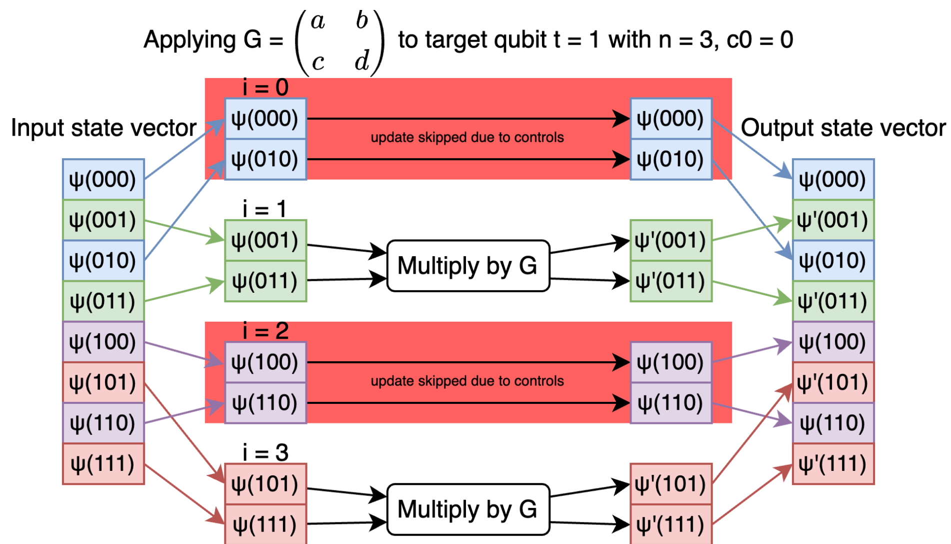


Figure 3.6: Direct iteration processing applying a gate, G, to the second qubit $(t = 1)$ of a three-qubit system with a single control on the first qubit $(c_0 = 0)$. The red boxes indicate skipped iterations due to the control qubit.

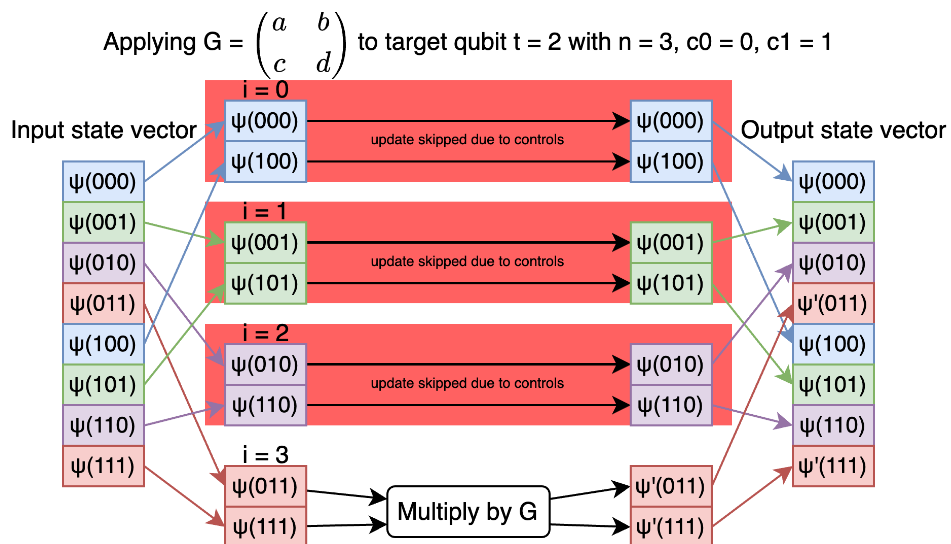


Figure 3.7: Direct iteration processing applying a gate, G, to the third qubit $(t = 2)$ of a three-qubit system with the other qubits acting as controls $(c_0 = 0$ and $c_1 = 1)$. The red boxes indicate skipped iterations due to the controls. Since each control halves the number of iterations which update the state vector, the number of required iterations here is a fourth of the original.

(of the amplitude's index in the state vector in binary) in the same position as the control qubit in the register is 1. It can be observed that since the two elements constituting a pair will only differ in the target qubit, then the controls will cancel whole iterations at a time (as opposed to possibly updating a single element of the pair); i.e. pairs are either entirely updated together or not at all.

This is typically handled by a check in the execution of an iteration, which determines whether the memory access should go ahead after the pair indices have been computed based on the target qubit's index. On a CPU/GPU with dynamically scheduled iterations, this does not cause a loss in performance as no cycles are scheduled for reading and writing to memory until the control flow determines that this should happen. On an FPGA, however, the potential memory accesses are statically scheduled and so clock cycles are always allocated for them; wasting cycles when the control flow determines that no computation should take place.

We observe that we can schedule exactly as many iterations which perform memory accesses as are required for the gate, taking the gate's controls into account. To demonstrate this, we introduce the concept of an iteration index set in the context of full state vector simulation. As described above, to simulate a quantum gate over an n-qubit register, 2^{n-1} iterations are required, giving an iteration index set of $[0, 2^{n-1} - 1]$. Define this set as I_g, the **global iteration index set**. These are the indices which can be plugged into the `ithCleared(i,t)` function along with the target qubit t to return the index of the first element in the pair of amplitudes that need to be processed for any given iteration $i \in I_g$.

Our goal is to be able to schedule only the number of iterations that are required taking into account each added control, i.e. introduce a **reduced iteration index set** $I_r = [0, 2^{n-n_c-1} - 1]$, where n_c is the number of controls of the gate.

The challenge is to map this reduced set I_r back to the global set I_g, as directly scheduling I_r would result in incorrect calculations. The idea is to iteratively map the smaller iteration sets back to I_g, considering each control qubit, selecting the values of i from I_g (which can be plugged into `ithCleared`) that correspond to I_r taking into account the set of controls applied to the gate $C = \{c_0, c_1, ..., c_{n_c-1}\}$.

This means we need a map from the values of I_r to the values of I_g. For a single control, let I_{r_0} be the reduced iteration set; this will have half the cardinality of I_g. If we introduce a further control, let the corresponding reduced iteration set be I_{r_1} (which will have half the cardinality of I_{r_0}); as long as this further control is higher in the register than the first control, we can map from I_{r_1} to I_{r_0}, and then finally map from I_{r_0} to I_g. This is the general idea for handling higher numbers of controls: iteratively map from the smaller iteration sets until the global iteration index in I_g is reached.

To realise this, we map out the required memory accesses for differently controlled gates to

t = 0	0	1	2	3	4	5	6	7
t = 1	0	1	2	3	4	5	6	7
t = 2	0	1	2	3	4	5	6	7

Figure 3.8: Access patterns for gate applications for a 3-qubit register. The numbers on the state vector represent the index of the complex probability amplitude. The table shows how the amplitudes are accessed depending on different target qubits. Like-coloured boxes are accessed and computed on together.

find a pattern. We start from the access pattern demonstrated in Figure 3.8 and rearrange the amplitudes such that iteration pairs are contiguous (they are not actually rearranged in memory, rather this is just for demonstration). We then impose controls, in an ascending order, for every target qubit example. We make an exhaustive list of controls: for a 4-qubit example, there will be three cases with 1 control qubit, three cases with 2 control qubits, and one case with 3 control qubits. When controls are arranged in a logical order across different target examples, a pattern emerges in the iterations which are skipped.

For brevity, Figure 3.9 shows the example for a 3-qubit register. Note that the first row in each target example (representing the uncontrolled case) is now shown in binary, to make it easier to recognise the control condition for the controlled cases, and the whole table is rearranged such that iteration pair elements are contiguous. For each controlled case, we cross out the iteration pairs which do not satisfy the controls (where the control qubits are 0 in the binary representation). Regardless of the target qubit, the same **iteration skips** pattern emerges.

In order to encode these iteration skips, we start by introducing the concept of an **adjusted control**, which is a re-indexing of the control qubits relative to the target qubit: if the control qubit is greater than the target qubit, subtract one, otherwise keep its original value. This is shown in Eq. 3.1. This method gives the same values of adjusted controls for all the **control qubit enumerations** for different values of the target qubit; e.g. for the 3-qubit register demonstrated in Figure 3.9, the method gives us adjusted control values of $c_{adj} = 0$, $c_{adj} = 1$, and $c_{adj} = 0, 1$ for the three possible control qubit cases. We can then compute a **skip interval** corresponding to each adjusted control value as $2^{c_{adj}}$. Based on the computed skip interval, we can map any iteration index belonging to a reduced iteration index set ($i_{r_{k+1}}$) to a higher iteration index (i_{r_k}) by adding to it, following the iterative formula shown in Equation 3.2 where we can consider i_g, the global iteration index, as $i_{r_{-1}}$.

$$c_{adj} = \begin{cases} c - 1 & \text{if } c > t \\ c & \text{otherwise} \end{cases} \tag{3.1}$$

t = 0	000	001	010	011	100	101	110	111
t = 0, c = 1			2	3			6	7
t = 0, c = 2					4	5	6	7
t = 0, c = 1, 2							6	7

t = 1	000	010	001	011	100	110	101	111
t = 1, c = 0			1	3			5	7
t = 1, c = 2					4	6	5	7
t = 1, c = 0, 2							5	7

t = 2	000	100	001	101	010	110	011	111
t = 2, c = 0			1	5			3	7
t = 2, c = 1					2	6	3	7
t = 2, c = 0, 1							3	7

Figure 3.9: Access patterns for controlled gate applications for a 3-qubit register. For demonstration, the boxes representing the amplitudes are rearranged such that required pairs are contiguous, in contrast to Figure 3.8. Crossed out pairs indicate skipped iterations due to the controls imposed on the gate. The pattern of control skips is the same when controls are arranged in ascending order, as shown.

$$i_{r_k} = i_{r_{k+1}} + (\lfloor i_{r_{k+1}} / 2^{c_{adj}} \rfloor + 1) \times 2^{c_{adj}} \tag{3.2}$$

To illustrate this, we take the $t = 1$ case for a 3-qubit register as an example (the middle table in Figure 3.9). For 3 qubits, the global iteration index set is $I_g = \{0, 1, 2, 3\}$ (as there are $2^3 = 8$ amplitudes, the number of iterations is half the size of the state vector). When one control is imposed ($c_0 = 0$ or $c_0 = 2$), the reduced iteration index set is $I_{r_0} = \{0, 1\}$, and these are the values which the NDRange kernel will be scheduled with (i.e. the values that `get_global_id()` will return). For $c_0 = 0$, the adjusted control remains the same as $c_0 < t$, so $c_{0_{adj}} = 0$. The skip interval is then computed as $2^{c_{0_{adj}}} = 2^0 = 1$.

Then, following Equation 4, I_{r_0} can be mapped to I_g in this way: $\{0, 1\} \rightarrow \{0 + (0/1 + 1) \times 1, 1 + (1/1 + 1) \times 1\} = \{1, 3\}$, meaning the second and fourth global iterations are the required ones, which matches the result in the figure. For the case where $c_0 = 2$, $c_0 > t$ and so the adjusted control is $c_{0_{adj}} = c_0 - 1 = 1$, giving a skip interval of $2^1 = 2$. Following the same formula as the previous case, we can map the reduced iteration index set to the global one in this way: $\{0, 1\} \rightarrow \{0 + (0/2 + 1) \times 2, 1 + (\lfloor 1/2 \rfloor + 1) \times 2\} = \{2, 3\}$, meaning the last two global iterations are the desired ones, which again can is verified by the figure. Finally, for the case where we have two controls $c_0 = 0, c_1 = 2$, we perform two mappings: one from $I_{r_1} = \{0\}$ (corresponding to the two imposed controls) to $I_{r_0} = \{0, 1\}$ (corresponding

to one imposed control) and then to $I_{r_{-1}} = I_g = \{0, 1, 2, 3\}$, the global iteration index set.

To go from I_{r_1} to I_{r_0}, we take the first control $c_0 = 0$ and apply the formula: $c_{0_{adj}} = c_0 = 0$ as $c_0 < t$, and the skip interval is again $2^0 = 1$. The single element, $i_{r_1} = 0$ in I_{r_1} is then mapped to $i_{r_0} = 0 + (0/1 + 1) \times 1 = 1$ in I_{r_1}. We then perform the second mapping, for $c_1 = 2$: $c_{1_{adj}} = c_1 - 1 = 1$ as $c_1 > t$, and the skip interval is $2^1 = 2$. Applying the formula gives $i_g = 1 + (\lfloor 1/2 \rfloor + 1) \times 2 = 3$, meaning only the final iteration in the global iteration set is to be performed. The only condition for this formula to function correctly is that the controls have to be in ascending order, i.e. strictly $c_0 < c_1 < ... < c_{n_c-1}$.

With this formula, we can schedule the compute kernels with exactly the number of iterations that will always update the memory. We use the formula to iteratively go from a reduced iteration index to the equivalent global iteration index allowing us to use the previously used memory indexing strategy based on the `ithCleared` function.

3.2 Group-based Processing

While the technique described above lends itself to being easily expressible as a single loop computation for processing each gate in a quantum circuit, there are not many optimisations which can be applied to this formulation. In this section, group-based processing is described and a key optimisation, gate fusion, is presented in its context.

Regardless of the underlying implementation of the simulation architecture, the iteration-based and group-based approaches are different ways of expressing the same simulation computation with different loop structures. In the iteration-based approach, there is a single loop which loops over all the required iterations and uses a function to compute the state vector indices at every iteration. In the group-based approach however, the state vector is partitioned into **groups** such that each group contains a full set of amplitude pairs to be processed together. For a target qubit t, the stride between the amplitude pairs in memory remains the same at 2^t and thus to accomplish this, each group consists of 2^{t+1} elements; where the iteration pairs are separated by half the size of the group, i.e. we pick one element from the first half of the group and its corresponding element from the second half and together they form an iteration pair to be processed together. Each group then replaces 2^t iterations from the iteration-based approach. Thus, for the group-based approach, there are two nested loops required to perform the simulation: one which loops over all the groups and another to loop over the iterations inside the group. The following metrics can then be defined for the group-based approach, for target t and number of qubits n:

$$\text{Group size: } E = 2^{t+1} \tag{3.3}$$

$$\text{Inter-group iteration count: } I = 2^t \tag{3.4}$$

$$\text{Group count: } G = 2^{n-t-1} \tag{3.5}$$

In this approach, no bit-wise function is needed to compute the memory indices of the state vector elements as they can be computed from the two loop indices. For group loop index g going from 0 to G (outer loop), and for iteration index i from 0 to I (inner loop), the memory indices can be computed as:

$$\text{First pair element: } v_0 = g \times E + i$$

$$\text{Second pair element: } v_1 = g \times E + i + I$$

The loop structure for this approach of group-based processing is then demonstrated in Listing 3.4. In the listing, lines 1 and 2 define the loop bounds for each loop, where G and I are, as defined above, the group count, and the group iteration count, respectively. Line 3 and 4 then read from memory and we use E, the group size (number of elements in each group) to compute the required memory indices. The stride for computing the index of the second element in the pair is simply I, the inter-group iteration count, which is equivalent to the element pair stride, 2^t. Lines 5 and 6 then compute the evolution through matrix multiplication; lines 7 and 8 then write back the updated values to memory. In this case, since we are accessing each element in memory independently, we cannot benefit from burst memory access, since the two elements required to be accessed in the same loop iteration will have an exponential stride betweeen them.

```
1  for g=0; g < G; g++ do
2     for i=0; i < I; i++ do
3        in0 = vec[g*E + i];
4        in1 = vec[g*E + i + I]
5        out0 = mat0 * in0 + mat1 * in1;
6        out1 = mat2 * in0 + mat3 * in1;
7        vec[g*E + i] = out0;
8        vec[g*E + i + I] = out1;
9     endfor
10 endfor
```

Listing 3.4: Pseudocode for group-based quantum gate application. Two nested loops are involved in group-based processing, one which loops over the groups, and one over the iterations within each group.

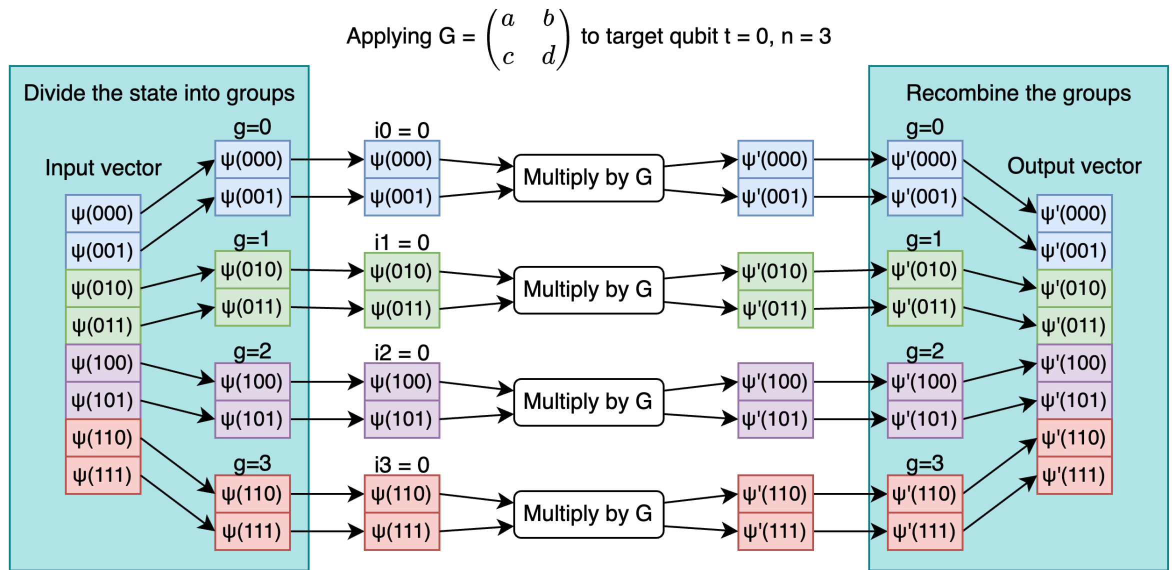


Figure 3.10: Group processing application of gate G, to the first qubit $(t = 0)$.

3.2.1 Examples

We now present some examples to demonstrate this approach.

Example: n = 3, t = 0

Figure 3.10 shows an example of applying a gate to the first qubit of a 3-qubit system using the group-based processing approach. In this case, the input vector is divided into $G = 2^{3-0-1} = 4$ groups, each having $I = 2^0 = 1$ iteration. The diagram shows the division into groups in the teal boxes on the sides. Each group is then assigned its own iteration index parameter (which will always be 0 in this case).

Example: n = 3, t = 1

Figure 3.11 shows an example of applying a gate to the second qubit of a 3-qubit system using the group-based processing approach. In this example, the state vector is divided into $G = 2^{3-1-1} = 2$ groups, each having $I = 2^1 = 2$ iterations.

Example: n = 3, t = 1, c0 = 0

Figure 3.12 shows an example of applying a controlled gate to the second qubit of a 3-qubit system using the group-based processing approach. The control in this case is on the first qubit. Similar to the direct-iteration example above, the controls cause half the iterations to be skipped. In theory, this could imply that entire groups would be skipped (though this is not the case in this example); this concept is expanded upon below in Section 3.4.

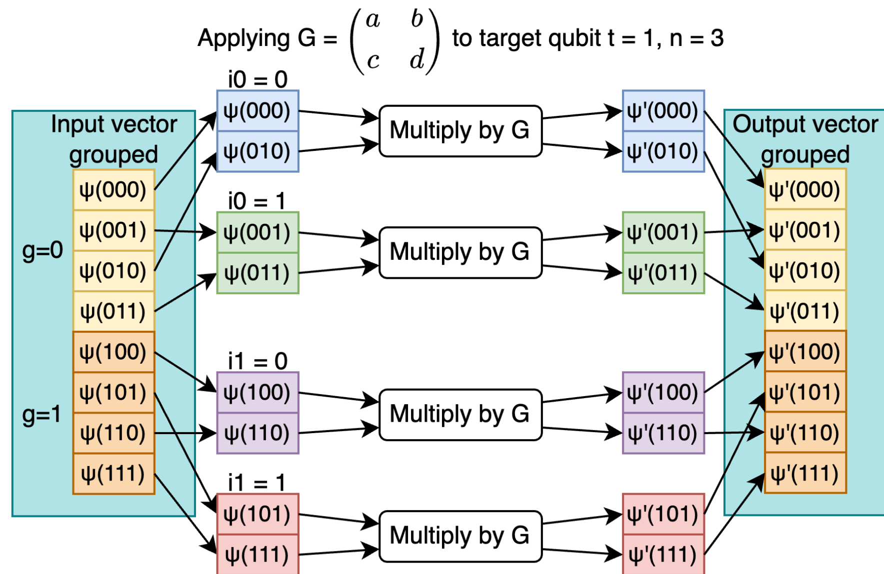


Figure 3.11: Group processing application of gate G, to the second qubit $(t = 1)$.

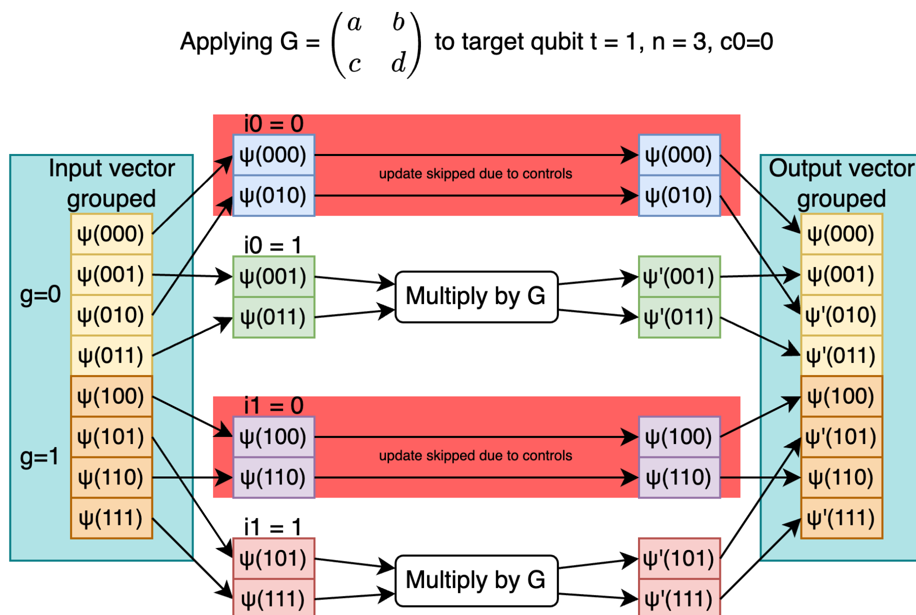


Figure 3.12: Group processing application of gate G, to the second qubit $(t = 1)$ with a control on the first qubit $(c0 = 0)$.

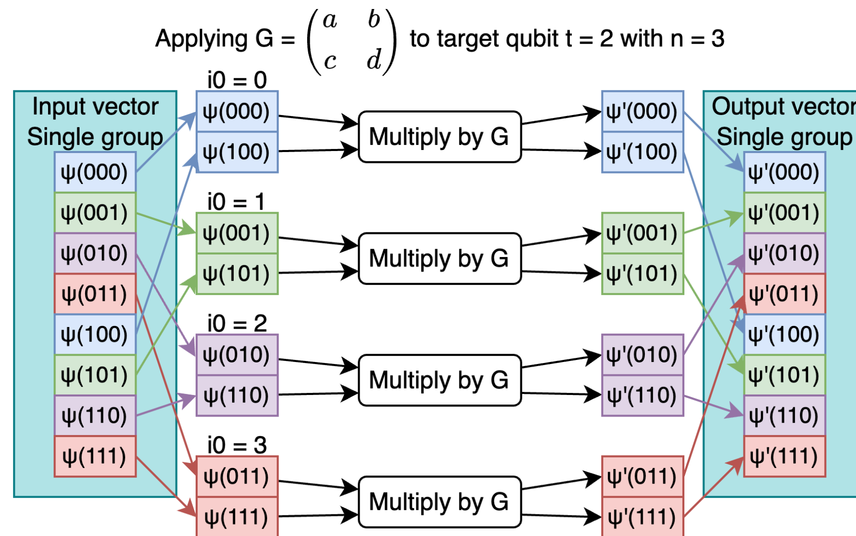


Figure 3.13: Group processing application of gate G, to the third qubit ($t = 2$).

Example: n = 3, t = 2

Finally, Figure 3.13 shows an example of applying a gate to the last qubit of a 3-qubit system using the group-based processing approach. The target qubit $t = 2$ implies that there will be only one group, $G = 2^{3-2-1} = 1$, with $I = 2^2 = 4$ iterations. This essentially becomes a case of direct-iteration processing over one group.

3.3 Adding a buffer: Iterating over Buffer Passes

Adding a buffer to the system allows an important optimisation by taking advantage of burst memory reads and writes. FPGAs have very fast BRAM bandwidth compared to the traditional DRAM or even HBM banks present in more recent FPGAs and GPUs. In direct iteration processing, a single amplitude is read at a time from the DRAM (and we need to read two amplitudes per iteration!), while the group-based approach allows using a buffer and burst-memory access to group together the memory read/write operation required to perform several iterations and reduce the time spent reading and writing to memory. This also allows for better overlap of compute time and memory access time.

Ideally, a buffer which can hold a group of any size would be used; however, since t can go from 0 to $n-1$, the largest possible group size is 2^n, i.e. the entire state vector. This is certainly unfeasible as the resource usage on the FPGA fabric would increase exponentially **with number of qubits** n. Thus, the buffer is limited to a fixed size defined at compile-time and an architecture is developed around this size. Let the number of state vector elements which the buffer can hold be L. It is also useful to define a derived parameter known as the buffer's **effective qubit size**: $l = \log_2 L$; this is the equivalent number of qubits correspond-

ing to a state vector which can fit in the buffer (note however that the buffer will in general never hold an entire state vector, only slices of one).

Since the buffer size is limited and the element pair stride is exponential with the target, it is not guaranteed that the element pairs required for a given iteration will be available to be read/written in one burst memory access. In particular, we can define two access pattern cases, corresponding to the condition $t < l$.

3.3.1 Motivating different buffer cases

Before presenting the cases formally, three examples are presented. Recall that a group refers to a block of contiguous amplitudes entirely containing a complete set of iterations required to process the simulation of a quantum gate on *part of a state vector*, meaning that we do not need to access any elements outside the group to process any single iteration fully.

n = 4, l = 2, t = 1

Consider the example of simulating a gate on the second qubit ($t = 1$) on a 4-qubit system: $n = 4 \implies$ we have $N = 16$ amplitudes to store in the DRAM. Let our architecture have a buffer with effective qubit size: $l = 2 \implies$ we have $L = 4$ amplitudes which can be stored in the buffer at a time. Based on n and t, we can compute the group metrics from Equations 3.5 and 3.3: the group count, $G = 2^{n-t-1} = 4$, and the group size, $E = 2^{t+1} = 4$. Thus we have 4 groups each containing 4 amplitudes to be processed. In this case, each group fits exactly within the buffer, and so we simply process each group by loading it entirely into the buffer and accessing its elements directly based on the element pair stride (2^t). Each group will have two iterations ($I = 2^t = 2$), and the element pair stride is $2^t = 2$. Instead of reading each amplitude individually (which would be the case since $t = 1$ implies that the element pair stride is 2, i.e. not contiguous for each iteration), we now read 4 amplitudes at a time, just by adding a buffer of size 4 elements; thus we go from 16 memory read/write cycles to only 4. Listing 3.5 demonstrates the structure of processing a gate with these parameters.

```
1  for g = 0; g < 4; g++ do // Group loop
2    mem_read: buffer[0:3] = read_from_mem(from: g*4, count: 4);
3
4    for i=0; i < 2; i++ do // Each group has two iterations
5      in0 = buffer[i];
6      in1 = buffer[i + 2]; // The element pair stride is 2, half the
         buffer size
7      out0 = mat0 * in0 + mat1 * in1;
8      out1 = mat2 * in0 + mat3 * in1;
9      buffer[i] = out0;
10     buffer[i + 2] = out1;
```

```
11     endfor
12
13     mem_write: write_to_mem(buffer[0:3], at: g*4, count: 4);
14 endfor
```

Listing 3.5: Gate processing pseudocode for n = 4, l = 2, t = 1 example. Each group's elements fit exactly into the buffer.

n = 4, l = 2, t = 2

In this example, we stay with the previous system of $n = 4$ qubits and a buffer qubits size of $l = 2$. However, we now consider the gate operating on the third qubit ($t = 2$) of the 4-qubit system. Again, we compute the group count and the group size to be: $G = 2^{n-t-1} = 2$ and $E = 2^{t+1} = 8$, respectively; and so the 16-dimensional state vector here divides into 2 groups of 8 amplitudes each. Our buffer with $L = 2^l = 4$ spaces cannot hold a full group at one time, and thus the groups have to be divided across different buffer passes, with each buffer pass performing two memory reads/writes, one from each half of the group. Recall that in general to process a group, the element pair stride implies that the iteration pairs always come from corresponding indices from each half of the group (i.e. the element pair stride is always half the size of the group); this is why each pass reads in this way.

In this case, each group divides across two buffer passes. To process the first group, the first pass needs to read the first two elements of the first half of the group ($\psi(0000)$ and $\psi(0001)$) and the first two elements of the second half of the group ($\psi(0100)$ and $\psi(0101)$) into the buffer. Then the buffer is operated on by the compute cores, processing the iteration pairs $(\psi(0000), \psi(0100))$ and $(\psi(0001), \psi(0101))$ by reading from each half of the buffer in each iteration. The buffer elements are then written back to their same locations in memory as read from. In the second pass, the third and fourth elements from the first half of the group ($\psi(0010)$ and $\psi(0011)$), and the third and fourth elements from the second half of the group ($\psi(0110)$ and $\psi(0111)$) are read into the buffer, which is again operated on just like in the first pass, following which the updated amplitudes are written back to the same locations in memory.

The second group then operates in the same fashion as the first, except with the elements of the second half of the state vector (which constitute the second group), $\psi(1xyz)$. This procedure is demonstrated in Listing 3.6.

```
1 for g = 0; g < 2; g++ do // Group loop
2     for p = 0; p < 2; p++ do // Buffer pass loop
3     mem_read1: buffer[0:1] = read_from_mem(from: g*8 + p*2, count: 2);
4     mem_read2: buffer[2:3] = read_from_mem(from: g*8 + p*2 + 4, count:
          2); // The element pair stride in memory is 4
```

```
5
6     for i=0; i < 2; i++ do // Each buffer pass has two iterations
7        in0 = buffer[i];
8        in1 = buffer[i + 2]; // The element pair stride is 2, half the
            buffer size
9        out0 = mat0 * in0 + mat1 * in1;
10       out1 = mat2 * in0 + mat3 * in1;
11       buffer[i] = out0;
12       buffer[i + 2] = out1;
13    endfor
14
15    mem_write: write_to_mem(buffer[0:1], at: g*8 + p*2, count: 2);
16    mem_write: write_to_mem(buffer[0:1], at: g*8 + p*2 + 4, count: 2); //
          The element pair stride in memory is 4
17 endfor
```

Listing 3.6: Gate processing pseudocode for n = 4, l = 2, t = 2 example. Each group's elements have to divided across two buffer passes as each group consists of 8 elements and the buffer size is 4.

n = 4, l = 2, t = 0

While the previous two examples demonstrated two cases corresponding to $t \geq l$, the first example is really a special case where because $l - t = 1$, each group fits exactly in the buffer and so we can process one group in one full contiguous buffer read. However, in cases where $t < l - 1$, we can fit more than one group in the buffer at any time.

Consider the same system with $n = 4$ and $l = 2$ but now we simulate the gate on the first qubit ($t = 0$). Again, we compute the group metrics: group count, $G = 2^{n-t-1} = 8$, and group size, $E = 2^{t+1} = 2$. Thus, we have 8 groups of 2 amplitudes each to process. If we follow the same approach as the first example (loading one group per buffer pass), we would need 8 buffer passes, and each buffer pass would not fill the entire buffer. However we observe that since everything is a power of 2, we can fit 2 groups in each buffer pass neatly (the number of groups per buffer pass is then the size of the buffer divided by the group size: $\frac{4}{2} = 2$); and since all the groups are contiguous (essentially by definition), we would need only one contiguous memory read/write cycle per buffer pass.

The total number of buffer passes can then be computed as the number of groups divided by the number of groups per buffer pass: $\frac{8}{2} = 4$. The first buffer pass begins by reading the first 4 contiguous elements of the state vector ($\psi(0000), \psi(0001), \psi(0010)$, and $\psi(0011)$). At this point, this can be considered as a smaller, 4-dimensional, state vector (essentially corresponding to a 2-qubit system) over which we want to apply the gate to the first qubit. We proceed with direct iteration processing, except now instead of accessing the DRAM of

the system, we access the on-board buffer, which is orders of magnitude faster. Per direct iteration processing, we have 2 iterations here to execute (corresponding correctly to $I = 2^t = 1$ iteration per group as per Eq. 3.4), and we can utilise the `ithCleared` in an iteration loop to determine the correct buffer indices of the amplitudes required for each iteration. After all the iterations are executed, the updated buffer is then written back to the same locations in memory as read from, in one contiguous burst write.

This process is then repeated for the remaining buffer passes, with the second pass processing elements $\psi(0100), \psi(0101), \psi(0110)$, and $\psi(0111)$, and so on, as demonstrated in Listing 3.7.

This same procedure can also be applied to our first example where $l - t = 1$ led to each group exactly fitting in the buffer. The `ithCleared` function and direct iteration processing on the buffer method will give us the correct buffer indices of the amplitudes.

```
1  for p = 0; p < 4; p++ do // Pass loop
2     mem_read: buffer[0:3] = read_from_mem(from: p*4, count: 4);
3
4     for g = 0; g < 4; g++ do // Groups per pass loop
5        for i=0; i < 2; i++ do // Each group has two iterations
6           // Apply direct iteration processing over the buffer
7           in0_index = ithCleared(i,t);
8           in1_index = in0_index + 2^t;
9
10          in0 = buffer[in0_index];
11          in1 = buffer[in1_index]; // The element pair stride is 2, half
               the buffer size
12          out0 = mat0 * in0 + mat1 * in1;
13          out1 = mat2 * in0 + mat3 * in1;
14          buffer[in0_index] = out0;
15          buffer[in1_index] = out1;
16       endfor
17    endfor
18
19    mem_write: write_to_mem(buffer[0:3], at: p*4, count: 4);
20 endfor
```

Listing 3.7: Gate processing pseudocode for n = 4, l = 2, t = 0 example. The buffer can fit two group's elements at a time.

3.3.2 Case 1 $t < l$: Single burst access per buffer pass

If $t < l$, then it is guaranteed that the number of elements required to process any complete group involved in the computation can fit in the buffer, and, since all the elements are contiguous in memory, only one burst access (read and write) is required to process the group.

Additionally, depending on the difference $l - t$, multiple contiguous groups can fit in the buffer and be processed with one burst access.

In this case, it is better now to consider **buffer passes** as the top level computation unit involving memory access. The groups can be grouped further together to fit into full buffer read/write passes. The number of groups which can be processed per buffer pass, G_P, and the total number of buffer passes, P, can then be defined as:

$$\text{Number of groups per pass} : G_P = \frac{L}{E} = \frac{2^l}{2^{t+1}} = 2^{l-t-1},$$

$$\text{Number of passes} : P = \frac{N}{L} = \frac{2^n}{2^l} = 2^{n-l},$$

where L is the buffer size, E is the group size, and N is the size of the state vector.

Listing 3.8 shows pseudocode for the processing of a case 1 gate. In the listing, line 1 sets up the buffer passes loop, as we know through the buffer size (always a power of 2) how many total contiguous memory accesses will be required to update the whole state. Line 2 issues a contiguous memory read request to fill the buffer starting from an index of the $p \times L$ in memory. We then perform the same group-based computation as in Listing 3.4 but iterating with a group count of G_P, i.e. only the groups that fit within the buffer. As a result of using on-board resources to buffer the data, the random-access reads and writes on lines 6, 7, 10, and 11 are much faster than the equivalent random-accesses to global memory without using a buffer in the previous listing. After processing all the groups that fit in the buffer, line 15 then writes back the buffer in a contiguous burst write to global memory, at the same location as the read on line 2.

```
1  for p = 0; p < P; p++ do
2     mem_read: buffer[0:L-1] = read_from_mem(from: p*L, count: L);
3
4     for g = 0; g < G_P; g++ do
5        for i=0; i < I; i++ do
6           in0 = buffer[g*E + i];
7           in1 = buffer[g*E + i + I]
8           out0 = mat0 * in0 + mat1 * in1;
9           out1 = mat2 * in0 + mat3 * in1;
10          buffer[g*E + i] = out0;
11          buffer[g*E + i + I] = out1;
12       endfor
13    endfor
14
15    mem_write: write_to_mem(buffer[0:L-1], at: p*L, count: L);
16 endfor
```

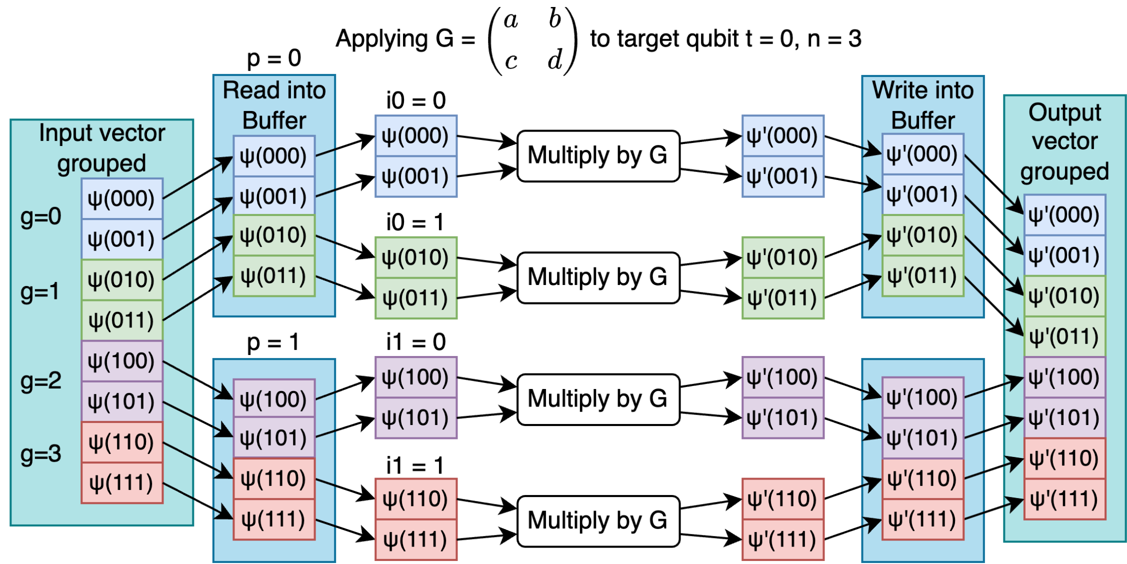


Figure 3.14: Buffered gate application of gate G, to the first qubit $(t = 0)$ for with buffer size 4 $(l = 2)$.

Listing 3.8: Buffer passes case 1 pseudocode. The primary goal of the buffer is to eliminate the number of memory reads and writes by coallescing contiguous accesses through an on-board buffer implemented as BRAMs or registers.

While above in the motivating examples, we used systems of 4 qubits, we now consider a simpler system of 3 qubits for the examples here to keep the diagrams manageable. We will consider the same examples considered for the group-based approach above in Section 3.2.1, now explicitly showing how buffering works in these cases.

Example: n = 3, l = 2, t = 0

Figure 3.14 shows an example of applying a gate to the first qubit of a 3-qubit system using the buffered approach. Consider our architecture has a buffer qubit size, $l = 2$, i.e. it can hold $2^2 = 4$ amplitudes at a time, the same as the motivating examples in Section 3.3.1. In this case, we have 4 groups, each requiring one iteration. Since the buffer can hold 4 elements, we can process two iterations per buffer, thus we need $P = \frac{8}{4} = 2$ buffer passes. In the figure, the red line separates these buffer passes, and the buffer is represented by the blue box. It is important to recognise that these buffer passes are executed sequentially, as there will be a single physical buffer on the board. However, the primary thing to notice here is that each buffer pass takes advantage of the contiguous burst memory access, and loads/stores several groups' worth of amplitudes at once.

We then operate on the groups within in the buffer. This can be implemented as either a

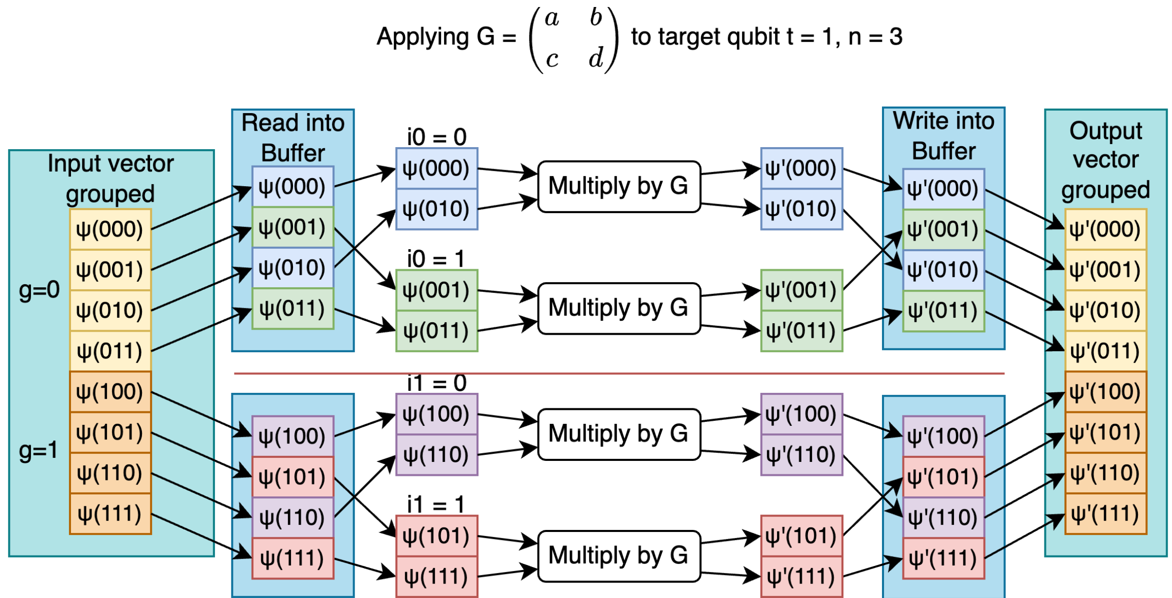


Figure 3.15: Buffered gate application of gate G, to the second qubit $(t = 1)$ for with buffer size 4 $(l = 2)$.

loop over the groups within the buffer, inside each we loop over each iteration, or as direct-iteration processing over the slice of the state vector stored in the buffer. In general, we take the latter approach when implementing for the FPGA as it is one fewer loop that the HLS tools would have to deal with.

Example: n = 3, l = 2, t = 1

Figure 3.15 shows an example of applying a gate to the second qubit of a 3-qubit system using the buffered approach. This example has 2 groups each having two iterations. Again we require two buffer passes, where each buffer pass processes one group. Notice that we load the same slices of the state vector as in the previous example in each buffer pass. However, the access pattern within the buffer is different, applying direct-iteration processing using the different target qubit.

3.3.3 Case 2 $t \geq l$: Two burst accesses per group pass

If instead $t \geq l$, then the group sizes are necessarily larger than the buffer size. However, as long as the buffer size is a power of 2 (i.e. $l \in \mathbb{Z}$), then any group can be divided evenly across multiple buffer passes. However, since the element pair stride is also necessarily larger than the buffer size, we cannot access all the elements required for a pass (i.e. a slice of the group) in one contiguous read/write. Instead each buffer pass now involves two reads/writes, with a size of half the buffer each and a stride of 2^t between them. Note here another difference

from the first case; where a single buffer passes was divided into/processed multiple groups, whereas in this case, a single group is divided across several buffer passes.

In this case, we can define the number of buffer passes *per group*, P_G, as the reciprocal of the number of groups per pass from case 1:

$$P_G = \frac{E}{L} = \frac{2^{t+1}}{2^l} = 2^{t+1-l}$$

Processing the group slice elements once they are read from memory into the buffer would now simply involve accessing an element from the first half of the buffer and its corresponding element in the same position in the second half; together they form an iteration pair.

Listing 3.9 demonstrates the processing of a case 2 gate. Line 1 sets up looping over the groups, and line 2 loops over the required number of buffer passes per group, P_G. Lines 3 and 4 do two contiguous burst memory reads, each sized half of the buffer, with a stride of $I = 2^t$ between them. Line 6 then sets up the computation loop, iterating a number of times equal to half the buffer size, and processing each element from the first half of the buffer with its corresponding pair element in the second half of the buffer. Lines 15 and 16 then write the buffer elements back to the same locations in memory using two contiguous burst writes.

```
1  for g = 0; g < G; g++ do
2     for p = 0; p < P_G; p++ do
3        mem_read1: buffer[0:L/2-1] = read_from_mem(from: g*E + p*L/2,
            count: L/2);
4        mem_read2: buffer[L/2:L-1] = read_from_mem(from: g*E + p*L/2 + I,
            count: L/2);
5
6        for i = 0; i < L/2; i++ do
7           in0 = buffer[i];
8           in1 = buffer[i + L/2];
9           out0 = mat0 * in0 + mat1 * in1;
10          out1 = mat2 * in0 + mat3 * in1;
11          buffer[i] = out0;
12          buffer[i + L/2] = out1;
13       endfor
14
15       mem_write1: write_to_mem(buffer[0:L/2-1], at: g*E + p*L/2, count:
            L/2);
16       mem_write2: write_to_mem(buffer[L/2:L-1], at: g*E + p*L/2 + I,
            count: L/2);
17    endfor
18 endfor
```

Listing 3.9: Buffer passes case 2 pseudocode. In this case, a single group cannot fit fully in the buffer so the groups are divided across several buffer passes.

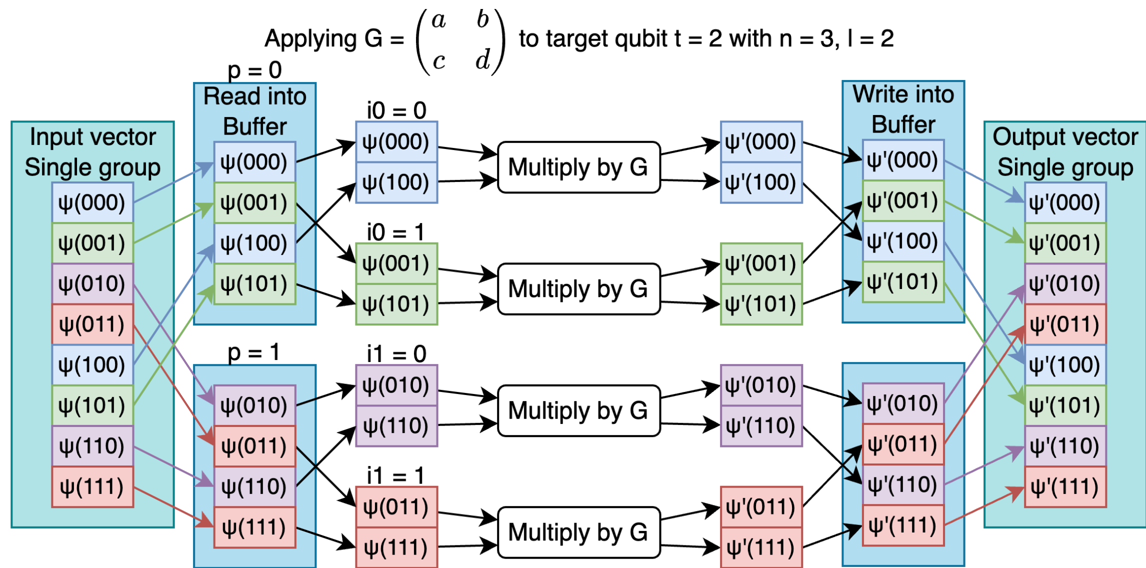


Figure 3.16: Buffered gate application of gate G, to the third qubit $(t = 2)$ for with buffer size 4 $(l = 2)$.

Example: n = 3, l = 2, t = 2

Figure 3.16 shows an example of applying a gate to the last qubit of a 3-qubit system using the buffered approach. As this is case 2 for buffering $(t \geq l)$, each buffer pass requires two contiguous global memory accesses. The first buffer pass loads two groups of amplitudes (each contiguous): $(\psi(000), \psi(001))$, and $\psi(100), \psi(101))$ into the two halves of the buffer. The stride between these amplitude groups is determined from the target as: $2^t = 4$. The buffer is then operated on, with each iteration choosing the corresponding element from each half of the buffer to form an iteration pair. After the iterations are processed, they are written back to the buffer, which is then written back to the global memory using the same pattern. The next buffer pass then proceeds reading a different slice of the state vector, and operating on it, before writing it back to memory.

3.4 Controlled gates in a buffered architecture

As using a buffer involves a more intricate memory access pattern, special consideration has to be given when adding controls to gates. Since a control halves the number of amplitudes which need to be updated, there may be cases where we can get away with skipping entire buffer passes. In this section, we present the different cases we encounter when controls are introduced. These can be complex and so for the FPGA implementations described in the next chapter, they are not used; however we include them here for completeness and as a base for potential future work.

As adding multiple controls can involve quite complicated group/pass skip patterns, we re-

strict ourselves to gates with only one control. We will define additional metrics to the ones defined above, including an inter-group iteration skip interval, a group skip interval, and, in some cases, a pass skip interval.

We can identify two control-related cases related to the difference between the target qubit index, t, and the control qubit index, c. If $c < t$, then no groups are fully skipped, and we have an **inter-group iteration skip interval**, $ISI = 2^c$. On the other hand, if $c > t$, then entire groups can be skipped, and we have a **group skip interval**, $GSI = 2^{c-t-1}$.

Considering these two control cases alongside the two buffering cases explained in the previous section, we get four possible cases for a buffered architecture processing a controlled gate.

3.4.1 Controlled Buffered Cases breakdown

Controlled Case 1, $t < l, c < t$**:** In this case, $t < l$ implies we process multiple groups in the same pass, and $c < t$ implies that we have to skip iterations within each group with an $ISI = 2^c$.

Controlled Case 2, $t < l, c > t$**:** Again, $t < l$ implies that we process multiple groups in the same buffer pass. However, now we can skip entire groups with a $GSI = 2^{c-t-1}$. Furthermore, we notice that if the group skip interval is greater than or equal to the number of groups per pass ($G_P = 2^{l-t-1}$, as defined above in Section 3.3.2), then we can skip entire passes, as all their groups will be skipped. From the definitions of GSI and G_P, we can express this condition in terms of c and l:

$$GSI \geq G_P \implies 2^{c-t-1} \geq 2^{l-t-1} \implies c-t-1 \geq l-t-1 \implies c \geq l$$

Thus, if $c \geq t$, we can define a **pass skip interval**, $PSI = \frac{GSI}{G_P} = 2^{c-l}$.

Controlled Case 3, $t \geq l, c < t$**:** In this case, $t \geq l$ implies that we process the same group across different buffer passes (case 2 buffering as described in the previous section), and $c < t$ implies that within the groups, we have an $ISI = 2^c$ (like in case 1 of control-buffering). However, since a group is divided across several passes, if the ISI is high enough, entire buffer passes could be skipped. In particular, if ISI is greater than or equal to the number of iterations which the buffer can hold ($\frac{L}{2}$), i.e. $ISI \geq \frac{L}{2}$ (where L is the buffer's element size, $L = 2^l$), then we have a $PSI = \frac{ISI}{L/2} = 2^{c-l+1}$. Again, we can express this condition in terms of c and l:

$$ISI \geq \frac{L}{2} \implies 2^c \geq 2^{l-1} \implies c \geq l-1 \implies c > l,$$

which is a slightly different condition for having entire pass skips than the previous cases.

Controlled Case 4, $t \geq l, c > t$: This case again processes the same group across different passes (with number of passes per group defined as in Section 3.3.3: $P = 2^{t+1-l}$), and $c > t$ implies that entire groups will be skipped with a $GSI = 2^{c-t-1}$. These conditions necessarily imply that entire passes will skipped, and with the same condition as controlled case 2: $t \geq l, c > t \implies c > l$. The PSI can be computed from the group skip interval and the number of passes per group to be:

$$PSI = P \times GSI = (2^{t+1-l})(2^{c-t-1}) = 2^{c-l},$$

as in the previous cases 1 and 2.

To summarise, across all of these cases, the important factor is whether $c < l$. If this condition is true, then no entire passes are skipped, and either groups will be skipped with a $GSI = 2^{c-t-1}$ (if $c > t$), or inter-group iterations are skipped with $ISI = 2^c$ (if $c < t$). If on the other hand, $c \geq l$, then entire passes are skipped (regardless of case 1 or case 2 buffer processing), with a $PSI = 2^{c-l}$ if $c > t$, or $PSI = 2^{c-l+1}$ if $c < t$.

3.4.2 Examples

Case 1 Example: n = 3, l = 2, t = 1, c0 = 0

Figure 3.17 shows an example of applying a controlled gate to the second qubit ($t = 1$) of a 3-qubit system using the buffered approach, with the control on the first qubit ($c0 = 0$).

Case 2 Example without pass skips: n = 3, l = 2, t = 0, c0 = 1

Figure 3.18 shows an example of applying a controlled gate to the first qubit ($t = 0$) of a 3-qubit system using the buffered approach, with the control on the second qubit ($c0 = 2$).

Case 2 Example with pass skips: n = 3, l = 2, t = 1, c0 = 2

Figure 3.19 shows an example of applying a controlled gate to the second qubit ($t = 1$) of a 3-qubit system using the buffered approach, with the control on the last qubit ($c0 = 2$). Since $c0 \geq l$, entire buffer passes can be skipped.

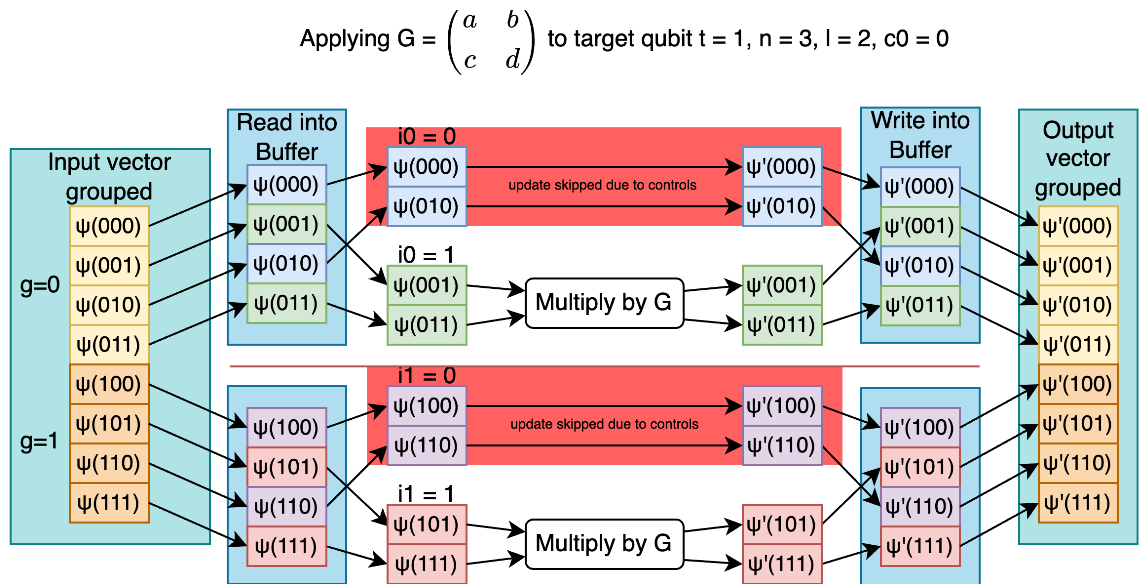


Figure 3.17: Buffered gate application of a controlled gate G, to the second qubit $(t = 1)$ for buffer size 4 ($l = 2$) with a control on the first qubit $(c0 = 0)$.

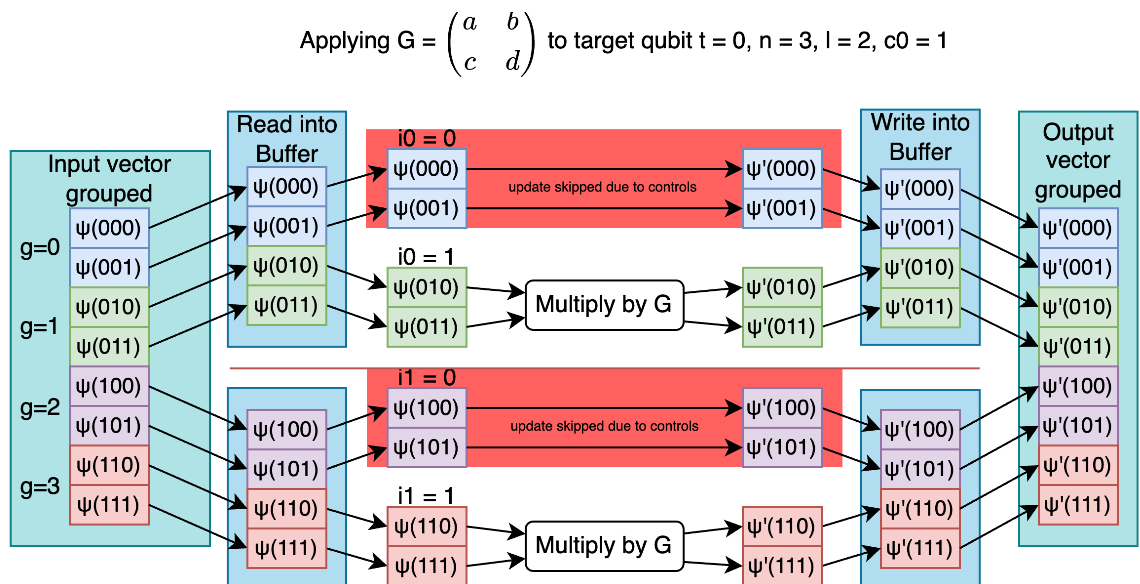


Figure 3.18: Buffered gate application of a controlled gate G, to the first qubit $(t = 0)$ for buffer size 4 ($l = 2$) with a control on the second qubit $(c0 = 1)$.

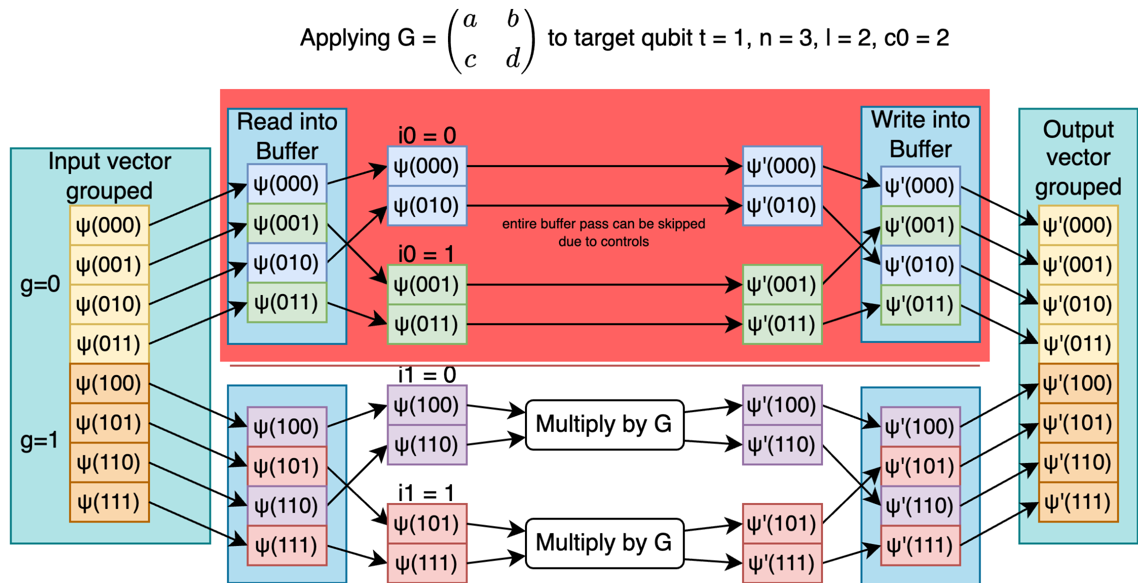


Figure 3.19: Buffered gate application of a controlled gate G, to the second qubit ($t = 1$) for buffer size 4 ($l = 2$) with a control on the last qubit ($c0 = 2$).

Case 3 Example without pass skips: n = 3, l = 2, t = 2, c0 = 0

Figure 3.20 shows an example of applying a controlled gate to the last qubit of a 3-qubit system using the buffered approach, with the control on the first qubit.

Case 3 Example with pass skips: n = 3, l = 2, t = 2, c0 = 1

Figure 3.21 shows an example of applying a controlled gate to the last qubit of a 3-qubit system using the buffered approach, with the control on the second qubit.

Case 4 Example: n = 3, l = 1, t = 1, c0 = 2

Figure 3.22 shows an example of applying a controlled gate to the second qubit ($t = 1$) of a 3-qubit system using the buffered approach, with the control on the last qubit ($c0 = 2$). The difference between this example and the others is we have reduced the buffer size to 2 ($L = 2$ with $l = 1$), in order to demonstrate $c > t \geq l$. Entire buffer passes can always be skipped in this case.

3.4.3 Summary

As demonstrated above, adding a single control can significantly increase the complexity in terms of scheduling and skips across the different loop-levels (iterations, groups, and buffer passes). As we aim to develop a general quantum circuit simulator which can operate on

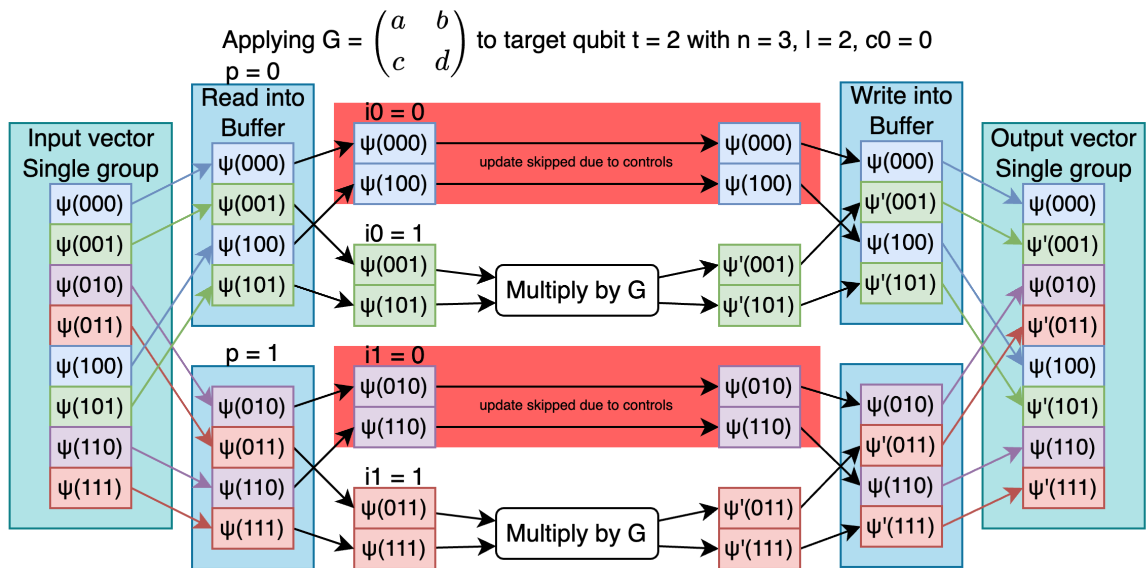


Figure 3.20: Buffered gate application of a controlled gate G, to the third qubit $(t = 2)$ for buffer size 4 $(l = 2)$ with a control on the first qubit $(c0 = 0)$.

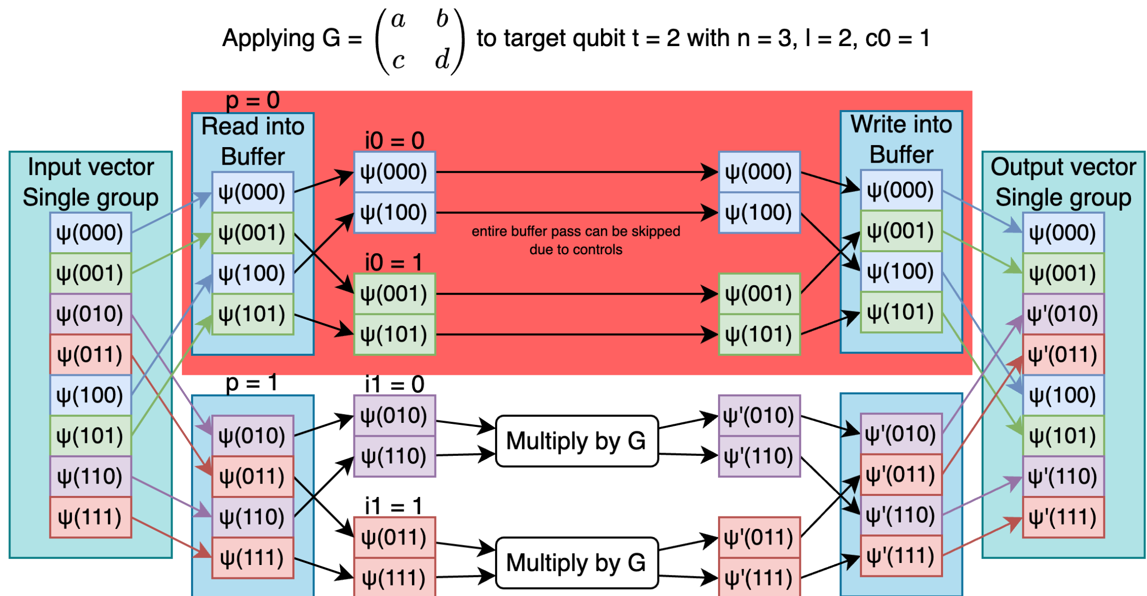


Figure 3.21: Buffered gate application of a controlled gate G, to the third qubit $(t = 2)$ for buffer size 4 $(l = 2)$ with a control on the second qubit $(c0 = 1)$.

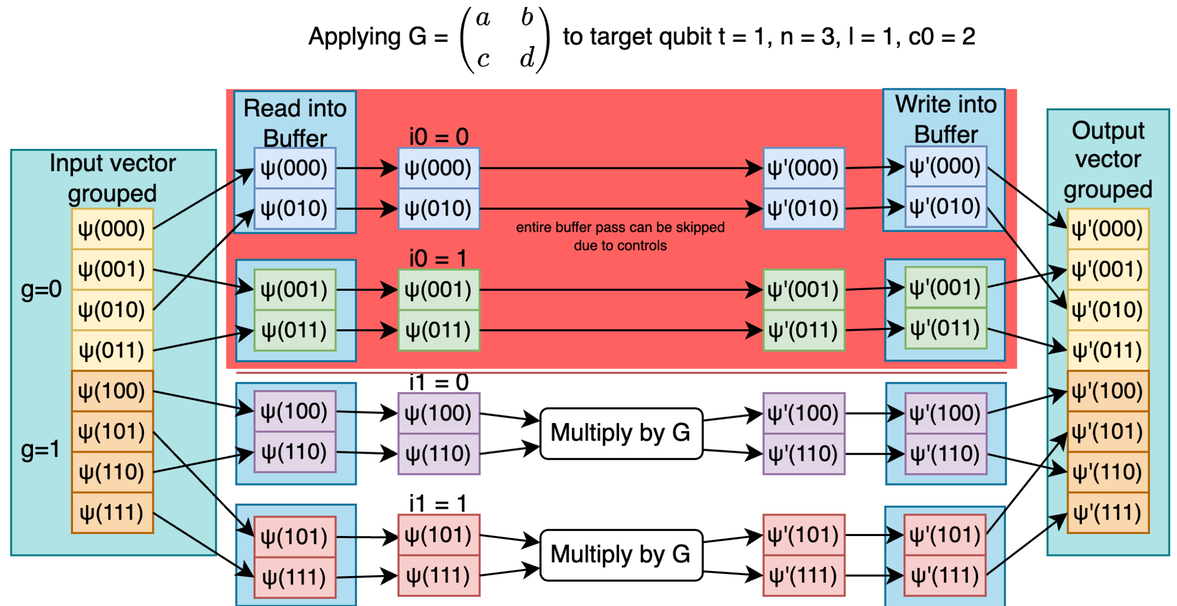


Figure 3.22: Buffered gate application of a controlled gate G, to the second qubit ($t = 1$) for buffer size 2 ($l = 1$) with a control on the last qubit ($c0 = 2$).

circuits containing gates with any number of controls (albeit our architectures would be parameterised by a maximum allowable number of controls at compile time), we have opted to not account for these cases as presented here in our buffered architectures. To maintain correctness, the buffered architectures instead compute a corresponding global iteration index from the indices of the different loop-levels and use this index for control checking, as in the direct iteration processing case. This is similar but far less complex to the technique described in Section 3.1.3 for optimising the scheduling of controlled gates in a direct iteration processing architecture.

3.5 Gate Fusion

The gate fusion optimisation further takes advantage of on-board memory by grouping (or fusing) together consecutive gates which all have consecutive groups of state vector elements that can fit in the buffer through one burst memory access (i.e. which are all case 1 gates with $t < l$) and performing all the fused gates on that slice of the state vector. This is achieved by recognising that, for gates satisfying this condition, each slice of the state vector that fills the buffer will contain all the group elements required to process the whole fused gate block for that slice. Since no elements from different slices are accessed, the gates can be applied to the entire slice, in order. The final result of applying all the gates will be preserved, while only performing the same number of memory accesses as would be required for a single gate otherwise. As discussed in the literature survey (Section 2.1.1), this optimisation was used in prior works for simulators targeted at CPUs and GPUs for the purpose of cache blocking.

In those cases, the computation is described in a way to implicitly let the OS keep reused elements in the last level cache. In our case, on an FPGA, the buffer acts as cache for which we have to explicitly specify all memory access read and write locations.

The first step to allow gate fusion is to identify sequences of gates which satisfy the condition $t < l$ and form fused **gate blocks**. This step operates only on the circuit description and can be performed on the host device. A key difference between our architectures which have this optimisation and those which do not is in the format of the input circuit description. Described in the next chapter (see Section 4.1.1), the default input circuit description for an FPGA will be parameterised by a max number of controls C_{max} allowable per gate. This description is a sequence of integers that looks like $< n >$ (gate)$+$ where n is the number of qubits, and a gate is described by an integer gate code, a target, and then a further C_{max} integers representing the controls of the gate; where if a control is the same as the target then there is no control indicated at that position. This scheme guarantees a static number of integers required to represent each gate: $2 + C_{max}$. In gate fusion, we operate on fused gate blocks instead of gates directly, and the input circuit description must account for this. Since a fused gate block can have any number of gates, this must be reflected in the description, and so the following scheme is proposed: $< n >$ (gate_block)$+$ where a gate block is $< N_{gb} >$ (gate)$+$, where N_{gb} is the number of gates in the gate block and the gates retain the same representation as in the previous circuit input description.

In practice, there is a maximum number of gates which can be fused into a gate block; because to allow for full on-board computation of a gate block on a state vector slice, the gate block gates should be loaded into an on-board BRAM. To achieve this description, a subroutine runs on the host which scans through the gates in the default representation and identifies sequences of consecutive gates which satisfy $t < l$ and groups them into a gate block with a representation as described above. The new input circuit description is then written to the global memory of the FPGA device and the computation can start.

Then, the first step on the FPGA is to read the entire gate block onto the on-board gate block buffer. The computation then proceeds as before for the buffered group-based computation by identifying which case to perform: $t_0 < l$ or $t_0 \geq l$, based on the target of the first gate in the gate block, t_0. For the latter case $t \geq l$, a gate block which contains exactly one gate is required to be processed and this is executed in the same way as in the buffered group-based architecture described in the previous section.

For the case where we read the target of the first gate to be less than l, i.e. $t_0 < l$, a gate block which may contain more than one gate is to be processed. The logic for processing a single gate over a buffer slice of the state vector is the same as in case 1 of the buffered architecture. However, the parameter G_P, describing the number of groups, and the parameter I, describing the iteration count per group, have to be recomputed for each gate based on

reading the target qubit from the gate block buffer. We also have to select the gate matrix on device using the gate's gate code read from the gate block buffer.

This flow is demonstrated in Listing 3.10. Like in Listing 3.8, line 1 sets up the buffer passes loop and line 2 issues a contiguous memory read request to fill the buffer. Line 4 then iterates through the gates in the fused gate blocks with a gate index `gate_i`. For each gate, we then proceed with case 1 buffered case 1 processing. We compute the gate-specific parameters: G_P, the number of groups per buffer pass, and I, the number of iterations per group, on lines 7-9. We then have to select the gate matrix since each gate is presumed to be different; this is done on lines 11-12. Lines 14-23 then perform the same group-based computation as in Listing 3.4. After all the gates are executed for this buffer pass, the updated state vector slice is then written back to global memory from the buffer on line 27.

```
1  for p = 0; p < P; p++ do
2     mem_read: buffer[0:L-1] = read_from_mem(from: p*L, count: L);
3
4     for gate_i = 0; gate_i < gate_count; gate_i++ do
5
6        // Compute G_P and I
7        t = target_for_gate_index(gate_i);
8        G_P = 2^(l-t-1)
9        I = 2^t
10       // Select gate matrix
11       g_code = gate_code_for_gate_index(gate_i)
12       mat0, mat1, mat2, mat3 = select_mat(g_code)
13
14       for g = 0; g < G_P; g++ do
15          for i=0; i < I; i++ do
16             in0 = buffer[g*E + i];
17             in1 = buffer[g*E + i + I]
18             out0 = mat0 * in0 + mat1 * in1;
19             out1 = mat2 * in0 + mat3 * in1;
20             buffer[g*E + i] = out0;
21             buffer[g*E + i + I] = out1;
22          endfor
23       endfor
24
25    endfor
26
27    mem_write: write_to_mem(buffer[0:L-1], at: p*L, count: L);
28 endfor
```

Listing 3.10: Gate fusion pseudocode. The code assumes some gate buffer location that stores that fused gate block information and is accessed using functions like `target_for_gate_index` and `gate_code_for_gate_index`.

Figure 3.23 shows an example of two fused gates being applied to a 3-qubit state vector. $G0$ is applied with $t = 1$, and $G1$ is applied with $t = 0$, thus, for $l = 2$, they both satisfy $t < l$ and can be fused. The data flow of the state vector through the buffer is demonstrated in the figure.

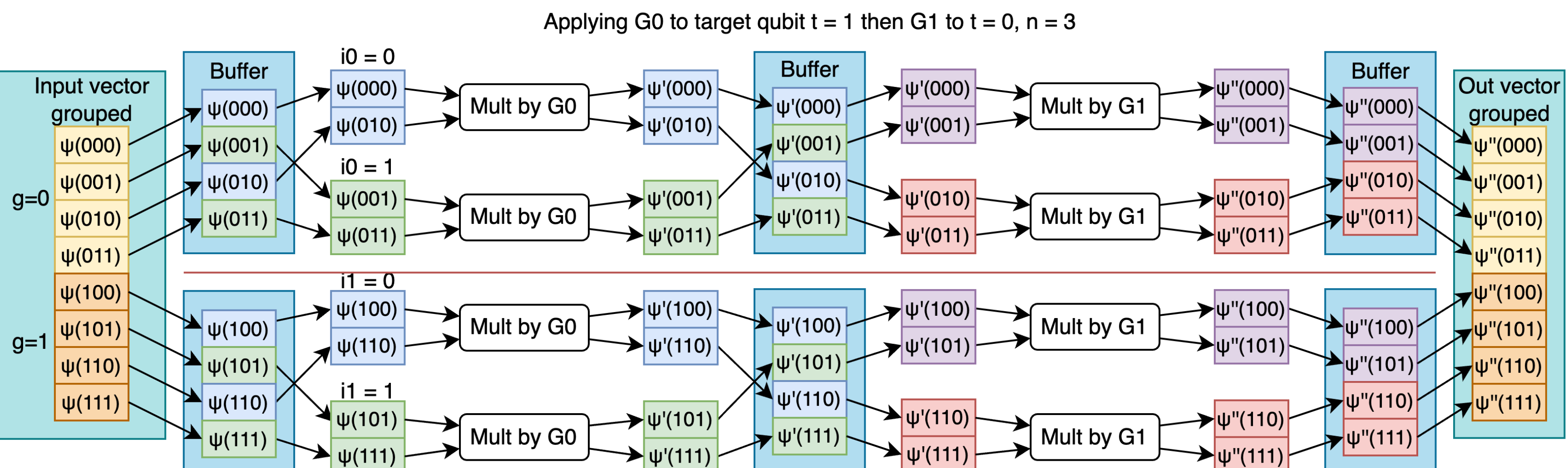


Figure 3.23: Demonstrating application of two quantum gates, with gate fusion, to a 3-qubit quantum register. The targets of both gates are such that they both are Case 1 for a buffered architecture, allowing them to be fused. For a buffer with $l = 2$, the 3-qubit state vector is divided into two buffer passes. Each buffer pass executes two iterations, for each of the fused gates. As before, the red line separates sequentially executing parts of the simulation (there is only one buffer). For each gate, the buffer is read with the same strategy as described for the Case 1 buffered architecture in Section 3.3.2. This allows for both gates to be simulated while performing the same number of memory accesses as needed for one gate.

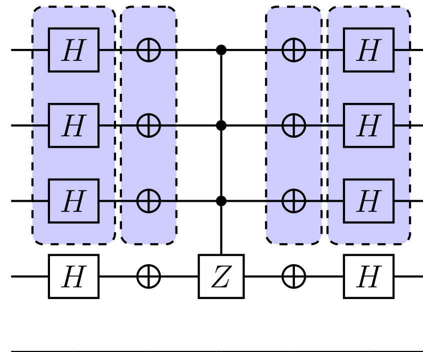


Figure 3.24: Grover's diffusion quantum circuit for a four qubit search register with gate fusion for a buffer qubit size $l = 3$. The blue boxes indicated fused quantum gates.

3.5.1 Grover's diffusion operator example

We take Grover's search circuit for a 4-qubit search register, described in Section 1.3.4. This is a 5-qubit circuit, where the last qubit is the oracle qubit. We focus on the diffusion operator, demonstrated in Figure 1.10.

Consider an example architecture where the buffer can hold 8 amplitudes, i.e. $l = 3$. This parameter enforces the constraint that the maximum target qubit index to fuse consecutively-executing gates is $t = 2$.

Depending on the execution order defined in the circuit description given to the simulation system, there are two ways to apply gate fusion to this circuit, shown in Figures 3.24 and 3.25, where the blue boxes represent fused gate blocks. If all the H gates are to be executed first before the NOT gates, then the gates can be fused as shown in Figure 3.24, creating four gate blocks of three gates each.

Alternatively, if the execution of the NOT gates is interleaved between H gates, then the circuit can be fused as shown in Figure 3.25, creating two gate blocks of six gates each, which would result in more efficiency in terms of global memory access.

If the amplitude buffer on the board were bigger, then we can fuse more gates together. For the example of $l = 4$, we would be able to fuse the entire diffusion operator, as shown in Figure 3.26.

For simulation, another restriction is the allocated size of the gate block buffer used to store the description of the currently-executing gate block. For example if the gate block buffer could only hold the description of four gates per block, then we would have to adhere to this limit when fusing gates. In the example of fused Grover's diffusion circuit ($l = 4$), the allowed fusion for the circuit is shown in Figure 3.27. For the latter part, we are able to fuse the multi-controlled-Z gate with three of the NOT gates, and then a single NOT gate with three of the H gates, leaving a single final H gate to be executed by itself.

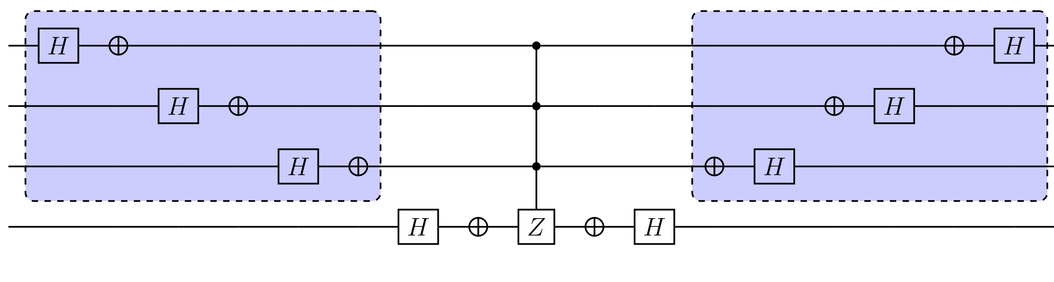


Figure 3.25: Alternative gate fusion ($l = 3$) scheme for Grover's diffusion circuit.

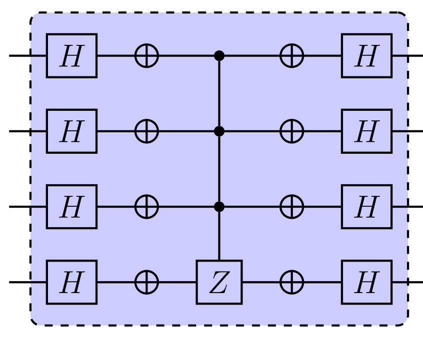


Figure 3.26: With $l = 4$, and no restriction on the number of gates which can be fused, the entire diffusion operator can be fused into a single gate block.

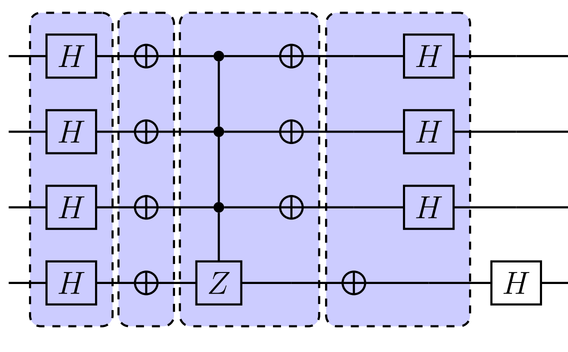


Figure 3.27: Fused gate blocks for $l = 4$ with a restrictive gate block buffer, with a maximum gate count of four per block.

3.5.2 Buffer Case 2 for Gate Fusion

We include here a description of an additional optimisation which may be applied in the context of gate fusion, though it was not implemented in the presented architectures.

While for case 1 gates, we can theoretically fuse as many consecutive gates as occur which all satisfy the case 1 condition, $t < l$; for case 2 gates, the primary barrier to gate fusion is that for gates where $t \geq l$, the amplitude slices in each buffer pass do not intersect between different gates. However, there is one edge case where we can still benefit from gate fusion for case 2 gates: which is is when several case 2 gates operate on the same target qubit consecutively. In this case, the amplitude slices which would be stored in the buffer at each buffer pass for each gate are exactly the same. And so, we can apply several gates on the slices without losing the logical ordering of the gate applications. An example of this is shown in Figure 3.28.

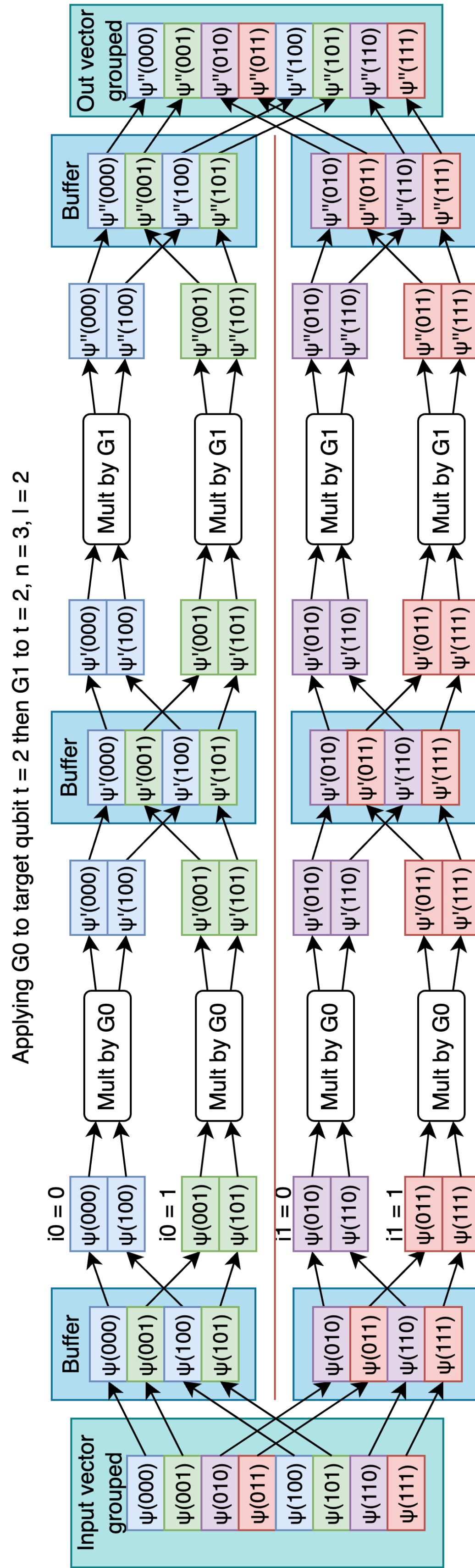


Figure 3.28: Application of two buffer case 2 quantum gates using gate fusion. The condition for fusing case 2 gates is that all gate target qubits must be the same; in this case both gates target the third qubit, $t = 2$. There are still two buffer passes, however each pass now requires two contiguous memory accesses instead of one.

Chapter 4

OpenCL FPGA Implementation of Full State Vector QCS

In this chapter, we present different classes of architectures which were developed to evaluate the simulation techniques presented in Chapter 3. We discuss details of these architectures, including a comparison of NDRange vs Loop-based, the effect of double buffering, how circuit reduction is implemented, and full on-board execution vs. offloading of the circuit gate-by-gate.

We begin by giving an overview of the presented architecture, followed by an introduction to OpenCL and its use for FPGA architecture development, before diving into the details of each architecture.

4.1 Architectures Overview

The architecture for quantum circuit simulation on FPGAs is designed to harness the parallel processing capabilities of these devices to efficiently simulate the behavior of quantum circuits. This architecture typically consists of two main components: the host code, which manages the simulation process and interacts with the FPGA, and the device code, which is executed on the FPGA itself to perform the quantum state transformations.

4.1.1 Quantum Circuit Representation and Execution Flow on FPGAs

Given a quantum circuit description, we have developed a workflow to convert it to an intermediate representation (IR), which the FPGA can process to simulate the quantum circuit.

A toolchain implementing this workflow is described in Section 4.2; the rest of this section describes the FPGA IR and the execution flow of a quantum circuit using our FPGA platform.

As described in Section 2.5, our goals for developing an FPGA simulator for quantum circuits are threefold: *universality* (ability to simulate any theoretical gate), *reuseability* (a recompilation process should not be necessary to simulate different circuits), and *scalability* (our architectures should scale with compute resources).

We achieve universality by ensuring the system has built-in at least a universal set of quantum gates. Our current architecture stores the state vector in the FPGA board's global memory and compute kernels corresponding to each quantum gate access the memory to perform the necessary computations. Since, in general, each gate application needs to access the entire memory space, we perform gate applications sequentially and attempt to optimise the performance of the application of a general gate.

Our reusability goal is achieved with some limitations: our architectures are parameterised by a number of parameters depending on the architecture. One parameter used across all our architectures however is the maximum number of controls allow for a gate in a circuit to be processed. This is to allow for static scheduling of the processing of controlled gates; we always check a fixed number of controls, which is the maximum. With this taken into account, our architectures can generally process circuits of any gate count (as long as the gates are within the supported gate set), of any qubit count (as long as the state vector can fit in the FPGA's DRAM), and up to some maximum number of controls for a given version of an architecture.

We show scalability by demonstrating a benefit from using fine and coarse-grained parallelism, generally through using replication of compute units and complete unrolling of loops (again as replicated hardware modules) on the FPGA. This provides a proof-of-concept for moving to FPGA clusters.

In this FPGA-based quantum circuit simulation architecture, the host code is designed to receive and process quantum circuits in a specialised representation that enables efficient simulation on the hardware. This representation is structured as a series of fixed-length instructions, where each instruction corresponds to a quantum gate operation. The architecture is parameterised by the maximum number of control qubits, denoted as NCONTROLS, that any gate in the circuit can have. Each instruction in the series is composed of 2 + NCONTROLS unsigned integers, which encode the information needed to simulate the corresponding gate on the FPGA.

The first integer in the instruction sequence is the gate code, which identifies the specific quantum gate to be applied. Based on this code, the appropriate gate matrix is selected and loaded onto the FPGA. The second integer represents the target qubit index within the

quantum register, indicating where the gate operation should be applied. The remaining integers are placeholders for control qubits. If a control qubit is used in the operation, its index in the quantum register is placed in the corresponding position. However, if the gate does not require as many controls as the architecture allows, the absence of a control is represented by setting the control placeholder integer to be equal to the target qubit index. This mechanism ensures that the representation remains uniform in length while allowing for flexibility in the number of controls associated with each gate.

Thus, the FPGA platform takes input in the form of an ASCII file containing the IR instructions as integers representing the quantum circuit to be simulated, and is independent of any compilation tool used. In theory, our FPGA platform can act as a backend for any quantum circuit description toolchain. We chose to build our own toolchain (described later in Section 4.2), as part of this work, to facilitate the description of the quantum circuits in a high-level language, and its compilation to this FPGA IR (dubbed QP, short for Quantum Problem). A typical circuit execution flow for the FPGA platform is shown in Figure 4.1.

With the exception of the OnBoardUnrolledLoops architecture, all our architectures are ”gate-by-gate” architectures, i.e. they process the quantum circuit by sending the gates one-by-one to the FPGA for execution. In the case of the gate fusion architectures, gates are replaced by gate blocks in these descriptions. Figure 4.1 shows a high-level description of the flow of these architectures.

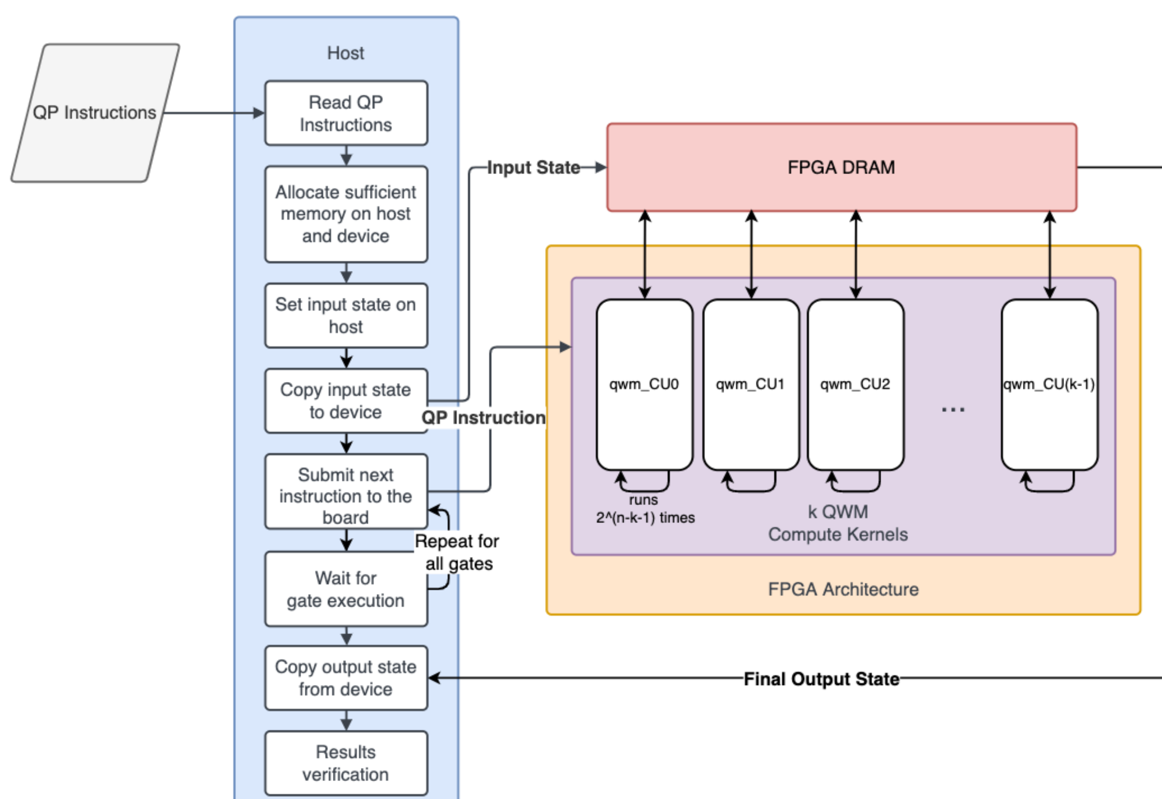


Figure 4.1: QProblem to FPGA flow for a gate-by-gate architecture. At each gate, the QP instruction is sent to the compute units over the host-FPGA PCIe connection.

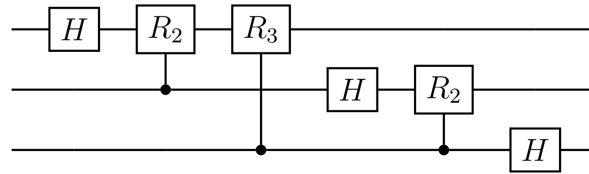


Figure 4.2: 3-qubit QFT circuit used for demonstrating the IR representation for the FPGA platform.

Example Representation: QFT

To illustrate the execution flow of this representation, consider the Quantum Fourier Transform (QFT), a fundamental quantum algorithm. The QFT can be decomposed into a sequence of Hadamard gates and controlled phase shift gates applied to the qubits in the quantum register. The QFT makes use of controlled phase shift gates, R_m, which are typically parameterised by an integer m: $R_m = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{2\pi}{2^m}} \end{bmatrix}$, as explained in Section 1.3.2. To take this parameterisation into account in our representation, this gate is chosen to have a gate code of $1000 + m$.

For example, consider an architecture where NCONTROLS, the maximum number of allowed controls per gate is 2. In a 3-qubit QFT, the sequence of operations would be:

1. Hadamard gate on qubit 0: The gate code for the Hadamard operation is chosen to be 0. The instruction would then be [0, 0, 0, 0], where 0 is the gate code, 0 is the target qubit index, and the remaining integers represent the absence of control qubits.
2. Controlled phase shift gate (R_2) with qubit 0 as target and qubit 1 as control: The corresponding instruction would be [1002, 0, 1, 0], where 1002 is the gate code, 0 is the target qubit, 0 is the control qubit index, and the last integer indicates that there are no additional control qubits.
3. Controlled phase shift gate (R_3) with qubit 0 as target and qubit 2 as control: The corresponding instruction would be [1003, 0, 2, 0], where 1003 is the gate code, 0 is the target qubit, 2 is the control qubit index, and the last integer indicates that there are no additional control qubits.
4. Hadamard gate on qubit 1: This operation would be represented by the instruction [0, 1, 1, 1], where the control placeholders indicate no controls.
5. Controlled phase shift gate with qubit 2 as control and qubit 1 as target: The instruction would be [1002, 1, 2, 1], indicating a phase shift gate between qubits 1 and 2.
6. Hadamard gate on qubit 2: The corresponding instruction would be [0, 2, 2, 2].

The resulting IR code is shown in Listing 4.1.

```
1 3 0 0 0 0 1002 0 1 0 1003 0 2 0 0 1 1 1 1002 1 2 1 0 2 2 2
```

Listing 4.1: FPGA IR representation of the 3-qubit QFT.

The eDSL function which generates this QP code is shown in Listing 4.2.

```
1 qft3 :: Qu -> Qu -> Qu -> Circ
2 qft3 q0 q1 q2 =
3    h q0 ++
4    control q1 (rm 2 q0) ++
5    control q2 (rm 3 q0) ++
6    h q1 ++
7    control q2 (rm 2 q1) ++
8    h q2
```

Listing 4.2: eDSL code for the 3-qubit QFT circuit shown in Figure 4.2 expressed in the Haskell toolchain.

Since this is an embedded language in Haskell (described in more detail in Section 4.2), we have full access to all the Haskell constructs, allowing us to define high-level circuit generators. For example, Listing 4.3 constructs an n-qubit QFT operating over the provided quantum register, independent of its size.

```
1 qft :: NQuGate
2 qft qReg = let n = length qReg in
3    concat $ concat [h (qReg!!i) : [control (qReg!!j) (rm (j-i+1)
       (qReg!!i)) | j <- [i+1..n-1]] | i <- [0..n-1]]
```

Listing 4.3: n-qubit QFT eDSL code in the Haskell toolchain. NQuGate is a type defintion for a function which takes a quantum register (as a list of qubit identifiers) and returns a Circ.

Once the host code reads the quantum circuit in this format from disk, it begins by initialising the FPGA, setting up the necessary memory buffers, and transferring the initial quantum state vector to the device. The circuit's instruction list is then parsed, and the host code iteratively sends the gate instructions to the FPGA. For each instruction, the host code extracts the gate code and the target and control qubits, configures the FPGA kernel with the appropriate gate matrix and qubit indices, and then enqueues the operation for execution. After all the gates are executed, the host reads back the resulting state vector from the global memory buffer of the FPGA, for verification and further processing if necessary. The is the flow shown in Figure 4.1.

4.2 Quantum Circuit Description to FPGA IR

Debugging complex quantum circuits at the level of this FPGA instruction set (or IR) can be very tedious, and so several higher level languages exist for expressing quantum algorithms, including Quipper [122], OpenQASM 3 [87, 88], and Microsoft Q# [123]. We decided to include a custom eDSL with our toolchain to maintain end-to-end control of the compilation system, and facilitate future development of architecture-specific optimisations in the instruction set. However, it is certainly feasible to implement frontends for these already existing high-level languages, which would allow for a tighter integration with the current ecosystem. Our toolchain is described in more detail in [124].

4.2.1 Core

The Core modules of the toolchain provide the constructs used in the specification of a quantum circuit. This includes primary definitions for types used throughout the tool, and an inner Circuit type to represent a circuit over an indexed quantum register (i.e. very close to what the FPGA will actually process). On top of this Circuit type, the eDSL constructs are defined. This includes utilities for referring to qubits by names instead of indices (essentially defining arbitrary ”quantum pointers”), arbitrary controls and anti-controls defined over a gate or a set of gates, circuit chaining, looping and tiling subcircuits, etc. A demonstrative example circuit definition for a generic Cuccaro full adder [2] is shown in Listing 4.4.

```
1 fullAdd :: QReg -> QReg -> Qu -> Qu -> Circ
2  fullAdd in1 in2 c z = if length in1 /= length in2 then error "fullAdd:
      Input qubit register lengths must be identical." else let
3   combinedRegister = c : interleave in2 in1
4   in
5     ladderQC 2 3 maj combinedRegister ++
6     cnot (last in1) z ++
7     reverseLadderQC 2 3 unmaj combinedRegister
```

Listing 4.4: Generic input size full Cuccaro adder example implementation in the presented Haskell eDSL.

The Core modules also include two simulators, implemented in Haskell, for convenience. One is a general full state-vector QWM simulator with no optimisations, which was used to model early QWM-based simulators. Additionally, a logic-based simulator is also provided which can simulate circuits containing only the NOT gate with any number of controls; this is useful for quickly debugging circuits which operate only in the computational basis. Since only one state is set at any point in the circuit, simulating such circuits can be performed in linear time and memory.

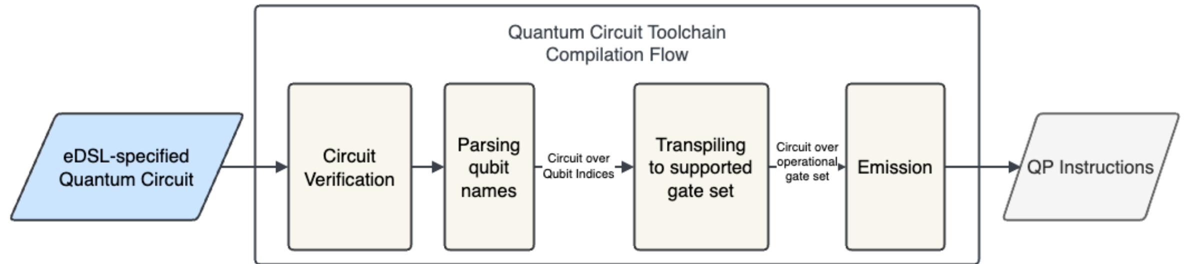


Figure 4.3: eDSL to FPGA intermediate representation.

4.2.2 Testing

As is the case with classical software, effective unit testing of quantum circuits is very important. This involves running quantum circuits with different input states and checking that the output states fit some expectations. To facilitate automating this task, a testing framework is provided with the toolchain. Currently, the side of this which interacts with the FPGA simulator takes qubit preparations which it uses to compute the input state of the circuit. The circuit is then simulated by the FPGA and the output state is passed back into the toolchain (currently this is manual but automation is planned), decoded, and checked against the test expectations. There is also an option to run the tests against the included logic simulator, which proved particularly useful for debugging complex computational-basis circuits.

4.2.3 Compiler

The compiler modules include functions and tools for going from the eDSL representation to the FPGA instructions. Alternatively, the compiler can also read a QASM-like file specifying the circuit.

The compilation process is demonstrated in Figure 4.3. First, the specified circuit is verified, ensuring all qubits used are valid (have an index in the register) and no gates are specified with invalid target/controls. Then the named qubit identifiers are parsed away, and the qubits are mapped to an index in the quantum register. At this point in the process, some constructs are still available to the tool which would not necessarily be available to the FPGA (like direct calls to a SWAP gate, or a high number of controls), which need to be reduced away. SWAP gates are replaced with their equivalent CNOT specifications, negative controls are reduced by negating the control qubits before and after the gate, and gates with a higher number of controls than supported are expanded to several gates with fewer controls. This results in a circuit which is ready to be converted to a QP (Quantum Problem) file which is simply the list of integers specifying the circuit, described above in Section 4.1.1. Taking into account the maximum number of controls allowed by the architecture, each emitted gate consists of its opcode, target qubit, followed by a constant number of controls. The resulting

list is then written to disk, ready to be read by the simulator host.

4.2.4 Circuit Qubit Reduction

Quantum circuits representing quantum algorithms which employ computational-basis encoding can have some qubits reduced out by generating two different circuits for each possible qubit input. In this way, the total memory required for one run of the circuit is reduced by half for each qubit reduced out. Qubits which are only used as controls throughout the circuit are ideal candidates for this type of reduction, and the toolchain provides functionality to automate this. While this approach is useful for reducing the total memory required across any platform, it is especially good for an FPGA which would, in theory, be able to run both (for one reduced qubit) circuits concurrently on completely independent memory spaces, which improves data locality. This circuit reduction technique was the subject of the works published as part of this PhD work [56, 125], and is described in detail in Chapter 5.

4.3 OpenCL

OpenCL (Open Computing Language) is an open, royalty-free standard for parallel programming of heterogeneous systems that encompass CPUs, GPUs, FPGAs, and other processors. It provides a framework for writing programs that execute across these diverse platforms, offering a unified programming model that abstracts the hardware specifics, thereby allowing developers to write portable, high-performance code. OpenCL consists of an API for coordinating parallel computation across heterogeneous platforms, and a cross-platform programming language that enables the development of kernels to execute on these devices. The OpenCL spec by the Khronos group [4] defines the OpenCL APIs.

The adoption of OpenCL for FPGA programming brings several advantages. First, it abstracts the underlying hardware complexities, allowing developers to focus on optimising algorithms rather than dealing with low-level hardware details. This abstraction layer enables the development of portable code, which can be executed on different FPGA platforms with minimal modifications. Second, OpenCL facilitates the efficient utilisation of FPGA resources by enabling fine-grained control over hardware resources and memory hierarchy, which is critical for achieving optimal performance in quantum circuit simulation tasks.

In the rest of this chapter, the implementation of various architectures for simulating quantum circuits on FPGAs using OpenCL is explored. These architectures leverage the parallel processing capabilities of FPGAs to simulate the complex operations of quantum circuits efficiently. By utilising OpenCL, the design and development process is streamlined, allowing for rapid prototyping and iterative optimisation of the simulation algorithms. The

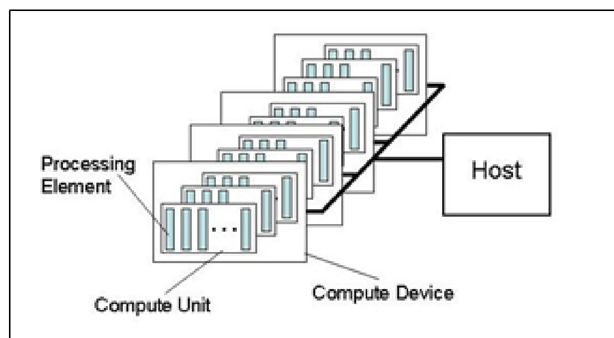


Figure 4.4: OpenCL platform model. From [4].

research presented here demonstrates the potential of OpenCL in bridging the gap between high-level algorithm design and low-level hardware implementation, thereby contributing to the advancement of quantum computing simulation on reconfigurable hardware platforms.

4.3.1 OpenCL Programming Model

The OpenCL programming model is designed to facilitate parallel computing across heterogeneous platforms, including CPUs, GPUs, DSPs, and other processors. It provides a standardised approach for writing programs that can execute on diverse hardware architectures. The core components of the OpenCL programming model encompass the platform model, execution model, memory model, and programming model, each playing a vital role in harnessing the capabilities of various devices.

The platform model in OpenCL consists of a host and one or more OpenCL devices. The host, typically a CPU, runs the main application and is responsible for managing the execution environment, including the allocation and deallocation of resources. Devices can be CPUs, GPUs, or other processors that execute OpenCL kernels. Each device comprises multiple compute units, which in turn consist of processing elements. Figure 4.4 shows a graphical demonstration of the OpenCL platform model.

The execution model, demonstrated in Figure 4.5 defines how kernels are executed on devices. A kernel, written in OpenCL C, is the fundamental unit of computation executed on the OpenCL device. Commands for execution, such as kernel execution and memory operations, are submitted to command queues. Each device can have one or more command queues. Kernels are executed over an N-dimensional range (NDRange) of work-items, defining the global dimensions of the problem space. The NDRange is divided into work-items, the smallest units of execution, which are further grouped into work-groups. Work-groups are executed independently and may synchronise within themselves. An example of the mapping of these work-items into work-groups and their indexing is shown in Figure 4.6.

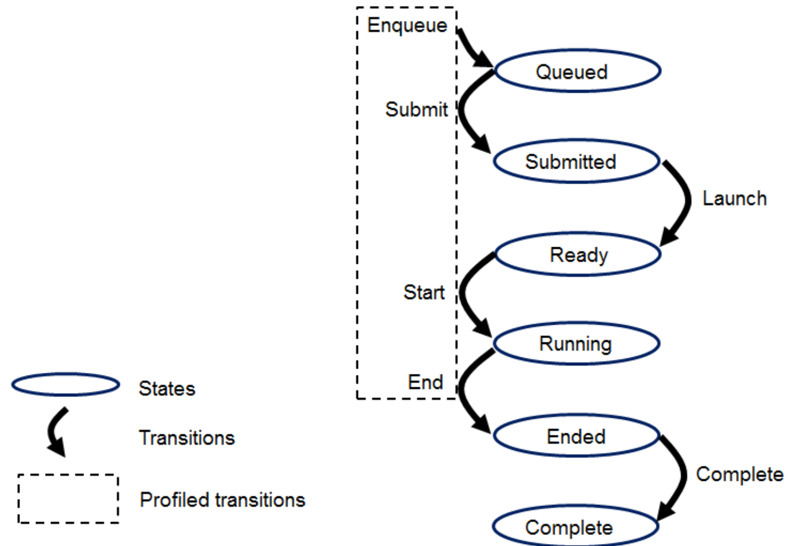


Figure 4.5: OpenCL execution model [4].

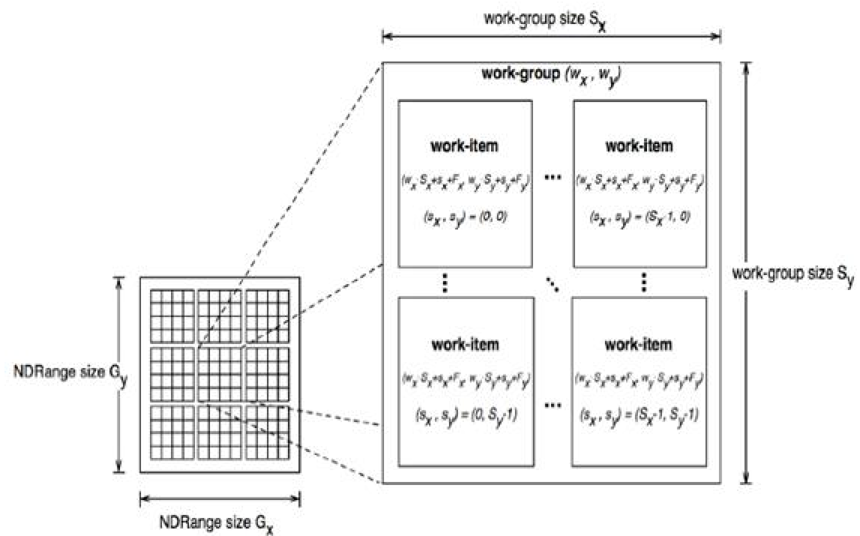


Figure 4.6: OpenCL NDRange kernels work-items indexing and mapping example. From [4].

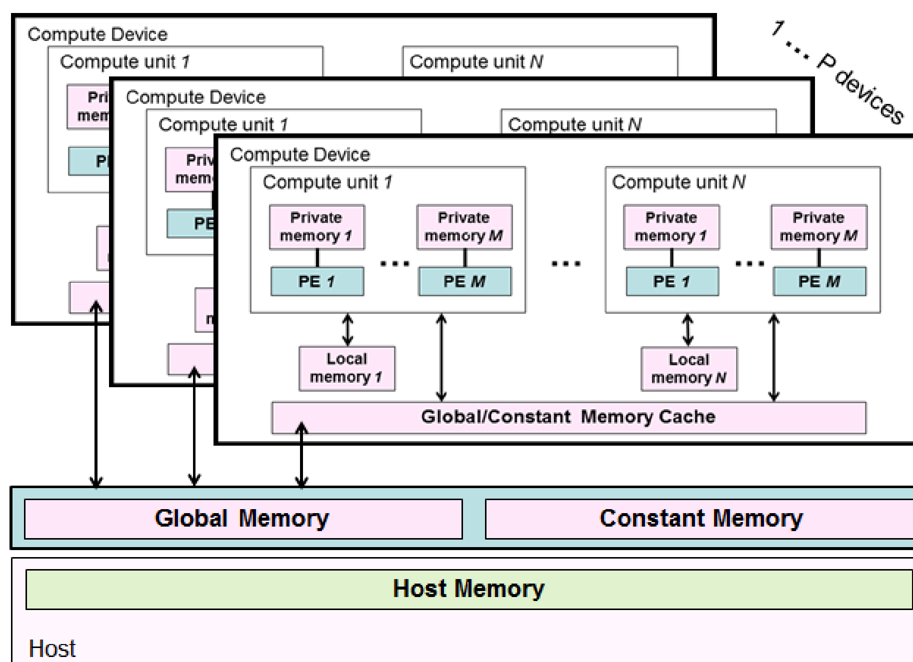


Figure 4.7: OpenCL memory model [4], showing the differences between, global, constant, local, and private memory regions.

The memory model outlines the types of memory available and how data is shared between the host and devices. **Global memory** is accessible by all work-items and the host, characterised by high latency and lack of caching. **Constant memory** is a read-only region of global memory that remains constant during kernel execution. **Local memory** is shared among work-items within a single work-group, offering lower latency compared to global memory. **Private memory** is exclusive to each work-item, typically stored in registers or on-chip memory. This model is shown in Figure 4.7.

The programming model includes the API and language constructs used to develop OpenCL applications. Kernels are written in OpenCL C, a subset of C99 with extensions for parallelism, and are compiled at runtime or offline. The host program utilises the OpenCL API to manage devices, create contexts, build programs, and submit kernels for execution. This model allows for the efficient and flexible development of parallel programs.

An example workflow in OpenCL begins with the initialisation phase, where the platform and device are queried and selected, and a context is created. Following this, programs and kernels are created, where kernels are written in OpenCL C, compiled, and kernel objects are instantiated. Memory management involves allocating and transferring data between the host and device memory. During the execution phase, kernel arguments are set, kernels are enqueued for execution on the device, and synchronisation is managed. Finally, the cleanup phase involves releasing resources, including memory objects, kernels, and the context.

4.3.2 OpenCL for Intel-based FPGAs

Building upon the foundational understanding of OpenCL and its applicability to FPGA-based quantum circuit simulation, it is essential to delve into the specific optimisations and specialisations required for Intel FPGAs. The Intel FPGA SDK for OpenCL [5, 126] offers a comprehensive suite of tools and guidelines designed to maximise performance and efficiency when deploying OpenCL applications on Intel's FPGA architecture. This section explores these strategies in detail.

Efficient memory access is another important factor in achieving high performance on FPGAs. One effective strategy is kernel unification, where kernels that produce and consume data are combined into a single kernel. This approach eliminates the need to store intermediate results in global memory, significantly reducing memory access overhead and enhancing overall performance. Additionally, while traditional GPU optimisations often involve avoiding local memory bank conflicts, Intel's FPGA compiler automatically generates hardware to manage these conflicts, allowing developers to simplify their code. Optimising global memory accesses is also essential for high-performance FPGA implementations. The default burst-interleaved configuration for global memory on Intel FPGAs is designed to balance load across memory banks effectively.

By adhering to these best practises and using the specialised tools and options provided by the Intel FPGA SDK for OpenCL, developers can significantly enhance the performance and efficiency of their OpenCL applications on Intel FPGAs. These optimisations are important in the context of quantum circuit simulation, where the parallel nature of quantum operations requires careful utilisation of compute resources to achieve a high level of efficiency and maximum possible throughput.

Note that while the Intel platform for FPGAs was chosen as the target for deploying our architectures in this study, the architectures and optimisation techniques described in this chapter are certainly not limited to this platform. The Xilinx HLS platforms, Vivado and Vitis, allow for implementing the same architectures for deployment on Xilinx boards. Intel's platform was chosen purely out of convenience as it was what was most easily accessible for conducting this research.

4.3.3 FPGA OpenCL Model

Using the OpenCL programming model on FPGAs involves several unique steps and considerations due to the inherent differences between FPGAs and other processors like CPUs and GPUs. FPGAs are reconfigurable hardware devices that offer fine-grained parallelism and can be tailored to specific computational tasks, providing high performance and efficiency

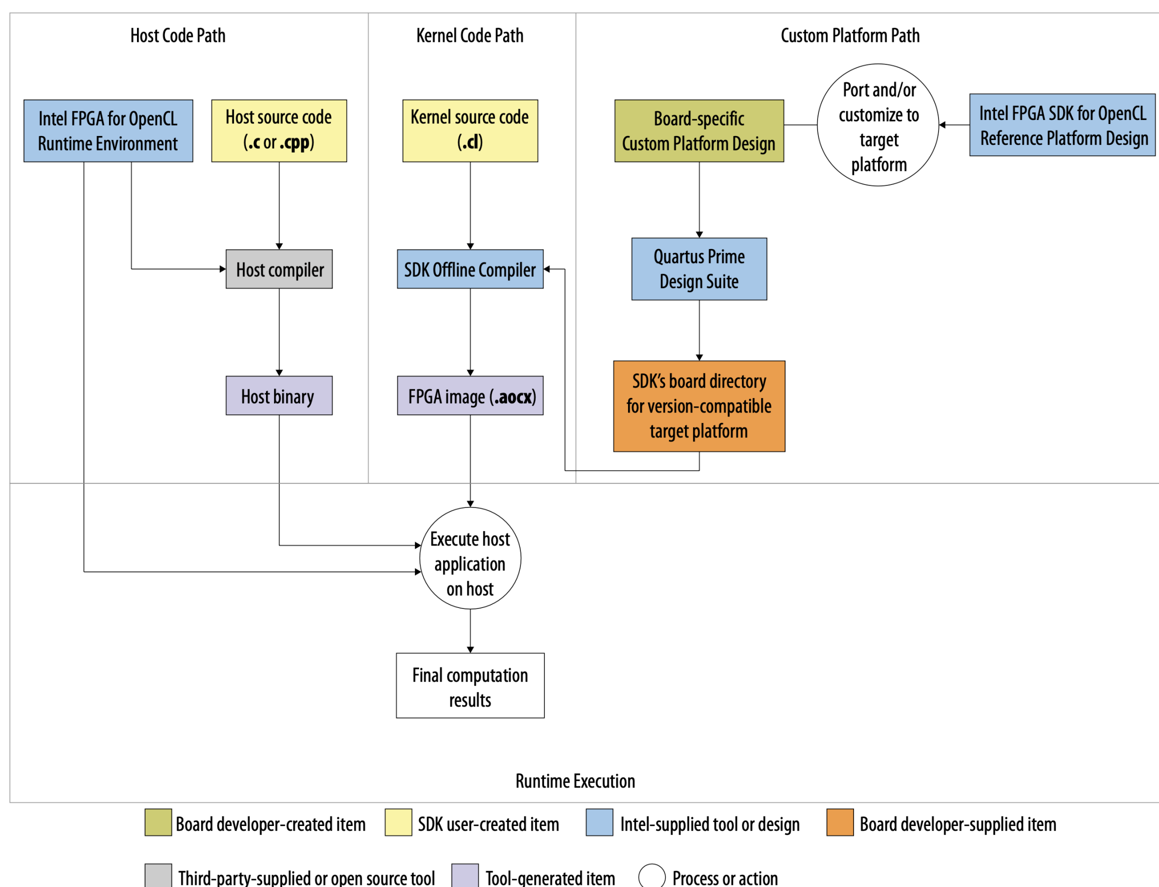


Figure 4.8: Intel SDK for FPGAs programming flow. From [5].

for specialised applications such as quantum circuit simulation. A high-level diagram of the programming flow using the Intel SDK for FPGAs [5] is shown in Figure 4.8.

Platform Model on FPGAs

In the context of FPGAs, the platform model remains consistent with the general OpenCL model but involves additional steps for hardware configuration:

- Host: The host remains responsible for managing the execution environment, including communication with the FPGA, initialising the OpenCL runtime, and handling data transfer.
- FPGA Device: The FPGA acts as the OpenCL device, where the kernels are mapped onto its reconfigurable fabric. This requires generating a hardware bitstream that configures the FPGA to execute the specific OpenCL kernels.

Execution Model

The execution model for FPGAs adapts to their reconfigurable nature:

- Kernel Execution: Kernels are written in OpenCL C, similar to other platforms, but are compiled into a hardware description language (HDL) like VHDL or Verilog during the synthesis process. This HDL is then used to generate the FPGA configuration bitstream.
- Command Queue: The host submits commands to the FPGA's command queue, which orchestrates the execution of kernels and manages memory operations.
- NDRange and Work-Groups: The mapping of the NDRange and work-groups to FPGA resources is a critical aspect. Unlike CPUs or GPUs, where work-items and work-groups are managed by fixed hardware resources, FPGAs allow custom hardware pipelines to be created for these work-items, providing parallelism and pipelining at the hardware level.

Memory Model on FPGAs

The memory hierarchy on FPGAs differs significantly from traditional processors, and is summarised by the following:

- Global Memory: On FPGAs, global memory typically maps to external DRAM. This memory is used for large datasets and is accessible by all work-items, but has higher latency compared to on-chip memory.
- Constant Memory: Similar to other platforms, constant memory on FPGAs is a read-only segment of global memory.
- Local Memory: FPGAs provide on-chip memory blocks (such as BRAMs) that can be used as local memory, offering lower latency and higher bandwidth compared to global memory. These memory blocks are shared among work-items within a work-group.
- Private Memory: Private memory for individual work-items is implemented using registers or small on-chip memory segments, providing the lowest latency.

Programming Model on FPGAs

Developing OpenCL applications for FPGAs involves additional tools and steps:

- Kernel Development: Kernels are written in OpenCL C, focusing on maximising parallelism and minimising dependencies to leverage the FPGA's parallel processing capabilities.

- Compilation: The OpenCL kernels are compiled into an intermediate representation, which is then synthesised into HDL. This synthesis process involves optimisations specific to the FPGA architecture, such as loop unrolling and pipelining.
- Hardware Generation: The synthesised HDL is used to generate the FPGA bitstream, which configures the FPGA hardware to execute the kernels. This step is time-consuming compared to compiling for CPUs or GPUs.
- Host API: The host program uses the OpenCL API to manage the FPGA device, including loading the bitstream, allocating memory, transferring data, and launching kernels. The API abstracts the complexity of managing the FPGA hardware.

Key Considerations for FPGAs

- Optimisation: FPGA kernels must be carefully optimised to exploit the hardware's parallelism. This involves techniques like loop unrolling, pipelining, and efficient memory access patterns.
- Resource Management: Efficient use of FPGA resources (e.g., logic elements, memory blocks) is crucial. Developers must balance computational resources and memory bandwidth to avoid bottlenecks.
- Compilation Time: The synthesis and bitstream generation process can be time-consuming, requiring several hours for complex kernels. This necessitates thorough testing and iterative optimisation.

By leveraging OpenCL for FPGA development, developers can harness the FPGA's parallel processing capabilities while benefiting from a standardised programming model. This is particularly valuable for computationally intensive applications like quantum circuit simulation, where the FPGA's ability to be customised for specific tasks can lead to significant performance improvements.

4.4 Architectures Implementation Overview

4.4.1 Host Overview

The host code serves as the central controller of the simulation, responsible for initialising the FPGA environment, setting up the simulation parameters, managing data transfers, and coordinating the execution of quantum gate operations. It begins by setting up the necessary

computational context, including the creation of buffers for data storage and the compilation of kernels for execution on the FPGA. Depending on the specific implementation, the host code may also handle the loading and parsing of quantum circuit descriptions, which typically consist of a sequence of quantum gates and their associated parameters.

Once the initialisation phase is complete, the host code prepares the quantum state vector, based on an initial state provided by the user. In theory, this initial state can be any normalised 2^n-dimensional vector which can be read from a file. However, the most commonly used paradigm in our experiments is to specify this initial state as an integer at runtime (through a command line argument for example), such that the corresponding amplitude is set to 1 in the state vector. This state vector is then transferred from the host to the FPGA's memory, ensuring that the device code has access to the initial quantum state. The host code then enters a loop where it iterates through the quantum gates specified in the circuit description. For each gate, the relevant parameters—such as the target qubit, control qubits, and gate matrix—are passed to the FPGA, where the device code performs the required state transformations.

In terms of synchronisation, in the case of an Over-PCIe gate-by-gate architecture, the host code waits for the FPGA to complete each gate operation before moving on to the next, ensuring that the gates of the quantum circuit are simulated sequentially. After all gates have been processed, the final quantum state is read back from the FPGA to the host, where it can be analysed and verified, stored, or used for further computation. In other cases (e.g. On-board and Gate Fusion architectures), gate communication is different and detailed later in Section 4.11.

4.4.2 Device Overview

The device code, or kernel, is the component that executes directly on the FPGA and performs the actual quantum state transformations. The design of the device code can vary significantly depending on the specific requirements of the simulation and the capabilities of the FPGA. There are two general types of kernels used in our architectures: NDRange kernels and single-task kernels.

NDRange kernels operate over a range of data elements in parallel, with each work-item processing a specific portion of the quantum state vector. This approach is well-suited for FPGAs because it allows for a high degree of parallelism, enabling the simultaneous execution of multiple quantum gate operations. NDRange kernels are typically designed to take advantage of the FPGA's pipeline structure, which can process multiple operations concurrently, further enhancing performance.

In contrast, single-task kernels focus on processing one task at a time, often using a loop-

based approach. This design is more sequential but can be advantageous in scenarios where fine-grained control over the execution is needed. Single-task kernels may also be used when explicit buffering is required, allowing the FPGA to manage data more effectively.

The device code is responsible for reading the quantum state from the FPGA's memory, performing the necessary transformations based on the gate parameters, and writing the updated state back to memory. These transformations typically involve complex arithmetic operations, such as matrix-vector multiplications, which are fundamental to simulating the effects of quantum gates. The efficiency of the device code is the most important factor to the overall performance of the simulation, and various optimisation techniques, such as loop unrolling, pipelining, vectorisation, and parallel execution, are often employed to maximise throughput.

Parallelism through Compute Units

In general, the primary method to achieve parallelism in this context is by replication computational elements in hardware on the FPGA. This can be implemented in two different ways: full kernel replication (coarse-grained parallelism, through the `num_compute_units` OpenCL attribute), and loop unrolling (fine-grained parallelism). The architectures we explore here make use of and contrast both approaches.

Generally, parallelism in NDRange kernel approaches is achieved through full kernel replication and distribution of work groups across different compute units; whereas in the single-task kernel approach, parallelism is achieved by unrolling critical high-compute loops.

Core Kernel Computation

Each kernel invocation is responsible for processing a specific quantum gate across multiple iterations. For each gate, the kernel operates over a series of iterations, each of which corresponds to a unique set of qubit states. The iteration indices, which are either derived from an NDRange global ID or a loop index in a single-task kernel, determine which amplitudes in the state vector are being modified. These iteration indices are essential because they guide the kernel in selecting the appropriate elements of the state vector to apply the quantum gate operation.

Iteration and Amplitude Index Calculation Given an iteration index i, the kernel calculates two critical indices in the state vector: the "zero state" and the "one state." The "zero state" index is determined by the function `ithCleared(i, t)`, where t represents the target qubit for the gate operation. This function clears the t-th bit in the binary representation of i, effectively calculating the index of the state where the target qubit is in the $|0\rangle$ state. The

"one state" index is then obtained by setting the t-th bit of the "zero state" index, corresponding to the state where the target qubit is in the $|1\rangle$ state. These indices allow the kernel to identify the specific amplitudes in the state vector that are influenced by the quantum gate, corresponding to this iteration. Listing 4.5 demonstrates this.

```
1 uint i = get_global_id(0); // Alternatively, the iteration index i could
     be a loop index in a single-task kernel
2 int zero_state = ithCleared(i, t);
3 int one_state = zero_state | (1 << t);
```

Listing 4.5: Computation of indices used in direct iteration processing through the global work item ID of the NDRange kernel and the `ithCleared()` function.

Control Qubit Processing After determining the relevant indices, the kernel evaluates whether the current iteration should proceed with the gate operation by processing the control qubits. Controls are additional qubits that must satisfy certain conditions for the gate to be applied. For each control qubit, the kernel checks whether it is distinct from the target qubit and whether it is set to 1 in the current state. Listing 4.6 demonstrates how this check is computed

```
1  bool perform = true;
2  int cond = c0 != t;
3  perform &= (cond && ((1 << c0) & zero_state) > 0) || (!cond && perform);
4  #if NCONTROLS > 1
5  cond = c1 != t;
6  perform &= (cond && ((1 << c1) & zero_state) > 0) || (!cond && perform);
7  #endif
8  #if NCONTROLS > 2
9  cond = c2 != t;
10 perform &= (cond && ((1 << c2) & zero_state) > 0) || (!cond && perform);
11 #endif
12 ...
```

Listing 4.6: Device code for processing controls in an iteration of DIP. At the end of the block, the `perform` flag determines whether the iteration meets the conditions imposed by the controls.

To perform this check, the kernel examines the binary representation of the "zero state" index using bitwise operations. If a control qubit is set in the current state (i.e., its corresponding bit in the "zero state" index is 1), and it differs from the target qubit, the condition for that control is met. The kernel then logically combines the results of these checks for all control qubits using an AND operation. This cumulative condition, stored in the variable `perform`, determines whether the gate operation should be executed for the current iteration.

Gate Application If the `perform` condition is satisfied, the kernel proceeds to apply the quantum gate to the selected state vector amplitudes. The gate operation involves matrix-vector multiplication, where the kernel multiplies the gate matrix by the input vector formed by the amplitudes at the ”zero state” and ”one state” indices. The resulting output vector is then written back to the state vector, replacing the original amplitudes at these indices. This operation effectively transforms the quantum state according to the gate being simulated. This core gate application is shown in Listing 4.7, where a `cpair` is simply a struct containing two complex floats and `cdot` computes the dot product of two `cpair`s.

```
1  if(perform) {
2     cpair inVec;
3     // Read from memory
4     inVec.a = state[zero_state];
5     inVec.b = state[one_state];
6
7     // Compute and write to memory
8     state[zero_state] = cdot((cpair){mat0, mat1}, inVec);
9     state[one_state] = cdot((cpair){mat2, mat3}, inVec);
10 }
```

Listing 4.7: Core on-device computation for updating an amplitude pair in an iteration, provided a `perform` flag. The `cdot()` function compute the complex dot product of two `cpair`s.

This control flow ensures that the quantum gate is only applied when the necessary conditions are met, which is critical for accurately simulating controlled operations in quantum circuits. The kernel’s ability to selectively execute the gate based on control qubits allows it to accurately simulate the complex behavior of quantum circuits with conditional operations.

4.4.3 Integrating the overall system

The integration of the host and device code is a critical aspect of the quantum circuit simulation architecture. This integration ensures that the FPGA can efficiently execute the quantum state transformations while the host manages the overall simulation process. The host code must carefully manage data transfers between the host and the FPGA, ensuring that the device code has timely access to the quantum state and gate parameters.

In some implementations, the integration may include explicit buffering strategies, where intermediate quantum states are stored in buffers on the FPGA, reducing the need for frequent data transfers (in particular in Gate Fusion approaches). This approach can significantly improve performance by minimising communication overhead and allowing the FPGA to operate more independently from the host.

4.4.4 Summary of Architectures

In the following, the different developed architectures are itemised and their key features are highlighted.

- **Direct Iteration Processing (DIP)**
  - BaselineNDRange: Baseline implementation that uses an OpenCL NDRange to schedule the iterations of each gate in Direct Iteration Processing.
  - UnrolledLoops: Schedules a single task kernel for each gate instead of the NDRange and implements parallelisation using loop unrolling.
  - OnBoardUnrolledLoops: Variation which sends the whole quantum problem to the device's global memory.
  - TwoCircuitNDRange: Variation on the BaselineNDRange architecture which allows for two circuits to be simulated in one kernel call; developed mainly to target the circuit width reduction techniques described in the upcoming Chapter 5.
- **OptimisedControlsNDRange**: Significant variant of the baseline NDRange-based approach which implements the controls scheduling optimisation described in Section 3.1.3.
- **Buffered Architectures**
  - SingleBuffered: Adds a buffer to the UnrolledLoops architecture to implement the buffering optimisation described in Section 3.3.
  - DoubleBuffered: Variation of the buffered architecture which uses two buffers for input and output state vector slices.
- **Gate Fusion Architectures**
  - SingleBufferedGateFusion: Adds the gate fusion optimisation to the buffered architecture, as described in Section 3.5. The primary difference is the device now receives a gate block description in global memory instead of the gate parameters as kernel arguments.
  - DoubleBufferedGateFusion: Variation of the buffered gate fusion architecture that implements double buffering.

During the make step, our platform parameterises the architectures with some compile-time constants. Every architecture is parameterised by at least the following two parameters:

NCONTROLS (also refered to as NC later), the maximum number of controls allowed for any gate in a circuit, and the NCU, the number of compute units that the HLS compiler will attempt to instantiate.

The buffered architectures (with and without gate fusion) have an additional parameter: BQS, the buffer qubit size. For these architectures, the NCU is determined by the BQS, and so specifying the BQS is sufficient and the NCU is computed at compile time during the make step. Finally, the gate fusion architectures additionally have GBGC as a parameter, the gate block gate count; this is the maximum number of gates allowed to be fused in a gate block. This is a limitation imposed because the gate block must be read onto a finite on-board buffer to allow for fast access during the iterations of the gate block execution.

The rest of this chapter details these different architecture implementations which were developed during this work.

4.5 Baseline NDRange Architecture

This section provides a description of a simple baseline implementation for an FPGA OpenCL-based quantum circuit simulator, including both the host and device code. This architecture forms the starting point upon all other architectures build/iterate. The host code orchestrates the execution of the simulation, while the device code (kernel) performs the actual quantum state transformations. Together, these components leverage the parallel processing capabilities of FPGAs to achieve efficient simulation of quantum circuits.

4.5.1 Host Code Description

The host code is responsible for initialising the simulation environment, managing data transfers between the host and the FPGA, and coordinating the execution of quantum gate operations. It begins by parsing command-line arguments to obtain the necessary parameters, such as the quantum problem file, the initial state index, and the output options. It then constructs the file paths for the kernel and quantum problem files, setting up the environment for the simulation.

Initialisation of the FPGA environment is handled by creating a custom context struct, which includes the OpenCL context, kernel, and command queue. This context manages the interactions with the FPGA device. The host code then proceeds to read the quantum problem file, which contains the quantum gates and their parameters, and prepares the state vector. The state vector is initialised with complex floating-point numbers (cfloat) in the host's memory. The initial value of the input state vector is set with a command line argument, either as a single unit value defined by an input state vector index, or if a file path is provided, the host

will attempt to read the input state vector from disk. The file should have the complex numbers defining the amplitudes as tuples in the form (x, y), separated by commas, and contain exactly as many amplitudes as required to define the n-qubit state required by the circuit to be processed; otherwise an error is thrown and the process exits.

Next, the host code allocates device memory buffers and transfers the initial state to the FPGA. This involves creating a buffer for the state vector and copying the data from the host to the device memory. The cl::Buffer object is used for this purpose, ensuring that the state vector is accessible to the kernel during execution.

The core of the host code lies in the execution loop, where it iterates through the quantum gates specified in the problem file. For each gate, the relevant parameters (target qubit, control qubits, and matrix elements) are extracted and set as kernel arguments. The kernel is then enqueued for execution on the FPGA, and the host code waits for its completion before proceeding to the next gate. This process ensures that each quantum gate operation is performed sequentially, respecting the dependencies between gates.

After all gates have been executed, the host code reads the final state vector back from the device to the host memory. The timing results are then written to a CSV file for further analysis, and the state vector is optionally logged to a file.

Finally, the host code measures and reports the total execution time, including the time spent on initial memory transfers, gate executions, and final memory transfers. This detailed timing information is required for evaluating the performance of the simulation.

4.5.2 Device Code (Kernel) Description

```
1  #include "device_header.h"
2
3  __attribute__((num_compute_units(NCU)))
4  __kernel void device_kernel(
5     __global cfloat * restrict state,
6     uint t,
7     #include "controls_as_args.h"
8     cfloat mat0, cfloat mat1, cfloat mat2, cfloat mat3) {
9
10    uint i = get_global_id(0);
11    int zero_state = ithCleared(i, t);
12    int one_state = zero_state | (1 << t);
13
14    #include "controls_check.h"
15
16    #include "baseline_compute.h"
17 }
```

Listing 4.8: Baseline device code for FSVQCS based on an NDRange kernel implementation.

The device code (shown in Listing 4.8), implemented as an NDRange kernel, performs the quantum state transformations required by the simulation. It is designed to operate on a global range, with each work-item processing a specific portion of the state vector based on the target qubit. The kernel takes several parameters, including the state vector, target qubit, control qubits, and matrix elements representing the quantum gate.

The kernel begins by calculating the indices for the zero and one states using the `ithCleared(i, t)` function. This function clears the t-th bit in the binary representation of the global ID to determine the zero state index, and then the t-th bit is to determine the one state index. These indices are used to access the relevant portions of the state vector.

Control qubits are processed to determine whether the gate operation should be performed. A series of conditional checks are used to evaluate the state of each control qubit, and the perform flag is set accordingly. If all control conditions are met, the kernel proceeds to perform the quantum gate operation. The `controls_check.h` include hides the code in Listing 4.6 for brevity. We use such includes throughout the listings in this chapter to hide unnecessary detail which would have been covered in earlier sections.

The quantum gate operation involves complex number multiplication, where the input state vector is multiplied by the matrix elements to produce the new state vector. The results are then written back to the global memory, updating the state vector with the transformed values. The `baseline_compute.h` include contains the core compute code described earlier in Listing 4.7.

Global memory gate buffer

Instead of passing the gate parameters as arguments to the kernel, we could also choose to instead pass a global memory buffer containing them. This adds an extra step for the host to write to the global memory of the device every gate, and for the device to read the gate parameters from the global memory. This is additional memory access overhead and so is not generally the approach taken. It is mentioned here as a precursor to implementing gate fusion discussed later in Section 4.11.

4.6 Unrolled Loops Architecture

In this single-task kernel architecture, the core computation for processing quantum gates is managed by a sequential, unrolled loop within the kernel. Unlike the NDRange approach, which relies on parallel execution over multiple work-items, this method employs a single kernel instance to iteratively process all iterations needed for a gate operation. We start by describing a version with no parallelism as it simply replaces the NDRange kernel with a loop sequentially going over all the iterations required to process each gate.

4.6.1 Non-parallel version

As before, the host code first reads and parses the quantum circuit instructions. It then initialises the quantum state vector, loading it into the device memory; and prepares the kernel arguments for each gate operation, including the quantum state vector, target qubit index, control qubit indices, and gate matrix elements. The host enqueues the kernel for execution and manages the process of setting the kernel arguments for each gate in sequence.

```
1  #include "device_header.h"
2
3  __kernel void device_kernel(
4     __global cfloat * restrict state,
5     uint iter_count,
6     uint t,
7     #include "controls_as_args.h"
8     cfloat mat0, cfloat mat1, cfloat mat2, cfloat mat3
9  ) {
10    #pragma ivdep
11    for(uint i = 0; i < iter_count; i++) {
12       int zero_state = ithCleared(i, t);
13       int one_state = zero_state | (1 << t);
14
15       #include "controls_check.h"
16
17       #include "baseline_compute.h"
18    }
19 }
```

Listing 4.9: Loop-based single-task kernel architecture for FSVQCS, with no parallelism through loop unrolling.

Within the kernel, a for loop iterates over all possible iterations (combinations of qubit states) determined by the size of the quantum register, as defined by iteration count. As described above, for each iteration, the kernel calculates the indices of the state vector that correspond

to the zero and one states of the target qubit. These indices are used to access the relevant amplitudes in the state vector, which are then subject to potential modification based on the gate operation. Control-processing is handled exactly as in the Baseline NDRange architecture above. This flow is shown in Listing 4.9.

4.6.2 Parallelism through unrolled loops

So far, this version of the architecture makes no use of parallelism, as the `num_compute_units` attribute is not utilised and no unrolling of loops happens.

```
1  #include "device_header.h"
2
3  __kernel void device_kernel(
4      __global cfloat * restrict state,
5      uint iter_count,
6      uint t,
7      #include "controls_as_args.h"
8      cfloat mat0, cfloat mat1, cfloat mat2, cfloat mat3
9  ) {
10
11     #pragma unroll
12     for(uint cu_id = 0; cu_id < NCU; cu_id++) {
13         #pragma ivdep
14         for(uint i = 0; i < iter_count/NCU; i++) {
15             int iter_index = i + cu_id * iter_count/NCU;
16             int zero_state = ithCleared(iter_index, t);
17             int one_state = zero_state | (1 << t);
18
19             #include "controls_check.h"
20
21             #include "baseline_compute.h"
22         }
23     }
24 }
```

Listing 4.10: Single-task kernel implementation of the device code, utilising loop unrolling to achieve compute unit parallelism.

Listing 4.10 shows the device code for the UnrolledLoops architecture. The kernel is designed to exploit the parallelism offered by the FPGA by distributing the workload across multiple CUs. The outer loop, which runs from 0 to NCU-1, is unrolled using `#pragma unroll`.

When a loop is unrolled, the iterations of the loop are replicated in hardware, effectively creating multiple instances of the loop body (which can be considered independent compute

units, CUs) that can execute in parallel. This increases the degree of parallelism and allows for higher throughput since multiple operations are performed simultaneously. However, it also increases the resource utilisation on the FPGA, as more logic units, registers, and memory bandwidth are required to support the additional parallelism. The static loop bound is crucial because it enables the compiler tools to precisely determine the extent of unrolling at compile time, optimising the hardware design for the specified number of iterations. This enables each CU to handle a portion of the iterations independently. This design allows the FPGA to process multiple iterations in parallel within each CU, effectively dividing the iteration count evenly among the available CUs.

Memory access is optimised in this approach by ensuring that each CU reads and writes to different parts of the state array. This is a given property of our memory access pattern, as no two iterations will ever access the same set of amplitudes (and no pair of amplitudes will ever share an element with another). As the memory access indices are computed dynamically at runtime, the compiler needs to be made aware that all the iterations access independent slices of the memory. This is accomplished using the `#pragma ivdep` pragma, which stands for ”ignore vector dependencies”. This avoids conflicts and maximises memory throughput, as each CU operates on its own subset of data. For each iteration, the zero state and one state indices are calculated based on the current iteration index, corresponding to the states in the quantum register before and after applying the quantum gate.

The host code remains largely unchanged, with the main difference being how it interacts with the kernel. It sets up the problem, initialises the state, and writes it to the device's memory buffer, then sets the kernel arguments and launches the kernel for each gate in the quantum circuit. The kernel execution now leverages the parallelism provided by multiple CUs, significantly enhancing performance. After kernel execution, the host reads back the results for further processing.

This approach's key advantages include increased parallelism, scalability, and efficient memory management. By fully utilising the FPGA's computational resources, the architecture is more suitable for large-scale quantum simulations.

4.7 OnBoard Circuit Execution

We introduce a variation of the DIP-based architectures which can process a quantum circuit entirely on the device in one single-task kernel call without requiring to send each gate over host-FPGA connection. The device code for this architecture is shown in Listing 4.11. The `qproblem` array contains the entire circuit with each gate represented by a block of unsigned integers (gate code, target qubit, and control qubits). For each gate, the code loads its information into a local `gate_buffer`, which holds the gate code, the target qubit index

(t), and up to NCONTROLS control qubit indices. This buffer allows for fast access to the gate's parameters during processing.

The kernel iterates over all the gates in the circuit (num_gates). For each gate, the appropriate matrix representing the quantum gate operation is determined using a switch statement based on the gate_code, a step which would have been done on the host side in the previously described architectures. After the gate parameters are prepared, execution proceeds similar to the UnrolledLoops architecture presented in the previous section.

Another primary difference from the previous architectures is that controls can now be checked using an fully unrolled static loop, since they are processed from the local gate_buffer.

```
1  #include "device_header.h"
2
3  __kernel void device_kernel(
4     __global cfloat * restrict state,
5     uint iter_count,
6     __global uint * restrict qproblem,
7     uint num_gates
8  ) {
9     __local uint gate_buffer[NCONTROLS+2];
10
11    for(uint g = 0; g < num_gates; g++) {
12       // Read the gate
13       #pragma unroll
14       for(uint i = 0; i < NCONTROLS+2; i++) {
15          gate_buffer[i] = qproblem[g * (NCONTROLS+2) + i];
16       }
17
18       uint gate_code = gate_buffer[0];
19       uint t = gate_buffer[1];
20
21       #include "device_gate_select.h"
22
23       #pragma unroll
24       for(uint cu_id = 0; cu_id < NCU; cu_id++) {
25          #pragma ivdep
26          for(uint i = 0; i < iter_count/NCU; i++) {
27             int iter_index = i + cu_id * iter_count/NCU;
28             int zero_state = ithCleared(iter_index, t);
29             int one_state = zero_state | (1 << t);
30
31             // unrolled controls check
32             bool perform = true;
33             #pragma unroll
34             for(int i = 0; i < NCONTROLS; i++) {
35                bool cond = gate_buffer[i+2] != t;
36                perform &= (cond && (((1 << gate_buffer[i+2]) &
                      zero_state) > 0)) || !cond;
```

```
37                  }
38
39                  #include "baseline_compute.h"
40              }
41          }
42      }
43  }
```

Listing 4.11: Device code for the OnBoard architecture, designed to execute the entire circuit in one single-task kernel invocation, requiring an extra loop for the gates in the circuit and full on-board gate description processing.

4.8 Parallel Execution of Different Circuits

In the upcoming Chapter 5, we present a technique for reducing the circuit width of quantum circuits to allow them to be simulated on systems with limited memory capacity by splitting them circuits with ”specialised” inputs that operate on fewer qubits. This method allows us to run circuits whose state vectors would otherwise not fit in the memory of a particular system by running several circuits sequentially instead, and combining the results (if needed, as described in the Section 5.9). In some cases however, we can also reduce circuits further than just enough to meet particular memory constraints, which would allow us to fit several reduced circuits' worth of state vectors in the memory system. These state vectors would be guaranteed to have no dependencies for the simulation of the larger ”main” circuit and thus would improve memory locality significantly. In this section, we present an architecture which can theoretically simulate two circuits concurrently aimed at this purpose.

Memory systems which have several independent memory banks (whether DRAM, or HBM) allow for the synthesis of separate memory interfaces to access each bank independently and in parallel. This allows us to run different circuits on the same board, and with sufficient compute resources, completely in parallel. In this section, the implementation of such an architecture is detailed.

As in Chapter 4, we present the implementation based on our OpenCL architectural pattern. In particular, we present two versions as modifications of the BaselineNDRange architecture from Section 4.5. For simplicity (as a proof-of-concept) we restrict ourselves to up to two circuits being run in parallel (this is also a restriction due to our experimental setup having an FPGA board with a memory system containing two DRAM banks).

The main change from the architecture presented in Section 4.5 is that the kernel takes two sets of arguments related to the different gates and instantiates two versions of the compute

hardware, as shown in Listing 4.12. If one of the circuits contains more gates than the other, the host compensates for this by adding identity gates to the smaller circuit.

```
1  #include "device_header.h"
2
3  __attribute__((num_compute_units(NCU)))
4  __kernel void device_kernel(
5     __global cfloat * restrict state0,
6     __global cfloat * restrict state1,
7     uint t0,
8     uint t1,
9     #include "controls0_as_args.h"
10    #include "controls1_as_args.h"
11    cfloat mat0_0, cfloat mat0_1, cfloat mat0_2, cfloat mat0_3,
12    cfloat mat1_0, cfloat mat1_1, cfloat mat1_2, cfloat mat1_3) {
13
14    uint i = get_global_id(0);
15
16    {
17       int zero_state = ithCleared(i, t0);
18       int one_state = zero_state | (1 << t0);
19
20       #include "controls0_check.h"
21       #include "baseline_compute0.h"
22    }
23    {
24       int zero_state = ithCleared(i, t1);
25       int one_state = zero_state | (1 << t1);
26
27       #include "controls1_check.h"
28       #include "baseline_compute1.h"
29    }
30 }
```

Listing 4.12: Device code for executing two gates on two state vectors in parallel, corresponding to the concurrent simulation of two quantum circuits.

4.9 Scheduling Optimisation for Controlled Gates

Section 3.1.3 presented a method for optimising the scheduling of controlled gates. The implementation of this method is discussed in this section. The primary idea of this method is by realising that every control added to a gate halves the number of iterations which actually access and modify the state vector stored in the global memory, we can schedule only as many iterations which have an effect on the state vector.

The goal is to be able to go from an index in the reduced set of iteration indices, $I_r = \{0, 1, ..., 2^{n-n_c-1}\}$, to its equivalent in the global iteration set (the set with cardinality 2^{n-1}). This is done through the iterative formula in Equation 3.2: $i_{r_k} = i_{r_{k+1}} + (\lfloor i_{r_{k+1}}/2^{c_{adj}} \rfloor + 1) \times 2^{c_{adj}}$, where $c_{adj} = \begin{cases} c-1 & \text{if } c > t \\ c & \text{otherwise} \end{cases}$.

With this formula, we are able to schedule the NDRange kernels with exactly the number of iterations that will always update the memory. We use the formula to iteratively go from a reduced iteration index to the equivalent global iteration index allowing us to use the previously used memory indexing strategy based on the `nthCleared` function. Listing 4.13 shows the implementation of the formula in the OpenCL kernel, for up to 2 controls. This block would go above the pair element index computation in the pseudocode for the baseline kernel shown above in Listing 4.5, and then i_g would be used in the computation of `zero_state` in place of i in the `ithCleared` function. The controls check block that utilises a boolean flag `perform` would no longer be needed and there would be no control flow checks on the memory access.

```
1  uint i_g = get_global_id(0); // start from the reduced index
2
3  // adjusted control:
4  uint c_adj0 = c0 > t ? c0-1 : c0;
5  // skip interval:
6  uint sInt0 = 1 << c_adj0;
7  i_g += ((i_g/sInt0) + 1) * sInt0;
8
9  uint c_adj1 = c1 > t ? c1-1 : c1;
10 uint sInt1 = 1 << c_adj1;
11 i_g += ((i_g/sInt1) + 1) * sInt1;
```

Listing 4.13: Implementation of the iterative formula for going from a reduced iteration index set to the global set in the OpenCL kernel. Example shown for up to 2 controls.

The final code of an NDRange kernel implementing this optimisation would then look like Listing 4.14. Notice the main achievement of this method is that the memory access is always scheduled and is not predicated on a boolean flag.

```
1  #include "global_index_computation.h"
2
3  int zero_state = ithCleared(i_g, t);
4  int one_state = zero_state | (1 << t);
5
6  cpair inVec;
7  inVec.a = state[zero_state];
8  inVec.b = state[one_state];
9
```

```
10 // Perform the computation and write to the memory
11 state[zero_state] = cdot((cpair){mat0, mat1}, inVec);
12 state[one_state] = cdot((cpair){mat2, mat3}, inVec);
```

Listing 4.14: NDRange kernel implementing the controls scheduling optimisation. The host would schedule this kernel with a global work item size of 2^{n-n_c-1} instead of 2^{n-1}.

4.10 Buffered Architecture

The buffered implementation of the device code is designed to exploit contiguous burst memory access, which can significantly enhance performance on memory-bound operations, such as those commonly encountered in quantum simulations. The idea behind using a buffered implementation is to minimise memory access latency by reading larger chunks of data in a single burst rather than accessing memory in smaller, scattered portions. This is achieved by reading and writing blocks of data to and from a buffer in the local memory of the device, which can be accessed more quickly than global memory.

4.10.1 Case 1: Single Memory Access per Pass

As explained in Section 3.3, in this implementation, there are two distinct cases for how the buffering is handled, depending on the value of t relative to l (`BUFFER_QUBIT_SIZE`). When t is less than l, the kernel operates in case 1 mode where the buffer can read from and write to the memory in one contiguous access to fill the whole buffer.

Listing 4.15 shows how the buffer is populated.

```
1 #pragma unroll
2 case1_mem_read: for(int i = 0; i < BUFFER_SIZE; i++) {
3    buffer[i] = state[p*BUFFER_SIZE + i];
4 }
```

Listing 4.15: Populating the buffer for a case 1 buffer pass in a buffered architecture.

Listing 4.16 shows how we operate on the buffer if the perform condition determined by the controls check is satisfied. The computation of the buffer access indices is done as described in Section 3.3 and is shown below in Listing 4.18.

```
1 if(perform) {
2    cpair inVec;
3    inVec.a = buffer[zero_state];
4    inVec.b = buffer[one_state];
```

```
5       buffer[zero_state] = cdot((cpair){mat0, mat1}, inVec);
6       buffer[one_state] = cdot((cpair){mat2, mat3}, inVec);
7   }
```

Listing 4.16: Updating an amplitude pair on-board through the buffer for a case 1 buffer pass.

Listing 4.17 shows how the buffer is written back to memory after being processed by the compute loop.

```
1   #pragma unroll
2   case1_mem_write: for(int i = 0; i < BUFFER_SIZE; i++) {
3       state[p*BUFFER_SIZE + i] = buffer[i];
4   }
```

Listing 4.17: Writing the buffer back to global memory for a case 1 buffer pass.

Listing 4.18 shows how the fragments above combine to form the whole case 1 computation. The include statements hide the above listings for brevity.

```
1   const uint pass_count = 1 << (n - BUFFER_QUBIT_SIZE);
2
3   #pragma ivdep
4   case1_pass_loop: for(int p = 0; p < pass_count; p++) {
5
6       #include "buffered_case1_read.h"
7
8       // Perform the computation
9       #pragma unroll
10      case1_compute: for(int i = 0; i < NCU; i++) {
11          int zero_state = ithCleared(i, t);
12          int one_state = zero_state | (1 << t);
13
14          int global_state_index = p*BUFFER_SIZE + zero_state;
15          #include "controls_check.h"
16
17          #include "buffered_case1_compute.h"
18      }
19
20      #include "buffered_case1_write.h"
21  }
```

Listing 4.18: Full case 1 buffer pass loop for a buffered architecture.

4.10.2 Case 2: Two Memory Accesses per Pass

Conversely, when t is greater than or equal to `BUFFER_QUBIT_SIZE`, the element pair stride is greater than can be read contiguously and fit within the buffer, so each group has to be divided across multiple buffer passes. In this scenario, the data is divided into chunks, each of which is processed independently in groups. The kernel reads half of the buffer from the lower half of the current group and the other half from the upper half, performing the necessary computations before writing the results back to the respective locations in global memory. This approach ensures that even for gates whose element pair strides are high, memory access remains as efficient as possible and maintains the advantage of burst access, albeit across more complex memory patterns. Listings 4.19, 4.20, and 4.21, show the three stages of executing a case 2 buffer pass.

```
1  #pragma unroll
2  case2_mem_read1: for(int i = 0; i < BUFFER_SIZE/2; i++) {
3     buffer[i] = state[gr*group_size + p*BUFFER_SIZE/2 + i];
4  }
5  #pragma unroll
6  case2_mem_read2: for(int i = 0; i < BUFFER_SIZE/2; i++) {
7     buffer[i + BUFFER_SIZE/2] = state[gr*group_size + group_size/2 +
         p*BUFFER_SIZE/2 + i];
8  }
```

Listing 4.19: Populating the buffer for a case 2 buffer pass in a buffered architecture.

```
1  // Perform the computation and write to the memory
2  if(perform) {
3     cpair inVec;
4     inVec.a = buffer[i];
5     inVec.b = buffer[i + BUFFER_SIZE/2];
6     buffer[i] = cdot((cpair){mat0, mat1}, inVec);
7     buffer[i + BUFFER_SIZE/2] = cdot((cpair){mat2, mat3}, inVec);
8  }
```

Listing 4.20: Updating an amplitude pair on-board through the buffer for a case 1 buffer pass.

```
1  #pragma unroll
2  case2_mem_write1: for(int i = 0; i < BUFFER_SIZE/2; i++) {
3     state[gr*group_size + p*BUFFER_SIZE/2 + i] = buffer[i];
4  }
5  #pragma unroll
6  case2_mem_write2: for(int i = 0; i < BUFFER_SIZE/2; i++) {
```

```
7       state[gr*group_size + group_size/2 + p*BUFFER_SIZE/2 + i] = buffer[i
            + BUFFER_SIZE/2];
8   }
```

Listing 4.21: Writing the buffer back to global memory for a case 1 buffer pass.

4.10.3 Double-Buffering

In the buffered architecture described above, we utilise only a single buffer for both reading and writing amplitudes. However, this means that we cannot overlap reading from the memory with writing updated amplitudes to the buffer. Using double-buffering introduces a pair of buffers—one dedicated to reading data from global memory and the other to writing data back to it. This strategy enhances performance by overlapping computation with memory operations, thereby reducing the idle time for both the computation units and the memory system.

The latency associated with memory access is reduced since the kernel can continuously process data without pausing to wait for memory operations to complete. By decoupling read and write operations, the kernel can start processing new data as soon as it is available in the read buffer, minimising delays.

With double-buffering, the architecture can more effectively exploit parallelism. While one buffer is being written to, the other can be prepared for the next batch of data to be read in. This leads to a more continuous flow of data through the computation pipeline, allowing for a smoother and more consistent utilisation of the processing resources.

In a double buffering architecture, the compute section of the code must always write to the output buffer, even if the perform variable, which indicates whether the computation should be executed based on control conditions, is false. This is necessary because the output buffer must be fully updated and consistent after each iteration of computation. When double buffering is used, one buffer holds the current state (input buffer), while the other holds the next state (output buffer). If the output buffer is not updated, either by writing the result of the computation or by simply copying the unchanged data when perform is false, the subsequent stages of computation could use stale or incorrect data, leading to incorrect results in the final quantum state. Thus, ensuring that the output buffer is fully populated, whether or not the computation was performed, guarantees data integrity and correctness. This is demonstrated in Listing 4.22.

```
1   cpair inVec, outVec;
2   inVec.a = in_buffer[zero_state];
3   inVec.b = in_buffer[one_state];
4   outVec.a = inVec.a;
```

```
5  outVec.b = inVec.b;
6
7  if(perform) {
8     outVec.a = cdot((cpair){mat0, mat1}, inVec);
9     outVec.b = cdot((cpair){mat2, mat3}, inVec);
10 }
11
12 out_buffer[zero_state] = outVec.a;
13 out_buffer[one_state] = outVec.b;
```

Listing 4.22: Double buffering compute snippet ensuring state vector data is propagated correctly to the output buffer before writing.

4.11 Gate Fusion Architecture

The core idea behind gate fusion is to exploit the spatial locality of the quantum state vector by processing multiple consecutive gates that share a target qubit index satisfying the property $t < l$ in a single pass. This allows the simulator to load a slice of the state vector into a buffer, apply multiple gates to this slice, and then write the updated slice back to memory, minimising the number of memory accesses and thus improving performance.

To implement gate fusion as described in Section 3.5, the device has to be modified to take a global memory buffer containing the gate block instructions. As described, these instructions are in a format similar to the overall quantum circuit representation which the host reads from file. Each gate block instruction begins with an integer indicating the number of gates in the gate block, followed by series of integers which indicate the gate parameters themselves, as in the earlier architectures. To construct this format, the host reads the quantum circuit problem file from disk and runs a function on the independent gates to identify gates which can be fused into blocks (gates satisfying the condition $t < l$). Then, for each gate block, the host has to transfer the gate block instructions into the global memory of the device. The device kernel then takes four arguments: a pointer to the state vector, an integer representing the number of qubits involved in the circuit, a pointer to the gate block instructions, and an integer representing the number of gates in the gate block.

The device code then also instantiates an additional local buffer (`gb_buffer`) to be implemented as a BRAM to store the gate block to allow for faster access times compared to reading each gate from the global memory. The host must take into account the maximum number of gates which this buffer can hold when constructing the fused gates representation.

On the device, each gate computation then starts by copying the gate block instructions from the global memory to the local memory buffer. We then determine whether to apply case 1 buffering or case 2 based on the number of gates in the gate buffer. Special care also has to be

taken in case there is a single gate; as even if the gate block consists of only one gate, it may still require case 1 buffering if the target qubit index of that gate is less than the buffer qubit size. Thus the condition to apply case 1 buffering is `gate_count > 1 || first_target < BUFFER_QUBIT_SIZE`.

4.11.1 Case 1: Single Memory Access per Pass

If this condition is satisfied, we proceed with a pass loop as usual for a case 1 buffering architecture. The primary difference now is instead of running the unrolled case 1 compute loop once, it is wrapped inside of a gate loop, as we can apply all the gates in the gate block to the same slice of the state vector stored in the buffer.

Another difference to the above buffering architectures is that we have to determine the gate matrix elements on the device instead of on the host. In the previous architectures, the device would only always process one gate and so it the gate matrix could be determined on the host and sent to the device as kernel parameters. However, since here we process several gates in the same buffer pass, the device has to determine the gate matrix at each gate loop. This is accomplished by a switch statement, similar to on the host.

Additionally, since the controls of each gate are stored in the gate block buffer, they can be processed in an unrolled loop, as demonstrated in Listing 4.23.

```
1 bool perform = true;
2 #pragma unroll
3 for(int i = 0; i < NCONTROLS; i++) {
4    bool cond = gb_buffer[g*(NCONTROLS+2)+2+i] != t;
5    perform &= (cond && (((1 << gb_buffer[g*(NCONTROLS+2)+2+i]) &
         global_state_index) > 0)) || !cond;
6 }
```

Listing 4.23: Processing controls in a gate fusion architecture for case 1 buffering. `gb_buffer` is the on-board gate block buffer.

This system comes together as shown in Listing 4.24.

```
1 const uint pass_count = 1 << (n - BUFFER_QUBIT_SIZE);
2
3 #pragma ivdep
4 case1_pass_loop: for(int p = 0; p < pass_count; p++) {
5
6    #include "buffered_case1_read.h"
7
8    #pragma ivdep
9    case1_gate_loop: for(int g = 0; g < gate_count; g++) {
```

```
10          const uint gate_code = gb_buffer[g*(NCONTROLS+2)];
11          const uint t = gb_buffer[g*(NCONTROLS+2)+1];
12
13          #include "device_gate_select.h"
14
15          // Perform the computation
16          #pragma unroll
17          case1_compute: for(int i = 0; i < NCU; i++) {
18             int zero_state = ithCleared(i, t);
19             int one_state = zero_state | (1 << t);
20
21             int global_state_index = p*BUFFER_SIZE + zero_state;
22
23             // check controls
24             #include "unrolled_controls_check_case1_gb.h"
25
26             #include "buffered_case1_compute.h"
27          }
28       }
29
30       #include "buffered_case1_write.h"
31    }
```

Listing 4.24: Case 1 buffer pass loop device code for a gate fusion architecture.

This method optimises simulation by reducing the number of memory accesses, as multiple operations are performed on data that is kept in fast-access buffers rather than repeatedly reading from and writing to slower global memory. By fusing gates that operate on the same slice of the state vector, the simulator efficiently leverages the buffer space and minimises the overhead associated with memory access, significantly reducing the time spent accessing memory.

4.11.2 Case 2: Two Memory Accesses per Pass

If the case 1 condition is not satisfied, the architecture proceeds with case 2 where only a single gate whose target qubit index is such that the groups cannot be contiguously read into the buffer in one burst access. This functions the same as discussed in the earlier section on buffered architectures. Another difference is how the controls for this single gate are read from the gate block buffer; this is demonstrated in Listing 4.25.

```
1  bool perform = true;
2  #pragma unroll
3  for(int i = 0; i < NCONTROLS; i++) {
4     bool cond = gb_buffer[2+i] != t;
```

```
5       perform &= (cond && (((1 << gb_buffer[2+i]) & global_state_index) >
            0)) || !cond;
6   }
```

Listing 4.25: Processing controls in a gate fusion architecture for case 2 buffering

The case 2 gate fusion system comes together as shown in Listing 4.26.

```
1  const uint gate_code = gb_buffer[0];
2  const uint t = gb_buffer[1];
3
4  #include "device_gate_select.h"
5
6  const uint pass_count = 1 << (t + 1 - BUFFER_QUBIT_SIZE);
7  const uint group_size = 1 << (t + 1);
8  const uint group_count = 1 << (n - t - 1);
9  const uint g = 0;
10
11 #pragma ivdep
12 case2_group_loop: for(int gr = 0; gr < group_count; gr++) {
13    #pragma ivdep
14    case2_pass_loop: for(int p = 0; p < pass_count; p++) {
15
16       #include "mem/buffered_case2_read.h"
17
18       #pragma unroll
19       case2_compute: for(int i = 0; i < NCU; i++) {
20          const uint global_state_index = gr*group_size + p*BUFFER_SIZE/2
                + i;
21
22          // check controls
23          #include "unrolled_controls_check_case2_gb.h"
24
25          #include "comp/buffered_case2_compute.h"
26       }
27
28       #include "mem/buffered_case2_write.h"
29    }
30 }
```

Listing 4.26: Case 2 buffer pass loop device code for a gate fusion architecture.

4.11.3 Double buffering for Gate Fusion

Adding double buffering for gate fusion involves similar steps to adding the double buffer for the normal buffered architecture, however extra care has to be taken in case 1 to ensure

that the buffers are accessed correctly. In particular, the roles of the buffers as inputs and outputs have to be reversed after each gate; this is implemented by checking if the gate index is even. If it is, then the initial input buffer remains the input and the initial output buffer remains the output; otherwise, vice versa.

The processing of a buffer pass starts the same as before, with the state vector slice being read into the input buffer, `in_buffer`. For the compute section, the device checks if the condition `g % 2 == 0` is satisfied (indicating that the gate is even), and if so, it reads from `in_buffer` and writes the results to `out_buffer`. If this condition is not satisfied (indication that the gate is odd) then the roles of the buffers are flipped and the device reads from `out_buffer`, which now holds the updated state from the previous gate, and writes the results to `in_buffer`.

After all gates have been applied for a given pass, the final buffer (which contains the updated state vector slice) is written back to global memory. The final buffer is identified based on whether the total number of gates in the fused gate block is even. If the number of gates is even, `in_buffer` will hold the final results, so `buffered_case1_write_flipped.h` is used to write the slice back to memory. Otherwise, the default `buffered_case1_write.h` is used if the number of gates is odd.

Listing 4.27 demonstrates this case 1 system coming together.

```
1  cfloat in_buffer[BUFFER_SIZE];
2  cfloat out_buffer[BUFFER_SIZE];
3
4  __local uint gb_buffer[GB_BUFFER_SIZE];
5
6  #pragma unroll
7  gb_read_loop: for(int i = 0; i < GB_BUFFER_SIZE; i++) {
8     gb_buffer[i] = gate_block[i];
9  }
10
11 uint first_target = gb_buffer[1];
12
13 if(gate_count > 1 || first_target < BUFFER_QUBIT_SIZE) {
14    const uint pass_count = 1 << (n - BUFFER_QUBIT_SIZE);
15
16    #pragma ivdep
17    case1_pass_loop: for(int p = 0; p < pass_count; p++) {
18
19       #include "mem/buffered_case1_read.h"
20
21       #pragma ivdep
22       case1_gate_loop: for(int g = 0; g < gate_count; g++) {
23          const uint gate_code = gb_buffer[g*(NCONTROLS+2)];
24          const uint t = gb_buffer[g*(NCONTROLS+2)+1];
25
26          #include "device_gate_select.h"
```

```
27
28            // Perform the computation
29            if(g % 2 == 0) {
30               #pragma unroll
31               case1_even_g_compute: for(int i = 0; i < NCU; i++) {
32                  int zero_state = ithCleared(i, t);
33                  int one_state = zero_state | (1 << t);
34
35                  int global_state_index = p*BUFFER_SIZE + zero_state;
36
37                  // check controls
38                  #include "unrolled_controls_check_gb.h"
39
40                  #include "comp/buffered_case1_compute_gur.h"
41               }
42            } else {
43               #pragma unroll
44               case1_odd_g_compute: for(int i = 0; i < NCU; i++) {
45                  int zero_state = ithCleared(i, t);
46                  int one_state = zero_state | (1 << t);
47
48                  int global_state_index = p*BUFFER_SIZE + zero_state;
49
50                  // check controls
51                  #include "unrolled_controls_check_gb.h"
52
53                  #include "comp/buffered_case1_compute_gur_flipped.h"
54               }
55            }
56         }
57
58         if(gate_count % 2 == 0) {
59            #include "mem/buffered_case1_write_flipped.h"
60         } else {
61            #include "mem/buffered_case1_write.h"
62         }
63      }
64   }
```

Listing 4.27: Device code for a gate fusion architecture using double-buffering. Only case 1 is shown.

Case 2 in the double buffered version of the architecture operates the same as in the single-buffered version, with the only difference being the use of the two different buffers for reading and writing from and to global memory. However, since only one gate is processed in this case, no distinction based on parity is required like in case 1.

4.12 Summary

This chapter presented a comprehensive exploration of the implementation of various FPGA-based quantum circuit simulation architectures, focusing on techniques that we expect to improve the simulation performance of large-scale quantum circuits. The primary architectures introduced include Direct Iteration Processing (DIP), Buffered Architectures, and Gate Fusion. This represents the first work to implement Gate Fusion on FPGA-based simulation platforms. Key findings include the successful implementation of control scheduling optimisations, which we expect to significantly improve the simulation of control-heavy quantum circuits. A comprehensive evaluation of the implemented architectures in terms of raw performance and energy efficiency is presented in Chapter 6.

Chapter 5

Circuit Transformations and Width Reduction Techniques

One of the key optimisations developed to facilitate quantum circuit simulation on FPGAs in this work relates to circuit transformations, which enable another level of parallelisation for some circuits (i.e. in addition to parallelisation of the simulation of individual gates or fused gate blocks through the techniques described in the previous chapter). In this chapter, these transformations are presented, and examples are given in the context of quantum circuits for computational fluid dynamics (CFD) and arithmetic circuits. This type of optimisation was originally presented by the author in collaboration with Steijl and Vanderbauwhede in [56] and demonstrated on further arithmetic quantum circuit examples in [125]. Additionally, we demonstrate in this chapter how these transformations can be used to facilitate execution of circuits which do not operate in the computational-basis encoding, which is unpublished work.

5.1 Contributions

As the work for these publications was conducted in collaboration with my supervisors, the following is a summary of my contributions from these works. Development of the reduction techniques was done by my supervisor, Dr René Steijl, as well as the development of the D1Q3 and divider circuits, and their reduced versions.

- Development of the circuits by encoding them in the compiler toolchain described in Section 4.2.
- Validation of both the original circuits and their reduced versions through the development of test suites.

- Development and evaluation of the FPGA architectures used to test the reduced circuits, described in Section 4.5.
- Development and evaluation of the FPGA architectures used to run the reduced circuits in parallel, described in Section 4.8.
- Development of the proof-of-concept that the reduced circuits can include superposition and entanglement, despite being reduced in the computational basis, described in Section 5.9.

5.2 Motivation

The motivation behind the quantum circuit transformation techniques presented in this work is multifaceted, driven primarily by the need to facilitate efficient simulation of large and complex quantum circuits on hardware accelerators such as FPGAs (Field Programmable Gate Arrays).

One of the core motivations is to overcome the significant memory constraints that impede the simulation of large quantum circuits. Given that simulating quantum circuits involves handling exponentially growing state vectors, memory limitations of current FPGA and local CPU/GPU systems pose a substantial challenge. Without such circuit transformation techniques, the state vectors for circuits with high circuit width would not fit within the available memory resources. By reducing the number of qubits required and optimising the circuit design, these transformations enable the simulation of circuits that would otherwise be infeasible to handle.

Another significant motivation is the potential for reducing execution times through these transformations. Smaller circuits, even if they fit within the memory limits without the described transformations, can benefit from reduced execution times by leveraging coarse-grained parallelism on the FPGAs by replicating the components which simulate entire circuits. This approach not only speeds up the simulation but also maximises the use of available hardware resources, thereby enhancing overall computational efficiency.

The described circuit transformations result in a reduction in the width of the transformed circuit by creating two altered circuits for each reduced qubit. Thus, we no longer simulate the original circuit and instead simulate two lower circuit-width circuits (either sequentially or in parallel, if memory capacity allows); the results of the two (for one reduced qubit) circuits can then either be interpreted as they are, or combined to find the result of the original circuit.

In addition, as the structure does not generally change significantly, the impact of noise and

fault-tolerance can still be tested on these circuits to get an intuition for how the circuit would behave on a real quantum computer.

5.3 Background

The quantum circuit transformation approaches presented here were designed to reduce quantum circuits performing quantum arithmetic operations. In the intended application, the quantum circuit considered represents a larger, and more general quantum algorithm where the arithmetic represents a part of the computational work. Quantum arithmetic operations typically rely on a specific type of data encoding. The differences between the computational-basis and amplitude-basis encoding are explained in Section 1.2.5.

Since the circuit transformation approaches introduced here target quantum arithmetic operations, key aspects of computational basis encoding (as explained in Section 1.2.5) were used in creating these transformation steps. Specifically, identifying qubits that throughout a quantum computation are guaranteed to remain in either state $|0\rangle$ or $|1\rangle$ relies on this type of encoding throughout the computation or at least for the considered part of the algorithm performing arithmetic operations. We also show in Section 5.9 that these transformations can still be used in some cases where the qubits to be reduced are in amplitude-encoding (and entangled with other qubits).

Relating to the use of the transformation approaches in quantum algorithm development process, the following observations need to be made:

- The quantum arithmetic operations in circuits using computational basis encoding considered here can be efficiently simulated on a classical computer using a logic based simulator—these circuits act as classical reversible circuits when not used as part of a larger quantum algorithm;
- As arithmetic blocks which operate in the computational basis are often parts of larger quantum algorithms, quantum circuit simulations often require a full-state vector simulation approach;
- For quantum circuit simulations where the effect of (modeled) quantum errors are included, the full-state vector simulation approach is generally required even for algorithms operating entirely (through computation) with computational basis encoding.

In the published works, the focus of the application of the transformation techniques is on circuits which exclusively use the computational basis encoding. Thus, the focus of most of this chapter is also only on such algorithms. However, in the last part of this chapter,

we show how these techniques can be applied to circuits that make use of superposition and entanglement, in the amplitude-basis encoding.

5.4 Circuit Width Reduction by Qubit Specialisation

Quantum circuits representing quantum algorithms which employ computational-basis encoding often can have some qubits reduced out by generating two different circuits for each possible qubit input. In this way, the total memory required for one run of the circuit is reduced by half for each qubit reduced out. Qubits which are only used as controls throughout the circuit are ideal candidates for this type of reduction. The toolchain presented in Section 4.2 provides functionality to automate this. While this approach is useful for reducing the total memory required across any platform, it is especially good for an FPGA which would, be able to run both (for one reduced qubit) circuits concurrently on completely independent memory spaces, allowing full parallelisation of their simulation. This will be demonstrated in Section 4.8.

As a prelude to this chapter, the reduction transformation is demonstrated on two simple example quantum-circuits: a 3-qubit QFT-based adder, a circuit for which the reduction can be easily automated, and a 3-qubit Cuccaro modulo adder, an example which requires more logical reasoning to reduce, and for which the reduction is not so straightforward to automate.

5.4.1 QFT-Based Adder

Draper [6] introduces a QFT-based adder, which employs the QFT to perform addition operations. The QFT is a quantum analogue of the discrete Fourier transform that transforms a quantum state into a superposition where the amplitude of each state is a function of the input state. This transformation encodes information into the phases of the quantum states, enabling arithmetic operations in the transformed domain.

In Draper's QFT adder, the process begins with preparing two numbers, 'a' and 'b', which are both in the computational basis. The QFT is applied to a, transforming it into the phase basis, resulting in a superposition of states with phase factors dependent on a. This transformation is then evolved using b by applying controlled phase shift gates that introduce a phase shift equivalent to adding b to a in the phase domain. The inverse QFT is finally applied to retrieve the sum $a + b$ in the computational basis. A circuit diagram implementing this for 3-qubit inputs is shown on the left of the Figure 5.1. Three possible specialisations for the value of the input a are shown on the right of the figure.

The reduction of the Draper QFT based adder utilises a basic reduction technique: since the input values of the $|a2|a1|a0\rangle$ register are known and the qubits constituting the register

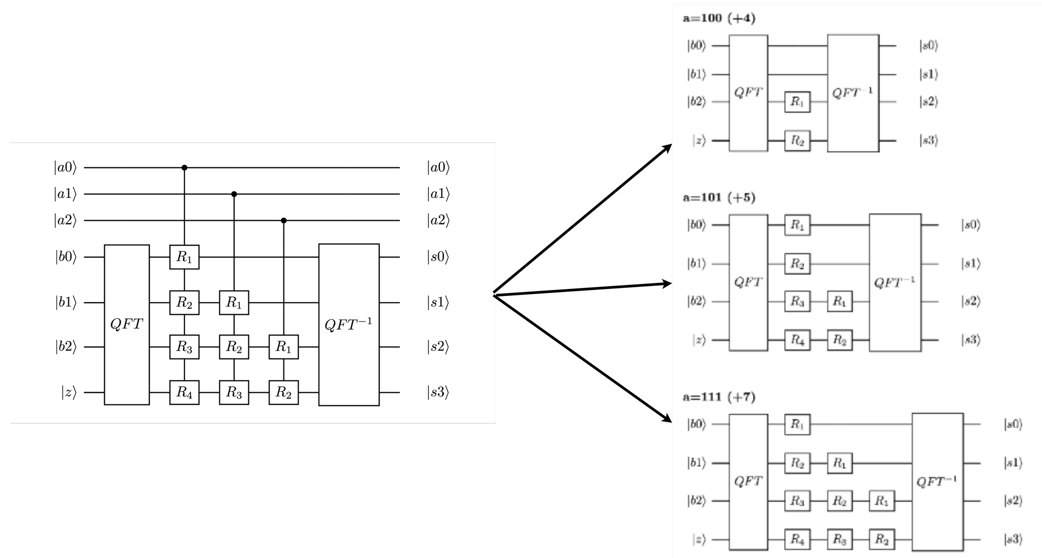


Figure 5.1: Example specialisations of the Draper QFT-based adder [6], for input a values of $+4, +5$, and $+7$. Note that most significant bits of the input integers are shown at the bottom of the integer's quantum register (e.g. for $a = +4$, we have $a2 = 1, a1 = 0, a0 = 0$).

are only used as controls and never as targets, then the qubits can be directly reduced by checking if their input values satisfy the control conditions for the gate. In the top example on the right side of Figure 5.1, $|a2|a1|a0\rangle = |100\rangle$ (adding 4 to b), thus only the phase shift gates which are controlled by the qubit $|a2\rangle$ will be applied. So we can remove the gates which are controlled by the other qubits in the $|a2|a1|a0\rangle$ register. Reduction for the other input cases of $|a2|a1|a0\rangle$ follows the same logic.

5.4.2 Step-by-step reduction of the Cuccaro Modulo Adder

As a demonstrative example of the reduction technique, we start with a simple 3-qubit full Cuccaro adder, taking two input qubit registers: $|a2|a1|a0\rangle$, $|b2|b1|b0\rangle$, and a carry qubit $|c\rangle$. The carry qubit is required due to the use of the modular design utilising the majority and unmajority blocks in the circuit. This adder performs an "in-place" modulo addition, such that the sum of the input registers is stored in the register consisting of $|b2|b1|b0\rangle$ at the end of the computation. From this circuit, shown in Figure 5.2, we derive circuits specialised for the input qubits $|a2|a1|a0\rangle$ having values of $|001\rangle$, $|010\rangle$, and $|011\rangle$, corresponding to adding 1, 2, and 3 to the register $|b2|b1|b0\rangle$, respectively.

Specialising for $|a2|a1|a0\rangle = |001\rangle$

To reduce the circuit in Figure 5.2 for the input $|a2|a1|a0\rangle = |001\rangle$, we begin by observing the structure of the gates in the middle part of the circuit. Since the X (or NOT) gate is

its own inverse, and therefore controlled version of X are also their own inverse, including $CNOT$ and $TOFF$ ($CCNOT$), then consecutive gates of each of these types can cancel each other out. So we can cancel out the two $TOFF$ gates in the middle of the circuit, and them immediately cancel out the $CNOT$ gates surrounding them as well. Then by observing that the qubit $|a2\rangle$ has the value $|0\rangle$, the $CNOT$ gate controlled by $|a2\rangle$ acting on $|b2\rangle$ will have no effect, and can be cancelled out as well.

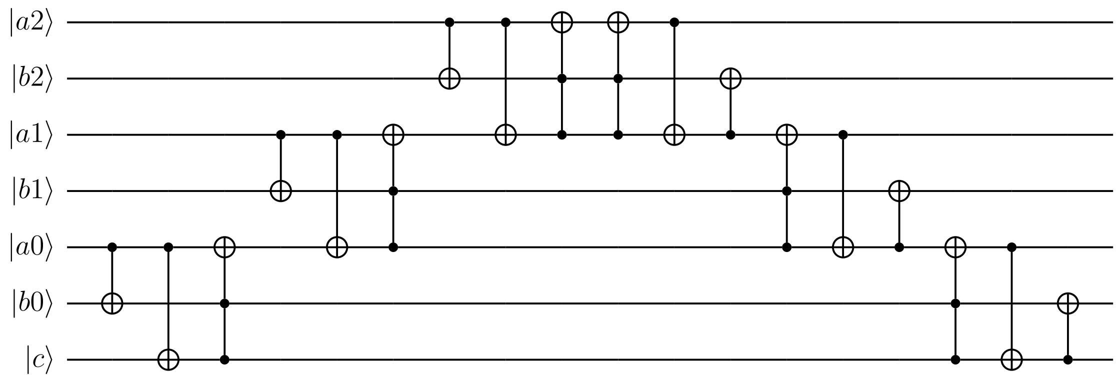


Figure 5.2: Original 3-qubit Cuccaro Modulo adder unpacking the MAJ and $UMAJ$ blocks, as opposed to Figure 1.14.

By removing these five gates, the circuit now looks like Figure 5.3. With no gates acting on $|a2\rangle$, this qubit is effectively eliminated and can be considered reduced.

Next, we observe that $|a1\rangle = |0\rangle$ implies that any gate which has it as a control will have not effect, and can be eliminated. And so the $CNOT$ gates acting on $|b1\rangle$ and $|a0\rangle$ (with $|a1\rangle$ as the control) can be removed. Then we observe that whatever the starting value of $|b1\rangle$ is, the $TOFF$ gate which has $|b1\rangle$ and $|a0\rangle$ as controls and which acts on $|a1\rangle$, gets uncomputed after the $CNOT$ $a1$ $b2$ gate. Thus, after this uncomputation, $|a1\rangle$ will definitely still have the value $|0\rangle$, and so any gates with it as a control can also be safely eliminated; so we can remove the $CNOT$ acting on $|a0\rangle$ in the second half of the circuit.

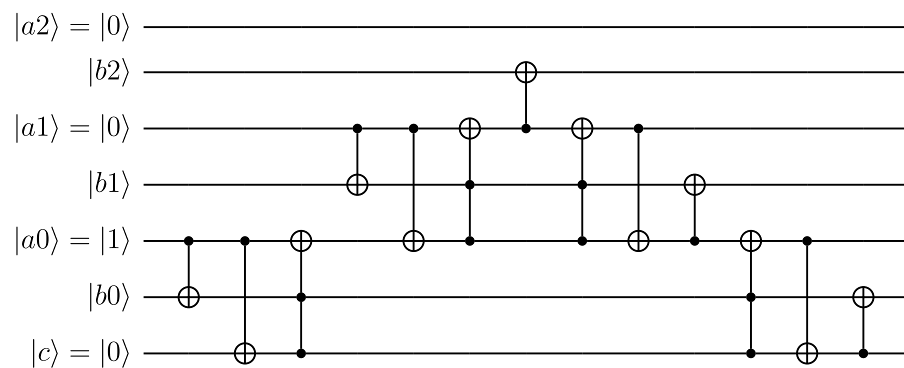


Figure 5.3: Step 1 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$.

After this step, the circuit becomes as Figure 5.4.

Next we use the value of $|a0\rangle$ being $|1\rangle$ to see the first two $CNOT$ gates in the circuit will always be executed. Thus the first $CNOT$ can simply be replaced by an X gate acting on qubit $|b0\rangle$. The carry qubit $|c\rangle$ will always start with the value $|0\rangle$, and thus after the execution of the second $CNOT$, it will have the value $|1\rangle$. This bring our attention to the third gate in the circuit, the $TOFF$ acting on $|a0\rangle$ with $|b0\rangle$ and $|c\rangle$ as controls. Since at this stage, $|c\rangle = |1\rangle$, the control part of this gate on this qubit is always satisfied, so we can remove this qubit as a control from this gate; and replace this $TOFF$ with a $CNOT$ acting on $|a0\rangle$ with $|b0\rangle$ as the control.

Similarly, for the $TOFF$ acting on the same qubit with the same controls at the end of the circuit; the qubit $|c\rangle$ will retain its value of $|1\rangle$ at this point in the circuit and so can again be removed as a control from this gate; replacing it with a $CNOT$ $b0$ $a0$ gate. We then recognise that as far as $|a0\rangle$ is concerned, these two $CNOT$ gates uncompute each other, reverting $|a0\rangle$ to its original value at the start of the circuit of $|1\rangle$. And so for the last two gates in the circuit, the $CNOT$ $a0$ c will always execute, reverting the value of $|c\rangle$ back to $|0\rangle$. The last gate in the circuit now is a $CNOT$ with $|c\rangle$ as the control, which can be safely eliminated since $|c\rangle = |0\rangle$ at this point.

After this step, the circuit is now as Figure 5.5, with the qubit $|c\rangle$ effectively eliminated from the circuit, as no gates act or depend on it.

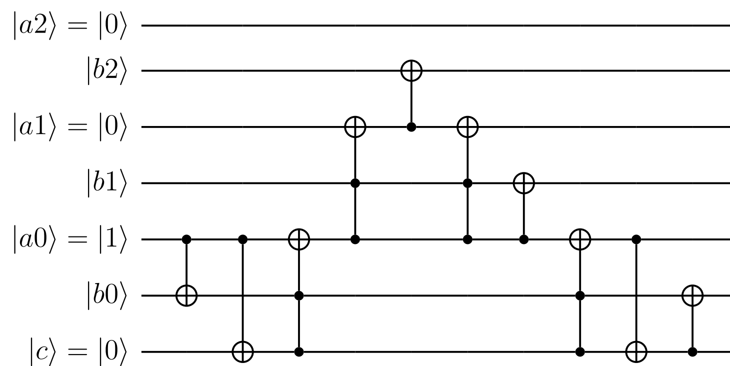


Figure 5.4: Step 2 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$.

Now, with the X gate on $|b0\rangle$ followed by a $CNOT$ controlled by $|b0\rangle$ acting on $|a0\rangle$, we recognise that effectively the qubit $|a0\rangle$ is flipped if the initial value of $|b0\rangle$ is $|0\rangle$. This is equivalent to having an X gate on $|a0\rangle$ anti-controlled by $|b0\rangle$. However, to maintain consistency in the rest of the circuit, we also have to transform the last gate in Figure 5.5, which is the same $CNOT$ $b0$ $a0$ to an anti-control version followed by X gate on $|b0\rangle$ to reset it to its original value. This replacement is shown in Figure 5.6.

Another observation we make at this step is regarding the three gates in the middle of the circuit diagram (Figure 5.5). Since qubit $|a1\rangle$ always starts with the value $|0\rangle$, then it is only flipped by the $TOFF$ gate if $|a0\rangle$ and $|b1\rangle$ are both $|1\rangle$ at this point in the circuit. This

qubit's value is then only used to conditionally flip $|b2\rangle$ in the next $CNOT$ gate, before it is uncomputed by the $TOFF$ that follows. This means that in reality, the $CNOT$ which flips $|b2\rangle$ actually depends on the values of $|a0\rangle$ and $|b1\rangle$ at that point in the circuit. And so these gates can effectively be replaced by one $TOFF$ gate acting on $|b2\rangle$ and controlled by the qubits $|a0\rangle$ and $|b1\rangle$, bypassing the dependency on $|a1\rangle$. We refer to this from now as **qubit dependency propagation**. This is also shown in Figure 5.6. At this point, qubit $|a1\rangle$ is effectively eliminated from the circuit.

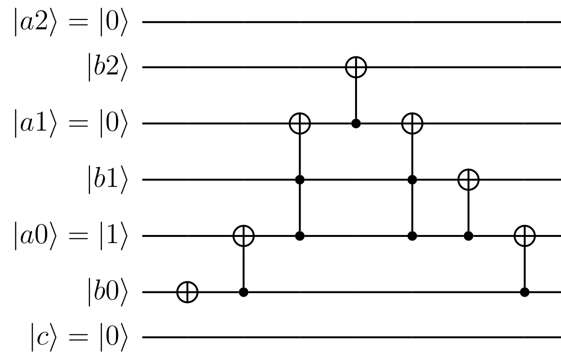


Figure 5.5: Step 3 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$.

At this point, the circuit only consists of the five gates as shown in Figure 5.6. Similar to the reduction of the three middle gates in the last step, we recognise that qubit $|a0\rangle$ acts as an intermediary dependency for the $TOFF$ and $CNOT$ gates in the middle of the circuit at this stage; with the real dependency being on qubit $|b0\rangle$. Since we know $|a0\rangle$ starts with the value $|1\rangle$, then we can say if $|b0\rangle$ starts with value $|0\rangle$, then the $|a0\rangle$ will be flipped to $|0\rangle$ (since it is an anti-control gate) and the middle gates will not have an effect (since they depend on $|a0\rangle$ having the value $|1\rangle$). However, if $|b0\rangle$ starts with value $|1\rangle$, then $|a0\rangle$ will retain its starting value $|1\rangle$ and the middle gates will remain. From this we infer that the middle gates in reality depend on $|b0\rangle$ having the value $|1\rangle$. And since the last anti-control gate guarantees that $|a0\rangle$ will be uncomputed, we can safely replace all four gates with two gates: a $TOFF$ gate acting on $|b2\rangle$ with controls $|b1\rangle$ and $|b0\rangle$, and a $CNOT$ gate acting on $|b1\rangle$ with $|b0\rangle$ as a control. The last X gate on $b0$ remains as it is.

The final version of this circuit after these transformations is shown in Figure 5.7. With the register $|a2|a1|a0\rangle$ and the qubit $|c\rangle$ eliminated from the circuit, this is now essentially a 3-qubit circuit which adds 1 to the input 3-bit integer $|b2|b1|b0\rangle$.

Specialising for $|a2|a1|a0\rangle = |010\rangle$

Now we discuss the example of specialising for the an input register $|a2|a1|a0\rangle$ having the value $|010\rangle$. Since like in the previous example, $|a2\rangle$ starts with value $|0\rangle$, and the structure

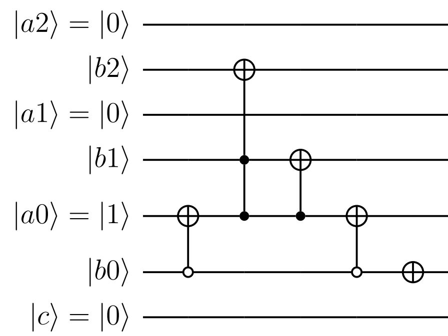


Figure 5.6: Step 4 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$.

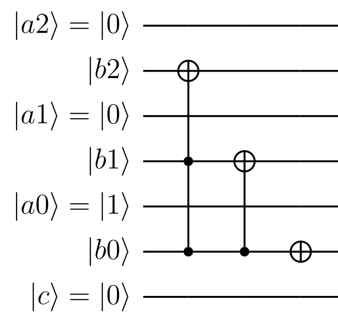


Figure 5.7: Step 5 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |001\rangle$.

of the middle part of the circuit is the same; we can apply the exact same gate elimination as in step 1 of the last example. Thus, our starting point is the circuit structure shown in Figure 5.3.

From this starting point, we tackle the middle part of the circuit. We recognise the $CNOT$s acting on $|b1\rangle$ and $|a0\rangle$ controlled by $|a1\rangle$ will always flip these qubits since $|a1\rangle = |0\rangle$ at this point. Tackling only the second $CNOT$ for now (the one acting on $|a0\rangle$), we can see that this gets uncomputed after the $TOFF$-$CNOT$-$TOFF$ sequence in the middle of the circuit. And since this sequence also guarantees that $|a1\rangle$ will be back to its original value of $|1\rangle$ at the end of the sequence, then this uncomputation will always have its effect. This means we can convert the control on $|a0\rangle$ of both $TOFF$s to an anti-control and eliminate the surrounding $CNOT$s.

Now looking again at the $CNOT$ acting on $|b1\rangle$ controlled by $|a1\rangle$; even though this does not get uncomputed after the $TOFF$-$CNOT$-$TOFF$ sequence, we can still eliminate it and convert the control on $|b1\rangle$ of the two middle $TOFF$s to an anti-control by adding an uncontrolled X gate on $|b1\rangle$ after the sequence.

After these transformations, the circuit looks as in Figure 5.8.

Next, we use the fact that $|a0\rangle = |c\rangle = |0\rangle$ to recognise the three gates at the start of the circuit and the three gates at the end of the circuit will have not effect at all and can be

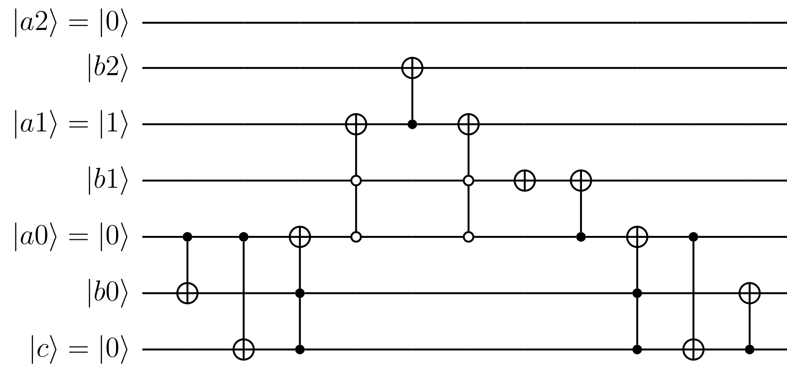


Figure 5.8: Step 2 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |010\rangle$.

eliminated. And so now we can see that we can remove the dependency on $|a0\rangle$ of the middle anti-$TOFF$s since it will always be $|a0\rangle$ will always be $|0\rangle$ at this stage of the circuit. The $CNOT$ acting on qubit $|b1\rangle$ controlled by $|a0\rangle$ can also be safely removed as a result of this. The circuit now is as in Figure 5.9.

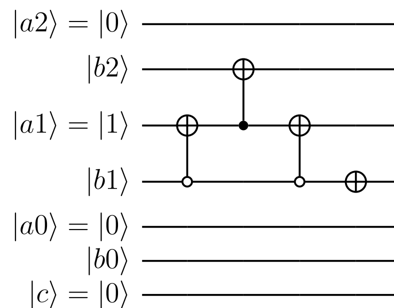


Figure 5.9: Step 3 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |010\rangle$.

Finally, we can apply dependency propagation similar to steps 4 and 5 in the previous example to remove the dependency of the $CNOT$ on $|a1\rangle$, to be directly dependent on qubit $|b1\rangle$. The X gate on $|b1\rangle$ at the end of the circuit remains untouched.

Figure 5.10 now shows the final version of this circuit. This is effectively a 3-qubit circuit which adds 2 to the 3-bit integer input $|b2|b1|b0\rangle$.

Specialising for $|a2|a1|a0\rangle = |011\rangle$

The final example is for specialising the circuit for the input register $|a2|a1|a0\rangle$ having the value $|011\rangle$. Just like the previous example, we can apply the same transformations to reach Figure 5.3, as $|a2\rangle$ still starts at $|0\rangle$. Also, since $|a1\rangle = |1\rangle$ like in the previous example, the second step follows exactly from the previous example, making our starting point here Figure 5.8.

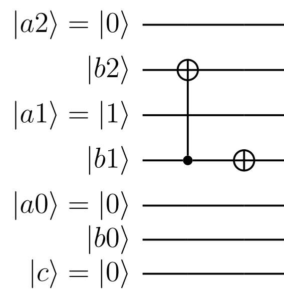


Figure 5.10: Step 4 of reducing 3-qubit Cuccaro Adder for $|a2|a1|a0\rangle = |010\rangle$.

To reduce qubit $|c\rangle$ here, we use the fact that $|a0\rangle$ starts as $|1\rangle$ and $|c\rangle$ starts as $|0\rangle$. Looking at the first 3 gates in the circuit, the first two $CNOT$s will always flip $|b0\rangle$ and $|c\rangle$ since the control, $|a0\rangle$, is $|1\rangle$. Then $|c\rangle$ will have a value of $|1\rangle$ and can be removed as a control of the following $TOFF$ gate since its control condition will always be satisfied. The resulting X on $|b0\rangle$ can then be absorbed into the control on $|b0\rangle$ of this $TOFF$ gate as an anti-control. The $TOFF$ now effectively become an X gate on $|a1\rangle$ anti-controlled by $|b0\rangle$. Note that as a result of this absorption, we later have to flip $|b0\rangle$ at the end of the circuit. Looking at the last 3 gates in the circuit, we apply the same to the $TOFF$ here; using the fact that a control on $|b0\rangle$ was absorbed, and that at this point $|c\rangle$ is still $|1\rangle$, this $TOFF$ also becomes an X gate on $|a1\rangle$ anti-controlled by $|b0\rangle$. Then the $CNOT$ on $|c\rangle$ has the effect of flipping it back to $|0\rangle$ (since $|a0\rangle$ retains its starting value of $|1\rangle$ due to uncomputation. And since $|c\rangle = |0\rangle$ again now, the final $CNOT$ on $|b0\rangle$ has no effect and can be safely removed. Finally, we have to add an X gate on $|b0\rangle$ as a result of the earlier absorption.

The circuit now looks as Figure 5.11.

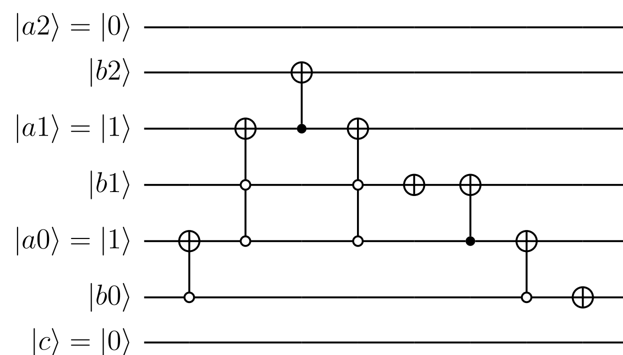


Figure 5.11: Step 3 of reducing 3-qubit Cuccaro Modulo Adder for $|a2|a1|a0\rangle = |011\rangle$.

The resulting circuit in Figure 5.13 is effectively a 3-qubit circuit adding 2 to the 3-bit input integer $|b2|b1|b0\rangle$.

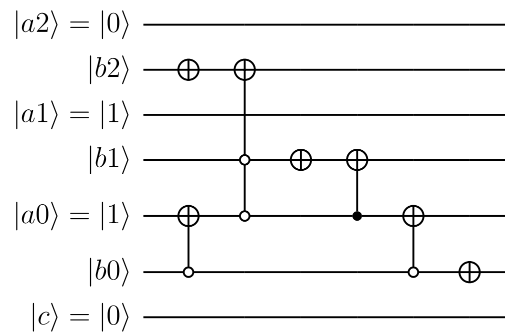


Figure 5.12: Step 4 of reducing 3-qubit Cuccaro Modulo Adder for $|a2|a1|a0\rangle = |011\rangle$.

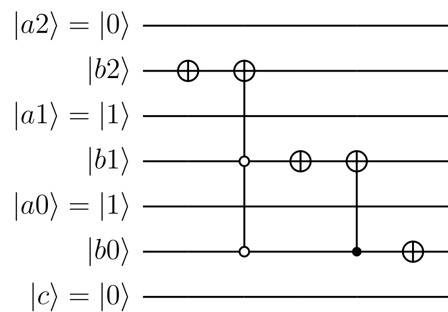


Figure 5.13: Step 5 of reducing 3-qubit Cuccaro Modulo Adder for $|a2|a1|a0\rangle = |011\rangle$.

5.5 Context and Published Works

The two published works [56, 125] present novel circuits targeting two different applications, and both present techniques for reducing those circuits by applying the strategies presented in the previous section. In this section, the contributions of both works are summarised; and for the purpose of demonstration, we focus on one of the two presented circuits, the D1Q3 CFD circuit presented in [56], and demonstrate how this circuit's width is reduced to fit in limited memory systems and allow for parallel simulation of different specialisations corresponding to different input values.

5.5.1 Contributions of the published works

The earlier work [56] was conducted in particular in the context of facilitating the simulation of quantum circuits which solve a particular Lattice Boltzmann Method model in the field of Computational Fluid Dynamics. In this section, this context is briefly described. The relevant contributions from this work are summarised in the following:

- Demonstration of how quantum algorithms including non-linear terms can be derived by employing quantum computational basis encoding. Although the use of this type of

encoding in general precludes exponential speed-ups relative to classical algorithms, it is expected that this approach has significant potential as part of larger quantum algorithms and when achieving polynomial speed-up is sufficient;

- Introduction of a quantum algorithm defining the non-linear equilibrium distribution function for the D1Q3 lattice model in the quantum computational basis;
- Introduction of a quantum-floating point format with reduced precision with key features of IEEE-754 standard, i.e. use of hidden-qubit approach for mantissa, the use of sub-normal numbers and consistent rounding (here, rounding-down to nearest);
- Detailed demonstration of how the derived circuits for the D1Q3 equilibrium distribution function in quantum computational basis can be transformed to facilitate efficient simulation on FPGAs;
- Introduction of a Haskell-based toolchain and eDSL for specifying and compiling quantum circuits for an FPGA-based architecture;

The latter work's contributions are summarised in the following [125]:

- Demonstration of quantum circuit width reduction transformations based on Circuit Parameterisation and Qubit Specialisation for quantum algorithms performing arithmetic operations as part of the computational work;
- Demonstration of the derivation steps used in creating parameterised quantum circuits for quantum arithmetic. The formulation in parameterised form for a quantum comparator and a quantum subtractor are detailed. To the best of the authors' knowledge, these parameterised circuits have not been considered in the literature before. The derivations detailed here also show how similar parameterisation can be applied to a wider range of arithmetic circuits;
- Analysis of the quantum circuit design of integer dividers in terms of suitability for the proposed quantum circuit width transformations;
- Demonstration how for the quantum divider exemplar the pre-computed and verified comparator and subtractor circuits in parameterised form can be imported into the complete quantum circuit implementation, followed by the automated selection of qubits suitable for specialisation and the automated specialisation for different user-defined inputs;
- Analysis in terms of circuit complexity of specialised quantum integer divider circuits as obtained from the transformation techniques introduced. The correctness of the circuits is verified using a quantum circuit simulator.

For most of the rest of this chapter, we focus on the earlier work's presented circuit, the D1Q3 circuit. These papers made these contributions with an overlap of demonstrating the circuit reduction technique. Based on these works, the primary contribution of this chapter is summarising the D1Q3 circuits and their reduction, as a demonstration of the circuit-width reduction techniques, showcasing a real-life example algorithm of their application.

5.6 The D1Q3 Circuit

We omit most of the CFD and Lattice-Boltzmann Method background regarding the D1Q3 circuit, and the method of encoding floating point numbers in the circuit; and instead focus directly on presenting the D1Q3 circuit, and explaining the input and output qubits of the utilised quantum register. A summary of the D1Q3 model was presented in Section 1.3.6. Parts of this section and the following section, showing the reduction of the circuit, are taken verbatim from the published work [56]. The biased quantum floating-point encoding method used in this work (including the hidden qubit approach) is demonstrated in Appendix A.

The circuit is shown across Figures 5.14 and 5.15

The quantum circuit was designed with input data encoded in qubits at the top of the circuit (most significant qubits), followed by qubits representing the output of the computation performed. The remaining qubits further 'down' (i.e. less significant in the employed memory indexing) generally act as ancillae qubits or as workspace. These ancillae and workspace qubits are all initialised in state $|0\rangle$ and will be returned to $|0\rangle$ at the time of completion of the quantum circuit.

$$\vec{g}^{eq} = \begin{pmatrix} -\frac{u}{2} + \frac{u^2}{2} \\ -\frac{u^2}{2} \\ -\frac{u^2}{2} \\ \frac{u}{2} + \frac{u^2}{2} \end{pmatrix} \tag{5.1}$$

Eq. 5.1 is the equation which the presented D1Q3 circuit is designed to compute. The specific component to compute is determined by the direction index encoded by the first two qubits in the circuit.

For this circuit, the qubit register for $N_M = 4$ and $N_E = 3$ can be summarised as follows:

$\|dv1\|dv0\rangle$	indices for 4 discrete velocities
$\|eu2\|eu1\|eu0\rangle$	3-bit representation of exponent of u
$\|su\rangle$	sign bit of input velocity u
$\|mu2\|mu1\|mu0\rangle$	3-bit representation of mantissa of u
$\|eg2\|eg1\|eg0\rangle$	3-bit representation of exponent of g^{eq}
$\|sg\rangle$	sign bit of output g^{eq}
$\|mg2\|mg1\|mg0\rangle$	3-bit representation of mantissa of g^{eq}
$\|esq2\|esq1\|esq0\rangle$	3-bit representation of exponent of u^2
$\|msq2\|msq1\|msq0\rangle$	3-bit representation of mantissa of u^2
$\|cut\rangle$	state $\|1\rangle$ defines that u^2 is truncated to 0
$\|r4\|qu3\|r3\|qu2\|r2\|qu1\|r1\|qu0\rangle$	workspace qubits ordered for 4-qubit addition
$\|c\rangle$	workspace qubit named to represent 'carry' bit in 4-qubit add
$\|r7\|r6\|r5\rangle$	workspace qubits ordered for 4-qubit squaring
$\|anc\rangle$	workspace qubit - mostly used as control qubit

This shows that 37 qubits are required in this original design. Data related to g_0 and g_3 are stored in the vector for $|dv1|dv0\rangle = |00\rangle$ and $|dv1|dv0\rangle = |11\rangle$. Similarly, states with $|dv1|dv0\rangle = |01\rangle$ and $|dv1|dv0\rangle = |10\rangle$ represent data for the (identical) distribution functions g_1 and g_2. For the floating-point representations of input u and each component of output $\vec{g}^{eq}$, 7 qubits are needed using the hidden-qubit approach. For the temporary storage of u^2, the sign bit can be omitted. In the following, for $N_M = 4$ and $N_E = 3$ an exponent 'bias=8' is used in the floating-point representations.

The following sections explain this circuit in more detail. We describe the operation of the various components of the circuit including the $0011a$, $0011b$, $0011c$, 0110, and $CCu2$ operators, as well as the squaring operators. For brevity, we omit the circuit implementations of some of these operators, and present their reduced versions directly in Section 5.7.

5.6.1 D1Q3 Part 1

Figure 5.14 shows the first part of the quantum circuit designed to compute the equilibrium distribution function $\vec{g}^{eq}$ for $N_M = 4$ and $N_E = 3$. The first step involves setting $|icut\rangle = |1\rangle$ for the cases with a guaranteed truncation of u^2 - for $N_M = 4$ and $N_E = 3$ this truncation always occurs for $|eu2|eu1|eu0\rangle = |000\rangle$, $|001\rangle$ and $|010\rangle$. In the next step, the mantissa of u is set into the required positions in the workspace, defined by $|qu3|qu2|qu1|qu0\rangle$, for all cases

without truncation of u^2 to 0. The operation $SQ4$ then computes the square of the mantissa and stores the results in $|r7|r6|\ldots|r0\rangle$. Operation CC_{u2} then creates a (temporary) 'copy' of the u^2 defined in the quantum-floating format in the qubits $|esq2|esq1|esq0\rangle$ (defining exponent) and $|msq2|msq1|msq0\rangle$ (defining mantissa using hidden-qubit approach). For u^2 no sign qubit is needed. To clear the workspace for further use, operation $ISQ4$ un-computes the mantissa squaring, followed by the removal of the copy of the mantissa of u from qubits $|qu3|qu2|qu1|qu0\rangle$.

To define the equilibrium distribution functions for directions e_0 and e_3 (defined by $|dv1|dv0\rangle = |00\rangle$ and $|dv1|dv0\rangle = |11\rangle$), the addition of $\pm u$ and u^2 is required. This addition operator for the signed values defined in the $N_M = 4$ and $N_E = 3$ format is performed using a modulo-5 Cuccaro adder, denoted by $MA5$. Operation $0011a$ initialises this addition step, followed by the operation $0011b$ that uses the created result to define the equilibrium distribution functions for directions e_0 and e_3 in qubits $|eg2|eg1|eg0\rangle$ (exponent), $|sg\rangle$ (sign qubit) and $|mg2|mg1|mg0\rangle$ (mantissa qubits using hidden-qubit approach).

5.6.2 Shift-and-add Squaring

The quantum circuit implementations for $SQ4$ and $ISQ4$ are shown in Figure 5.16 and Figure 5.17, respectively. Here, $FAdd$ represents a 4-qubit Cuccaro full adder, and Rmv the un-computation of this adder. The required shift in the used shift-and-add approach are performed by Sh (in $SQ4$) and Sh' (shift in reversed direction in $ISQ4$). Following the definition of u^2 in quantum floating-point format, the equilibrium distribution for directions e_1 and e_2 (defined by $|dv1|dv0\rangle = |01\rangle$ and $|dv1|dv0\rangle = |10\rangle$) is defined using the operator 0110 in Figure 5.14.

5.6.3 D1Q3 Part 2

Figure 5.15 shows the second part of the quantum circuit designed to compute the equilibrium distribution function $\vec{g}^{eq}$ for $N_M = 4$ and $N_E = 3$. The first step involves the un-computation of the modulo-5 addition (denoted by $UMA5$), followed by operation $0011c$ used to clear the inputs to this addition. Upon completion of operation $0011c$, the 14 workspace qubits are all in state $|0\rangle$. However, at this stage, the temporary copy of u^2 still resides in qubits $|esq2|esq1|esq0\rangle$ and $|msq2|msq1|msq0\rangle$. To clear these qubits to state $|0\rangle$, the square of the mantissa of u^2 needs to be re-computed using $SQ4$. Then, CC_{u2} is used to set the 6 qubits defining u^2 in quantum floating point format to $|0\rangle$. Then, $ISQ4$ un-computes the mantissa squaring step. Finally the remaining workspace qubits can be cleared along with the qubit $|cut\rangle$, which for cases with truncation of u^2 to 0 is re-set to $|0\rangle$.

At this stage, the output of the quantum circuit has the required format: the qubits $|eg2|eg1|eg0\rangle$ (exponent), $|sg\rangle$ (sign qubit) and $|mg2|mg1|mg0\rangle$ define the equilibrium distribution function for all 4 directions in the modified D1Q3 model, while the rest of the qubits is left unchanged.

In operator 0110, the previously computed term u^2 is used to set $-u^2/2$ for the two 'rest' directions ($|dv1|dv0\rangle = |01\rangle$ and $|dv1|dv0\rangle = |10\rangle$). For these directions the '-' sign can be trivially introduced by setting the sign qubit of $|g\rangle^{eq}$, $|sg\rangle = |1\rangle$. For the direction 0 defined by $|dv1|dv0\rangle = |00\rangle$, the equilibrium distribution function is of the form $-u/2 + u^2/2$. For this direction (and for direction 3 with distribution function of the form $u/2 + u^2/2$), first the terms $-u + u^2$ and $+u + u^2$ are computed using a 5-qubit modulo adder $MA5$ (and its reverse $UMA5$), while the division by 2 is introduced when setting the result in quantum-floating point format. The sign change to $' - u'$ for direction 0 is created by switching to 2's complement notation or the 4 mantissa qubits. Based on the assumption of positive input velocity u, the 5 input qubits (as 'a' input to Cuccaro modulo adder) representing mantissa of u are initially set as $|0|mu3|mu2|mu1|mu0\rangle$ (with $|mu3\rangle = |0\rangle$ for sub-normal numbers). The quantum-circuit implementation of operator $SgnA5$ performing this sign change is shown in Figure 5.18. With $u \ll 1$ in the D1Q3 model, the term $-u + u^2$ will always be a negative number. Since the 4 mantissa qubits of $\vec{g}^{eq}$ are not stored in 2's complement formulation for negative values (i.e. the sign is defined by $|sg\rangle$), a similar sign change to $SgnA5$ needs to be applied on the output of the 5-qubit modulo adder. With the guaranteed negative result for direction 0, it is sufficient to perform the 2's complement conversion on only to the 4 qubits (for $N_M = 4$ considered here) used to define mantissa qubits of $\vec{g}^{eq}$. The operator $SgnB4$ performs this change. Its quantum-circuit implementation is also shown in Figure 5.18.

5.7 Reducing the D1Q3 Circuit

The quantum-circuit implementation for the computation of the equilibrium distribution functions of the modified D1Q3 model shown in Figure 5.14 and Figure 5.15 will be transformed in this section to facilitate a more efficient quantum circuit simulation using FPGA acceleration. The 'original' circuit designed for $N_M = 4$ and $N_E = 3$ uses 37 qubits. As a first step toward 'reduced' circuits, circuit re-ordering and partial specialisation of one or more qubits is used.

The key ideas behind the considered reduction/transformation are as follows:

- The circuit evaluates the distribution functions for 4 directions based on a single input u defined in quantum floating-point format - therefore specialised circuits can be created based on the selected input for u;

- The specialised input is defined using 7 qubits: $|eu2|ue1|e0\rangle$ (exponent), $|su\rangle$ (sign) and $|mu2|mu1|mu0\rangle$ (defines mantissa using hidden-qubit appraoch) - so reduced circuits with 7 fewer can be created for particular values of u;
- The two most significant qubits in the design shown in Figure 5.14 and Figure 5.15, i.e. $|dv1|dv0\rangle$ define the direction vector ($i \in [0, 3]$). If required, a further reduction or transformation can be performed, so that only the equilibrium distribution function for a single direction is evaluated. This allows a further reduction by 2 qubits relative to the original circuit;
- The qubit $|cut\rangle$ only depends on the exponent of u, so at compile time for specialised circuits this information regarding truncation of u^2 is known. Therefore, also $|cut\rangle$ can be eliminated in transformed circuits, specialised for specific u input

Using the above transformation steps, reduced circuits with 27 qubits can be created for $N_M = 4$ and $N_E = 3$, as compared to the original number of 37.

5.7.1 Further reduction

The transformation steps detailed in the previous section enabled a reduction to computational kernels with 27 qubits as compared to the 37-qubit full circuit. For a further reduction, the 14-qubit workspace needs to be transformed. Specifically, the arithmetic operations in $SQ4$, $ISQ4$, $MA5$ and $UMA5$ need to be transformed and specialised for specific inputs. Since the Quantum Computer simulator used here employs a memory allocation with the top qubits in the circuits shown acting as the most significant qubits, the overall circuit design shown in Figure 5.14 and Figure 5.15 needs to be changed: the qubits defining workspace need to be moved towards the top of the quantum circuit, and therefore, the qubits storing u^2 and $\vec{g}^{eq}$ in quantum floating-point format need to be moved down toward less significant bit locations.

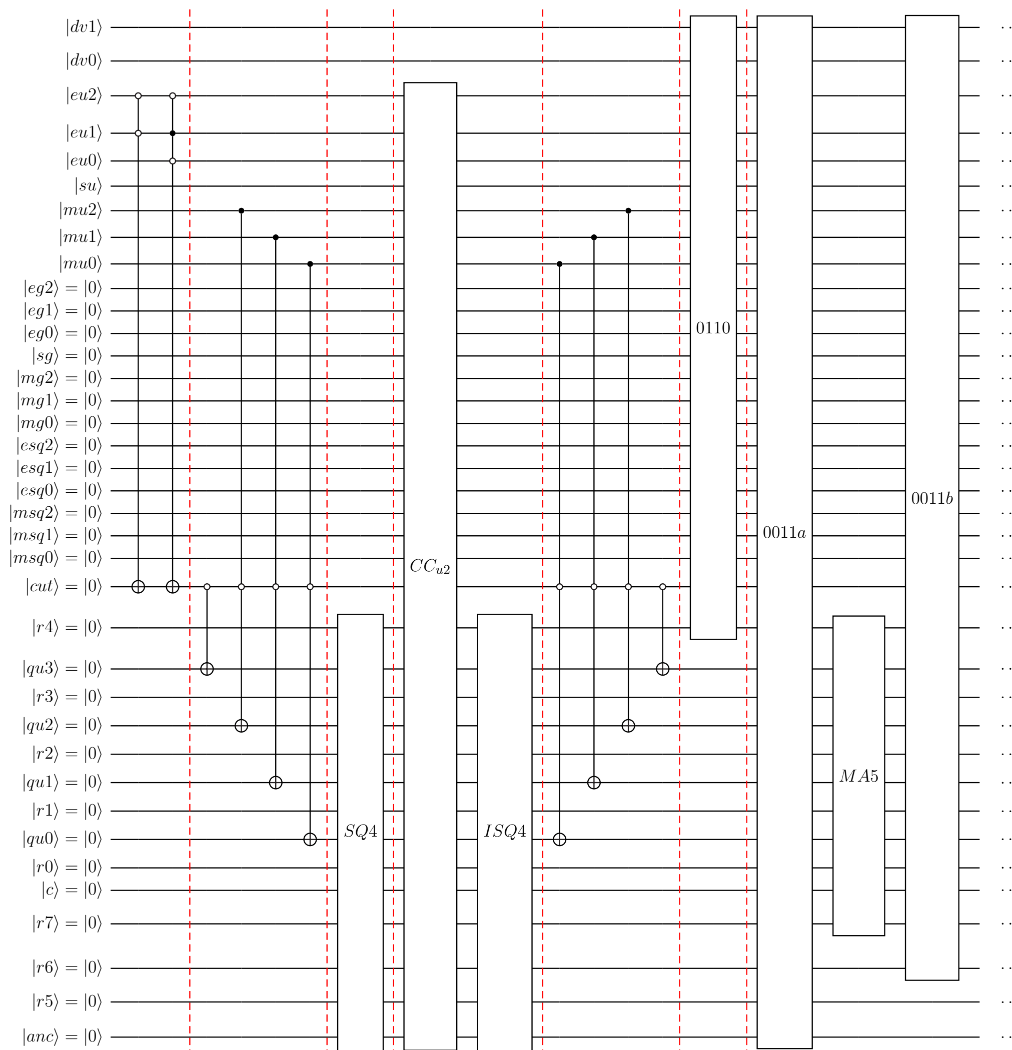
Following this re-ordering, a further reduction requires specialising (and factoring out) one or more mantissa qubits, starting from the most significant mantissa qubit of u, i.e. $|qu3\rangle$, followed by $|mu2\rangle$, etc. Using this approach, requires transformation to the u^2 computations as well as modulo-additions, as discussed in following sections.

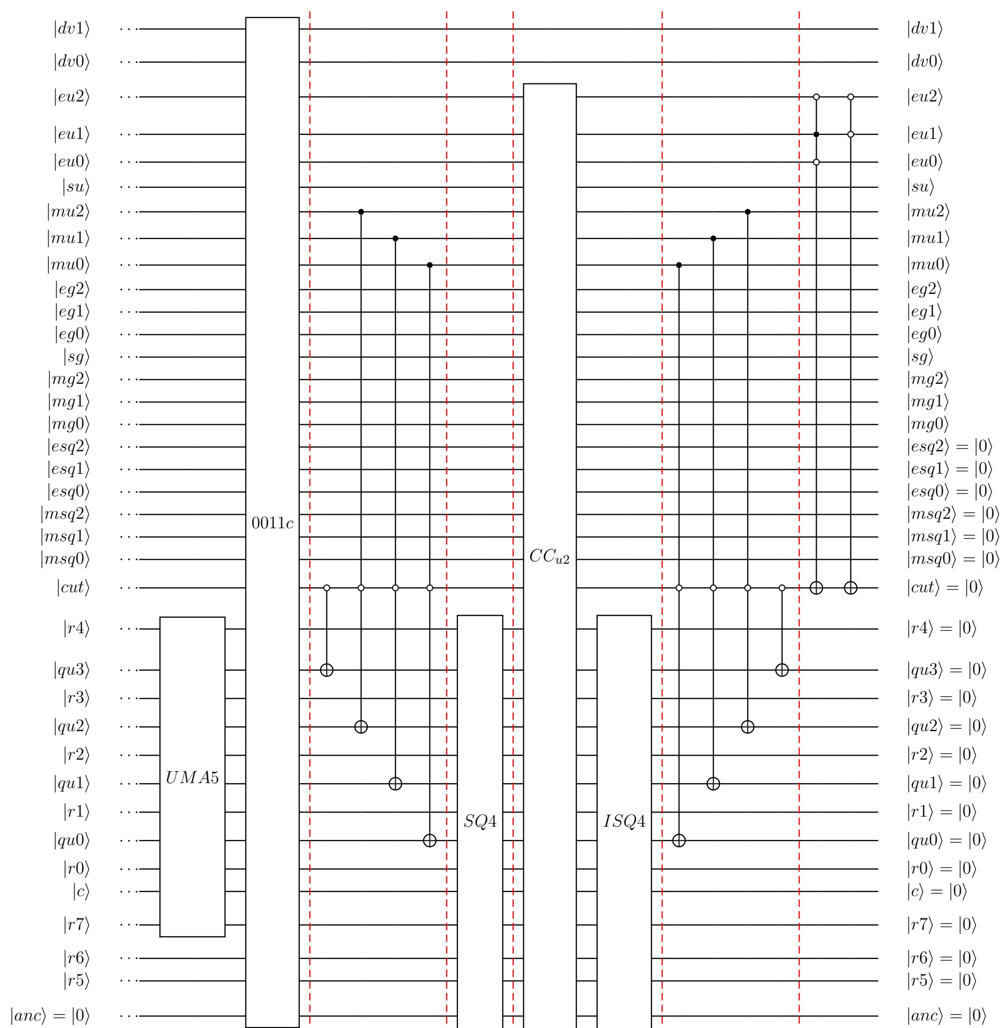
5.7.2 Reduction of the u^2 computation

To illustrate the further reduction of the number of qubits, the computation of u^2 is analyzed first. In the interest of clarity, this section will consider the reduced circuits for the evaluation of g_1^{eq} ($|dv1|dv0\rangle = |01\rangle$) and g_2^{eq} ($|dv1|dv0\rangle = |10\rangle$), since these identical terms

only involve the term $u^2/2$. Figure 5.19 shows the reduced quantum circuit with 26 qubit resulting from eliminating the two most-significant mantissa qubits in squaring operations for $N_M = 4$ and $N_E = 3$. As can be seen, the circuit only involves the two mantissa qubits $|qu1|qu0\rangle$, and $SQ4$ and $ISQ4$ represent reduced squaring (and un-computation) for specific choices of $|qu3|qu2\rangle$. For the further reduction by 2 qubits to 26-qubits, the operations $SQ4$ and $ISQ4$ involve 12 qubits in the workspace. The operation CC_{u2} sets the squared-velocity values in terms of quantum-floating point format in $|esq2|esq1|esq0\rangle$ (exponent) and $|msq2|msq1|msq0\rangle$ (mantissa), and for the considered reduction to 26 qubits, the quantum circuit implementations for CC_{u2} are detailed for $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$ or $|101\rangle$ in Figure 5.20. For $|eu2|eu1|eu0\rangle = |110\rangle$, the operator is identical to that for $|101\rangle$, apart from the NOT operation on $|esq1\rangle$ that for $|eu2|eu1|eu0\rangle = |110\rangle$ is performed on $|esq2\rangle$ instead.

Figure 5.21 shows the reduced quantum circuit with 25 qubit resulting from eliminating the three most-significant mantissa qubits in squaring operations for $N_M = 4$ and $N_E = 3$. As can be seen, the circuit only involves the mantissa qubit $|qu0\rangle$, and $SQ4$ and $ISQ4$ represent reduced squaring (and un-computation) for specific choices of $|qu3|qu2|qu1\rangle$. For the further reduction by 3 qubits to 25-qubits, the operations $SQ4$ and $ISQ4$ involve 11 qubits in the workspace. For $N_M = 4$, a further reduction to 24 qubis can be made, by reducing out $|qu0\rangle$ from the $SQ4$ and $ISQ4$ operations. For this further reduction to 24 qubits, quantum-circuit implementation of operations $SQ4$ and $ISQ4$ are shown in Figure 5.25. The addition and remove operators are specialised for the 4 mantissa qubits $|qu3|qu2|qu1|qu0\rangle = |1001\rangle$.


|dv1⟩
|dv0⟩
|eu2⟩
|eu1⟩
|eu0⟩
|su⟩
|mu2⟩
|mu1⟩
|mu0⟩
|eg2⟩ = |0⟩
|eg1⟩ = |0⟩
|eg0⟩ = |0⟩
|sg⟩ = |0⟩
|mg2⟩ = |0⟩
|mg1⟩ = |0⟩
|mg0⟩ = |0⟩
|esq2⟩ = |0⟩
|esq1⟩ = |0⟩
|esq0⟩ = |0⟩
|msq2⟩ = |0⟩
|msq1⟩ = |0⟩
|msq0⟩ = |0⟩
|cut⟩ = |0⟩
|r4⟩ = |0⟩
|qu3⟩ = |0⟩
|r3⟩ = |0⟩
|qu2⟩ = |0⟩
|r2⟩ = |0⟩
|qu1⟩ = |0⟩
|r1⟩ = |0⟩
|qu0⟩ = |0⟩
|r0⟩ = |0⟩
|c⟩ = |0⟩
|r7⟩ = |0⟩
|r6⟩ = |0⟩
|r5⟩ = |0⟩
|anc⟩ = |0⟩
SQ4
CC_u2
ISQ4
0110
0011a
MA5
0011b


|dv1⟩
|dv0⟩
|eu2⟩
|eu1⟩
|eu0⟩
|su⟩
|mu2⟩
|mu1⟩
|mu0⟩
|eg2⟩
|eg1⟩
|eg0⟩
|sg⟩
|mg2⟩
|mg1⟩
|mg0⟩
|esq2⟩
|esq1⟩
|esq0⟩
|msq2⟩
|msq1⟩
|msq0⟩
|cut⟩
|r4⟩
|qu3⟩
|r3⟩
|qu2⟩
|r2⟩
|qu1⟩
|r1⟩
|qu0⟩
|r0⟩
|c⟩
|r7⟩
|r6⟩
|r5⟩
|anc⟩ = |0⟩
UMA5
0011c
SQ4
CCu2
ISQ4
|dv1⟩
|dv0⟩
|eu2⟩
|eu1⟩
|eu0⟩
|su⟩
|mu2⟩
|mu1⟩
|mu0⟩
|eg2⟩
|eg1⟩
|eg0⟩
|sg⟩
|mg2⟩
|mg1⟩
|mg0⟩
|esq2⟩ = |0⟩
|esq1⟩ = |0⟩
|esq0⟩ = |0⟩
|msq2⟩ = |0⟩
|msq1⟩ = |0⟩
|msq0⟩ = |0⟩
|cut⟩ = |0⟩
|r4⟩ = |0⟩
|qu3⟩ = |0⟩
|r3⟩ = |0⟩
|qu2⟩ = |0⟩
|r2⟩ = |0⟩
|qu1⟩ = |0⟩
|r1⟩ = |0⟩
|qu0⟩ = |0⟩
|r0⟩ = |0⟩
|c⟩ = |0⟩
|r7⟩ = |0⟩
|r6⟩ = |0⟩
|r5⟩ = |0⟩
|anc⟩ = |0⟩

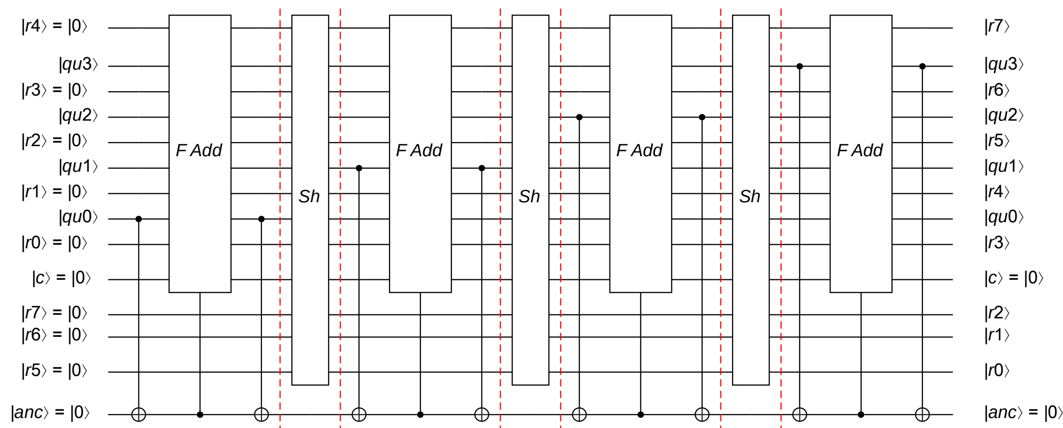


Figure 5.16: Quantum circuit defining $SQ4$ with 4-qubit mantissa qubits ($N_M = 4$, using hidden qubit approach) and a 3-qubit exponent ($N_E = 3$).

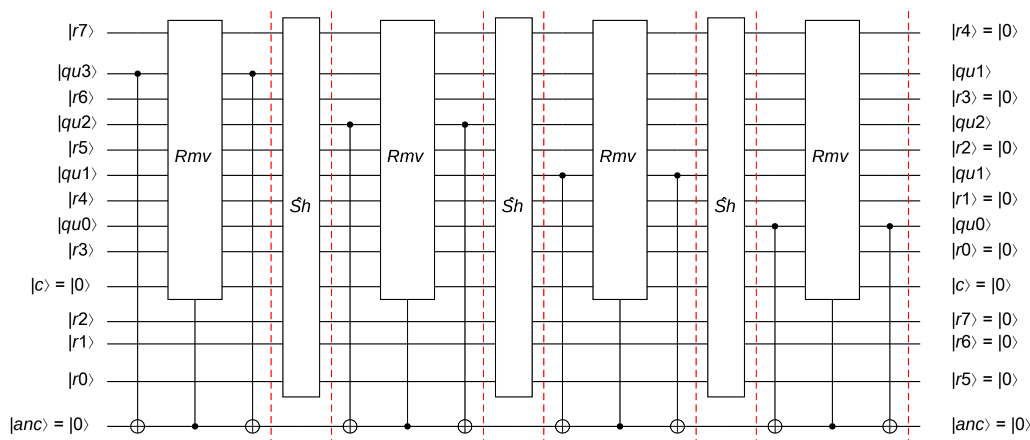


Figure 5.17: Quantum circuit defining $ISQ4$ with 4-qubit mantissa qubits ($N_M = 4$, using hidden qubit approach) and a 3-qubit exponent ($N_E = 3$).

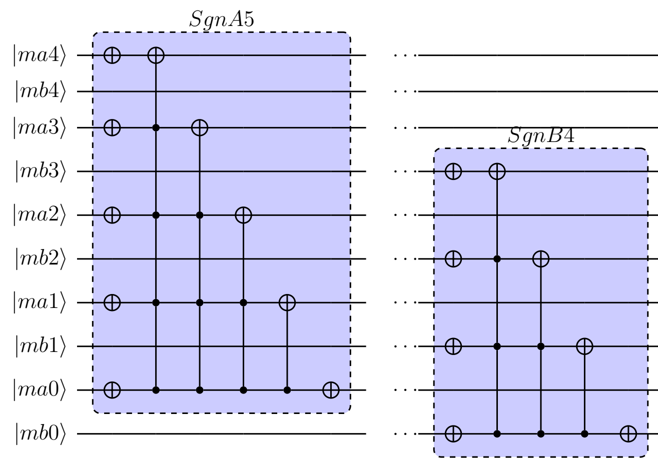


Figure 5.18: Quantum circuit-implementations of $SgnA5$ and $SgnB4$ used to introduced sign change in 5-qubit modulo addition.

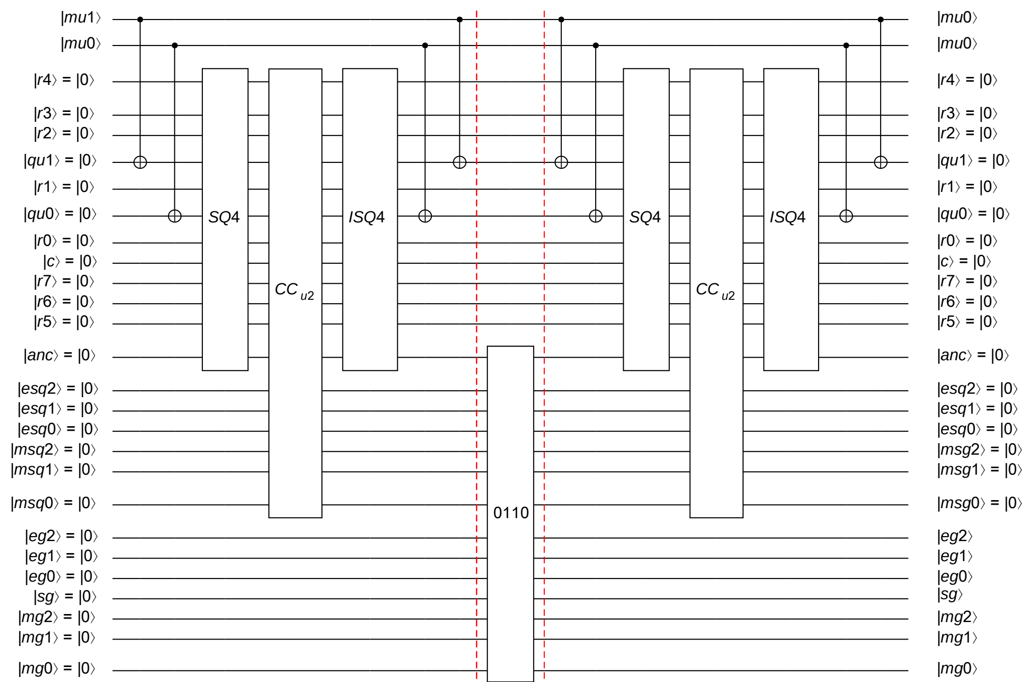


Figure 5.19: Reduced circuit with 26 qubits: $|dv1|dv0\rangle = |01\rangle$ or $|10\rangle$, $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$, $|101\rangle$ or $|110\rangle$ (and therefore $|cut\rangle = |0\rangle$).

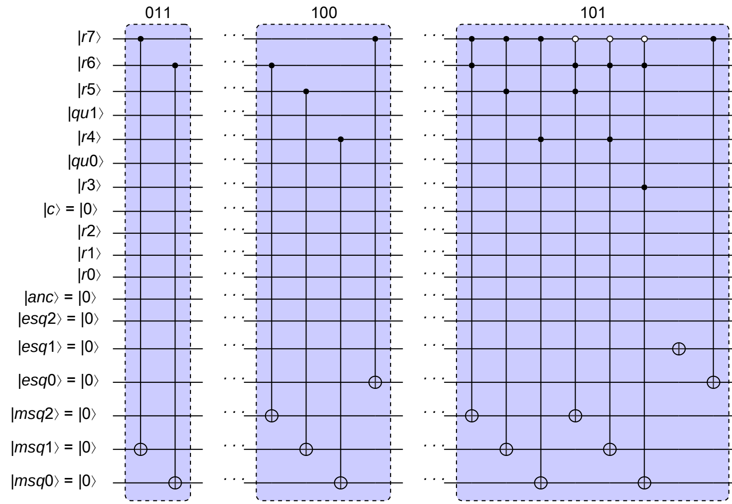


Figure 5.20: CC_{u2} for reduced circuit with 26 qubits. Squared velocity u^2 defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$. Label indicates for which $|eu2|eu1|eu0\rangle$ circuit was derived.

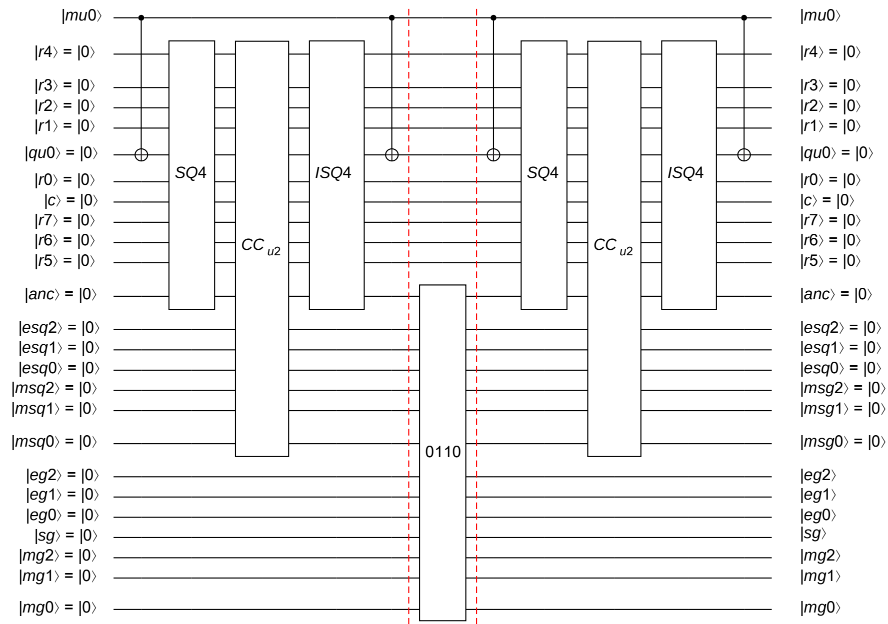


Figure 5.21: Reduced circuit with 25 qubits: $|dv1|dv0\rangle = |01\rangle$ or $|10\rangle$, $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$, $|101\rangle$ or $|110\rangle$ (and therefore $|cut\rangle = |0\rangle$).

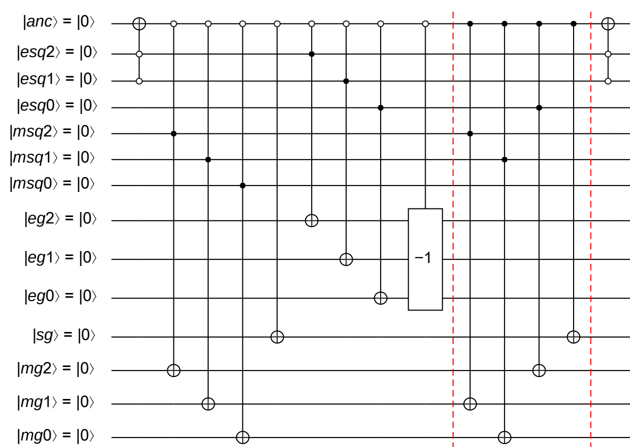


Figure 5.22: Operator 0110 for reduced circuit with 25 or 26 qubits. Squared velocity u^2 defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$.

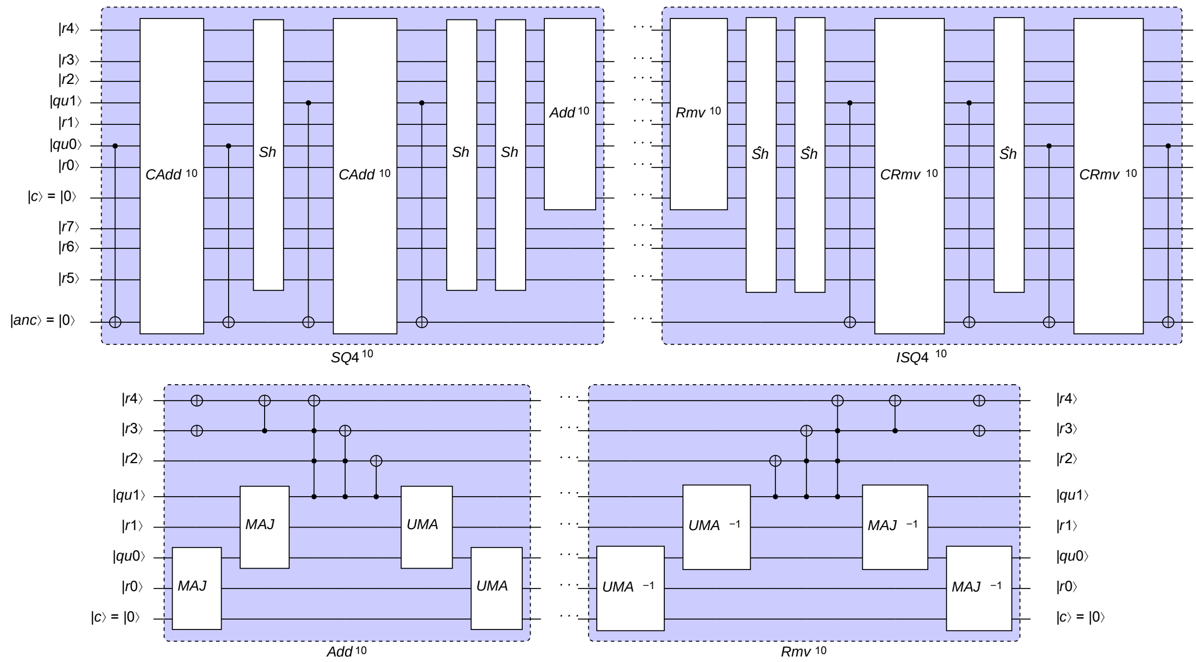


Figure 5.23: Definition of $SQ4$ and $ISQ4$ for reduced quantum circuit (26 qubit) for $|mu2\rangle = |0\rangle$ ($|qu3|qu2\rangle = |10\rangle$ for normalised input). $CAdd^{10}$ and $CRmv^{10}$ are defined as Add^{10} and Rmv^{10} with $|anc\rangle$ acting as control qubit.

The operation CC_{u2} defines squared-velocity values in terms of quantum-floating point format, and for the considered reduction to 25 qubits, the quantum circuit implementations follows directly from those discussed for the reduction to 26 qubits, since $|qu1|qu0\rangle$ are not involved in this step. The operation 0110 finally sets the equilibrium distributions functions g_1^{eq} (for $|dv1|dv0\rangle = |01\rangle$) and g_2^{eq} (for $|dv1|dv0\rangle = |10\rangle$) in quantum floating point format, based on the definition of u^2 in floating-point format defined previously. The quantum circuit implementation of 0110 for the reduced circuits with 25 and 26 qubits (identical in both cases) is shown in Figure 5.22.

5.7.3 Reduction including the modulo-adder

Following the analysis of the evaluation of the u^2 terms in quantum circuits with partial reduction of the workspace qubits, the focus now moves to the more complex case of also reducing the modulo adder (and its reverse) used in computing the summations $u + u^2$ and $-u + u^2$. The use of 2's complement method in the current implementation of $\vec{g}_{eq}$ evaluation poses particular challenges in the further reduction process, as detailed in this section. In the interest of clarity, this section will consider the reduced circuits for the evaluation of g_0^{eq} ($|dv1|dv0\rangle = |00\rangle$) and g_3^{eq} ($|dv1|dv0\rangle = |00\rangle$), since these represent the only directions for which the modulo-addition steps is required. Figure 5.27 shows the reduced quantum circuit with 25 qubit resulting from eliminating the three most-significant mantissa qubits

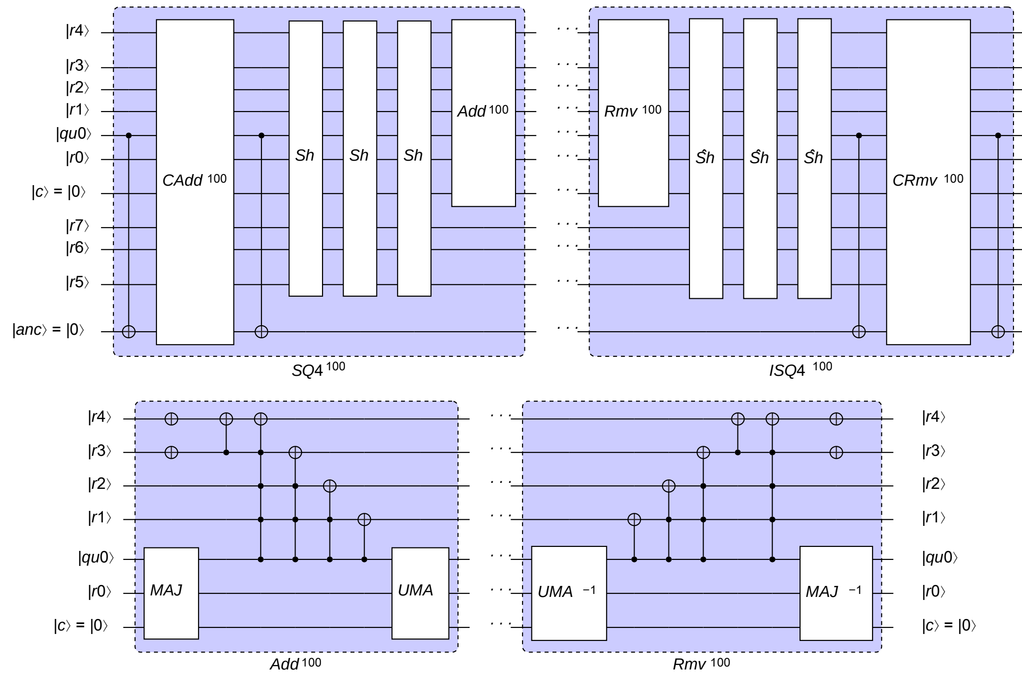


Figure 5.24: Definition of $SQ4$ and $ISQ4$ for reduced quantum circuit (25 qubit) for $|mu2|mu1\rangle = |00\rangle$ ($|qu3|qu2|qu1\rangle = |100\rangle$ for normalised input). $CAdd^{100}$ and $CRmv^{100}$ are defined as Add^{100} and Rmv^{100} with $|anc\rangle$ acting as control qubit.

in the arithmetic operations for $N_M = 4$ and $N_E = 3$. As can be seen, the circuit only involves the mantissa qubit $|qu0\rangle$. The operation $0011a$ prepares the modulo-adder step and depends on the mantissa qubits of u as well as on the exponent of u (represented by $|eu2|eu1|eu0\rangle$). Similarly $0011c$ un-computes these steps to reset workspace to $|0\rangle$ after completion of the addition step and setting g_0^{eq} and g_3^{eq} in quantum floating point format. The (reduced) modulo-5 adder $MA5$ (and its reverse $UMA5$) only depend on the specific choice of mantissa qubits that were reduced, i.e. in this step there is no dependency on exponent $|eu2|eu1|eu0\rangle$. Based on the outcome of modulo-5 adder, the operation $0011b$ sets g_0^{eq} and g_3^{eq} in $|mg2|mg1|mg0\rangle$ (mantissa), $|sg\rangle$ (sign) and $|eg2|eg1|eg0\rangle$ (exponents) for $|dv1|dv0\rangle = |00\rangle$ and $|dv1|dv0\rangle = |11\rangle$, respectively.

Following the process detailed in the previous section, for a further reduction by 2 qubits to 26 qubits would result in a similar quantum circuit, then involving $|qu1\rangle$ and $|qu0\rangle$. For brevity, this quantum circuit is not shown here.

The operation $0011a$ comprises two parts. The first part uses quantum-floating point representation of u^2 to set the 'b' register of the modulo-5 Cuccaro adder (i.e. the register that gets overwritten with addition result). Here, a shift is used to account for the difference in exponent of u and u^2. This part of the $0011a$ is not affected by the partial reduction of workspace qubits. The second part of $0011a$ set $-u$ (for $|dv1|dv0\rangle = |00\rangle$) or u (for $|dv1|dv0\rangle = |11\rangle$)

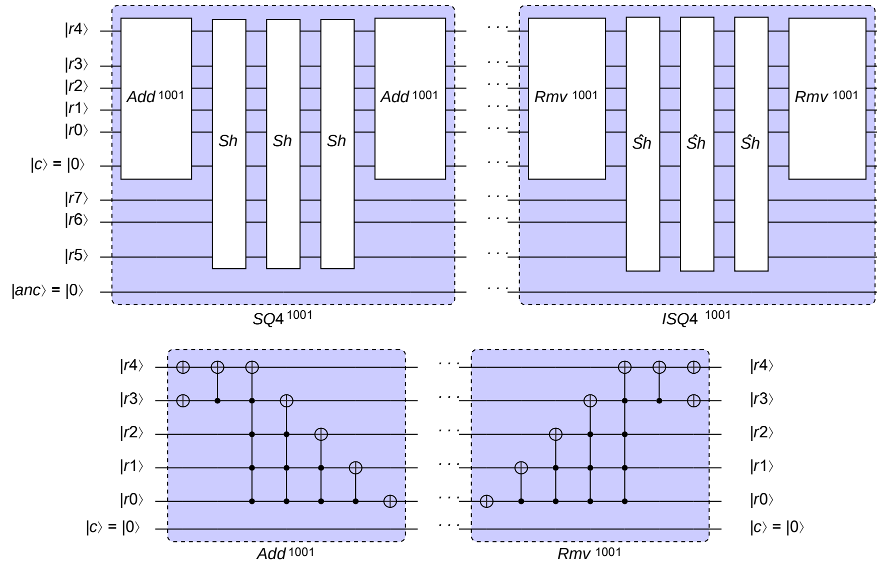


Figure 5.25: Definition of $SQ4$ and $ISQ4$ for reduced quantum circuit (24 qubit) for $|mu2|mu1|mu0\rangle = |001\rangle$ ($|qu3|qu2|qu1|qu0\rangle = |1001\rangle$ for normalised input). Controlled add/remove units not needed for the further reduction by 4 qubits (all mantissa qubits for $N_M = 4$).

into ’a’ register of modulo-5 adder. Figure 5.28 shows the quantum-circuit implementation of this 2nd step before a reduction of workspace qubits is performed. It can be seen that in a partial reduction of the workspace qubits, one or more of the qubits in the ’a’ register need to be removed. For the example of the reduction to 26 qubits, the qubits $|a4|a3|a2\rangle$ will be eliminated. For the reduction to 25 qubits, qubits $|a4|a3|a2|a1\rangle$ will be involved as indicated with the red box in Figure 5.28.

The quantum circuit illustrated in Figure 5.28 first ’copies’ the mantissa qubits (including the ’hidden’ qubit) into the ’a’ register for $|dv1|dv0\rangle = |00\rangle$ and for $|dv1|dv0\rangle$. To perform $-u + u^2$, the 2nd part of the circuit transforms the qubits to 2’s complement for $|dv1|dv0\rangle = |00\rangle$. Without this change to 2’s complement formulation, i.e. for $|dv1|dv0\rangle = |11\rangle$ the reduction of the workspace for the modulo-5 adder can be performed using the approach previously shown for the 4-qubit adders in the evaluation of u^2.

For $|dv1|dv0\rangle = |00\rangle$, in most cases, the need arises to use different computational kernels (derived for different choice of the reduced qubits), depending of the state of the remaining mantissa qubits of u. The following three examples illustrate this:

I. Reduction by 2 qubits to 26-qubit circuit, $|mu2|mu1\rangle = |0\rangle$. For normalised inputs, we then have $|qu3|qu2\rangle = |10\rangle$ in the representation without hidden qubit. For $|dv1|dv0\rangle = |00\rangle$: ’-u’ into modulo-5 adder, now 4 cases need to be considered:

- $|qu3|qu2|qu1|qu0\rangle = |1000\rangle$ and there for ’-u’: $|11000\rangle$

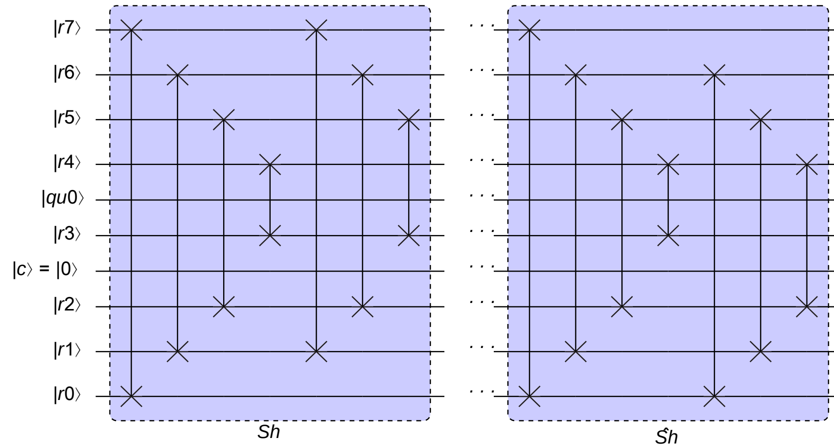


Figure 5.26: Shift operators used for left- and right-shifting of results register in reduced circuit with 25 qubits

- $|qu3|qu2|qu1|qu0\rangle = |1001\rangle$ and there for '-u': $|10111\rangle$
- $|qu3|qu2|qu1|qu0\rangle = |1010\rangle$ and there for '-u': $|10110\rangle$
- $|qu3|qu2|qu1|qu0\rangle = |1011\rangle$ and there for '-u': $|10101\rangle$

showing that modulo-5 kernels derived for $|a4|a3|a2\rangle = |110\rangle$ and for $|a4|a3|a2\rangle = |101\rangle$ are needed, depending on $|qu1|qu0\rangle$. In addition for $|qu0\rangle = |1\rangle$ a NOT operation on $|qu1\rangle$ just before and after performing the addition is needed;

II. Reduction by 3 qubits to 25-qubit circuit, $|mu2|mu1\rangle = |00\rangle$. For normalised inputs, we then have $|qu3|qu2|mu1\rangle = |100\rangle$ in the representation without hidden qubit. For $|dv1|dv0\rangle = |00\rangle$: '-u' into modulo-5 adder, now 2 cases need to be considered:

- $|qu3|qu2|qu1|qu0\rangle = |1000\rangle$ and there for '-u': $|11000\rangle$
- $|qu3|qu2|qu1|qu0\rangle = |1001\rangle$ and there for '-u': $|10111\rangle$

similar to the reduction-to-26-qubits example, it follows that modulo-5 kernels derived for $|a4|a3|a2\rangle = |1100\rangle$ and for $|a4|a3|a2\rangle = |1011\rangle$ are needed for this example, again with a switch between these kernels that depends on $|qu1|qu0\rangle$;

III. Reduction by 4 qubits to 24-qubit circuit, i.e. with all mantissa qubits for $N_M = 4$ reduced out in the transformed circuit. For the example $|mu2|mu1|mu0\rangle = |001\rangle$, the normalised input showing the hidden qubit is $|qu3|qu2|qu1|qu0\rangle = |1001\rangle$. For $|dv1|dv0\rangle = |00\rangle$, '-u' is then represented as $|10111\rangle$

For the first two examples, Figure 5.29 shows the quantum-circuit implementation of the reduced $MA5$ operation for $|dv1|dv0\rangle = |00\rangle$. The circuits were derived by first creating two

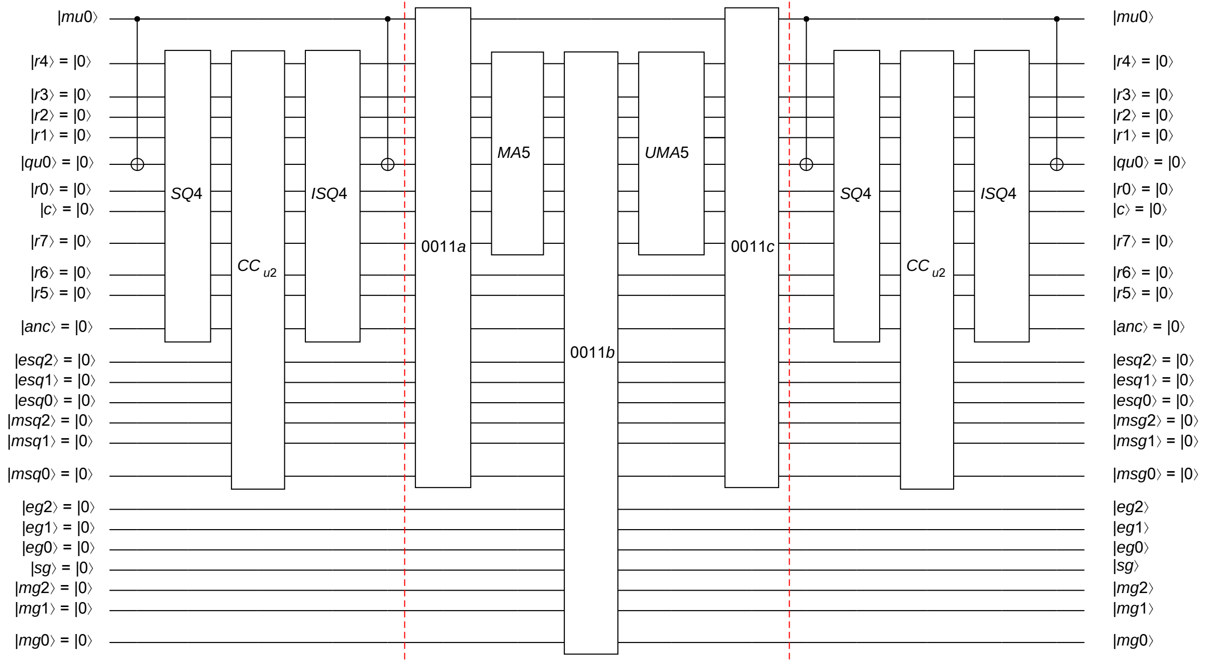


Figure 5.27: Reduced circuit with 25 qubits: $|dv1|dv0\rangle = |00\rangle$ or $|11\rangle$, $|eu2|eu1|eu0\rangle = |011\rangle$, $|100\rangle$, $|101\rangle$ or $|110\rangle$ (and therefore $|cut\rangle = |0\rangle$).

separate kernels, e.g. for $|a4|a3|a2\rangle = |110\rangle$ and for $|a4|a3|a2\rangle = |101\rangle$ in the reduction-by-2 qubits example. Then, the differences between the kernels are performed conditional and ancilla qubit $|anc\rangle$ in the combined circuits shown.

After reaching the final versions of the reduced circuits as above, they are implemented by encoding them in the Compilation tool's eDSL, which then generates the intermediate representation to facilitate the simulation by any of the developed architectures.

5.8 Complexity Analysis and Evaluation of Reduced D1Q3

For the full quantum circuit introduced here, the case with 4 mantissa qubits ($N_M = 4$) and 3 exponent qubits ($N_E = 3$) is described in detail in the present work. A crucial factor in applying this quantum algorithm as well as its simulation on classical hardware is its computational complexity in terms of space (number of qubits - or circuit width) and time (number of gate operations and circuit depth) as a function of N_M and N_E. For a well-conditioned computational problem such as the flow field governed by the D1Q3 model (with velocity u, squared velocity u^2 and re-scaled distribution functions all $\ll 1$ in magnitude), it can be expected that meaningful simulations can be performed with $N_E = 3$. However, to reduce the impact of rounding and truncation errors, realistic applications will involve

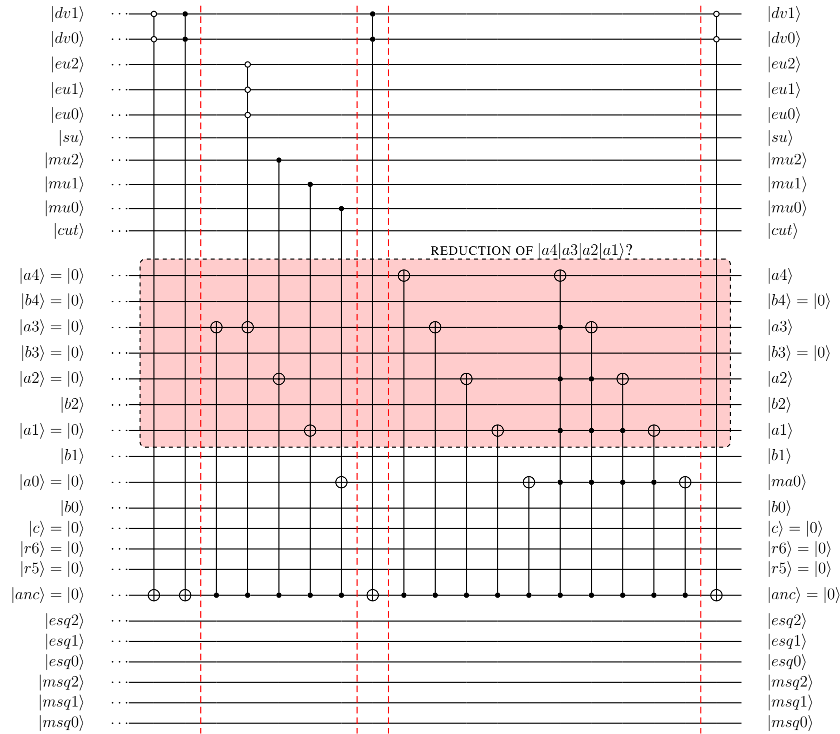
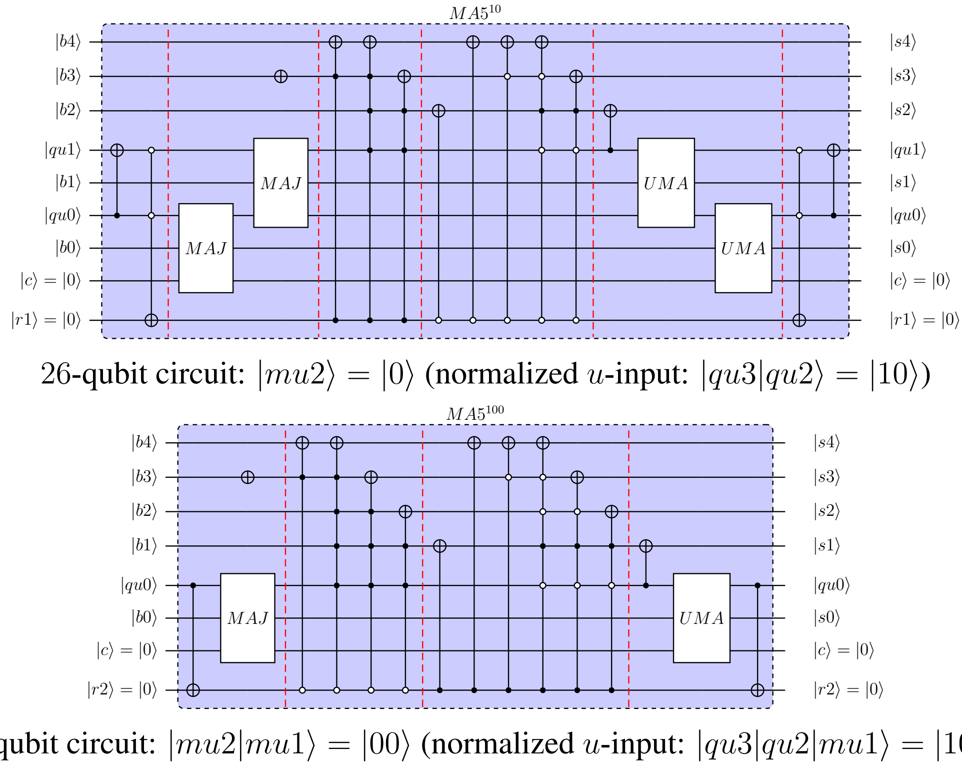


Figure 5.28: Re-arranged quantum circuit (without further reduction on workspace qubits). Quantum circuit shows setting of $-u$ (for $|dv1|dv0\rangle = |00\rangle$) or u (for $|dv1|dv0\rangle = |11\rangle$) into 'a' register of modulo-5 adder. Velocity u and squared velocity u^2 are defined as quantum floating-point numbers with $N_M = 4$ and $N_E = 3$.

$N_M > 4$. Therefore, the growth of circuit width and depth as function of increasing values of N_M is of particular interest in the complexity analysis presented here.

5.8.1 Full circuit - before reduction

For the D1Q3 model 2 qubits $|dv1|dv0\rangle$ are required independent of N_M and N_E. Using the hidden-qubit approach, the u-velocity is defined in terms of exponent, sign and mantissa using $N_E + 1 + N_M - 1 = N_E + N_M$ qubits. For the output $\vec{g}^{eq}$, similarly $N_E + N_M$ are required since the same floating-point format is used. A single qubit $|cut\rangle$ is used to identify cases with truncation to 0 of u^2. The temporary storage of u^2 in floating-point representation requires $N_E + N_M - 1$ qubits since a 'sign' qubit is not required. The remaining qubits represent the 'workspace' of the algorithm. Two computational steps are performed (as well as their un-computation) within this space: the shift-and-add based evaluation of u^2 and the 'signed' addition $\pm u + u^2$ for directions $|dv1|dv0\rangle = |00\rangle$ and $|dv1|dv0\rangle = |00\rangle$. For the example $N_M = 4$ this addition is performed using a 5-qubit modulo adder. The modulo adder requires $2N_M + 1$ qubits. The u^2 evaluation a $2N_M$ results register, N_M qubits for unsigned velocity input, 1 'carry' qubit as well as 1 ancilla qubit. In total, the u^2 evaluation therefore requires $3N_M + 2$ qubits. This exceeds the requirement of the modulo adder and



26-qubit circuit: $|mu2\rangle = |0\rangle$ (normalized u-input: $|qu3|qu2\rangle = |10\rangle$)

25-qubit circuit: $|mu2|mu1\rangle = |00\rangle$ (normalized u-input: $|qu3|qu2|mu1\rangle = |100\rangle$)

Figure 5.29: Quantum circuits for $MA5$ used in reduced circuit with 26 qubits and with 25 qubits for $|dv1|dv0\rangle = |00\rangle$.

therefore the workspace needs a total of $3N_M + 2$ qubits. The total space complexity of the original quantum circuit therefore is:

$$2 + 2\big(N_E + N_M\big) + 1 + \big(N_E + N_M - 1\big) + \big(3N_M + 2\big) = 4 + 3N_E + 6N_M$$

5.8.2 Quantum circuit after reduction steps

After the first reduction step introduced in Section 5.7, the following qubits were removed from the original circuit: 2 qubits $|dv1|dv0\rangle$, N_E+N_M (input 'u' in signed floating-point format) as well as the single $|cut\rangle$ qubit. Therefore, the total space complexity for the quantum circuit following this 1st reduction step is therefore:

$$4 + 3N_E + 6N_M - \big[2 + \big(N_E + N_M\big) + 1\big] = 1 + 2N_E + 5N_M$$

The further reduction detailed in Section 5.7.1, one or more of the N_M qubits defining the unsigned velocity input into the u^2 evaluation were eliminated. In its most aggressive form, this reduction step can eliminate all N_M qubits defining the unsigned velocity input, so that the space complexity reduces further to:

$$1 + 2N_E + 5N_M - N_M = 1 + 2N_E + 4N_M$$

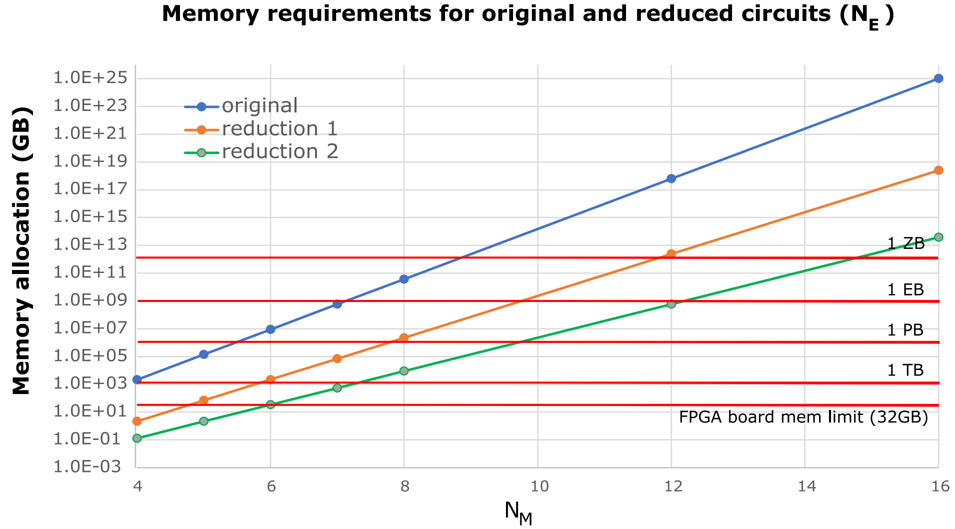


Figure 5.30: Memory requirements for the original circuit and the two types of reduced circuits as a function of N_M, for $N_E = 3$

To illustrate the complexity for different choices of N_M, Table 5.1 summarised the required number of qubits for $N_E = 3$ and increasing N_M for the original quantum circuits as well as the reduction steps 1 and 2.

Table 5.1: Required number of qubits for original and transformed quantum circuits ($N_E = 3$).

N_M	original	reduction 1	reduction 2
4	$4+9+24=37$	$1+6+20=27$	$1+6+16=23$
5	$4+9+30=43$	$1+6+25=32$	$1+6+20=27$
6	$4+9+36=49$	$1+6+30=37$	$1+6+24=31$
7	$4+9+42=55$	$1+6+35=42$	$1+6+28=35$
8	$4+9+48=61$	$1+6+40=47$	$1+6+32=39$
12	$4+9+72=85$	$1+6+60=67$	$1+6+48=55$
16	$4+9+96=109$	$1+6+80=87$	$1+6+64=71$

Figure 5.30 shows the memory required to store the qubit information as a pair of 32-bit floating point numbers. For reference, the red lines show the FPGA board memory and 1 TB, 1 PB, 1 EB and 1 ZB. To put this into context: the UK's largest supercomputer, Archer, comprises 4920 nodes with each 64 GB of memory, so the total memory is still less than 1 PB.

5.8.3 Examples of D1Q3 quantum circuit reduced to 25 qubits

In this section, two examples are considered of reduced circuits with 25 qubits:

- Example 1: velocity u defined as $|01000|110\rangle = +8/32 = +1/4$ or $|01001|110\rangle =$

$+9/32$. Alternatively, in terms of unsigned hidden-qubit formulation for mantissa: $|mu2|mu1|mu0|eu2|eu1|eu0\rangle = |000|110\rangle$ or $|mu2|mu1|mu0|eu2|eu1|eu0\rangle = |001|110\rangle$;

- Example 2: velocity u defined as $|01100|101\rangle = +12/64 = +3/16$ or $|01101|101\rangle = +13/64$. In terms of unsigned hidden-qubit formulation for mantissa: $|mu2|mu1|mu0|eu2|eu1|eu0\rangle = |100|101\rangle$ or $|mu2|mu1|mu0|eu2|eu1|eu0\rangle = |101|101\rangle$.

With a further reduction by 3 qubits, this means that only $|mu0\rangle$ acts as input qubit, and therefore the reduced circuits represented by both examples only compute 2 separate input velocities u each, as itemised above.

The equilibrium distribution functions g_0^{eq} (defined by $|dv1|dv0\rangle = |00\rangle$) and g_3^{eq} (defined by $|dv1|dv0\rangle = |11\rangle$) are shown in Table 5.2.

Table 5.2: Input and output of examples for 25-qubit example circuits. Mantissa is shown without 'hidden qubit'.

u	u^2	$g_0^{eq} = -u/2 + u^2/2$	$g_3^{eq} = u/2 + u^2/2$
$\|0\|110\|000\rangle = 8/32$	$\|0\|100\|000\rangle = 8/128$	$\|1\|101\|100\rangle = -6/64$	$\|0\|101\|010\rangle = 10/64$
$\|0\|110\|001\rangle = 9/32$	$\|0\|100\|010\rangle = 10/128$	$\|1\|101\|110\rangle = -7/64$	$\|0\|101\|011\rangle = 11/64$
$\|0\|101\|100\rangle = 12/64$	$\|0\|011\|001\rangle = 9/256$	$\|1\|100\|010\rangle = -10/128$	$\|0\|100\|110\rangle = 14/128$
$\|0\|101\|101\rangle = 13/64$	$\|0\|011\|010\rangle = 10/256$	$\|1\|100\|011\rangle = -11/128$	$\|0\|100\|111\rangle = 15/128$

Here, the term $-u + u^2$ required in g_0^{eq} was evaluated as follows using a modulo-5 adder. For positive numbers, a 5-qubit input with a leading $|0\rangle$, followed by the hidden qubit and $N_M - 1 = 3$ mantissa qubits is used, while for negative numbers, the 4-qubit representation (including hidden qubit) is transformed into its 5-qubit 2's complement. Furthermore, mantissa qubits are shifted where necessary to account for difference in exponents of the two inputs. Such shifts are performed before transformation to 2's complement. Then, for the 4 examples in the table above, the 'signed' addition can be summarised as:

$$
\begin{aligned}
u = 8/32 \quad (|0|110|000\rangle): \quad & |11000\rangle + |00010\rangle = |11010\rangle \\
& \rightarrow -u + u^2 = |1|101|100\rangle = -6/32 = -12/64 \\
u = 9/32 \quad (|0|110|001\rangle): \quad & |10111\rangle + |00010\rangle = |11001\rangle \\
& \rightarrow -u + u^2 = |1|101|110\rangle = -7/32 = -14/64 \\
u = 12/64 \quad (|0|101|100\rangle): \quad & |10100\rangle + |00010\rangle = |10110\rangle \\
& \rightarrow -u + u^2 = |1|101|010\rangle = -10/64 \\
u = 13/64 \quad (|0|101|101\rangle): \quad & |10011\rangle + |00010\rangle = |10101\rangle \\
& \rightarrow -u + u^2 = |1|101|011\rangle = -11/64
\end{aligned}
$$

As can be seen, in the top two examples, the outcomes $-6/32$ and $-7/32$ result from the modulo-5 mantissa addition, so that a re-normalisation is required. This leads to the nor-

malised numbers $-12/64$ and $-14/64$, respectively. For the bottom two examples, such a re-normalisation was not required.

5.8.4 Conclusion of D1Q3 Reduction

Quantum circuits for the non-linear equilibrium distribution function for the D1Q3 lattice-based model were introduced as a step towards more complete models such as D2Q9 and D3Q27. A key feature of the derived circuits is the use of the quantum computational basis encoding along with the use of a reduced-precision floating-point representation. This in contrast to existing work typically employing fixed-point representation in quantum algorithms using the quantum computational basis encoding. It is demonstrated that for modest precision (e.g. using 4-bit mantissas) quantum circuits with fewer than 40 qubits can be derived. Even with further ancillae qubits that result from transpiling the circuit to native gates available on quantum hardware, this shows that for Noisy Intermediate-Scale Quantum (NISQ) Computer-era hardware, demonstration of the introduced quantum circuits is feasible. The quantum-circuit transformation introduced and detailed in this work show that starting from the full circuit, reduced circuits can effectively be created so that step-by-step the behaviour of the full quantum circuit can be analysed using more limited resources, while taking advantage of the fine-grained parallelism offered by FPGAs.

We demonstrated reductions from 37 to 25 qubits for the quantum circuit computing the equilibrium distribution function of D1Q3 model with input velocity defined in floating-point format with a 4-qubit mantissa and 3-qubit exponent. For increased mantissa qubit count of 16, a reduction from 109 to 71 qubits occurs. In Chapter 6 We show that the reduced circuit can be successfully run on an FPGA system and that the FPGA simulation has more than $3\times$ better performance per Watt compared to the CPU simulation.

We note one important limitation which is that the quantum circuit reductions shown here were performed manually and certainly pose major challenges for automation by circuit compilation methods. This would be the subject of future work.

5.9 Circuits with Amplitude-Basis Encoding

This chapter has so far demonstrated how these techniques can be used to reduce the width of circuits that operate using computational-basis encoding. However, to truly take advantage of quantum computing, most algorithms use superposition and entanglement (i.e. utilise the amplitude basis). While we showed an example of applying this method to the QFT-based Draper adder in Section 5.4.1 (which makes use of these quantum phenomena); in that circuit, the qubits which were reduced out are always in the computational basis. We can however also apply these techniques to reduced qubits which themselves use amplitude-basis encoding as well.

In this section, we present a simple example to showcase how the presented reduction techniques can be used to simulate circuits which utilise both superposition and entanglement. Particularly important is the case where the qubit to be reduced out of the circuit is in superposition.

Consider the circuit shown in Figure 5.31, which acts on a 3-qubit register $|q_0|q_1|q_2\rangle$, and generates two different entanglement configurations for the bottom (w.r.t. the circuit diagram) two qubits, $|q_1\rangle$ and $|q_2\rangle$, based on the value of the top qubit, $|q_0\rangle$. In the examples in this section, the top qubit is always the most significant qubit. Recall the Hadamard gate's matrix is: $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

We can manually simulate this circuit here based on the value of the top qubit, and assuming an initial value for the bottom qubits of $|q_1|q_2\rangle = |00\rangle$, though of course we can choose any value for the bottom qubits in theory. If the top qubit's value is $|1\rangle$, then only the first two gates have an effect, and the system is entangled into the state: $\frac{1}{\sqrt{2}}(|100\rangle + |111\rangle)$. Otherwise, if $|q_0\rangle = |0\rangle$, then only the last three gates have an effect and the system is entangled into the state $\frac{1}{\sqrt{2}}(|000\rangle - |011\rangle)$.

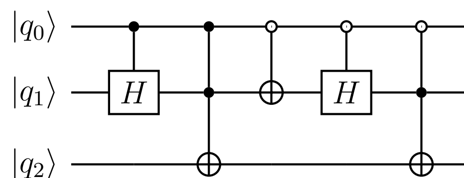


Figure 5.31: Proposed quantum circuit for demonstration of application of circuit reduction techniques to circuits utilising amplitude-basis encoding.

We can apply the circuit reduction techniques described in the earlier sections to split this circuit into two, shown in Figure 5.32. The first circuit (C_1, corresponding to $|q_0\rangle = |1\rangle$) is the usual entanglement circuit composed of a Hadamard gate, followed by a $CNOT$. The

second circuit (C_2, corresponding to $|q_0\rangle = |0\rangle$) is the same but flips the first qubit before entangling.

Figure 5.32: Split circuits based on the applying the circuit reduction techniques to $|q_0\rangle$ in the circuit shown in Figure 5.31. The circuit on the left corresponds to $|q_0\rangle = |1\rangle$, and the one on the right is for $|q_0\rangle = |0\rangle$.

Simulating these circuits individually on an initial state of $|q_1|q_2\rangle = |00\rangle$, we get the following two state vector results:

$$s_1 = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}, s_2 = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\-1\end{bmatrix}$$

In order to reconstruct the original 3-qubit state vectors, we must choose one of the initial values for the top qubit which we reduced out, and take its tensor product with the corresponding state resulting from the simulation of the reduced circuit.

Choosing $|q_0\rangle = |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$, we take the tensor product of this qubit with s_1 to get:

$$S_1 = \begin{bmatrix}0\\1\end{bmatrix} \otimes \frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}(|100\rangle + |111\rangle)$$

Similarly, for the choice of $|q_0\rangle = |0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$, we get:

$$S_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}(|000\rangle - |011\rangle)$$

These are the results we expect based on our initial simulation of the original unreduced circuit.

5.9.1 Applying superposition to the reduced qubit

Now consider an alternate version of the original circuit, where the initial state of the top qubit (the one to be reduced) is in a superposition. This would be such that the circuit looks like Figure 5.33.

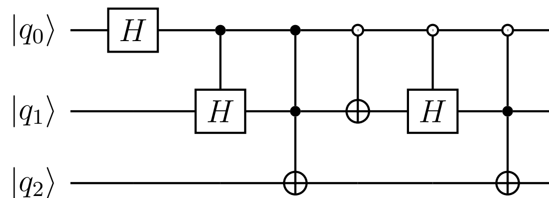


Figure 5.33: Alternate version of the circuit in Figure 5.31, adding an H gate to apply superposition to the qubit to be reduced.

The result of manually simulating this circuit directly is $\frac{1}{2}(|000\rangle - |011\rangle + |100\rangle + |111\rangle)$, which is essentially some combination of the two resulting state vectors we had earlier from the simulation of the the reduced circuits, S_1 and S_2. Our primary goal is to find a way to operate independently on two separate 2-qubit state vectors corresponding with the circuits in Figure 5.32 derived from the original circuit (Figure 5.31).

First, we observe that we cannot directly use the results of simulating the split circuits as they are, as each circuit was derived based on the assumption that the top qubit is in one of the computational-basis states, $|0\rangle$ or $|1\rangle$. Thus, we have to find another way to take into account the superposition of $|q_0\rangle$.

To do this, we assume the starting value of the whole register is such that $|q_0|q_1|q_2\rangle = |000\rangle$. Since the splittable part of the circuit starts after the initial H gate which cause the

superposition in the top qubit, we have to consider the value of the state vector after this gate acts. So we simulate this gate on the state vector $|000\rangle$ to get:

$$H \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{matrix} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{matrix} \tag{5.2}$$

Looking at the state vector indices on the right of the result in Eq. 5.2, we observe that the only non-zero values in the state vector correspond to different values of the top qubit. Thus we can divide this state vector into two and apply the reduced circuits to the corresponding half of this initial state vector. While the sub-state vectors are not themselves normalised, once they are combined again after the simulation, they will be normalised.

Applying C_1 to the second half of this state vector (corresponding to $|q_1\rangle = |1\rangle$):

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \end{bmatrix} \xrightarrow{H_1} \begin{bmatrix} \frac{1}{2} \\ 0 \\ \frac{1}{2} \\ 0 \end{bmatrix} \xrightarrow{CNOT_{1,2}} \begin{bmatrix} \frac{1}{2} \\ 0 \\ 0 \\ \frac{1}{2} \end{bmatrix} = s_1$$

Applying C_2 to the first half (corresponding to $|q_1\rangle = |0\rangle$):

$$\begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \end{bmatrix} \xrightarrow{X_1} \begin{bmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix} \xrightarrow{H_1} \begin{bmatrix} \frac{1}{2} \\ 0 \\ -\frac{1}{2} \\ 0 \end{bmatrix} \xrightarrow{CNOT_{1,2}} \begin{bmatrix} \frac{1}{2} \\ 0 \\ 0 \\ -\frac{1}{2} \end{bmatrix} = s_2$$

Then combining these two resulting sub-state vectors:

$$S = \begin{bmatrix} \frac{1}{2} \\ 0 \\ 0 \\ -\frac{1}{2} \\ \frac{1}{2} \\ 0 \\ 0 \\ \frac{1}{2} \end{bmatrix} = \frac{1}{2}(|000\rangle - |011\rangle + |100\rangle + |111\rangle),$$

which is our expected result.

Thus, we have shown how the circuit-width reduction techniques presented in the published work [56] can still be applied to simulate circuits which use the amplitude-basis encoding, including reduction of qubits in superposition. This comes at the cost of some precomputation: the effect of the qubits in superposition on the state vector has to be accounted for before the reduced circuits operate on the reduced state vectors.

Removing the requirement to store the full state vector

However, we can still show that the full state vector never needs to be stored. We carry on with the example above and demonstrate how the precomputations only need to be done on the independent qubits to be reduced, before computing the tensor product with the rest of the qubits' state vector.

At the start of the circuit shown in Figure 5.33, all the qubits are in state $|0\rangle$, thus the individual qubits can be described by the following state vectors:

$$|q_0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |q_1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |q_2\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This is the uncombined formulation of the state of the system. As long as no entanglement occurs, we can describe the system in this formulation. Notice that we need to store only $2n$ complex numbers instead of the 2^n needed to describe the state vector of the whole combined state.

We can now apply the initial H gate to $|q_0\rangle$ by itself:

$$|q_0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \xrightarrow{H} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Now, in order to simulate the reduced circuits (from Figure 5.32) on $|q_1\rangle$ and $|q_2\rangle$, we have to take their tensor product to describe their combined state:

$$|q_1 q_2\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Finally, in order to find the initial slices of the state vectors to be used as inputs to each of the reduced circuits, we should take the tensor product of $|q_0\rangle$ (after its preparation in superposition) with this initial state of $|q_1 q_2\rangle$:

$$(H\,|q_0\rangle) \otimes |q_1 q_2\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\ \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{bmatrix}$$

This state is exactly the initial state which we obtained in Eq. 5.2, and divided to find each of the slices of the state vector to be processed by each reduced circuit. This shows that we can simply take the scalar components of reduced qubits $|q_0\rangle$, after precomputation, and multiply each of them by the initial state vector to be used for the reduced circuits, to obtain the inputs of the reduced circuits. If we do this sequentially for each slice, we do not need to store the entire state vector at any point.

5.9.2 Summary

In this section, we demonstrated how the circuit reduction techniques discussed earlier can be applied not only to circuits operating purely in the computational basis but also to those involving superposition and entanglement (and thus utilise the amplitude-basis). By leveraging these optimisations, quantum circuits with amplitude-basis encoding were effectively reduced, maintaining the integrity of quantum phenomena like entanglement. A key takeaway was that even when qubits in superposition are reduced, the reduced circuit can still faithfully simulate the original system by operating on the resulting sub-state vectors independently and combining them after the simulation. Furthermore, we showed that we can

still perform the required precomputation, without having to store the full state vector of the entire system at any point.

Chapter 6

Evaluation of Developed Architectures

In this chapter, timing results of running different quantum circuits on the developed architectures are presented. We also give a power consumption analysis of the architectures compared with the Baseline architecture running on CPU and GPU. The primary aim of this evaluation is to assess the performance and efficiency of these architectures against traditional CPU and GPU implementations. By examining the runtime, resource utilisation, and energy consumption results, we aim to evaluate the effectiveness of our FPGA designs in executing various quantum algorithms. The insights derived from this analysis not only validate our architectural innovations but also lay the groundwork for future advancements in quantum computing simulation.

6.1 Evaluation Setup

The results presented in this chapter were obtained by running on a single-node FPGA system, hosting a **Nallatech PCIe-385N D5 FPGA board** with 8 GB RAM (DDR3) connected through a PCIe 2 connection, which features an Intel Stratix V GS D5 FPGA. Our host system has a dual Intel Xeon E5-2609 V2 2.5 GHz processor and 64 GB RAM (DDR3, 1.6 GHz). The system runs Scientific Linux 6.8 and we used the **Intel SDK for OpenCL version 17.1** to communicate with and program the FPGA device. The CPU used for evaluation is the same as the FPGA host. Our GPU evaluation system has access to an NVIDIA GK110B (GeForce GTX TITAN Black), with access to 6 GB of VRAM, and a quoted memory bandwidth of 336 GB/s [127]. Our FPGA's diagnosis tool reports 2.1 GB/s of DRAM global memory bandwidth. Table 6.1 reports the total FPGA resources available for our board.

As our FPGA board has 8GB of DRAM, we can store state vectors of systems with up to

Resource	Available
ALUTs	345200
FFs	690400
RAMs	2014
DSPs	1590

Table 6.1: Total available resources on the Intel Stratix V GS D5 FPGA.

29 qubits using a 32-bit floating point representation for the components of the complex numbers that make up the state vector. In this representation, we require $2^n \times 8$ bytes $= 2^{n+3}$ bytes to represent the state vector. For 29 qubits, this is approximately 4.29 GB; to move up to 30 qubits, we would require double this capacity and thus the state vector would not fit in our 8GB memory system.

6.2 Evaluation Circuits

We evaluate our architectures using four classes of quantum circuits: the QFT, Grover's search algorithm circuit, the D1Q3 circuit, and the streaming circuits. We choose the QFT (as introduced in Section 1.3.2) as it is a very common quantum operation used as a building block for different bigger quantum algorithms, including phase estimation, Shor's algorithm, and QFT-based arithmetic. The QFT's structure is high in the density of controlled gates, although it only has up to one control per gate. On the other hand, Grover's algorithm (described in Section 1.3.4) has up to a very high number of controls, while having a very low density of controlled gates, allowing us to evaluate our architectures' performance on contrasting algorithms (in terms of maximum number of controls, and controlled-gate density). The streaming circuits (introduced in Section 1.3.6), which are also used in CFD applications, are an example of a highly controls-dominant circuit and give us a best case to evaluate our Controls Scheduling Optimisation (discussed in Section 3.1.3).

The reduced D1Q3 (defined in Section 5.8.3) was chosen to demonstrate our circuit reduction technique and to attempt to show two different circuits running in parallel on our TwoCircuit-NDRange architecture. As these circuits also have high controlled-gate density, they are also useful for evaluating our buffered architectures (with and without gate fusion) for circuits with this property. We were unable to compile these architectures for the required 29 maximum number of controls for the Grover's and streaming circuits, as due to the complexity of the hardware design at this maximum number of controls, the HLS tools failed to produce a valid FPGA binary. Therefore, the D1Q3 is a reasonable middle ground for evaluating these architectures on this type of circuits, as they only require a maximum number of controls of 8.

We begin by evaluating all the developed FPGA architectures to determine the best performing ones for our chosen evaluation circuits. We then compare those against the BaselineNDRange and OptContNDRange versions running on the CPU and the GPU. Timing data was collected with the *wall-clock time* method, using the `clock_gettime()` from the `time.h` header in the C standard library.

Note about data reporting in tables In this chapter, we include several tables that present data about resource utilisation of architecture implementations, accompanied by average runtime of circuits (circuits are run 10 times each then averaged). For the resource utilisation parts of these tables, the percentages indicate the ratio of total available resources utilised. For the runtime sections, either percentages or times improvement will be included next to the reported runtime; unless otherwise stated, these indicate improvement relative to the first column of the table. If a percentage is reported, it indicates the percentage time saved compared to the first column (e.g. $|412.6|260.2(37\%)|$ would mean that the time reported in the second column is 37% shorter than in the first). Negative percentages are used when appropriate to indicate a longer runtime compared to the first column. If a times improvement ($\times$) is reported, then it is the ratio improvement compared to the first column (e.g. $|17.7|1.7(10.1\times)|$ would indicate that the second column performed $10.1\times$ faster than the first column).

6.3 Direct Iteration Processing Architectures Evaluation

In this section, we study the scaling of the Direct Iteration Processing (DIP) architectures (i.e. no buffering, gate fusion, or controls-based optimisations) described in Section 3.1.

6.3.1 BaselineNDRange

We begin by studying the performance and scaling of the BaselineNDRange architecture, described in Section 4.5. This is the most direct implementation of QWM (Qubit-Wise Multiplication, see Section 2.1.1), and represents the baseline for most of our architecture comparisons in the rest of this chapter. The CPU and GPU baseline implementations use the same OpenCL source code for this architecture.

QFT Circuits

Figure 6.1 shows the total circuit runtimes for the QFT circuit of with different qubit counts, for the BaselineNDRange architecture parameterised with 1 maximum controls per gate and for different numbers of compute units (NCU). Notice that we use a log scale in the y-axis as we expect the time it takes to execute circuits to roughly double with every additional qubit added. As expected, we observe exponential behaviour in the time taken, which gets very regular at higher qubit counts, starting at 16 qubits.

We can see that running with 8 compute units gives a performance benefit compared to running with 1 compute unit. From Table 6.2, we see this is a $36 - 37\%$ improvement for the highest qubit counts. We do not observe a regular scaling pattern with the number of compute units. Running with 2 or 4 compute units demonstrates marginally worse performance compared to the single compute unit version.

We explain the difference in performance through the overhead of adding compute units (the cost of parallelisation), which is not amortised until the 8 NCU case. The primary cause of this is the clock frequency that the HLS tool is able to achieve. From the table, notice that Fmax drops considerably moving from 1 to 2, and 2 to 4 compute units; however the jump to 8 is not as drastic, leading us to suspect that it would saturate for higher number of compute units. If we correct for the frequency, we would see the expected behaviour for scaling across number of compute units.

Table 6.2 shows that the limiting factor to synthesising more compute units is the ALUT utilisation which is 54% for 8 NCU and is much more than 100% if we attempt to compile a 16 NCU version. However, since we have shown that this architecture can scale, we can expect improved performance when going up to a more advanced FPGA. Our FPGA, the Stratix V GS D5, has 172600 ALMs [128] made up from its 345200 available ALUTs. The highest tier Stratix 10 board (S10 GX 10M) has 3466080 ALMS [129]. This gives a total of 6932160 ALUTs available on the device. This is about $37\times$ the utilised limiting resource for our architecture, indicating that we can potentially scale up to at least 256 compute units. This more advanced board would certainly consume more power, however, in terms of raw performance we would still see an improvement, assuming the memory bandwidth is not saturated.

Reduced D1Q3 Circuits

The 25-qubit reduced D1Q3 circuits were evaluated using a set of architectures compiled with 8 maximum number of controls (NC=8). We run the 8 specalisations (for different values of $|dv\rangle$ (representing which of the discrete-velocity directions is considered) and $|eu\rangle$ (defining the exponent of the input velocity in the quantum floating-point representation

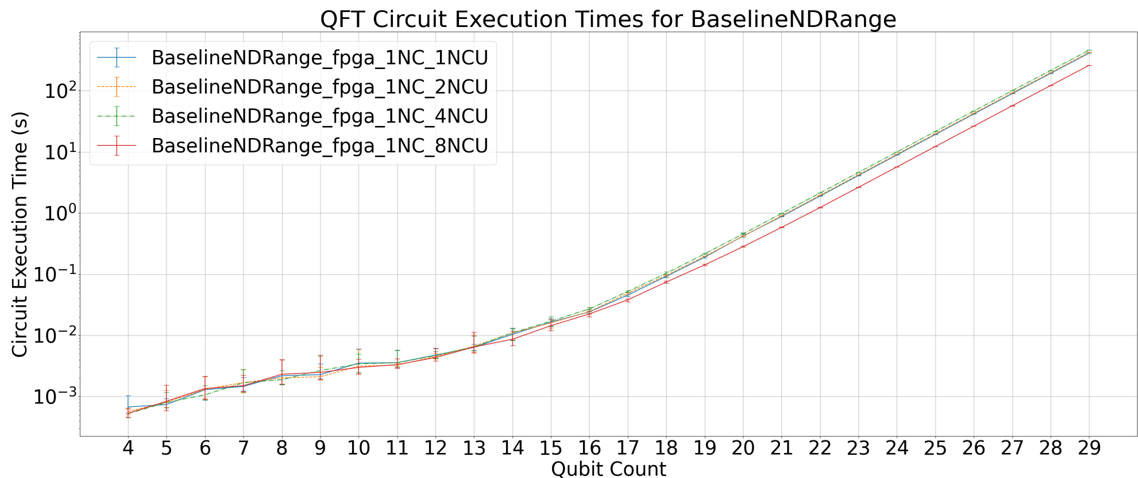


Figure 6.1: Total circuit runtimes for the QFT circuit of with different qubit counts, for the BaselineNDRange architecture parameterised with 1 maximum controls per gate and for different numbers of compute units (NCU).

Table 6.2: FPGA resource utilisation for a BaselineNDRange architecture with up to 8 Compute Units and 1 maximum control per gate; and average circuit runtimes for high qubit count QFT circuits on these architectures.

	NCU	1	2	4	8
Resources	ALUTs	59715(17%)	77922(23%)	114334(33%)	187266(54%)
	FFs	71741(10%)	89888(13%)	126201(18%)	199542(29%)
	RAMs	408(20%)	467(23%)	585(29%)	821(41%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	297	287	259	258
Avg Time (s)	QFT29	412.6	423.2(−3%)	460.2(−12%)	260.2(37%)
	QFT28	192.5	197.4(−3%)	214.9(−12%)	121.5(37%)
	QFT27	89.5	91.9(−3%)	100.1(−12%)	56.7(37%)
	QFT26	41.6	42.7(−3%)	46.5(−12%)	26.4(37%)
	QFT25	19.2	19.8(−3%)	21.6(−13%)	12.2(36%)

employed)) and average their runtime. The results are shown in Table 6.3 for the BaselineNDRange architecture, along with the resource utilisation, for different NCU values.

Grover's and Streaming Circuits

As described in Section 1.3.4, Grover's algorithm circuits consist of a repeating oracle and diffusion operators. The oracle operator contains a controlled gate with the maximum number of controls allowed in the circuit (i.e. if the circuit has n qubits, including the oracle qubit, this gate has $n-1$ controls); and the diffusion operator has a controlled gate with one fewer control. As our architectures are parameterised by the number of controls at compile time, in order to simulate Grover's circuits, we have to compile a set of our architectures for

Table 6.3: FPGA resource utilisation for a BaselineNDRange architecture with varying with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture.

	NCU	1	2	4	8
Resources	ALUTs	60231(17%)	78954(23%)	116398(34%)	191394(55%)
	FFs	71887(10%)	90180(13%)	126785(18%)	200710(29%)
	RAMs	408(20%)	467(23%)	585(29%)	821(41%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	291	286	277	267
Avg Time (s)	D1Q3	28.7	28.9(0%)	28.9(0%)	17.9(38%)

at least 28 maximum controls per gate. This gives us an opportunity to investigate how the resource usage varies as we change the number of controls. We compiled our architectures for 29 maximum controls per gate to run the Grover's algorithm circuits (in theory we could run a 30-qubit Grover's circuit, which would require 29 qubits to be used as controls, if we had sufficient memory space). We also reuse these architectures to run the streaming circuits (described in Section 1.3.6).

We planned to evaluate Grover's circuits for qubit registers ranging from 3 to 29, the same as for the QFT circuits. Grover's search optimally requires $\frac{\pi}{4}\sqrt{N}$ iterations (for $N = 2^n$, for an n-qubit search register) of the Grover's iterate (oracle followed by diffusion circuits) to obtain the search result with sufficiently high probability. However, for large search registers, this results in a very large number of gates, ranging from hundreds of thousands to millions. For the 29-qubit version (28-qubit search register), this is at least 1.4 million gates! Therefore, we elected to run only 10 iterations of the Grover's iterate for all the circuits, regardless of circuit qubit count. We still generally get a higher probability of finding our search target state, and as the point of running these experiments is to evaluate our circuit simulator's performance, we still get meaningful results.

Table 6.4 shows the resource utilisation of the BaselineNDRange architecture parameterised with 29 maximum controls for the four NCU configurations we can compile. We see that, across the board, the added maximum allowed controls make a very marginal difference in resource usage compared to the 1 maximum number of controls shown in Table 6.2.

Figure 6.2 shows the timing results for running Grover's circuits on the BaselineNDRange architecture parameterised with 29 maximum controls for four different numbers of compute units. We observe a similar pattern to that observed for the QFT experiments. The timing section in Table 6.4 also demonstrates a similar pattern, although at 8NCU, the improvement over 1NCU has dropped to 30% from 36% for Grover's circuits. This can be attributed to the lower clock frequency that the HLS tools could achieve for this architecture (about 7% lower). This decrease in improvement between NCU variants does not appear in the case of the Streaming circuits, and this highlights the impact of the memory access pattern, which

depends on the target qubit indices of the gates as well as the involved controls and number of controls, on the performance to be gained from increased parallelism.

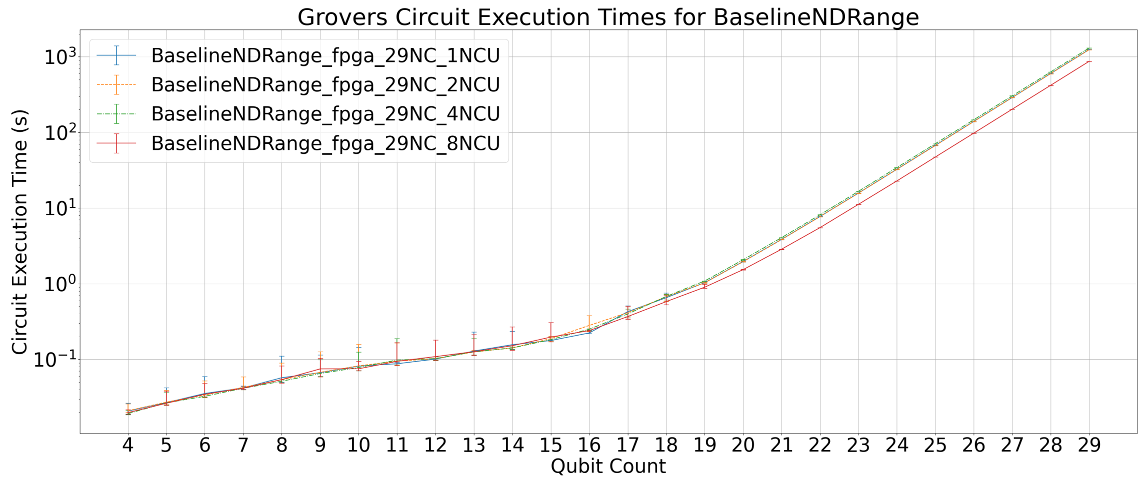


Figure 6.2: Total circuit runtimes for the Grover's circuit with different qubit counts, for the BaselineNDRange architecture parameterised with 29 maximum controls per gate and for different numbers of compute units (NCU).

6.3.2 UnrolledLoops

Figure 6.3 shows the QFT runtimes of the UnrolledLoops architecture (described in Section 4.6 with different numbers of compute units against the runtime of the best-performing BaselineNDRange architecture, the 8 compute units version.

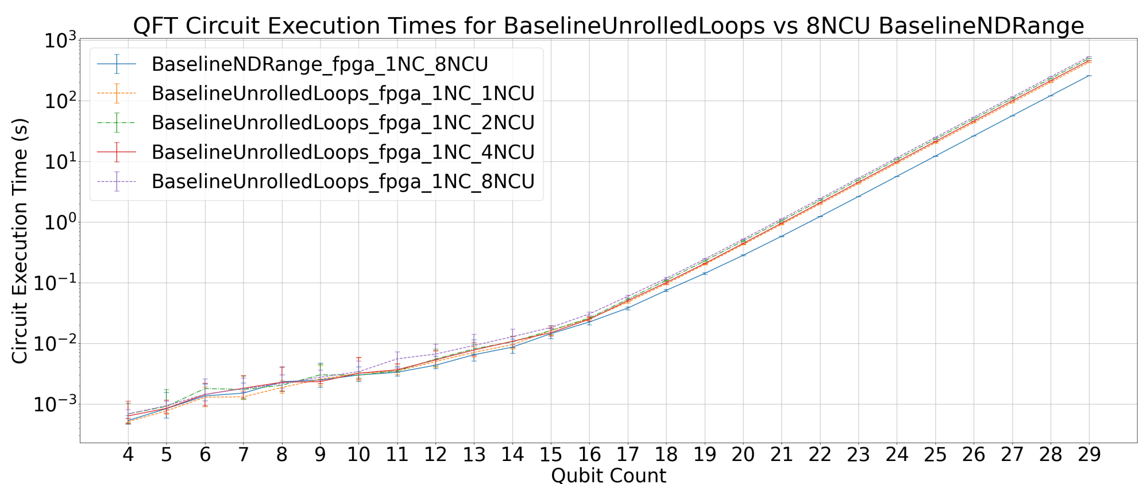


Figure 6.3: QFT circuit runtimes for the UnrolledLoops architecture with varying numbers of compute units, compared to the best-performing BaselineNDRange architecture with 8 NCUs.

Table 6.4: FPGA resource utilisation for a BaselineNDRange architecture with up to 8 Compute Units and 29 maximum controls per gate; and average circuit runtimes for high qubit count Grover's circuits on these architectures.

	NCU	1	2	4	8
Resources	ALUTs	61584(18%)	81660(24%)	121810(35%)	202218(59%)
	FFs	72201(10%)	90808(13%)	128041(19%)	203222(29%)
	RAMs	408(20%)	467(23%)	585(29%)	821(41%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	287	288	258	240
Avg Time (s)	Grovers29	1245.4	1238.6(1%)	1304.3(−5%)	869.9(30%)
	Grovers28	601.3	598.0(1%)	630.3(−5%)	419.8(30%)
	Grovers27	290.1	288.3(1%)	304.3(−5%)	202.4(30%)
	Grovers26	139.7	138.9(1%)	146.8(−5%)	97.5(30%)
	Grovers25	67.2	66.9(0%)	70.7(−5%)	47.0(30%)
	Streaming29	27.8	27.6(1%)	30.9(−11%)	17.7(36%)
	Streaming28	13.4	13.3(1%)	14.9(−11%)	8.6(36%)
	Streaming27	6.5	6.4(2%)	7.2(−11%)	4.1(37%)
	Streaming26	3.1	3.1(0%)	3.5(−13%)	2.0(35%)
	Streaming25	1.5	1.5(0%)	1.7(−13%)	1.0(33%)

From the figure and the QFT29 runtime numbers in Table 6.5, we can see that this architecture does not scale at all with number of compute units. We see as well that the NDRange-based implementation outperforms this implementation.

This is accounted for by the NDRange scheduler that the HLS tool uses being more optimised. By comparing the 1NCU times from this table and Table 6.2, we see that the UnrolledLoops architecture has an added overhead, likely caused by the looping structure. From Table 6.5, it can observed that the tool achieves a much lower clock frequency for the 8 NCU version.

However, we demonstrate later in Section 6.5 that we can gain a performance improvement over the baseline by adding buffering. As we do not observe any improvement over the baseline from this architecture, or any scaling, we evaluated only the QFT target.

6.3.3 OnBoardUnrolledLoops

The OnBoardUnrolledLoops architecture (described in Section 4.7 does not queue each gate as its own OpenCL task; instead the host transfers the whole quantum circuit description to the device's memory along with the gate count and the device proceeds to process each gate sequentially using the UnrolledLoops-based method.

Figure 6.4 shows the performance of this architecture compared to the 8 compute units BaselineNDRange architecture. We observe behavior similar to the last architecture: this

Table 6.5: FPGA resource utilisation for a UnrolledLoops architecture with up to 8 Compute Units and 1 maximum control per gate.

	NCU	1	2	4	8
Resources	ALUTs	59928(17%)	77788(23%)	114899(33%)	184900(54%)
	FFs	72007(10%)	90481(13%)	127490(18%)	199400(29%)
	RAMs	408(20%)	468(23%)	620(31%)	828(41%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	282	286	287	227
Avg Time (s)	QFT29	431.5	498.3(−15%)	460.4(−7%)	533.5(−24%)
	QFT28	201.5	232.3(−15%)	214.4(−6%)	248.7(−23%)
	QFT27	93.7	108.1(−15%)	99.5(−6%)	115.7(−23%)
	QFT26	43.5	50.1(−15%)	46.1(−6%)	53.7(−23%)
	QFT25	20.2	23.2(−15%)	21.3(−5%)	24.8(−23%)

approach does not scale with number of compute units and is still beaten by the BaselineNDRange approach for high qubit counts.

For low qubit count (< 16), we notice that this approach does perform better compared to the BaselineNDRange approach, however in general we are more concerned with the performance at a high qubit count. Like in the case of the UnrolledLoops architecture above, we only evaluated the QFT for this architecture since we did not observe any performance improvement at higher qubit counts.

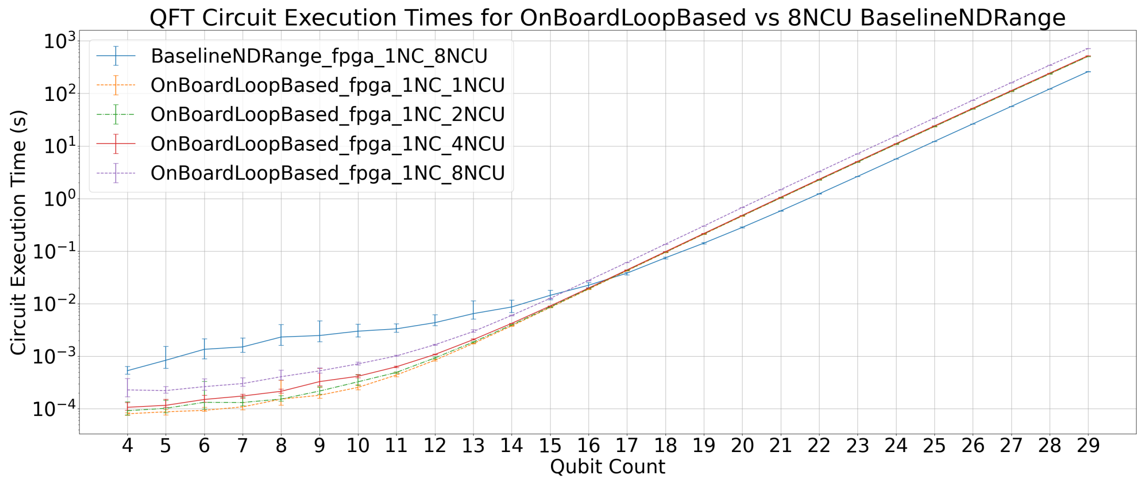


Figure 6.4: Performance comparison between the OnBoardUnrolledLoops and BaselineNDRange architectures for QFT circuits with varying qubit counts for different numbers of compute units.

From these experiments, we conclude that the BaselineNDRange architecture is the best-performing implementation of the direct iteration processing approach, without the controls scheduling optimisation.

Table 6.6: FPGA resource utilisation for the OnBoardUnrolledLoops architecture with up to 8 Compute Units and 1 maximum control per gate.

	NCU	1	2	4	8
Resources	ALUTs	60944(18%)	77898(23%)	112506(33%)	187041(54%)
	FFs	75559(11%)	93547(14%)	130825(19%)	210592(31%)
	RAMs	430(21%)	501(25%)	643(32%)	866(43%)
	DSPs	14(1%)	28(2%)	56(4%)	112(7%)
	Fmax (MHz)	290	276	247	218
Avg Time (s)	QFT29	508.3	505.8(0%)	523.5(−3%)	715.2(−41%)
	QFT28	235.8	235.7(0%)	243.5(−3%)	344.4(−46%)
	QFT27	109.6	109.5(0%)	113.1(−3%)	160.3(−46%)
	QFT26	50.6	50.7(0%)	52.3(−3%)	74.3(−47%)
	QFT25	23.4	23.4(0%)	24.1(−3%)	34.0(−45%)

6.3.4 TwoCircuitNDRange

We evaluate the architecture designed to simulate two circuits in parallel introduced in Section 4.8 in this section.

The TwoCircuitNDRange architecture, described in Section 4.8, was primarily designed to be able to simulate different quantum circuit specialisations created using the circuit-width reduction technique described in Chapter 5. We tested this architecture only on the QFT target, as the resulting performance results did not indicate that the HLS tool could accomplish the required result of executing the circuits in parallel with memory banks connected through independent memory interfaces. Therefore, further evaluations would not have been indicative of the desired behaviour of this architecture.

The largest NCU value for which we can compile this architecture is 4. This is because each NCU performs double the amount of computations compared to the baseline. Figure 6.5 shows a comparison of the TwoCircuitNDRange ran for 3 NCU variations for the QFT circuits compared against the 8 NCU BaselineNDRange architecture. We can see that the two-circuit architecture underperforms compared to the baseline.

Ideally we would want to see the QFT28 (which is the largest QFT we are able to simulate on this architecture due to memory space limitation), on 4NCU TwoCircNDRange, perform close to the QFT29 baseline on 8NCU, which would indicate full parallel execution of the two QFT28 circuits on separate memory banks. Instead we see from Table 6.7 that it performs about $1.5 - 2\times$ worse than the baseline. From the generated toolchain reports, we were not able to verify that the tool is indeed assigning the two state vectors, corresponding to the two circuits, to separate memory banks with their own memory interfaces, and thus even if the iteration operations are being run in parallel, they are not benefiting from their state vectors being stored in separate memories.

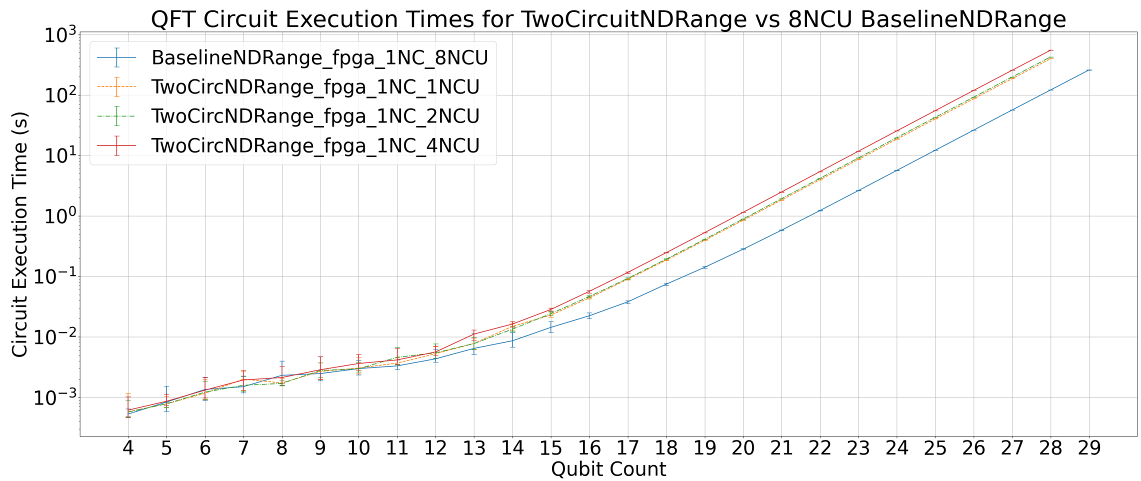


Figure 6.5: Performance comparison of the TwoCircuitNDRange architecture for different numbers of compute units, against the 8 NCU BaselineNDRange architecture for QFT circuits.

Table 6.7: FPGA resource utilisation for a TCNDRange architecture with up to 8 Compute Units and 1 maximum control per gate.

	NCU	1	2	4	8
Resources	ALUTs	77209(22%)	112908(33%)	188231(55%)	327378(95%)
	FFs	89531(13%)	125487(18%)	201249(29%)	343104(50%)
	RAMs	467(23%)	585(29%)	760(38%)	1293(64%)
	DSPs	32(2%)	64(2%)	128(8%)	256(16%)
	Fmax (MHz)	300	297	231	–
Avg Time (s)	QFT28	400.1	427.4(−7%)	552.0(−38%)	–
	QFT27	186.6	199.0(−7%)	257.1(−38%)	–
	QFT26	86.8	92.5(−7%)	119.4(−38%)	–
	QFT25	40.3	42.8(−6%)	55.3(−37%)	–
	QFT24	18.7	19.8(−6%)	25.6(−37%)	–

While the TwoCircuitNDRange architecture was designed to enhance parallelism through the simultaneous simulation of two circuits, the evaluation revealed limitations in its implementation, highlighting the need for further refinement in memory management and resource allocation to better evaluate its potential.

6.4 Evaluating the Controls Scheduling Optimisation

In this section, the method designed to optimise scheduling controlled gates introduced in Section 3.1.3 for DIP-based architectures is evaluated.

6.4.1 QFT Experiments

Figure 6.6 shows the timing results of the Controls optimisation implemented for the NDRange kernel with 4 NCU variations against the 8NCU BaselineNDRange for the QFT circuit.

Table 6.8 shows that the resource utilisation of this architecture is very similar to the BaselineNDRange architecture (compared to Table 6.2). The timing of the QFT circuits is also shown in this table and we can see that in the across all NCU cases, this architecture demonstrates a significant performance benefit over the baseline ($1.5\times$ faster in the 8NCU QFT29 experiment).

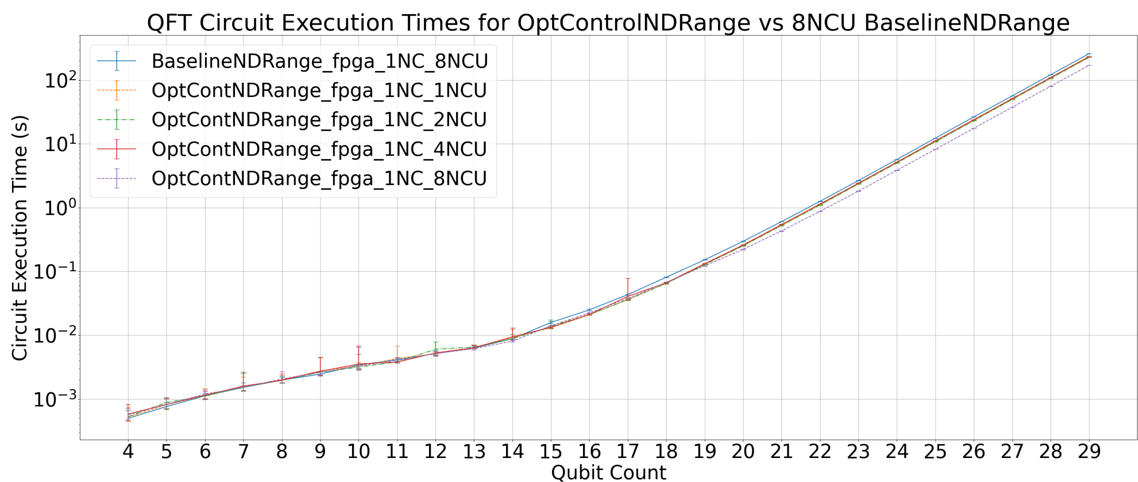


Figure 6.6: Timing results of the Controls Scheduling Optimisation for the NDRange kernel with varying numbers of compute units, compared to the 8 NCU BaselineNDRange architecture for QFT circuits.

We importantly note that the performance benefit from this optimisation is circuit-dependent, particularly on the density of controls in the circuit. When we look at gate timings (not shown here), we indeed observe that the controlled gates in the QFT (which have one control, thus requiring only half the iterations of a normal gate to be scheduled) on average take half the time of an uncontrolled gate. Thus, it is important to study this architecture in the context of other circuits as well.

Table 6.8: FPGA resource utilisation for the OptContNDRange architecture with up to 8 Compute Units and 1 maximum number of controls per gate.

	NCU	1	2	4	8
Resources	ALUTs	59797(17%)	78086(23%)	114662(33%)	187922(54%)
	FFs	71800(10%)	90006(13%)	126437(18%)	200014(29%)
	RAMs	408(20%)	467(23%)	585(29%)	821(41%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	303	299	280	233
Avg Time (s)	QFT29	228.3	226.9(1%)	234.0(−2%)	170.5(25%)
	QFT28	106.8	106.1(1%)	109.5(−3%)	79.9(25%)
	QFT27	49.8	49.5(1%)	51.2(−3%)	37.4(25%)
	QFT26	23.2	23.1(0%)	23.9(−3%)	17.5(25%)
	QFT25	10.8	10.8(0%)	11.1(−3%)	8.4(22%)

6.4.2 Reduced D1Q3 Circuits

In this section we evaluate the performance of the OptContNDRange architecture on the reduced D1Q3 circuits. These circuits provide a useful benchmark for testing the optimised control scheduling in cases with a high density of controlled gates. We present results for various numbers of compute units and compare the performance with the baseline architecture.

Table 6.9 shows the resource utilisation for the 8 maximum number of controls variants of this architecture, as well as the average timing results for the reduced 25-qubit D1Q3 circuits. We see that observe a similar pattern as in the 1 maximum number of controls case as well as the baseline in terms of scaling. The timing result at 8NCU demonstrates a $1.4\times$ advantage over the 8NCU baseline, in line with the improvement observed for the QFT.

Table 6.9: FPGA resource utilisation for a OptContNDRange architecture with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture.

	NCU	1	2	4	8
Resources	ALUTs	61290(18%)	81072(23%)	120634(35%)	199866(58%)
	FFs	72522(11%)	91450(13%)	129325(19%)	205790(30%)
	RAMs	408(20%)	467(23%)	585(29%)	821(41%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	316	297	277	236
Avg Time (s)	D1Q3	14.1	14.2(−1%)	14.5(−3%)	12.8(9%)

6.4.3 Grovers and Streaming Circuits Experiments

Similar to the treatment for the BaselineNDRange, we compile a set of variations for the OptContNDRange architecture with 29 maximum number of controls, for different NCU

values, and use them to evaluate the performance of the Grover's and the streaming circuits. Figures 6.7 and 6.8 show the timing results for these circuits, respectively.

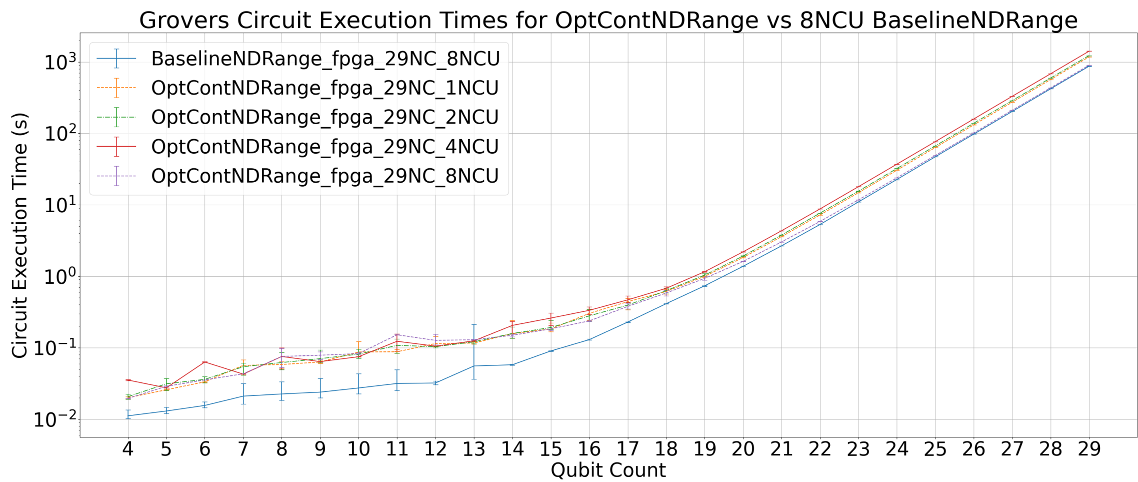


Figure 6.7: Total circuit runtimes for the Grover's circuit with different qubit counts, for the OptContNDRange architecture parameterised with 29 maximum controls per gate and for different numbers of compute units (NCU), shown against the BaselineNDRange with 8NCU.

We can see that in the case of Grover's circuit in Figure 6.7, the OptContNDRange 8NCU actually performs marginally worse the BaselineNDRange with the same NCU. This is down to Grover's being a very sparsely controlled circuit; while the controlled gates used in it require a large number of controls, the number of controlled gates compared to non-controlled is very small (approximately 87 non-controlled gates to every controlled gate in the case of Grovers29). This makes sense, since this is an optimisation targeting controlled gates in particular and there is an extra overhead in checking through this many possible control qubits, with no benefit to be gained since there are very few actually controlled gates.

The highly controlled streaming circuits however is where we see the most value from this approach. Figure 6.8 shows that we approach an order of magnitude better performance compared with the 8NCU baseline. Comparing the streaming circuits data from Tables 6.10 and 6.4, we see almost $5\times$ better performance for the OptContNDRange architecture for this circuit.

From the resources section in Table 6.8 a noticeable frequency drop was observed when scaling up to 8 NCUs. This reduction in clock frequency impacts the architecture's ability to fully realise the expected performance gains. This is likely due to the increased resource utilisation, which adds complexity to the circuit design, ultimately limiting the achievable clock frequency. The frequency drop highlights a trade-off between parallelism (i.e., more NCUs) and achievable clock speeds, which should be considered when scaling FPGA architectures. In the 8 NCU configuration, the Fmax dropped to 206-233 MHz, compared to

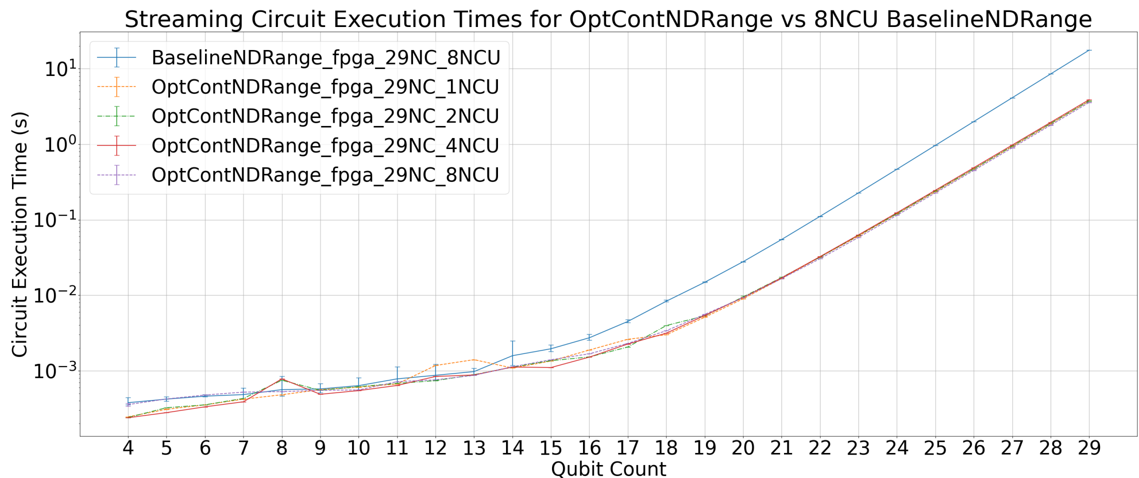


Figure 6.8: Total circuit runtimes for the Streaming circuits with different qubit counts, for the OptContNDRange architecture parameterised with 29 maximum controls per gate and for different numbers of compute units, shown against the BaselineNDRange with 8NCUs.

higher frequencies at lower NCU counts (e.g., 303-317 MHz for 1 NCU). Despite this reduction in clock speed at higher NCU values, the optimised control scheduling still provides a significant performance benefit, particularly in circuits with high control gate density like the streaming circuits.

6.5 Buffered Architectures Evaluation

In this section, we evaluate the different implementations of the buffered approach introduced in Section 3.3.

The evaluations are based on the architecture implementations described in Section 4.10. As discussed in that section, these architectures are parameterised by a **Buffer Qubit Size**, l, which is also referred to as BQS in this section; and the buffer size is determined as $L = 2^l$. The NCU for these architectures is determined as a function of the BQS, as each NCU should process exactly two elements from the buffer in each buffer pass. Thus the number of compute units is defined as NCU = 2^{l-1}.

We compare each version of the buffered architecture against the best performing Direct Iteration Processing architecture, the 8 NCU BaselineNDRange.

We compiled 3 versions of each buffered architecture with a BQS of 2, 3, and 4. However we only evaluate the 2 and 3 BQS versions as the HLS tool is unable to infer local arrays of a size more than 64 bytes as registers [5], which is important to achieve the desired performance. We tested the 4 BQS version and indeed it performs significantly worse (about an order of magnitude) than the 3 BQS version, because the buffer array gets implemented as BRAMs

Table 6.10: FPGA resource utilisation for the OptContNDRange architecture with up to 8 Compute Units and 29 maximum number of controls per gate.

	NCU	1	2	4	8
Resources	ALUTs	65554(19%)	89600(26%)	137690(40%)	233978(68%)
	FFs	74669(11%)	95744(14%)	137913(20%)	222966(32%)
	RAMs	413(21%)	477(24%)	605(30%)	861(43%)
	DSPs	16(1%)	32(2%)	64(4%)	128(8%)
	Fmax (MHz)	317	284	236	206
Avg Time (s)	Grovers29	1175.9	1223.6(−4%)	1412.2(−20%)	897.9(24%)
	Grovers28	567.2	595.0(−5%)	682.3(−20%)	434.2(23%)
	Grovers27	272.8	287.0(−5%)	329.6(−21%)	210.2(23%)
	Grovers26	131.6	138.5(−5%)	158.9(−21%)	101.4(23%)
	Grovers25	63.4	66.7(−5%)	76.6(−21%)	49.0(23%)
	Streaming29	3.7	3.8(−3%)	3.9(−5%)	3.6(3%)
	Streaming28	1.9	1.9(0%)	2.0(−5%)	1.8(5%)
	Streaming27	0.9	0.9(0%)	1.0(−11%)	0.9(0%)
	Streaming26	0.5	0.5(0%)	0.5(0%)	0.5(0%)
	Streaming25	0.2	0.2(0%)	0.2(0%)	0.2(0%)

instead of registers. The effect of this on the critical path can be seen in the significantly lower clock frequency reported in Tables 6.11 and 6.13.

The benefit of using this type of buffered architectures depends does depend on the target of the gate being simulated; with gates having a target index less than the BQS of the architecture ($t < l$) requiring a single contiguous memory access while, and gates not satisfying this condition requiring two memory accesses. Thus the benefit gained from this architecture is not uniform across all gates; nevertheless both cases of gates should see an improvement compared with the baseline DIP-based architecture.

6.5.1 Single Buffering

In this section, we assess the performance of the UnrolledSingleBuffer architecture across different BQS values and compare its results with the baseline UnrolledLoops and BaselineNDRange architectures, for the QFT and D1Q3 circuits.

QFT Circuits

Figure 6.9 shows the performance of the UnrolledSingleBuffer architecture with 2 buffer sizes against the 8 compute unit BaselineNDRange architecture for the QFT.

From this figure and Table 6.11, we see that by adding the buffering technique to the unrolled loops architectures, we start seeing a benefit over the baseline of about 12% in the QFT 29 case, comparing the 3 BQS buffered version against the 8 NCU baseline version.

This already demonstrates a gained benefit from adding the buffering techniques to the UnrolledLoops architecture. We also see that this architecture has the potential to scale with higher numbers of compute units (and a higher BQS), as the 4 NCU (3 BQS) version shows around a 40% improvement over the 2 NCU (2 BQS) version.

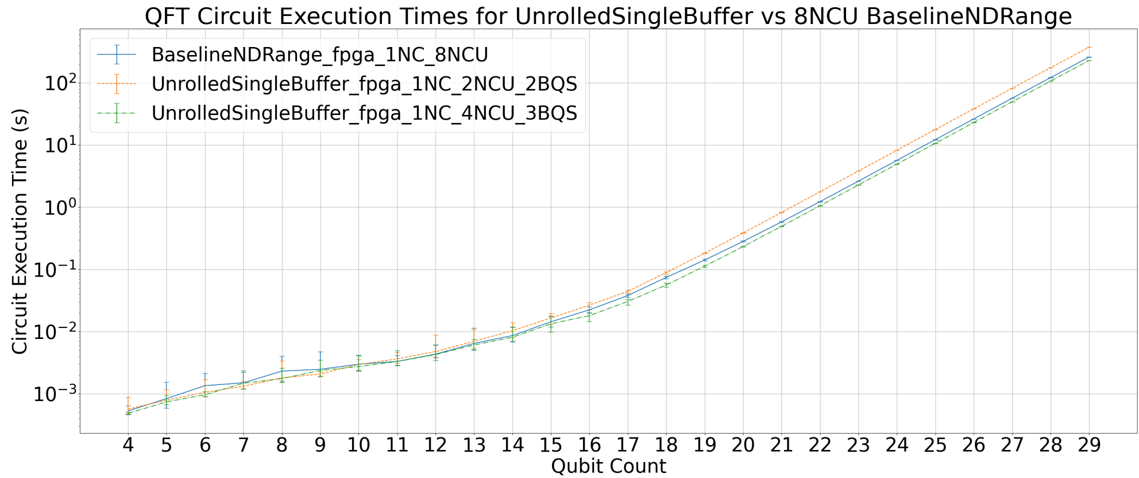


Figure 6.9: Performance comparison of the UnrolledSingleBuffer architecture with varying buffer sizes (BQS) and the 8 NCU BaselineNDRange architecture for QFT circuits.

Table 6.11: FPGA resource utilisation for a UnrolledSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	118754(34%)	190740(55%)	303052(88%)
	FFs	–	112017(16%)	145645(21%)	217920(32%)
	RAMs	–	549(27%)	523(26%)	632(31%)
	DSPs	–	64(4%)	128(8%)	256(16%)
	Fmax (MHz)	–	293	269	211
Avg Time (s)	QFT29	–	375.2	229.5(39%)	–
	QFT28	–	175.6	106.9(39%)	–
	QFT27	–	82.0	49.7(39%)	–
	QFT26	–	38.2	23.1(40%)	–
	QFT25	–	17.8	10.7(40%)	–

Reduced D1Q3 Circuits

Table 6.12 shows the results for the D1Q3 circuit on the UnrolledSingleBuffer architecture with 8 maximum number of controls. For this circuit, the buffering approach gives a 4% improvement over the baseline.

Table 6.12: FPGA resource utilisation for a UnrolledSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	118941(34%)	192956(56%)	–
	FFs	–	106649(15%)	145987(21%)	–
	RAMs	–	511(25%)	523(26%)	–
	DSPs	–	64(4%)	128(8%)	–
	Fmax (MHz)	–	260	267	–
Avg Time (s)	D1Q3	–	31.3	17.1(45%)	–

6.5.2 Double Buffering

In this section, we evaluate the double buffering technique, which was introduced to further improve the performance of the UnrolledSingleBuffer architecture by allowing for concurrent input and output buffering. The double-buffered design aims to reduce the overhead associated with memory accesses. We present the results for QFT and D1Q3 circuits, comparing the performance against the baseline architecture and the single-buffered architecture.

QFT Circuits

Figure 6.10 shows the performance of the UnrolledDoubleBuffer architecture compared to the single-buffered 3 BQS version. From Table 6.13 and the previous Table 6.11, we can see that using double buffering for this approach does not provide any improvement and gives slightly worse timings compared to the single buffered case. This is attributed to the lower clock frequency achieved by the tools.

Table 6.13: FPGA resource utilisation for a UnrolledDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	114834(33%)	177258(51%)	301366(87%)
	FFs	–	106115(15%)	140921(20%)	240804(35%)
	RAMs	–	511(25%)	523(26%)	912(45%)
	DSPs	–	64(4%)	128(8%)	256(16%)
	Fmax (MHz)	–	293	262	178
Avg Time (s)	QFT29	–	375.8	232.3(38%)	–
	QFT28	–	176.0	108.3(38%)	–
	QFT27	–	82.2	50.4(39%)	–
	QFT26	–	38.3	23.4(39%)	–
	QFT25	–	17.8	10.8(39%)	–

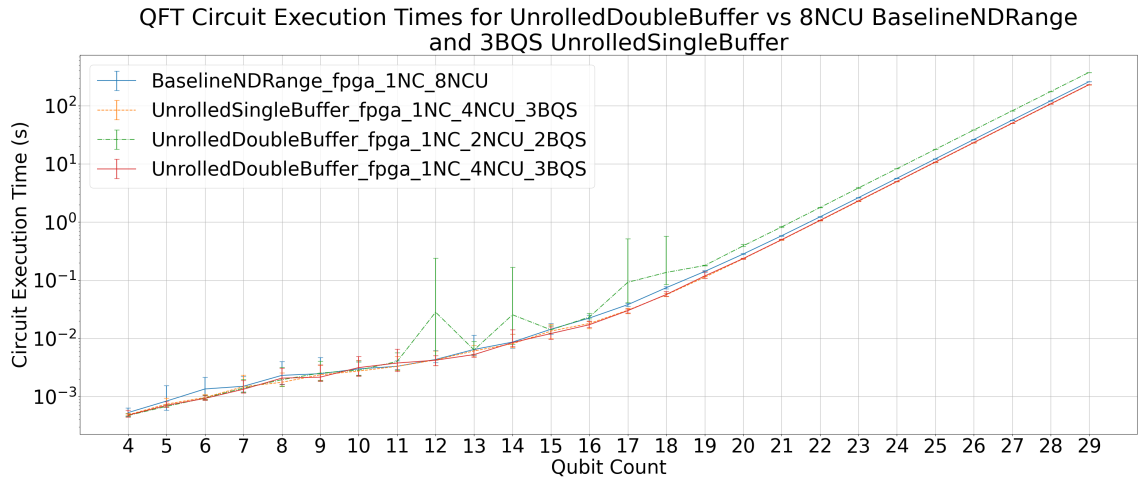


Figure 6.10: Performance comparison between the UnrolledDoubleBuffer architecture and the 3 BQS single-buffered architecture for QFT circuits.

Reduced D1Q3 Circuits

Table 6.14 shows the results for the D1Q3 circuit on the UnrolledDoubleBuffer architecture with 8 maximum number of controls. In this case, the architecture performs almost identically to the single-buffered approach, also giving a 4% improvement over the baseline.

Table 6.14: FPGA resource utilisation for a UnrolledDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	116130(34%)	179355(52%)	–
	FFs	–	106386(15%)	141023(20%)	–
	RAMs	–	511(25%)	523(26%)	–
	DSPs	–	64(4%)	128(8%)	–
	Fmax (MHz)	–	272	267	–
Avg Time (s)	D1Q3	–	30.0	17.1(43%)	–

6.6 Gate Fusion Evaluation

The Gate Fusion architectures aim to reduce the overhead associated with scheduling and processing individual quantum gates by fusing multiple gates into a single block operation. This technique decreases the number of kernel invocations and memory accesses, thereby improving the performance. In this section, we evaluate their performance, both in their single-buffered and double-buffered variations, and compare them against the baseline and buffered architectures. We focus on the QFT and D1Q3 circuits.

6.6.1 Single-Buffered Gate Fusion

QFT Circuits

Figure 6.11 shows the performance of 2 Gate Fusion architectures, with a single buffer (BQS = 2 and 3), compared to the 8NCU BaselineNDRange architecture and the 3 BQS single-buffered architecture (the best performing buffered architecture). The figure and Table 6.15 shows that we achieve a slight performance benefit when using the 3 BQS version of this Gate Fusion architecture compared to the normal buffered architecture. We can also see that the 3 BQS version performs better than the 2 BQS version indicating that we could potentially scale to a larger buffer.

Looking at clock frequency in Table 6.15 for this architecture, we see that the frequency drop between 2 BQS and 3 BQS is very small compared to the drop between 3 BQS and 4 BQS; which is attributed to the tool not instantiating a BRAM bank instead of registers for the 4 BQS case. Thus we only expect a scaling performance gain if the larger buffer can be inferred entirely as registers.

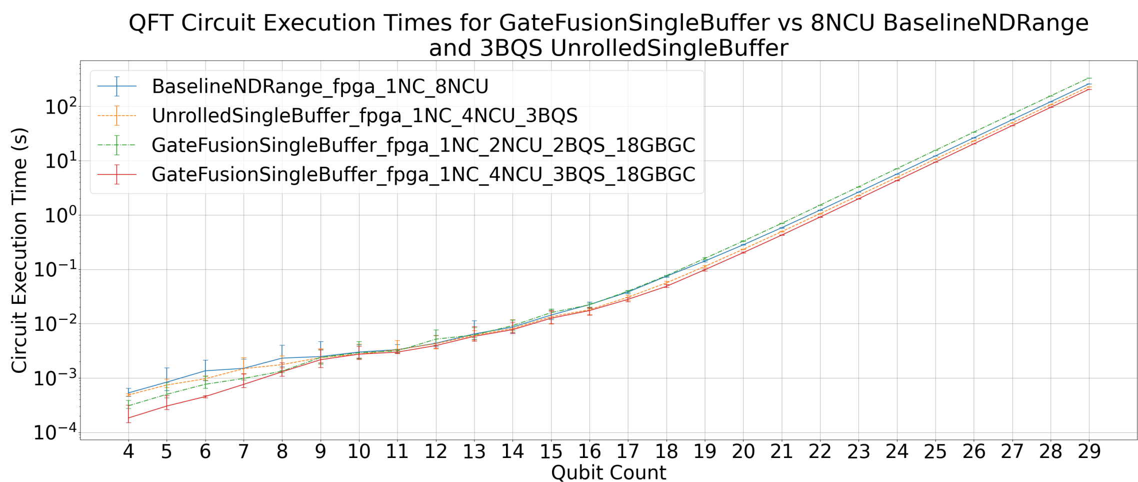


Figure 6.11: Performance of GateFusionSingleBuffer architectures with different buffer qubit sizes (BQS) compared to the 8 NCU BaselineNDRange architecture and the 3 BQS SingleBuffered architecture for QFT circuits.

Reduced D1Q3 Circuits

Table 6.16 shows the results for the D1Q3 circuit on the GateFusionSingleBuffer architecture with 8 maximum number of controls. Like in the QFT case, we see that this architecture demonstrates good scaling with an almost 50% reduction in execution time when moving from 2BQS to 3BQS. We can see that compared to the baseline, we get a better improvement than the single buffered architecture without gate fusion, of nearly 20%.

Table 6.15: FPGA resource utilisation for a GateFusionSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate. Also shown is the QFT circuits' performance for this architecture.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	116018(34%)	182482(53%)	282641(82%)
	FFs	–	120056(17%)	159210(23%)	220971(32%)
	RAMs	–	616(31%)	659(33%)	702(35%)
	DSPs	–	56(4%)	112(7%)	224(14%)
	Fmax (MHz)	–	277	274	196
Avg Time (s)	QFT29	–	334.0	206.4(38%)	–
	QFT28	–	155.6	95.7(38%)	–
	QFT27	–	72.3	44.2(39%)	–
	QFT26	–	33.5	20.4(39%)	–
	QFT25	–	15.5	9.4(39%)	–

Table 6.16: FPGA resource utilisation for a GateFusionSingleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	124577(36%)	192212(56%)	–
	FFs	–	168168(24%)	207442(30%)	–
	RAMs	–	897(45%)	940(47%)	–
	DSPs	–	56(4%)	112(7%)	–
	Fmax (MHz)	–	218	216	–
Avg Time (s)	D1Q3	–	28.1	14.6(48%)	–

6.6.2 Double-Buffered Gate Fusion

We now evaluate the performance of the double-buffered gate fusion architecture. Building on the single-buffered gate fusion technique, this architecture uses two buffers to store and process state vector slices. Here, we compare the performance of the double-buffered gate fusion architecture against its single-buffer counterpart for both QFT and D1Q3 circuits.

QFT Circuits

Figure 6.12 shows the effect of adding double buffering to the Gate Fusion based architecture. Recall from Section 4.11 that adding double buffering to gate fusion necessitates that the buffers alternate as input and output buffers between gates when processing a gate block; requiring extra control flow to check whether the gate index is even or odd when accessing the buffer. This architecture significantly underperforms compared to its single buffer counterpart (being 4-6x slower), and we attribute this underperformance primarily to this

additional control flow causing a significantly lower clock frequency, as reported in Table 6.17.

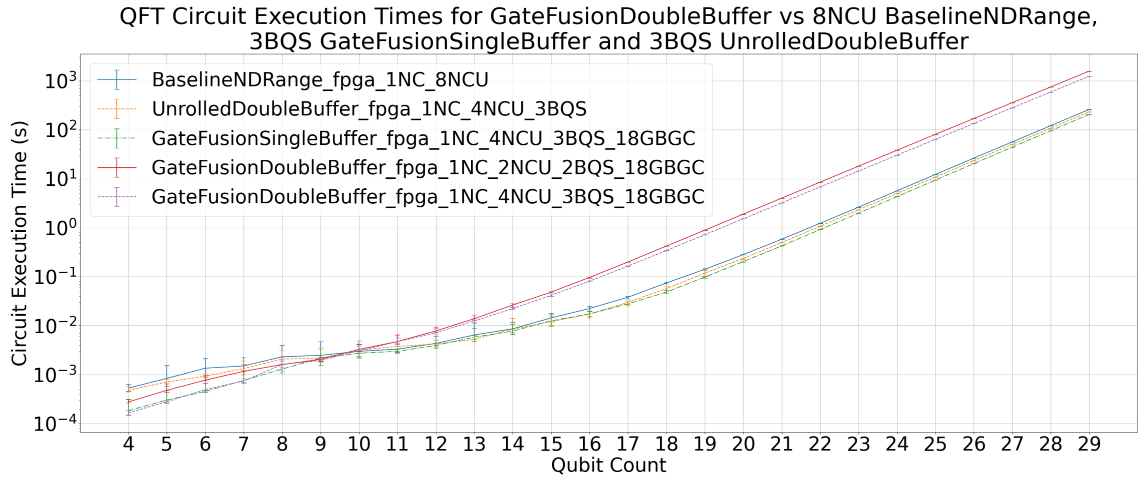


Figure 6.12: Performance of the DoubleBuffered Gate Fusion architecture compared to its single-buffer counterpart for QFT circuits.

Table 6.17: FPGA resource utilisation for a GateFusionDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 4, with up to 8 Compute Units and 1 maximum control per gate. Also shown is the QFT circuits' performance for this architecture.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	144877(42%)	234737(68%)	389492(113%)
	FFs	–	134043(19%)	181640(26%)	286516(42%)
	RAMs	–	589(29%)	615(31%)	951(47%)
	DSPs	–	84(5%)	168(11%)	336(14%)
	Fmax (MHz)	–	253	227	–
Avg Time (s)	QFT29	–	1571.8	1234.2(21%)	–
	QFT28	–	750.0	590.1(21%)	–
	QFT27	–	357.3	281.7(21%)	–
	QFT26	–	169.9	134.3(21%)	–
	QFT25	–	80.7	63.9(21%)	–

Reduced D1Q3 Circuits

Table 6.14 shows the results for the D1Q3 circuit on the GateFusionDoubleBuffer architecture with 8 maximum number of controls. Like the 1 maximum number of controls case, this variant of the architecture significantly underperforms, being about 7× slower than the baseline.

These results indicate that double-buffering is not suitable to implement for gate fusion based architectures. While in the buffered architecture without gate fusion, the drop in performance was marginal, this drop is much more substantial when using gate fusion.

Table 6.18: FPGA resource utilisation for a GateFusionDoubleBuffer architecture with BQS (the buffer qubit size, or parameter l) from 2 to 3, with up to 8 Compute Units and 8 maximum number of controls per gate. Also shown is the reduced D1Q3 average circuit performance for this architecture.

	BQS (NCU)	1 (1)	2 (2)	3 (4)	4 (8)
Resources	ALUTs	–	152984(44%)	242112(70%)	–
	FFs	–	182387(26%)	228455(33%)	–
	RAMs	–	869(43%)	894(44%)	–
	DSPs	–	84(5%)	168(11%)	–
	Fmax (MHz)	–	218	216	–
Avg Time (s)	D1Q3	–	138.1	126.0(9%)	–

6.7 FPGA vs CPU vs GPU

In this section, we compare the best performing FPGA architectures for each circuit evaluation target against the CPU and GPU, without the controls scheduling optimisation.

Figure 6.13 shows the performance of the best performing architecture from each architecture class against the NDRange architecture running on the GPU for the QFT circuits. We see that with this approach, both CPU and GPU outperform the FPGA. Table 6.19 shows that against the best performing FPGA architecture (the single-buffered 3BQS Gate Fusion architecture) without the controls scheduling optimisation, the GPU outperforms the FPGA with a factor of $20-22\times$ for the QFT. For Grover's and the Streaming circuits, we compare against the BaselineNDRange 8NCU FPGA architecture, in Figures 6.14 and 6.15, respectively. In the case of Grovers, the GPU outperforms the FPGA roughly $18\times$, and for streaming about $10\times$.

The significant performance difference between the GPU and FPGA can largely be attributed to the disparity in memory bandwidth. The GPU operates at 336 GB/s, providing it with a substantial advantage in terms of data throughput, enabling it to handle state vector updates more effectively. This high memory bandwidth allows the GPU to quickly access and manipulate large datasets In contrast, the FPGA's memory bandwidth is only 2.1 GB/s, severely limiting its ability to transfer data at the same speed. While FPGAs offer energy efficiency and reconfigurability, their lower memory bandwidth means that they are often bottlenecked when handling memory-intensive operations. This results in slower overall performance compared to GPUs, which are specifically designed to handle high-throughput, parallelised computations. Consequently, for tasks that require fast memory access and large data transfers, GPUs far outpace FPGAs in raw execution speed.

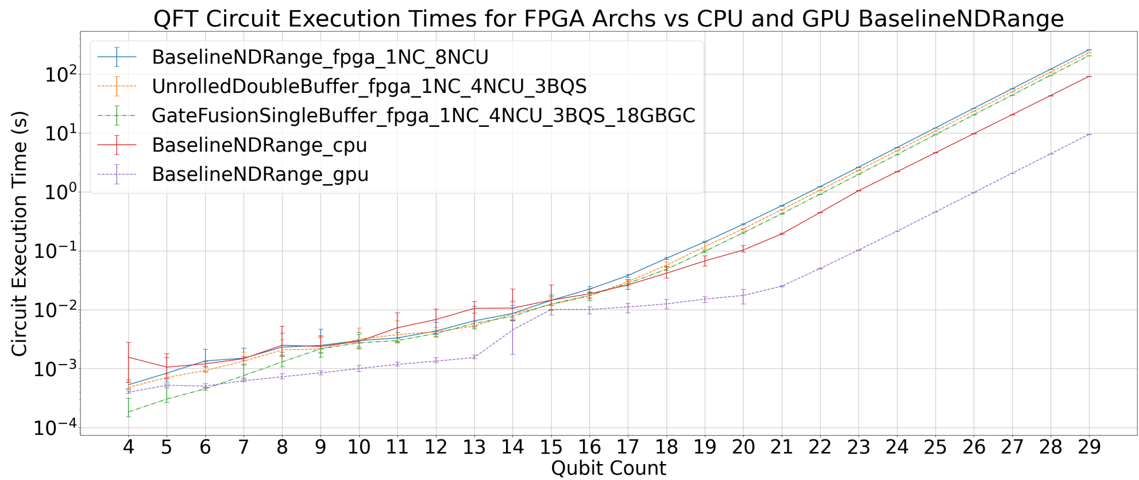


Figure 6.13: Performance comparison between the best-performing FPGA architectures from each class and the NDRange kernels on the CPU and GPU for QFT circuits.

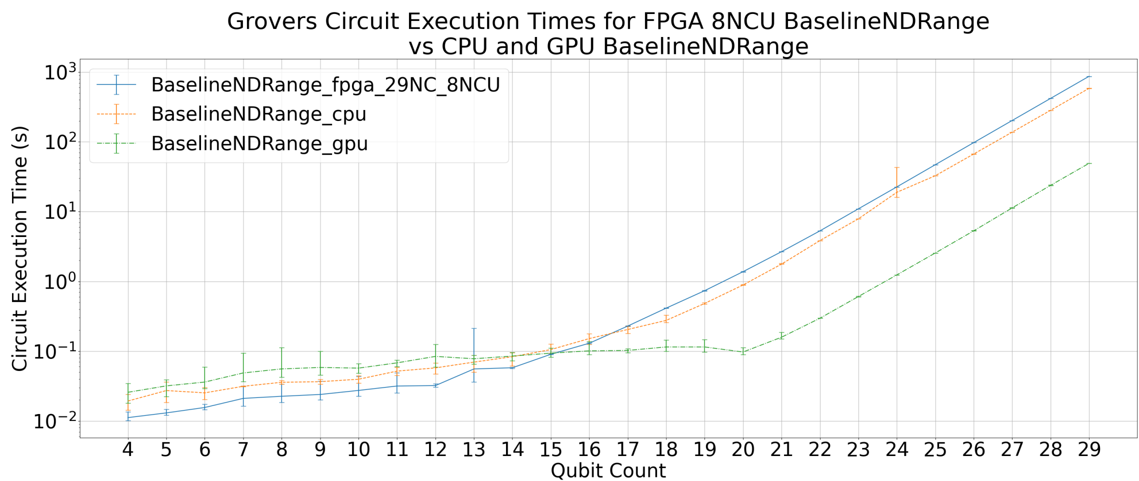


Figure 6.14: Performance comparison between the best-performing FPGA architecture (BaselineNDRange) and CPU/GPU for Grover's algorithm circuits.

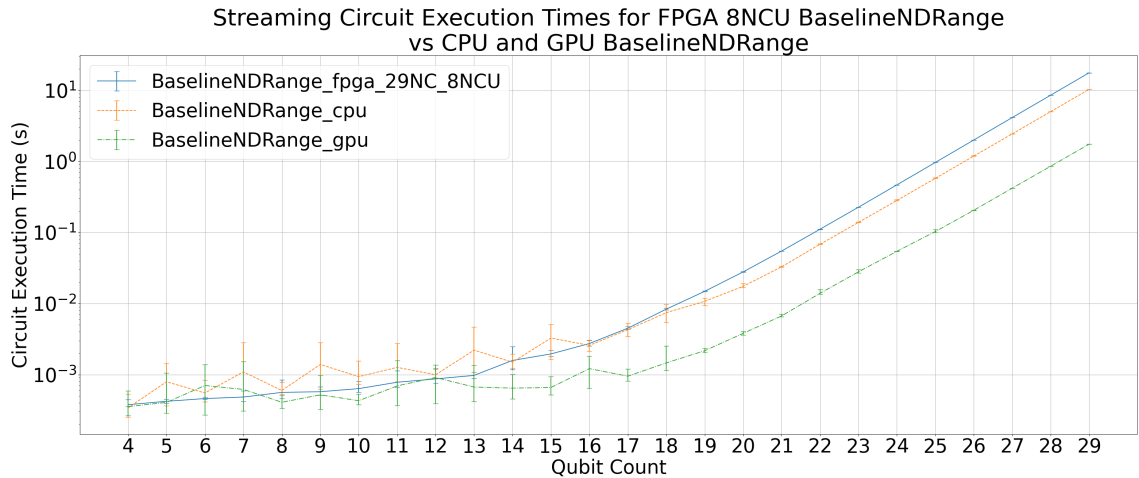


Figure 6.15: Performance comparison between the best-performing FPGA architecture (BaselineNDRange) and CPU/GPU for streaming circuits.

Table 6.19: Best-performing FPGA architectures without the optimised controls scheduling optimisation against the NDRange kernels on CPU and GPU. All runtimes are in seconds. For the FPGA architectures, the percentage improvement shown in parenthesis compared to the FPGA BaselineNDRange 8NCU architecture. For CPU and GPU, the performance improvement is shown compared to the best available FPGA architecture (SingleBuffered GateFusion 3BQS in the case of the QFT circuits).

	FPGA			CPU	GPU
Circuit	Baseline 8NCU	SingleBuff 3BQS	SingleBuffer GateFusion 3BQS	NDRange	NDRange
QFT29	260.2	229.5(12%)	206.4(21%)	91.9(2.2×)	9.5(21.8×)
QFT28	121.5	106.9(12%)	95.7(21%)	43.4(2.2×)	4.4(21.5×)
QFT27	56.7	49.7(12%)	44.2(22%)	20.6(2.1×)	2.1(21.2×)
QFT26	26.4	23.1(12%)	20.4(23%)	9.8(2.1×)	1.0(20.9×)
QFT25	12.2	10.7(12%)	9.4(23%)	4.6(2.0×)	0.5(20.5×)
Grovers29	869.9	–	–	585.2(33%)	49.3(17.7×)
Grovers28	419.8	–	–	283.4(32%)	23.8(17.6×)
Grovers27	202.4	–	–	137.8(32%)	11.3(17.9×)
Grovers26	97.5	–	–	67.0(31%)	5.3(18.4×)
Grovers25	47.0	–	–	32.7(30%)	2.6(18.4×)
Streaming29	17.7	–	–	10.4(41%)	1.7(10.1×)
Streaming28	8.6	–	–	5.0(42%)	0.9(10.0×)
Streaming27	4.1	–	–	2.4(41%)	0.4(9.9×)
Streaming26	2.0	–	–	1.2(40%)	0.2(9.7×)
Streaming25	1.0	–	–	0.6(40%)	0.1(9.4×)
D1Q3	17.9	17.1(4%)	14.6(18%)	8.2(44%)	0.7(20.1×)

6.7.1 Controls Scheduling Optimisation on CPU and GPU

Since the controls scheduling optimisation (introduced in Section 3.1.3 and evaluated on the FPGA in Section 6.4) can benefit all types of hardware systems, in this section, its performance on FPGA, CPU, and GPU is evaluated separately to the other architectures.

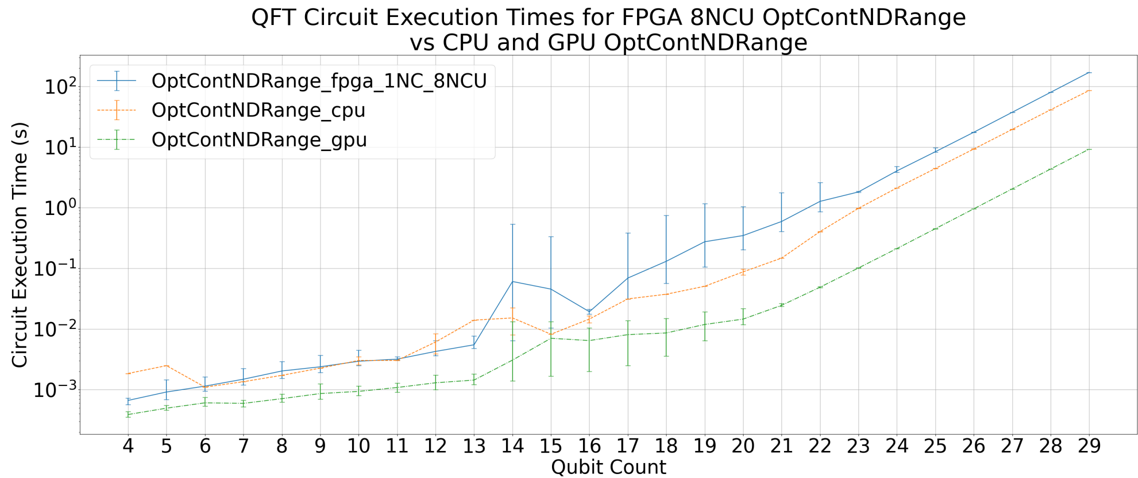


Figure 6.16: Timing performance of QFT circuits across FPGA, CPU, and GPU platforms using the OptContNDRange architecture.

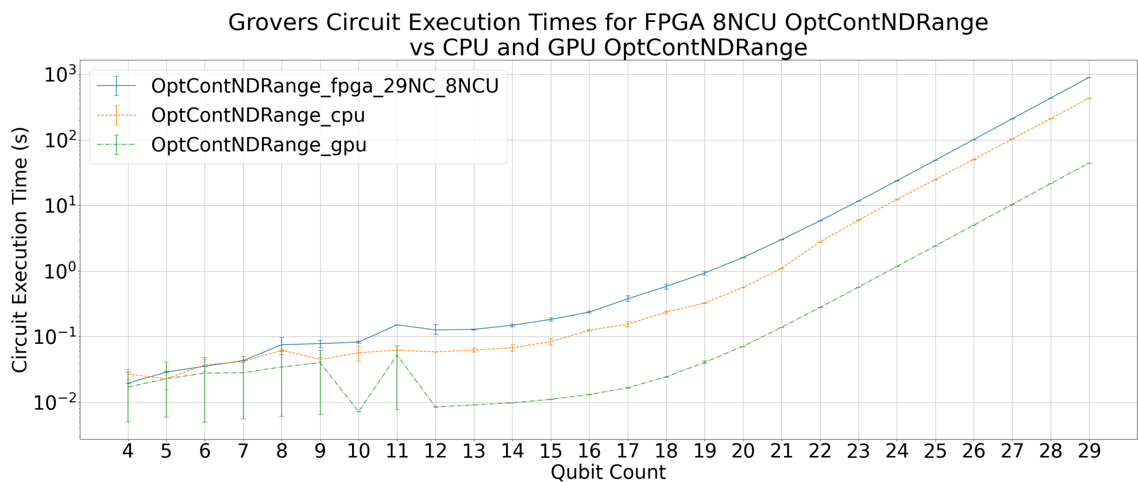


Figure 6.17: Timing performance of Grover's algorithm circuits across FPGA, CPU, and GPU platforms using the OptContNDRange architecture.

Figures 6.16, 6.17, and 6.18 show the timing performance of QFT, Grovers, and the Streaming circuits, respectively, for the three platforms. From Table 6.20, we can see that the difference in performance between the FPGA and the GPU get smaller in the cases of the QFT and Streaming circuits, whereas it gets larger for Grover's circuits; reflecting the effect of the density of controlled gates.

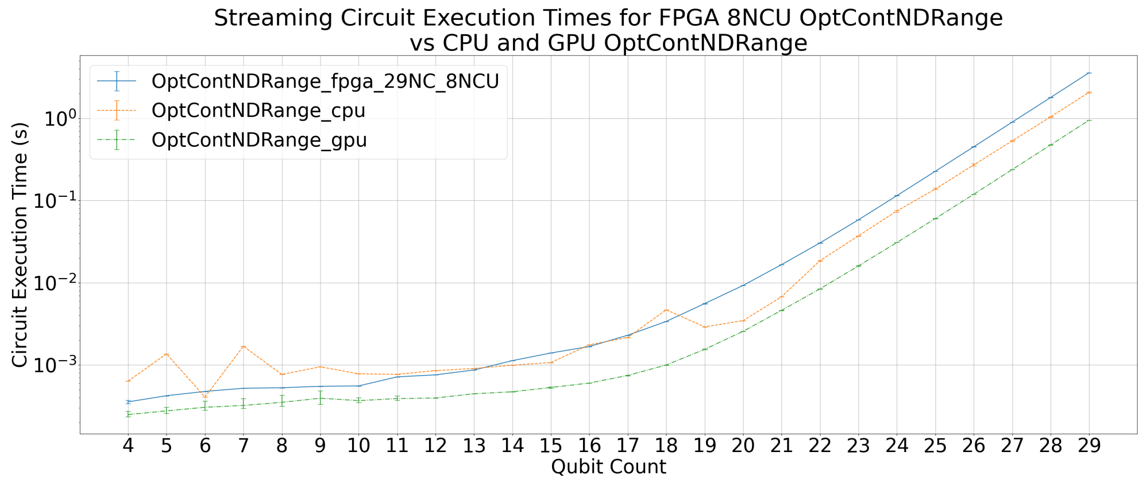


Figure 6.18: Timing performance of streaming circuits across FPGA, CPU, and GPU platforms using the OptContNDRange architecture.

Table 6.20: Controls scheduling optimised NDRange architectures runtime comparison (in seconds) across different platforms.

Circuit	FPGA 8NCU	CPU	GPU
QFT29	170.5	85.6(2.0×)	9.2(18.5×)
QFT28	79.9	41.2(1.9×)	4.3(18.4×)
QFT27	37.4	19.6(1.9×)	2.0(18.3×)
QFT26	17.5	9.3(1.9×)	1.0(18.3×)
QFT25	8.4	4.4(1.9×)	0.5(18.6×)
Grovers29	897.9	433.4(2.1×)	44.6(20.1×)
Grovers28	434.2	211.3(2.1×)	21.6(20.1×)
Grovers27	210.2	103.3(2.0×)	10.4(20.2×)
Grovers26	101.4	50.6(2.0×)	5.0(20.2×)
Grovers25	49.0	24.8(2.0×)	2.4(20.2×)
Streaming29	3.59	2.08(1.7×)	0.95(3.8×)
Streaming28	1.8	1.04(1.7×)	0.48(3.8×)
Streaming27	0.9	0.53(1.7×)	0.24(3.8×)
Streaming26	0.45	0.27(1.7×)	0.12(3.8×)
Streaming25	0.23	0.14(1.6×)	0.06(3.8×)
D1Q3	12.8	7.6(1.7×)	0.7(18.3×)

6.8 Energy Consumption Analysis

While the timing results presented in the previous section show that the GPU generally performs $10 - 20\times$ better than the FPGA, and the CPU $1.5 - 2\times$, another important factor to consider is the energy consumption of the devices during simulation. In order to be competitive with distributed computing simulation methods, it will be necessary to move to clusters of FPGAs. In the case of supercomputing clusters, energy consumed becomes an important metric, and so, in this section, we focus on the energy consumed by a device to simulate an evaluation circuit.

As we do not have a way to directly measure the power utilisation during the execution of the circuit on the devices, we rely on the quoted maximum power rating of each device to give us an estimate of the energy consumed during simulation. According to [130], our FPGA consumes at 25W for applications with similar clock speeds and utilisation. Our CPU consumes 160W and the GPU consumes 250W. To compute an estimate of the energy consumed during the execution of a circuit, we multiply the total time required to simulate the circuit by the target platform's rated power consumption.

6.8.1 Architectures without Controls Scheduling Optimisation

As with the timing analysis, first, we evaluate the best-performing FPGA architecture, with no controls scheduling optimisation, against the CPU and GPU NDRange in terms of energy. Figures 6.19, 6.20, and 6.21 show the energy consumption for the QFT, Grovers, and streaming circuits, respectively.

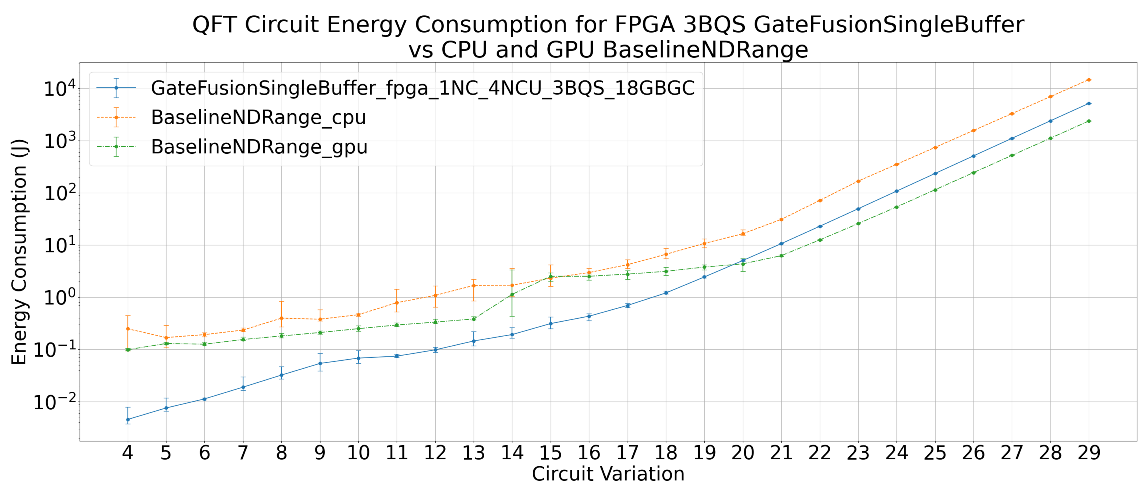


Figure 6.19: Energy consumption comparison for QFT circuits using the best-performing FPGA architecture (GateFusionSingleBuffer 3BQS) compared to CPU and GPU platforms.

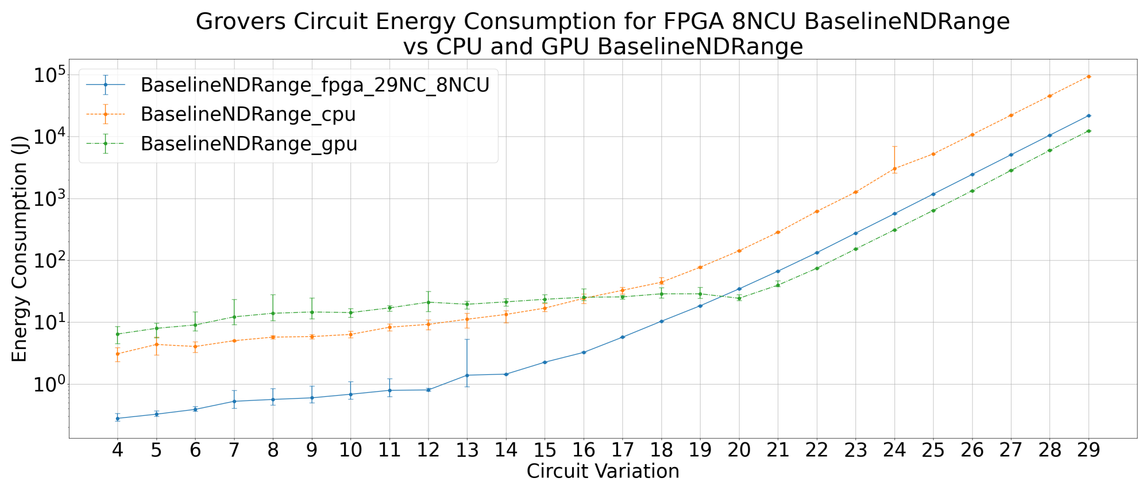


Figure 6.20: Energy consumption comparison for Grover's algorithm circuits using the best-performing FPGA architecture compared to CPU and GPU platforms.

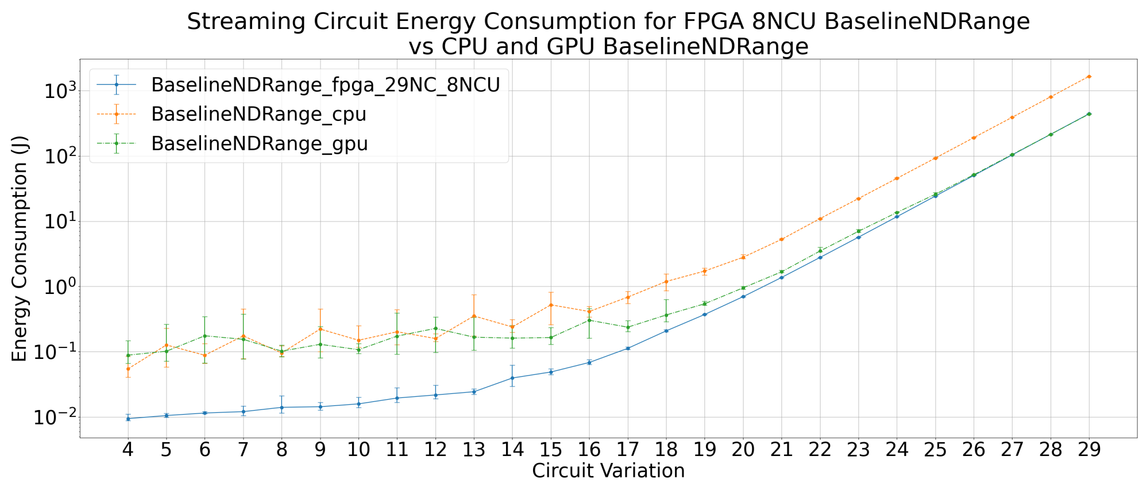


Figure 6.21: Energy consumption comparison for streaming circuits using the best-performing FPGA architecture compared to CPU and GPU platforms.

When evaluating the energy efficiency, it is notable that the FPGA, despite performing up to $20\times$ slower than the GPU, consumes significantly less energy, leading to competitive performance-per-Watt ratios. This efficiency could position FPGAs as a compelling choice for quantum circuit simulations in environments with stringent power constraints, including utilisation of FPGA clusters in HPC centres.

6.8.2 OptContNDRange FPGA vs CPU and GPU

The controls scheduling optimisation shows significant improvements in both runtime and energy efficiency across all platforms, but the energy savings on the FPGA are particularly noteworthy. The optimisation reduces the overhead associated with iterating over control

Table 6.21: Energy consumption comparison for the best performing architectures without the controls scheduling optimisation.

Circuit	FPGA Best-Performing	CPU NDRange	GPU NDRange
	GateFusion SingleBuffer 3BQS		
QFT29	5159.6	14710.8(0.4×)	2371.2(2.2×)
QFT28	2392.1	6944.7(0.3×)	1112.3(2.2×)
QFT27	1106.0	3292.3(0.3×)	521.4(2.1×)
QFT26	510.1	1564.2(0.3×)	244.2(2.1×)
QFT25	234.8	740.8(0.3×)	114.4(2.1×)
D1Q3	365.0	1312.0(0.3×)	175.0(2.1×)
	BaselineNDRange 8NCU		
Grovers29	21746.9	93628.8(0.2×)	12316.0(1.8×)
Grovers28	10494.6	45347.4(0.2×)	5961.8(1.8×)
Grovers27	5060.4	22042.4(0.2×)	2827.6(1.8×)
Grovers26	2438.1	10713.5(0.2×)	1327.3(1.8×)
Grovers25	1175.3	5230.8(0.2×)	638.8(1.8×)
Streaming29	442.3	1657.0(0.3×)	437.5(1%)
Streaming28	214.1	803.7(0.3×)	214.0(0%)
Streaming27	103.6	390.4(0.3×)	105.1(1.0×)
Streaming26	50.1	190.4(0.3×)	51.4(1.0×)
Streaming25	24.2	92.6(0.3×)	25.8(0.9×)

gates by directly addressing them within the NDRange kernel. This leads to a substantial reduction in both execution time and power consumption, particularly in control-heavy circuits such as the streaming circuits. Figures 6.22, 6.23, and 6.24 demonstrate the energy consumption on the different platforms for the OptContNDRange architecture, for QFT, Grovers, and streaming circuits, respectively. We can see from the graphs that, compared to the equivalent runtime graphs for QFT and Grover's (Figures 6.16 and 6.17), the performance gap between the FPGA and the GPU gets narrower when we consider energy consumption. For the streaming circuits, the FPGA outperforms the GPU by a factor of $2.5\times$, as shown in Table 6.22, for the 29-qubit streaming circuit.

6.8.3 Relative benefit of Optimised Controls Scheduling

In Section 6.4, we included a brief discussion comparing the OptContNDRange architecture to the BaselineNDRange performance on the FPGA. We reiterate these findings here and present them alongside the equivalent results for CPU and GPU, to highlight the benefit per each platform that this optimisation offers.

While the controls scheduling optimisation presented in this work certainly benefits all three

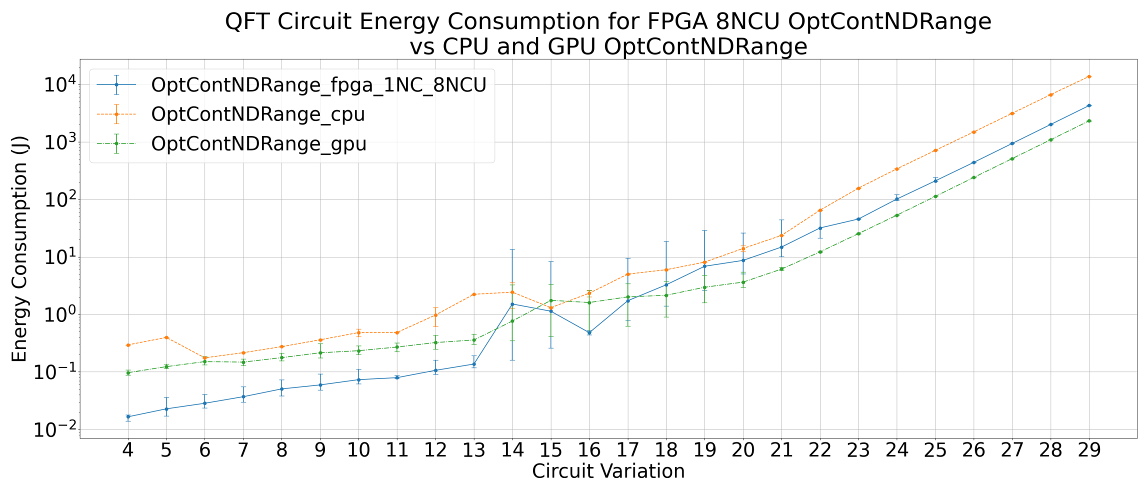


Figure 6.22: Energy consumption comparison for QFT circuits using the OptContNDRange architecture on FPGA, CPU, and GPU platforms.

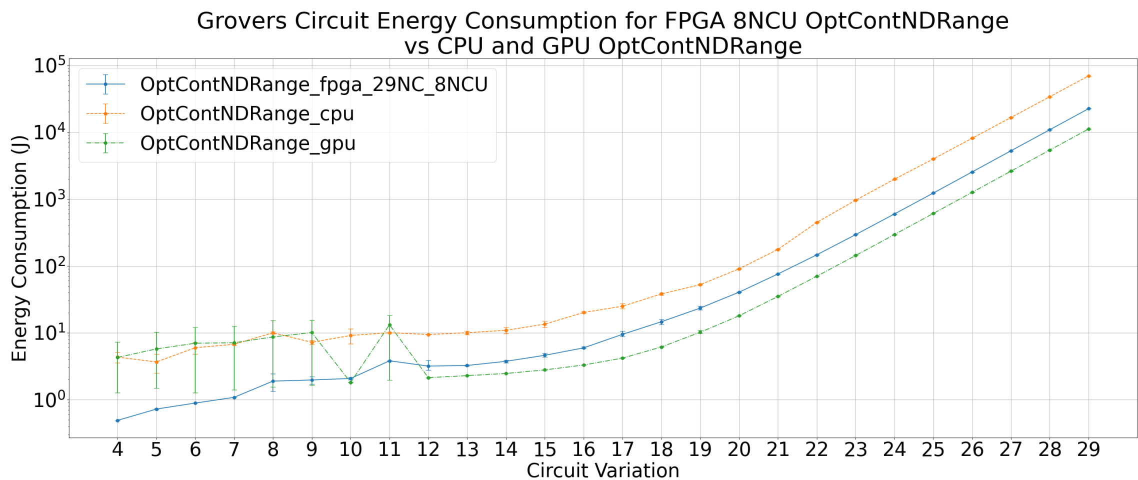


Figure 6.23: Energy consumption comparison for Grover's algorithm circuits using the OptContNDRange architecture on FPGA, CPU, and GPU platforms.

platforms (fewer NDRange work items will mean less execution time), we postulated that the benefit for the FPGA relative to the baseline implementation would be considerably higher than those for the CPU and GPU. To check this, Table 6.23 shows a comparison between the BaselineNDRange implementation and the OptContNDRange implementation, highlighting the benefit gained for each platform across our range of test circuits. The numbers are quoted in terms of timing results in seconds; however the performance gained metrics would be the same for energy consumption, since we are comparing different architectures on the same device platform.

As expected, circuits with a higher density of controlled gates benefit most from this optimisation. The FPGA stands out in its performance improvement, achieving an energy reduction of up to $1.5\times$ for the QFT circuits (which have at most one control on a gate) and approxi-

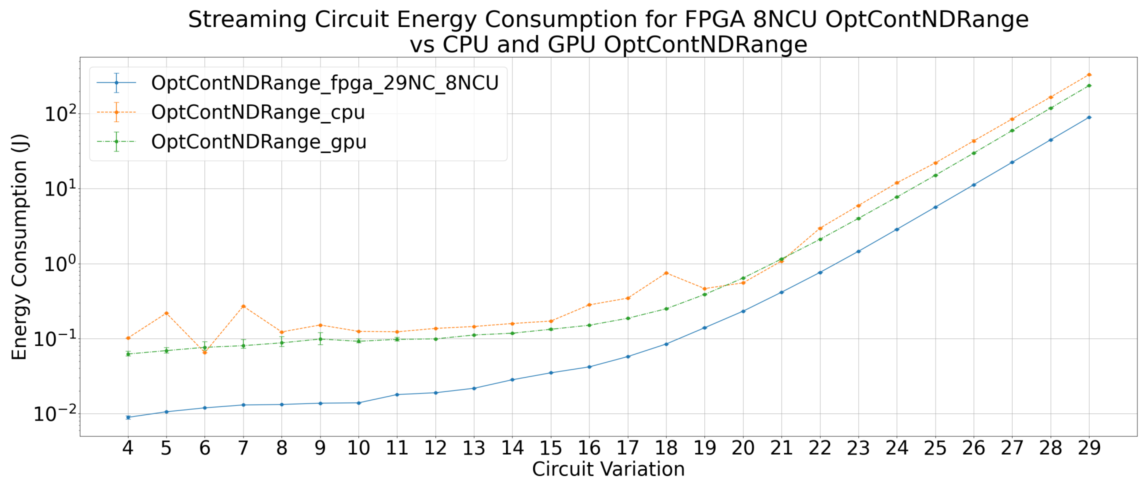


Figure 6.24: Energy consumption comparison for streaming circuits using the OptCont-NDRange architecture on FPGA, CPU, and GPU platforms.

mately $4.9\times$ for the streaming circuits when the controls scheduling optimisation is applied. In comparison, the CPU and GPU platforms see much smaller gains, with improvements of $4-7\%$ for the QFT. The GPU sees just $1.8\times$ improvement for the streaming circuits, while the FPGA sees up to $5\times$.

On the other hand, for control-sparse circuits such as Grovers, the FPGA performs marginally worse using the optimised controls optimisation. The CPU actually demonstrates the highest performance benefit for this circuit.

Table 6.22: Energy consumption comparison for the NDRange architectures with the controls scheduling optimisation (OptContNDRange). Results shown in Joules.

Circuit	FPGA 8NCU OptContNDRange	CPU OptContNDRange	GPU OptContNDRange
QFT29	4261.9	13688.7(0.3×)	2305.9(1.8×)
QFT28	1998.5	6597.1(0.3×)	1085.7(1.8×)
QFT27	935.9	3128.6(0.3×)	510.2(1.8×)
QFT26	438.5	1485.3(0.3×)	239.6(1.8×)
QFT25	209.2	709.0(0.3×)	112.5(1.9×)
Grovers29	22447.5	69337.0(0.3×)	11142.1(2.0×)
Grovers28	10854.2	33807.3(0.3×)	5392.9(2.0×)
Grovers27	5254.9	16531.9(0.3×)	2605.2(2.0×)
Grovers26	2536.2	8092.3(0.3×)	1257.8(2.0×)
Grovers25	1225.3	3967.2(0.3×)	607.1(2.0×)
Streaming29	89.7	332.5(0.3×)	237.3(0.4×)
Streaming28	44.9	166.9(0.3×)	118.9(0.4×)
Streaming27	22.5	85.2(0.3×)	59.6(0.4×)
Streaming26	11.3	43.5(0.3×)	29.9(0.4×)
Streaming25	5.7	22.1(0.3×)	15.1(0.4×)
D1Q3	320.0	1216.0(0.3×)	175.0(1.8×)

Table 6.23: Comparison of the performance improvements from the controls scheduling optimisation (OptContNDRange) across FPGA, CPU, and GPU platforms. The table shows the runtimes for each platform before and after applying the optimisation, with percentage improvements indicated in parentheses. In general, the FPGA demonstrates the largest relative benefit, particularly for control-heavy circuits like the streaming circuits.

	FPGA		CPU		GPU	
Circuit	Baseline	OptCont	Baseline	OptCont	Baseline	OptCont
QFT29	260.2	170.5(34%)	91.9	85.6(7%)	9.5	9.2(3%)
QFT28	121.5	79.9(34%)	43.4	41.2(5%)	4.4	4.3(2%)
QFT27	56.7	37.4(34%)	20.6	19.6(5%)	2.1	2.0(5%)
QFT26	26.4	17.5(34%)	9.8	9.3(5%)	1.0	1.0(0%)
QFT25	12.2	8.4(31%)	4.6	4.4(4%)	0.5	0.5(0%)
Grovers29	869.9	897.9(−3%)	585.2	433.4(26%)	49.3	44.6(10%)
Grovers28	419.8	434.2(−4%)	283.4	211.3(25%)	23.8	21.6(9%)
Grovers27	202.4	210.2(−4%)	137.8	103.3(25%)	11.3	10.4(8%)
Grovers26	97.5	101.4(−4%)	67.0	50.6(24%)	5.3	5.0(6%)
Grovers25	47.0	49.0(−4%)	32.7	24.8(24%)	2.6	2.4(8%)
Streaming29	17.7	3.6(4.9×)	10.4	2.1(5.0×)	1.7	0.9(1.8×)
Streaming28	8.6	1.8(4.8×)	5.0	1.0(4.8×)	0.9	0.5(1.8×)
Streaming27	4.1	0.9(4.6×)	2.4	0.5(4.6×)	0.4	0.2(1.8×)
Streaming26	2.0	0.5(4.4×)	1.2	0.3(4.4×)	0.2	0.1(1.7×)
Streaming25	1.0	0.2(4.3×)	0.6	0.1(4.2×)	0.1	0.1(0%)
D1Q3	17.9	12.8(28%)	8.2	7.6(7%)	0.7	0.7(0%)

6.9 Summary

In this chapter, we presented a comprehensive evaluation of the developed FPGA architectures for Full State Vector Quantum Circuit Simulation. The performance of these architectures was analysed in terms of execution time and energy consumption, with comparisons against CPU and GPU implementations for various quantum circuits, including QFT, Grover's algorithm, streaming circuits, and the reduced D1Q3 circuits.

We began by evaluating the Direct Iteration Processing architectures, where the BaselineNDRange approach served as a foundation for comparison. The results showed that while increasing the number of compute units led to improvements in performance, the scaling behaviour was limited by the achieved clock frequency for lower numbers of compute units, but the cost of parallelism was amortised for our highest achievable number of compute units. The UnrolledLoops and OnBoardUnrolledLoops architectures, which sought to reduce overhead by avoiding the NDRange kernel structure, did not outperform the baseline, highlighting the efficiency of the HLS scheduler for DIP architectures.

Next, we evaluated the Controls Scheduling Optimisation with OptContNDRange. This optimisation showed significant performance improvements, particularly in control-heavy circuits like the streaming circuits, where the scheduling of controlled gates reduced the overhead associated with kernel invocations. For these circuits, we observed up to a $5\times$ improvement in performance compared to the baseline, whereas control-sparse circuits, such as Grover's algorithm, saw limited benefit from the optimisation.

The Buffered Architectures were introduced to further improve memory access efficiency. Both the SingleBuffered and DoubleBuffered architectures were evaluated. The SingleBuffered version showed a 12% benefit over the baseline, and while the DoubleBuffered version showed a similar benefit over the baseline, it actually marginally underperformed compared to its single-buffered counterpart.

We then assessed the Gate Fusion Optimisation, which processes multiple quantum gates as blocks, reducing the memory access overhead. The Single-Buffered Gate Fusion architecture demonstrated a noticeable performance improvement over the standard single-buffered approach, especially for the QFT circuits, of up to 20%. However, the Double-Buffered Gate Fusion architecture did not show any performance gains, as the additional control logic for alternating input/output buffers introduced overhead that outweighed the benefits of gate fusion, significantly underperforming compared to its single-buffered counterpart.

Finally, we compared the best-performing FPGA architectures against CPU and GPU implementations. While the GPU generally outperformed the FPGA by $10 - 20\times$ in raw execution time, the FPGA showed competitive performance when energy efficiency was considered, particularly in control-dense circuits where the control scheduling optimisation

provided significant benefits (up to $2.5\times$).

Overall, this chapter demonstrated the trade-offs between different FPGA architectures and optimisations, highlighting the importance of tailoring designs to the specific characteristics of the target quantum circuits. The findings suggest that while FPGAs do not match the performance of GPUs in raw execution time, their energy efficiency and flexibility still make them a viable option for scalable quantum circuit simulation, especially in energy-conscious high-performance computing environments. We also show that it is possible to develop optimisations which, while can be applied to all hardware platforms, they benefit the FPGA significantly more than the CPU and GPU. Future work should explore scaling these architectures and optimisations to larger FPGAs and FPGA clusters.

Chapter 7

Conclusions

This chapter presents a summary and a discussion of the research conducted in this thesis, highlighting the main findings, contributions, and addressing the research questions posed at the outset. The work explored the design and evaluation of FPGA architectures for Full State Vector Quantum Circuit Simulation (FSVQCS), aiming to optimise performance and energy efficiency. Several architectures were implemented and benchmarked against CPU and GPU platforms, with specific emphasis on the benefits and trade-offs of FPGA-based solutions.

As stated in the Section **??**, we set out in this work to investigate the potential of using FPGAs for FSVQCS. A key motivator of this work is the realisation that in order to perform simulation of meaningful, real-world, quantum circuits, a simulation platform needed to support as high a number of qubits as possible. As FSVQCS requires the storage of the full state vector, moving to distributed memory systems is required as the memory requirement grows exponentially in the number of qubits, and thus energy consumption is a key factor to consider.

7.1 Contributions to the Field

In this section, we summarise the main novel contributions presented in this work. We presented several FPGA architecture implementations and compared their performance to a baseline, which does not implement any FPGA-specific optimisations.

7.1.1 Novel Universal FPGA-based QC Simulators

In this work, we presented and evaluated several quantum circuit simulation architectures, based on the Full State Vector approach, on FPGAs. As stated in Section 2.5, our primary goals for developing the simulator were:

- **Universality**: The simulator should be able to simulate universal quantum computing algorithms, and not be limited in gate set supported.
- **Scalability**: The simulator's performance should scale with compute resources.
- **Reusability**: The FPGA binary should not be specific to a particular quantum circuit, i.e., a compilation process should not be necessary between different circuits runs.

In the developed simulators, the goal of universality is achieved as long as quantum circuits can be decomposed to single-target multi-controlled gates. Since the single-target gates are simulated using matrix-vector multiplication, this set of gates certainly enables universal quantum computation. Scalability is achieved for some architectures as we do see an improvement in performance with increased compute unit count. Reusability is mostly achieved, as the only compile-time parameterisation that limits the circuits which can be simulated on a particular instance of the simulator is the maximum number of controls allowed in the circuits. Otherwise, different circuits can be freely simulated on the same FPGA binary, with no required recompilation step. There is also no limit on the number of qubits possible to simulate on our device, and there is no difference in operation for simulating circuits of different circuit-widths.

To our knowledge, we believe this quantum circuit simulation platform is the first FPGA-based platform which can simulate generally any quantum circuit (i.e. not specialised for a particular circuit) at a demonstrably high number of qubits. As discussed in Section 2.4, pre-exisitng FPGA-based simulators were either limited in number of qubits (< 20) for general quantum circuit simulators, or were specialised to tackle particular quantum algorithms such as the Quantum Wavelet Transform and Grover's Search algorithm (with up to 32 qubits). We demonstrate that our simulation platform can simulate up to 29 qubits for any quantum algorithm with a universal gate set supported, and has the potential to scale beyond this number, given sufficient memory resources.

7.1.2 Gate Fusion FPGA Architectures

To our knowledge, this is the first attempt at applying the Gate Fusion optimisation (see Section 3.5) to an FPGA architecture. While our results were limited in studying the scaling of such an architecture due to limitations of the utilised HLS tools, they still indicated that such an architecture has the potential to scale on larger FPGAs. In addition, even with the current limitations on scaling, we still show an improvement over the baseline FPGA architecture.

7.1.3 Controls Scheduling Optimisation

As we explored the development of Direct Iteration Processing (DIP) architectures on FPGAs, we realised that for control-heavy gates, a large number of iterations were being scheduled which did not modify the state vector. While on CPU and GPU, this scheduling does not add much overhead, on the FPGA, where the circuit logic has to be handled statically, many iterations were being scheduled and wasted. Therefore, we developed the optimisation introduced in Section 3.1.3, with the aim of scheduling only as many iterations which would modify the state vector. This required a formula to compute the global iteration equivalent of an index from a reduced iteration set. We saw significant improvements in performance from this optimisation for controlled gates. Using this optimisation, a single-controlled gate generally takes half the time needed to simulate compared to a non-controlled gate; whereas in the baseline, the difference is less pronounced. This performance improvement is consistent for higher number of controls, with each added control halving the required time.

7.1.4 Circuit Width Reduction

Chapter 5 introduced novel circuit reduction techniques which were developed as part of this work. The main goals of these techniques are two-fold.

First is to reduce the width (number of qubits) of circuits whose state vectors would not otherwise fit in the memory system of the simulation device, by specialising them for particular values of the reduced qubits, simulating each specialised circuit separately, then combining their results to find the simulation result of the original circuit. We demonstrated this on a circuit which originally required 37 qubits, which would have needed over a terabyte to simulate; by showing how it can be reduced to a set of 25-qubit circuits, each requiring only ~ 270MB to simulate. The results indicate that significant gains in simulation performance can be achieved, paving the way for more scalable quantum simulations.

Second is to be able to simulate several reduced circuits in parallel through utilisation of independent memory banks. While we were not able to show benefit from the architecture developed to demonstrate this goal, in theory this should still be possible.

These techniques were the primary subject of the published works, [56, 125], which focused on circuits utilising the computational basis. However, in Section 5.9, we expanded upon the published work by demonstrating how these techniques can benefit circuits which utilise either the computational or amplitude bases.

7.2 Summary of Results

The quantum circuit simulation experiments provided insights into the performance and scalability of different architectures, with comparisons to baseline methods as well as CPU and GPU platforms. Each architecture demonstrated varied results in terms of performance, energy efficiency, and scalability, especially when applied to control-heavy circuits and high-qubit systems. In this section, the key takeaways from the quantum circuit simulation experiments reported in Chapter 6 are summarised.

7.2.1 Direct Iteration Processing

The 8NCU BaselineNDRange architecture proved to be the best performing DIP-based architecture without the controls processing optimisation. The alternative approaches, UnrolledLoops and OnBoardUnrolledLoops, did not scale with the number of compute units and showed worse performance than the baseline. The 8NCU OptContNDRange, which added the controls processing optimisation to our baseline, resulted in a $1.5 - 5\times$ improvement in performance, depending on the target circuit.

Compared to the GPU, the 8NCU BaselineNDRange is outperformed nearly $27\times$ in terms of runtime. In terms of energy consumption, however, the GPU's advantage drops to $2.7\times$ for the QFT at high numbers of qubits.

We saw the OptContNDRange architecture benefit all three platforms. However, the performance gain was more pronounced for the FPGA; as the GPU only saw a $1.0 - 1.8\times$ improvement compared to the baseline. Using this optimisation also resulted in the FPGA seeing a performance-per-Watt advantage over the GPU for specific circuits, as it consumed $2.5\times$ less energy for the streaming circuits at high numbers of qubits.

The TwoCircuitNDRange architecture was developed primarily to facilitate parallel simulation of the circuits generated by the width-reduction techniques described in Chapter 5. Due to the limitations we faced while implementing this architecture, we cannot conclude anything meaningful about its performance based on our results. Our results clearly indicate that we were not able to implement the simulation of the circuits in parallel using independent memory banks. Thus, further work is required to see if this is feasible.

7.2.2 Buffered and Gate Fusion Architectures

The SingleBuffered architecture adds buffering to the UnrolledLoops architecture to make better use of memory accesses. This architecture sees a 12% improvement over the baseline at high qubit counts. The DoubleBuffered variation performed marginally slower compared

to the single buffered version. Compared to the GPU, the SingleBuffered architecture was outperformed $24\times$ in raw execution time.

The GateFusionSingleBuffer architecture is the best performing architecture on the FPGA without the controls scheduling optimisation, seeing almost a 20% benefit over the baseline. Compared to the GPU, it is outperformed $22\times$ in terms of execution time, but only $2.2\times$ in energy consumption. The GateFusionDoubleBuffer architecture underperformed significantly compared to its single-buffered counterpart, being $4 - 6\times$ slower.

7.3 Discussion

This work represents a step towards answering the primary research question: *"For FSVQCS, can FPGAs outperform, in terms of raw performance, or performance-per-Watt, the traditional simulation platforms, CPUs and GPUs?"*. Through the development and testing of various architectural approaches, it became clear that while FPGAs offer unique advantages in terms of energy efficiency and flexibility, they face significant challenges in terms of raw performance compared to CPU and GPU implementations. The experimental results highlighted the trade-offs between energy consumption, computational speed, and resource utilisation, particularly for circuits with high qubit counts and controlled gates.

Our work contributes to the growing body of research focused on using reconfigurable hardware to address the computational demands of quantum circuit simulation, emphasising the importance of optimisation strategies like buffering, gate fusion, and control gate scheduling. These techniques significantly improved the performance of FPGA-based simulators, particularly for control-heavy circuits, but also exposed limitations in terms of hardware resource utilisation and the ability to scale beyond certain thresholds.

7.3.1 Research Questions

In this section, we revisit the research question posited in Section **??** and discuss the insights our findings provide towards answering them.

Can scalable FPGA architectures for Full State Vector Quantum Circuit Simulation be designed?

The evaluation demonstrated that scalable FPGA architectures for FSVQCS are possible but face significant challenges, particularly in terms of resource utilisation and clock frequency. The BaselineNDRange architecture scaled relatively well with up to 8 NCUs, achieving a

37% performance improvement for high qubit counts in QFT circuits. However, further scaling was limited by resource constraints.

The SingleBuffered architecture showed better scalability than the DIP-based architectures, with the 3BQS version achieving up to 40% improvement over the 2BQS. However, the DoubleBuffered architectures introduced additional complexity without significant gains. This suggests that memory management optimisations, such as buffering, can enhance scalability but require careful balancing of resource usage.

The Gate Fusion technique proved to be a promising approach for improving scalability, especially in combination with buffering. The 3BQS Single-Buffered Gate Fusion architecture achieved 48% faster performance than the 2BQS, indicating that the gate fusion architecture is the best scaling with hardware utilisation.

How do the designed FPGA architectures compare against CPU and GPU implementations of this application?

The GPU outperformed all FPGA architectures in terms of raw execution time, with speedups of $10-22\times$ for QFT circuits and $18\times$ faster for Grover's algorithm. The CPU also performed $1.5-2\times$ faster than the FPGA in most cases. However, the FPGA showed significant energy efficiency advantages for evaluation targets with a high density of controlled gates. Although slower than the GPU, the FPGA consumed much less power, leading to competitive performance-per-Watt ratios, especially for control-dense circuits like the streaming circuits, where the FPGA achieved up to $5\times$ better performance with optimisations compared to the baseline FPGA.

The FPGA architectures also benefited most from optimisations like control scheduling and buffering, with the OptContNDRange architecture showing a $5\times$ performance improvement for streaming circuits. These optimisations narrowed the performance gap with the CPU and GPU, particularly in terms of energy efficiency, and in some cases caused the FPGA to demonstrate better energy efficiency than the GPU.

Is there a performance-per-Watt benefit to using an FPGA over a GPU for this application?

Yes, the evaluation clearly demonstrated a performance-per-Watt advantage for the FPGA over the GPU, particularly in control-heavy circuits like the streaming circuits. While the GPU was $10 - 20\times$ faster in raw execution time, the FPGA consumed significantly less power, making it a more energy-efficient option. We estimate that in particular, moving to larger FPGA boards and FPGA clusters will yield even better results in terms of energy efficiency.

Although the GPU and CPU are faster overall, the FPGA's lower power consumption makes it an appealing choice for distributed quantum circuit simulations in HPC clusters, where energy consumption is a critical factor. The FPGA's ability to achieve competitive performance-per-Watt suggests that it can be integrated into low-power quantum computing solutions.

What types of circuits lend themselves best to FPGA-based FSVQCS implementations?

Control-dense circuits, such as the streaming circuits, benefit the most from FPGA-based FSVQCS implementations. The OptContNDRange architecture, showed up to $5\times$ performance improvements for these circuits. This highlights the FPGA's strengths in handling circuits where controlled gates are heavily used.

QFT circuits, with their high controlled-gate density but lower number of controls per gate, also performed well on the FPGA architectures, especially with buffering and gate fusion optimisations. The Single-Buffered Gate Fusion architecture achieved 21% better performance than the baseline for QFT circuits.

Control-sparse circuits, such as Grover's algorithm, did not see the same level of performance improvements with the controls-scheduling optimisation. On the FPGA, these circuits performed best using the GateFusionSingleBuffer architecture.

Overall, this indicates that the choice of FPGA architecture to use should depend on the circuit structure, suggesting that a model can be developed to analyse quantum circuits pre-simulation to inform the choice of simulation architecture.

How feasible are HLS techniques for compiling such designs?

HLS techniques proved to be feasible for compiling FPGA architectures for FSVQCS, but several challenges were encountered, particularly related to clock frequency constraints and resource utilisation.

The BaselineNDRange and OptContNDRange architectures compiled successfully with up to 8 NCUs, but further scaling was limited by ALUT utilisation on our evaluation board. Buffering and gate fusion were successfully implemented using HLS, but the DoubleBuffered architectures introduced additional complexity that reduced performance. The HLS tools could not infer the required registers to handle large buffer sizes efficiently, limiting us to a BQS of 3. We estimate that we would see even better scaling for higher BQS values, and it would also allow us to get a better benefit from gate fusion.

Overall, HLS techniques are feasible for compiling FPGA designs for quantum circuit simulation, but manual optimisation and careful resource management are necessary to achieve

optimal performance. Future improvements in HLS tools could help overcome some of these challenges, particularly for more complex FPGA designs.

7.3.2 Limitations

In this section, we summarise the main limitations of the developed architectures and techniques.

Optimised Controls Scheduling formula

In the context of the controls scheduling optimisation introduced in Section 3.1.3, the formula for converting an iteration index from the reduced iteration set to its equivalent in the global iteration set is currently iterative. This leads to a higher critical path latency when implemented on an FPGA. We are currently unsure if we can write a closed form formula for performing this conversion.

Manual Circuit-Width Reduction

For the circuit-width reduction techniques introduced in Chapter 5, we can currently automate the simplest case of qubit reduction using our quantum circuit toolchain (described in Section 4.2). However, the more complex cases of qubit reduction (where the qubit to be reduced is used as more than just a control qubit for gates with other qubits as targets) are yet to be automated and all width-reductions demonstrated in this thesis were done by hand for these cases.

Measuring power usage

The energy consumption analysis in this work relied on maximum rated power draws for each platform (FPGA, CPU, and GPU), which provided only a worst-case estimate of power consumption. Without access to real-time power monitoring tools during the simulations, it was not possible to accurately capture dynamic power variations throughout the execution of the circuits. This limitation impacts the precision of the performance-per-Watt analysis, as actual power consumption is likely to vary depending on the circuit's workload, memory accesses, and control gate density.

HLS Tools

The use of HLS techniques to compile FPGA designs introduced several limitations. While HLS allowed for more rapid development and testing of architectures, the synthesis process

often resulted in suboptimal hardware configurations. Additionally, HLS tools were limited in optimising resource usage, which affected the scalability of designs as buffering and gate fusion techniques were applied. More advanced FPGA design tools, or manual tuning at the RTL level, may be necessary to fully realise the performance potential of the architectures.

Parallel Simulation of Multiple Circuits The TwoCircuitNDRange architecture was designed with the intention of simulating two quantum circuits concurrently within the same kernel call, to utilise the circuit-width reduction techniques developed earlier in the thesis. However, we were unable to fully evaluate this architecture due to the inability to ensure that the state vectors were assigned to separate memory banks; and indeed our results indicate that they certainly were not, as we observe significantly worse performance compared to the baseline. As a result of these limitations, we could not properly evaluate the potential performance gains of the TwoCircNDRange architecture.

Focus on Specific Quantum Circuits

The circuits evaluated in this thesis, such as QFT, Grover's algorithm, streaming, and D1Q3 circuits, represent a subset of the broad range of quantum algorithms that could be simulated on FPGA-based platforms.

Other quantum algorithms with different gate structures, qubit interconnections, or error-correction requirements were not explored in this work. Expanding the evaluation to include a broader variety of quantum circuits could provide a more comprehensive understanding of the suitability of FPGA architectures for different quantum applications.

Lack of FPGA Cluster Evaluation

While this thesis explores single FPGA architectures for quantum circuit simulation, the use of multi-FPGA clusters was not evaluated due to time and resource constraints. Given the demonstrated energy efficiency of individual FPGAs, a cluster-based approach could further enhance scalability and performance for large-scale quantum simulations. However, evaluating a multi-FPGA setup would require additional infrastructure and synchronisation mechanisms to coordinate the simulation across devices, which was beyond the scope of this research.

7.4 Future Work

Looking ahead, several key areas emerge for further research, building on the foundations laid in this work.

7.4.1 Multi-FPGA Clusters and Distributed Architectures

One of the most promising directions for future work is the development and evaluation of multi-FPGA clusters. While this thesis focused on single FPGA architectures, scaling up to clusters of FPGAs could significantly enhance the scalability and performance of quantum circuit simulations, especially for large-scale circuits. FPGA clusters would leverage parallelism across multiple devices to handle higher qubit counts, while maintaining the energy efficiency demonstrated in this thesis.

The key challenges to consider while implementing such a system is:

- **Inter-device communication**: Efficient data transfer and synchronisation between FPGAs will be critical to minimise overhead.
- **Memory partitioning**: Developing strategies to partition quantum states and gates across multiple FPGAs will be necessary to maximise parallelism. This work could be inspired by existing CPU/GPU-based distributed quantum circuit simulation systems.

Evaluating the energy efficiency and performance-per-Watt gains of multi-FPGA clusters, especially in high-performance computing environments, could further validate the suitability of FPGAs for quantum circuit simulation in power-constrained settings.

7.4.2 Real-Time Power Monitoring

Another area of future work involves improving the accuracy of the energy consumption analysis through the integration of real-time power monitoring tools. This would allow for more precise measurements of dynamic power usage during simulation, providing a clearer understanding of how energy is consumed across different stages of the quantum circuit.

7.4.3 Addressing HLS Limitations

The use of HLS tools in this thesis enabled faster development and testing of FPGA architectures but also introduced limitations, particularly in terms of resource utilisation and clock frequency constraints. Future work could explore **manual optimisation at the register-transfer level**, using custom Verilog/VHDL code, which would allow for finer control over the architecture's resource allocation, timing, and parallelism.

Optimising double-buffered architectures The DoubleBuffered and DoubleBuffered Gate Fusion architectures encountered performance bottlenecks due to clock frequency drops and increased complexity. Manual RTL optimisation could help resolve these issues by fine-tuning the buffering mechanism and improving data handling between fused gate block gates.

Improving synthesis for large buffer sizes HLS tools struggled with efficiently managing large buffer sizes. RTL optimisation could lead to better handling of on-board memory management and allow for scaling up the buffered architectures, while maintaining a performance gain.

Ensuring allocation of state vector buffers to separate banks We were not able to get the HLS tools to ensure that the two state vectors used for the circuits in the evaluation of the TwoCircuitNDRange architecture were allocated to separate memory banks. Future work should look into finding a workaround for this, in order to properly evaluate this architecture and experimentally show the full benefit of the circuit-width reduction technique for the FPGA.

7.4.4 Evaluating Other Quantum Circuits and Applications

The circuits evaluated in this thesis, such as QFT, Grover's algorithm, and streaming circuits, represent only a small subset of potential quantum algorithms. Future research should extend the evaluation to include a broader variety of circuits, particularly those that incorporate:

- **Quantum error correction (QEC)**: Evaluating how FPGAs handle the overhead of error correction codes and fault-tolerant gates, which are crucial for practical quantum computing.
- **Hybrid quantum-classical algorithms**: Investigating the performance of circuits used in hybrid algorithms like Variational Quantum Eigensolver (VQE) or Quantum Approximate Optimisation Algorithm (QAOA), which require feedback between quantum and classical computations.

Additionally, it would be valuable to classify and explore circuits with different gate structures and entanglement patterns to understand how FPGA architectures perform across a more diverse set of quantum workloads. Future work could explore the simulation of emerging quantum applications (combining a variety of the quantum circuits, discussed in this work and otherwise) across different domains, such as quantum chemistry, optimisation, cryptography, and machine learning. By utilising the hardware-agnostic simulators developed in this work, it will be possible to benchmark and optimise these applications at a theoretical level, independent of current hardware constraints. Such explorations could uncover patterns in algorithmic behaviour, resource utilisation, and scalability that inform the design of future quantum architectures.

7.4.5 Improved Memory Management and utilising HBM

Future work could also focus on developing advanced memory management strategies to overcome the limitations encountered in this thesis, particularly for the TwoCircuitNDRange architecture. Potential strategies for this include:

- **Memory partitioning**: Dividing the quantum state vector across multiple memory banks in a High-Bandwidth Memory (HBM) system should significantly improve parallel access.
- **Improving controls-processing for buffered architectures**: Finding an equivalent for the controls scheduling optimisation for buffered architectures would be desirable. This could build on the cases and examples demonstrated in Section 3.4.1.

7.4.6 Improving Gate Fusion and Further Evaluation

For Gate Fusion architectures, this work explored how the performance varies across different numbers of compute units (determined by the BQS parameter). However, these architectures are also parameterised by the Gate Block Gate Count (GBGC) parameter, which determines the maximum number of gates which can be fused into a gate block based on the size of the on-board buffer. We did not evaluate how performance varies for different values of GBGC. Such an evaluation would help to determine more precisely the full potential of this optimisation on the FPGA.

In addition, gate fusion for case buffering (described in Section 3.5.2) was not implemented during this work. This method extends gate fusion to gates whose target qubit index, t, is greater than the buffer qubit size, l, as long as all fused gates share the same target qubit. Implementing this extension can enable the gate fusion technique to see even better improvement in performance across circuits. In particular, we expect the QFT circuit, which has long sequences of gates acting on the same qubit, to see a significant benefit from this optimisation.

7.5 Closing Remarks

This thesis has explored the frontier of FPGA-based quantum circuit simulation, contributing novel architectures and optimisation techniques for Full State Vector Quantum Circuit Simulation. Through the implementation and evaluation of these architectures, we have demonstrated that while FPGAs present significant challenges, they also offer promising advantages, particularly in energy efficiency for specific circuit types, such as control-heavy quantum circuits.

The research presented here has expanded our understanding of how FPGA technology can be used to perform large-scale quantum circuit simulations, an important step in quantum algorithm development and testing. The gate fusion and control scheduling optimisations have proven to be effective in narrowing the performance gap between FPGAs and more traditional CPU/GPU platforms, while also emphasising the energy efficiency that makes FPGAs an appealing option for future quantum circuit simulation infrastructure.

Despite the advancements made, several limitations and challenges have been identified. However, these also provide clear directions for future work. The potential of multi-FPGA clusters, improved memory management, and better utilisation of High-Level Synthesis techniques offer opportunities for continuing research. In particular, addressing these challenges could enable FPGAs to not only compete with but surpass other platforms in both raw performance and performance-per-Watt.

In conclusion, this thesis has laid the groundwork for a future where FPGAs are not only viable but potentially superior platforms for large-scale quantum simulations. As the field of quantum computing continues to evolve, so too must the tools used to simulate and optimise quantum circuits. The architectures and techniques presented here represent a step towards that future, providing valuable insights into the scalability, performance, and energy efficiency of FPGA-based quantum circuit simulations.

Appendix A

Biased Quantum Floating Point Representation

In the current implementation of the D1Q3 model, the idea is to represent the (scaled) distribution function values $g_i, i \in [0,3]$ and u/c with a specialised (biased) quantum floating-point format. The motivation for choosing this format is the wider range of numbers that can be represented than in a fixed-point representation for the same number of qubits used. To minimise quantum-circuit width, a reduced number of mantissa and exponent bits is used as compared to IEEE-754 single-precision format. However, for the considered problems a scaling was used so that the ranges of numbers to be represented is both limited and predictable. For the D1Q3 model, where the numbers will be $\ll 1$, this choice of parameters is further detailed later this section.

The quantum floating-point representation used here builds on earlier work by Steijl[131] and involves reduced-bit representations of exponent and mantissa relative to single-precision in the IEEE-754 standard to facilitate quantum circuit implementations on current and near-future quantum hardware with relatively small number of qubits (< 100). A key feature of the used quantum floating-point representation is that following the IEEE-754 standard, it employs sub-normal numbers and consistent rounding (here, rounding-down to nearest). The number of mantissa qubits is defined by N_M, where only $N_M - 1$ mantissa qubits are stored following the 'hidden-qubit' approach from IEEE-754. Then, the number of qubits storing the exponent is defined by N_E. In the present work, $N_E = 3$, and exponent 0 (exponent qubits in state $|000\rangle$) represent sub-normal numbers and zero as in the IEEE-754 standard. The maximum value for exponent is 7 (exponent qubits in state $|111\rangle$) refers to 'overflow' conditions, as used in the IEEE-754 standard. For $N_E = 3$, an 'unbiased' exponent formulation would be equivalent to an exponent bias of 3. To optimise for the small numbers occurring in considered problems, a bias toward smaller numbers is used here. Clearly, the choice of N_M is crucial in achieving the required accuracy. For equilibrium distribution

functions, terms linear and quadratic in u/c are combined, and $N_M = 3$ was found to lead to excessive rounding or truncation for most values of u/c. Clearly, $N_M \geq 4$ should be used. However, since the lattice-based models considered here represent conservation of mass, momentum and energy in the fluid, rounding of numbers will have a significant effect on accuracy, particularly in multiple time-step simulations. Therefore, realistically it can be expected that $N_M \in [8, 16]$ is needed for realistic engineering applications. In this work, circuits for $N_M = 4$ are shown as illustration, and the complexity analysis shown in Section 5.8 addresses in detail how the circuit width increases with N_M.

Following IEEE-754, signed floating-point numbers are stored as '$|\text{sign}|\text{exponent}|\text{mantissa}\rangle$', while the sign qubit is omitted where unsigned numbers are used in the quantum-circuit implementation. For 'signed' additions or subtractions of two numbers, two $(N_M + 1)$-qubit registers as input to a modulo adder are created as follows. First, the hidden qubits are added to the mantissa, followed by re-normalization (to account for possible difference in exponent) to create two $(N_M + 1)$-qubit inputs with $|0\rangle$ as most-significant qubit. Where required a conversion to 2's complement is performed, so that negative numbers have the most-significant qubit in state $|1\rangle$. After addition/subtraction, the outcome is converted back to the quantum floating-point format, where a sign-qubit defines the sign and with mantissa no longer in 2's complement, i.e. back into $(N_M - 1)$-qubit 'hidden-qubit' representation.

For $N_M = 4$ and $N_E = 3$, the sub-normal numbers with the corresponding 'negative' $(N_M + 1)$-qubit mantissa representation using 2's complement method are shown in Table A.1 for 'bias=3' and 'bias=8'. Here, 'bias=3' corresponds to the 'standard' floating-point format with a symmetric bias. As can be seen from Table A.1, with symmetric bias ('bias=3'), terms involving u^2 ($O(10^{-2})$ will always be sub-normal and often truncated to 0. For 'bias=8', it can expected that u^2 can be represented either as non-zero sub-normal or normalised numbers.

For $N_M = 4$ and $N_E = 3$, the normalised numbers for $|e2|e1|e0\rangle = |110\rangle$ with the corresponding 'negative' using 2's complement method are shown in Table A.2 for 'bias=3' and 'bias=8'.

For the maximum exponent ($|e2|e1|e0\rangle = |110\rangle$), the values for 'bias=8' shown in Table A.2 indicate that despite the greatly reduced value of the maximum number that can be represented relative to the symmetric bias case ('bias=3'), the components of distribution function $\vec{g}$ and velocity u/c will be so small that these can still be represented without risking an overflow, as discussed next.

The choice of suitable values for N_E and exponent bias for the scaled and normalised D1Q3 model can be made based on the flow physics that is modelled. For the D1Q3 model, the lattice speed of sound is defined as $c_s = c/\sqrt{3}$. The iso-thermal model used here was derived for flows with weak or negligible compressibility effects. For a compressible fluid, a local

Table A.1: Sub-normal numbers for $N_M = 4$ and $N_E = 3$. Leading qubit acts as 'sign' qubit. In 2's complement 'hidden qubit' is represented.

bias = 3				
positive		negative		mantissa 2's complement
$\|0\|000\|000\rangle$	0	$\|0\|000\|000\rangle$	0	$\|00000\rangle$
$\|0\|000\|001\rangle$	$1/32$	$\|1\|000\|001\rangle$	$-1/32$	$\|11111\rangle$
$\|0\|000\|010\rangle$	$2/32$	$\|1\|000\|010\rangle$	$-2/32$	$\|11110\rangle$
$\|0\|000\|011\rangle$	$3/32$	$\|1\|000\|011\rangle$	$-3/32$	$\|11101\rangle$
$\|0\|000\|100\rangle$	$4/32$	$\|1\|000\|100\rangle$	$-4/32$	$\|11100\rangle$
$\|0\|000\|101\rangle$	$5/32$	$\|1\|000\|101\rangle$	$-5/32$	$\|11011\rangle$
$\|0\|000\|110\rangle$	$6/32$	$\|1\|000\|110\rangle$	$-6/32$	$\|11010\rangle$
$\|0\|000\|111\rangle$	$7/32$	$\|1\|000\|111\rangle$	$-7/32$	$\|11001\rangle$
bias = 8				
positive		negative		mantissa 2's complement
$\|0\|000\|000\rangle$	0	$\|0\|000\|000\rangle$	0	$\|00000\rangle$
$\|0\|000\|001\rangle$	$1/1024$	$\|1\|000\|001\rangle$	$-1/1024$	$\|11111\rangle$
$\|0\|000\|010\rangle$	$2/1024$	$\|1\|000\|010\rangle$	$-2/1024$	$\|11110\rangle$
$\|0\|000\|011\rangle$	$3/1024$	$\|1\|000\|011\rangle$	$-3/1024$	$\|11101\rangle$
$\|0\|000\|100\rangle$	$4/1024$	$\|1\|000\|100\rangle$	$-4/1024$	$\|11100\rangle$
$\|0\|000\|101\rangle$	$5/1024$	$\|1\|000\|101\rangle$	$-5/1024$	$\|11011\rangle$
$\|0\|000\|110\rangle$	$6/1024$	$\|1\|000\|110\rangle$	$-6/1024$	$\|11010\rangle$
$\|0\|000\|111\rangle$	$7/1024$	$\|1\|000\|111\rangle$	$-7/1024$	$\|11001\rangle$

Mach number can be defined as: $M = |u|/c_s = \sqrt{3}|u|/c$. For the weakly compressible-flow conditions it is required that $M < 0.3$ or smaller. Therefore, u/c should generally be limited to maximum values of $0.1 - 0.15$. Clearly, in the modified and re-scaled D1Q3 model employing a symmetric bias will introduce a range of numbers far from optimal for the D1Q3 model. As shown in Table A.2, for $N_E = 3$, a bias of 8 gives ample margin at the upper end of the floating-point range when representing u/c. However, since the (scaled) and re-normalised equilibrium distribution functions combine terms $O(u)$ and $O(u^2)$ it was decided that a bias of 9 could potentially leave too little margin in the floating-point representation of g_i components.

Table A.2: Example normalised numbers for $N_M = 4$ and $N_E = 3$. Leading qubit acts as 'sign' qubit. In 2's complement 'hidden qubit' is represented.

bias=3				
positive		negative		mantissa 2's complement
$\|0\|110\|000\rangle$	8	$\|1\|110\|000\rangle$	-8	$\|11000\rangle$
$\|0\|110\|001\rangle$	9	$\|1\|110\|001\rangle$	-9	$\|10111\rangle$
$\|0\|110\|010\rangle$	10	$\|1\|110\|010\rangle$	-10	$\|10110\rangle$
$\|0\|110\|011\rangle$	11	$\|1\|110\|011\rangle$	-11	$\|10101\rangle$
$\|0\|110\|100\rangle$	12	$\|1\|110\|100\rangle$	-12	$\|10100\rangle$
$\|0\|110\|101\rangle$	13	$\|1\|110\|101\rangle$	-13	$\|10011\rangle$
$\|0\|110\|110\rangle$	14	$\|1\|110\|110\rangle$	-14	$\|10010\rangle$
$\|0\|110\|111\rangle$	15	$\|1\|110\|111\rangle$	-15	$\|10001\rangle$
bias=8				
positive		negative		mantissa 2's complement
$\|0\|110\|000\rangle$	$8/32$	$\|1\|110\|000\rangle$	$-8/32$	$\|11000\rangle$
$\|0\|110\|001\rangle$	$9/32$	$\|1\|110\|001\rangle$	$-9/32$	$\|10111\rangle$
$\|0\|110\|010\rangle$	$10/32$	$\|1\|110\|010\rangle$	$-10/32$	$\|10110\rangle$
$\|0\|110\|011\rangle$	$11/32$	$\|1\|110\|011\rangle$	$-11/32$	$\|10101\rangle$
$\|0\|110\|100\rangle$	$12/32$	$\|1\|110\|100\rangle$	$-12/32$	$\|10100\rangle$
$\|0\|110\|101\rangle$	$13/32$	$\|1\|110\|101\rangle$	$-13/32$	$\|10011\rangle$
$\|0\|110\|110\rangle$	$14/32$	$\|1\|110\|110\rangle$	$-14/32$	$\|10010\rangle$
$\|0\|110\|111\rangle$	$15/32$	$\|1\|110\|111\rangle$	$-15/32$	$\|10001\rangle$

Bibliography

[1] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, 10th ed. Cambridge ; New York: Cambridge University Press, 2010.

[2] S. A. Cuccaro, T. G. Draper, S. A. Kutin, and D. P. Moulton, "A new quantum ripple-carry addition circuit," *arXiv:quant-ph/0410184*, Oct. 2004, arXiv: quant-ph/0410184. [Online]. Available: http://arxiv.org/abs/quant-ph/0410184

[3] G. F. Viamontes, "Efficient Quantum Circuit Simulation," p. 230, 2007. [Online]. Available: http://web.eecs.umich.edu/~imarkov/pubs/diss/GFVdiss.pdf

[4] "The OpenCL™ specification - khronos registry," Apr 2024, accessed: 01/10/2024. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

[5] "Intel® FPGA SDK for OpenCL™ programming guide," 2017, accessed: 01/10/2024. [Online]. Available: https://cdrdv2-public.intel.com/704864/aocl_programming_guide-17-1-683846-704864.pdf

[6] T. G. Draper, "Addition on a quantum computer," 2000. [Online]. Available: https://arxiv.org/abs/quant-ph/0008033

[7] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, no. 6, pp. 467–488, Jun. 1982. [Online]. Available: https://doi.org/10.1007/BF02650179

[8] L. Gyongyosi and S. Imre, "A Survey on quantum computing technology," *Computer Science Review*, vol. 31, pp. 51–71, Feb. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1574013718301709

[9] M. M. Savchuk and A. V. Fesenko, "Quantum Computing: Survey and Analysis," *Cybernetics and Systems Analysis*, vol. 55, no. 1, pp. 10–21, Jan. 2019. [Online]. Available: http://link.springer.com/10.1007/s10559-019-00107-w

[10] M. M. Waldrop, "The chips are down for Moore's law," *Nature News*, vol. 530, no. 7589, p. 144, Feb. 2016. [Online]. Available: http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338

[11] D. J. Griffiths, *Introduction to Quantum Mechanics (2nd Edition)*, 2nd ed. Pearson Prentice Hall, Apr. 2004.

[12] M. Nauenberg, "Max Planck and the birth of the quantum hypothesis," *American Journal of Physics*, vol. 84, no. 9, pp. 709–720, Sep. 2016. [Online]. Available: https://pubs.aip.org/ajp/article/84/9/709/1057853/Max-Planck-and-the-birth-of-the-quantum-hypothesis

[13] A. B. Arons and M. B. Peppard, "Einstein's Proposal of the Photon Concept—a Translation of the Annalen der Physik Paper of 1905," *American Journal of Physics*, vol. 33, no. 5, pp. 367–374, May 1965, _eprint: https://pubs.aip.org/aapt/ajp/article-pdf/33/5/367/12040128/367_1_online.pdf. [Online]. Available: https://doi.org/10.1119/1.1971542

[14] W. A. Fedak and J. J. Prentis, "The 1925 Born and Jordan paper "On quantum mechanics"," *American Journal of Physics*, vol. 77, no. 2, pp. 128–139, Feb. 2009. [Online]. Available: https://doi.org/10.1119/1.3009634

[15] P. A. M. Dirac, "A new notation for quantum mechanics," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, no. 3, p. 416–418, 1939.

[16] "IBM Quantum Development Innovation Roadmap," 2024, accessed: 01/10/2024. [Online]. Available: https://www.ibm.com/quantum/assets/IBM_Quantum_Development_&_Innovation_Roadmap.pdf

[17] J. Koch, T. M. Yu, J. Gambetta, A. A. Houck, D. I. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, "Charge-insensitive qubit design derived from the cooper pair box," *Phys. Rev. A*, vol. 76, p. 042319, Oct 2007. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.76.042319

[18] J. A. Schreier, A. A. Houck, J. Koch, D. I. Schuster, B. R. Johnson, J. M. Chow, J. M. Gambetta, J. Majer, L. Frunzio, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf, "Suppressing charge noise decoherence in superconducting charge qubits," *Phys. Rev. B*, vol. 77, p. 180502, May 2008. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.77.180502

[19] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney,

M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5

[20] J. I. Cirac and P. Zoller, "Quantum computations with cold trapped ions," *Phys. Rev. Lett.*, vol. 74, pp. 4091–4094, May 1995. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.74.4091

[21] S. A. Moses, C. H. Baldwin, M. S. Allman, R. Ancona, L. Ascarrunz, C. Barnes, J. Bartolotta, B. Bjork, P. Blanchard, M. Bohn, J. G. Bohnet, N. C. Brown, N. Q. Burdick, W. C. Burton, S. L. Campbell, J. P. Campora, C. Carron, J. Chambers, J. W. Chan, Y. H. Chen, A. Chernoguzov, E. Chertkov, J. Colina, J. P. Curtis, R. Daniel, M. DeCross, D. Deen, C. Delaney, J. M. Dreiling, C. T. Ertsgaard, J. Esposito, B. Estey, M. Fabrikant, C. Figgatt, C. Foltz, M. Foss-Feig, D. Francois, J. P. Gaebler, T. M. Gatterman, C. N. Gilbreth, J. Giles, E. Glynn, A. Hall, A. M. Hankin, A. Hansen, D. Hayes, B. Higashi, I. M. Hoffman, B. Horning, J. J. Hout, R. Jacobs, J. Johansen, L. Jones, J. Karcz, T. Klein, P. Lauria, P. Lee, D. Liefer, S. T. Lu, D. Lucchetti, C. Lytle, A. Malm, M. Matheny, B. Mathewson, K. Mayer, D. B. Miller, M. Mills, B. Neyenhuis, L. Nugent, S. Olson, J. Parks, G. N. Price, Z. Price, M. Pugh, A. Ransford, A. P. Reed, C. Roman, M. Rowe, C. Ryan-Anderson, S. Sanders, J. Sedlacek, P. Shevchuk, P. Siegfried, T. Skripka, B. Spaun, R. T. Sprenkle, R. P. Stutz, M. Swallows, R. I. Tobey, A. Tran, T. Tran, E. Vogt, C. Volin, J. Walker, A. M. Zolot, and J. M. Pino, "A race-track trapped-ion quantum processor," *Phys. Rev. X*, vol. 13, p. 041052, Dec 2023. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.13.041052

[22] E. Gibney, "Inside Microsoft's quest for a topological quantum computer," *Nature*, Oct. 2016. [Online]. Available: https://doi.org/10.1038/nature.2016.20774

[23] D. Castelvecchi, "Physicists find best evidence yet for elusive 2d structures," 2020. [Online]. Available: https://www.nature.com/articles/d41586-020-01988-0

[24] B. Yirka, “Microsoft claims to have achieved first milestone in creating a reliable and practical quantum computer,” 2023. [Online]. Available: https://phys.org/news/2023-06-microsoft-milestone-reliable-quantum.html#google_vignette

[25] A. Imamoglu, “Are quantum dots useful for quantum computation?” *Physica E: Low-dimensional Systems and Nanostructures*, vol. 16, no. 1, pp. 47–50, 2003, proceedings of the Twelfth International Winterschool on New Developments in Solids State Physics, ”Low- Dimensional Systems: From 2D to Molecules”. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1386947702005817

[26] C. Adami and N. J. Cerf, “Quantum computation with linear optics,” in *Quantum Computing and Quantum Communications*, C. P. Williams, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 391–401.

[27] E. Knill, R. LaØamme, and G. J. Milburn, “A scheme for efficient quantum computation with linear optics,” 2001.

[28] P. Kok, W. J. Munro, K. Nemoto, T. C. Ralph, J. P. Dowling, and G. J. Milburn, “Linear optical quantum computing with photonic qubits,” *Rev. Mod. Phys.*, vol. 79, pp. 135–174, Jan 2007. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.79.135

[29] D. Bluvstein, S. J. Evered, A. A. Geim, S. H. Li, H. Zhou, T. Manovitz, S. Ebadi, M. Cain, M. Kalinowski, D. Hangleiter, J. P. Bonilla Ataides, N. Maskara, I. Cong, X. Gao, P. Sales Rodriguez, T. Karolyshyn, G. Semeghini, M. J. Gullans, M. Greiner, V. Vuletić, and M. D. Lukin, “Logical quantum processor based on reconfigurable atom arrays,” *Nature*, vol. 626, no. 7997, pp. 58–65, Feb. 2024. [Online]. Available: https://doi.org/10.1038/s41586-023-06927-3

[30] J. Preskill, “Quantum Computing in the NISQ era and beyond,” Jul. 2018, arXiv:1801.00862 [cond-mat, physics:quant-ph]. [Online]. Available: http://arxiv.org/abs/1801.00862

[31] P. W. Shor, “Scheme for reducing decoherence in quantum computer memory,” *Phys. Rev. A*, vol. 52, pp. R2493–R2496, Oct 1995. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.52.R2493

[32] A. M. Steane, “Error Correcting Codes in Quantum Theory,” *Physical Review Letters*, vol. 77, no. 5, pp. 793–797, Jul. 1996. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.77.793

[33] A. Steane, “Quantum Computing,” *Reports on Progress in Physics*, vol. 61, no. 2, pp. 117–173, Feb. 1998, arXiv: quant-ph/9708022. [Online]. Available: http://arxiv.org/abs/quant-ph/9708022

[34] ——, “Enlargement of Calderbank-Shor-Steane quantum codes,” *IEEE Transactions on Information Theory*, vol. 45, no. 7, pp. 2492–2495, Nov. 1999. [Online]. Available: http://ieeexplore.ieee.org/document/796388/

[35] S. J. Devitt, K. Nemoto, and W. J. Munro, “Quantum Error Correction for Beginners,” *Reports on Progress in Physics*, vol. 76, no. 7, p. 076001, Jul. 2013, arXiv: 0905.2794. [Online]. Available: http://arxiv.org/abs/0905.2794

[36] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, “Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels,” *Phys. Rev. Lett.*, vol. 70, pp. 1895–1899, Mar 1993. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.70.1895

[37] D. Aharonov, W. van Dam, J. Kempe, Z. Landau, S. Lloyd, and O. Regev, “Adiabatic quantum computation is equivalent to standard quantum computation,” *SIAM Review*, vol. 50, no. 4, pp. 755–787, 2008. [Online]. Available: https://doi.org/10.1137/080734479

[38] R. Raussendorf, D. E. Browne, and H. J. Briegel, “Measurement-based quantum computation on cluster states,” *Phys. Rev. A*, vol. 68, p. 022312, Aug 2003. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.68.022312

[39] D. Deutsch, “Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 400, no. 1818, pp. 97–117, Jul. 1985. [Online]. Available: http://rspa.royalsocietypublishing.org/cgi/doi/10.1098/rspa.1985.0070

[40] M. H. Freedman, M. Larsen, and Z. Wang, “A Modular Functor Which is Universal for Quantum Computation,” *Communications in Mathematical Physics*, vol. 227, no. 3, pp. 605–622, Jun. 2002. [Online]. Available: https://doi.org/10.1007/s002200200645

[41] A. Chi-Chih Yao, “Quantum circuit complexity,” in *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, Nov. 1993, pp. 352–361.

[42] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, “Elementary gates for quantum computation,” *Phys. Rev. A*, vol. 52, pp. 3457–3467, Nov 1995. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevA.52.3457

[43] W. K. Wootters and W. H. Zurek, “A single quantum cannot be cloned,” *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982, number: 5886 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/299802a0

[44] L. Ruiz-Perez and J. Garcia-Escartin, “Quantum arithmetic with the quantum fourier transform,” *Quantum Information Processing*, vol. 16, p. 152, 2017. [Online]. Available: https://doi.org/10.1007/s11128-017-1603-1

[45] P. W. Shor, “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer,” *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, arXiv: quant-ph/9508027. [Online]. Available: http://arxiv.org/abs/quant-ph/9508027

[46] L. K. Grover, “A fast quantum mechanical algorithm for database search,” *arXiv:quant-ph/9605043*, May 1996, arXiv: quant-ph/9605043. [Online]. Available: http://arxiv.org/abs/quant-ph/9605043

[47] D. Deutsch and R. Jozsa, “Rapid Solution of Problems by Quantum Computation,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 439, no. 1907, pp. 553–558, Dec. 1992. [Online]. Available: http://rspa.royalsocietypublishing.org/cgi/doi/10.1098/rspa.1992.0167

[48] S. McArdle, S. Endo, A. Aspuru-Guzik, S. Benjamin, and X. Yuan, “Quantum computational chemistry,” *Reviews of Modern Physics*, vol. 92, no. 1, p. 015003, Mar. 2020, arXiv: 1808.10402. [Online]. Available: http://arxiv.org/abs/1808.10402

[49] Y. Cao, J. Romero, J. P. Olson, M. Degroote, P. D. Johnson, M. Kieferová, I. D. Kivlichan, T. Menke, B. Peropadre, N. P. D. Sawaya, S. Sim, L. Veis, and A. Aspuru-Guzik, “Quantum Chemistry in the Age of Quantum Computing,” *Chemical Reviews*, vol. 119, no. 19, pp. 10 856–10 915, Oct. 2019. [Online]. Available: https://pubs.acs.org/doi/10.1021/acs.chemrev.8b00803

[50] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, “Quantum machine learning,” *Nature*, vol. 549, no. 7671, pp. 195–202, Sep. 2017. [Online]. Available: http://www.nature.com/articles/nature23474

[51] S. Lloyd, M. Mohseni, and P. Rebentrost, “Quantum algorithms for supervised and unsupervised machine learning,” *arXiv:1307.0411 [quant-ph]*, Nov. 2013, arXiv: 1307.0411. [Online]. Available: http://arxiv.org/abs/1307.0411

[52] R. Steijl and G. N. Barakos, “Parallel evaluation of quantum algorithms for computational fluid dynamics,” *Computers & Fluids*, vol. 173, pp. 22–28, Sep. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0045793018301841

[53] B. N. Todorova and R. Steijl, "Quantum algorithm for the collisionless Boltzmann equation," *Journal of Computational Physics*, vol. 409, p. 109347, May 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0021999120301212

[54] D. Coppersmith, "An approximate Fourier transform useful in quantum factoring," Jun. 1994, arXiv:quant-ph/0201067. [Online]. Available: http://arxiv.org/abs/quant-ph/0201067

[55] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[56] Y. Moawad, W. Vanderbauwhede, and R. Steijl, "Investigating hardware acceleration for simulation of CFD quantum circuits," *Frontiers in Mechanical Engineering*, vol. 8, p. 925637, Oct. 2022. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fmech.2022.925637/full

[57] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "QuEST and High Performance Simulation of Quantum Computers," *Scientific Reports*, vol. 9, no. 1, p. 10736, Dec. 2019. [Online]. Available: http://www.nature.com/articles/s41598-019-47174-9

[58] G. G. Guerreschi, J. Hogaboam, F. Baruffa, and N. P. D. Sawaya, "Intel Quantum Simulator: A cloud-ready high-performance simulator of quantum circuits," *Quantum Science and Technology*, vol. 5, no. 3, p. 034007, May 2020, arXiv: 2001.10554. [Online]. Available: http://arxiv.org/abs/2001.10554

[59] R. P. Feynman, "Space-time approach to non-relativistic quantum mechanics," *Rev. Mod. Phys.*, vol. 20, pp. 367–387, Apr 1948. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.20.367

[60] R. Feynman and L. Brown, *Feynman's Thesis: A New Approach to Quantum Theory*, ser. G - Reference,Information and Interdisciplinary Subjects Series. World Scientific, 2005. [Online]. Available: https://books.google.co.uk/books?id=5kowI7YgFbEC

[61] J. Chen, F. Zhang, C. Huang, M. Newman, and Y. Shi, "Classical Simulation of Intermediate-Size Quantum Circuits," May 2018, arXiv:1805.01450 [quant-ph]. [Online]. Available: http://arxiv.org/abs/1805.01450

[62] D. A. C. Ferreira, "Feynman Path-Sum Quantum Computer Simulator," 2023.

[63] J. Biamonte and V. Bergholm, "Tensor networks in a nutshell," 2017. [Online]. Available: https://arxiv.org/abs/1708.00006

[64] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," Nov. 2005. [Online]. Available: https://arxiv.org/abs/quant-ph/0511069v7

[65] D. Aharonov, Z. Landau, and J. Makowsky, "The quantum FFT can be classically simulated," *arXiv:quant-ph/0611156*, Mar. 2007, arXiv: quant-ph/0611156. [Online]. Available: http://arxiv.org/abs/quant-ph/0611156

[66] G. Viamontes, I. Markov, and J. Hayes, "High-performance QuIDD-based simulation of quantum circuits," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. Paris, France: IEEE Comput. Soc, 2004, pp. 1354–1355. [Online]. Available: http://ieeexplore.ieee.org/document/1269084/

[67] D. Rosenbaum, "Binary superposed quantum decision diagrams," *Quantum Information Processing*, vol. 9, no. 4, pp. 463–496, Aug. 2010. [Online]. Available: http://link.springer.com/10.1007/s11128-009-0153-6

[68] C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1959.tb01585.x. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1959.tb01585.x

[69] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. Santa Clara, CA, USA: IEEE Comput. Soc. Press, 1993, pp. 188–191. [Online]. Available: http://ieeexplore.ieee.org/document/580054/

[70] M. Fujita and P. C. Mcgeer, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation," *Formal Methods in System Design*, p. 21, 1997.

[71] M. Boyer, G. Brassard, P. Hoeyer, and A. Tapp, "Tight bounds on quantum searching," *arXiv:quant-ph/9605034*, May 1996, arXiv: quant-ph/9605034. [Online]. Available: http://arxiv.org/abs/quant-ph/9605034

[72] P. Niemann, R. Wille, and R. Drechsler, "On the "Q" in QMDDs: Efficient Representation of Quantum Functionality in the QMDD Data-Structure," in *Reversible Computation*, ser. Lecture Notes in Computer Science, G. W. Dueck and D. M. Miller, Eds. Berlin, Heidelberg: Springer, 2013, pp. 125–140. [Online]. Available: http://www.informatik.uni-bremen.de/agra/doc/konf/13_rc_eff_qmdd_representation.pdf

[73] P. Niemann, R. Wille, D. M. Miller, and R. Drechsler, "QMDDs: Efficient Quantum Function Representation and Manipulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 1, pp. 86–99, Jan. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7163590/

[74] A. Zulehner and R. Wille, "Advanced Simulation of Quantum Computations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 848–859, May 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8355954/

[75] A. Zulehner, S. Hillmich, and R. Wille, "How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Westminster, CO, USA: IEEE, Nov. 2019, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/8942057/

[76] D. Michael, M. Mitchell, and A. Thornton, *QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits*, 2006.

[77] D. Goodman, M. A. Thornton, D. Y. Feinstein, and D. M. Miller, "Quantum Logic Circuit Simulation Based on the QMDD Data Structure," p. 7, 2007.

[78] D. M. Miller, D. Y. Feinstein, and M. A. Thornton, "QMDD Minimization using Sifting for Variable Reordering," p. 18, 2007.

[79] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito, "Massively parallel quantum computer simulator," *Computer Physics Communications*, vol. 176, no. 2, pp. 121–136, Jan. 2007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0010465506003390

[80] F. Tabakin and B. Juliá-Díaz, "QCMPI: A parallel environment for quantum computing," *Computer Physics Communications*, vol. 180, no. 6, pp. 948–964, Jun. 2009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0010465508004141

[81] H. De Raedt, A. H. Hams, K. Michielsen, and K. De Raedt, "Quantum Computer Emulator," *Computer Physics Communications*, vol. 132, no. 1-2, pp. 1–20, Oct. 2000. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0010465500001326

[82] Z. Wang, Z. Chen, S. Wang, W. Li, Y. Gu, G. Guo, and Z. Wei, "A quantum circuit simulator and its applications on Sunway TaihuLight supercomputer,"

Scientific Reports, vol. 11, no. 1, p. 355, Dec. 2021. [Online]. Available: http://www.nature.com/articles/s41598-020-79777-y

[83] Z.-Y. Chen, Q. Zhou, C. Xue, X. Yang, G.-C. Guo, and G.-P. Guo, “64-qubit quantum circuit simulation,” *Science Bulletin*, vol. 63, no. 15, pp. 964–971, Aug. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2095927318302809

[84] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik, “qHiPSTER: The Quantum High Performance Software Testing Environment,” *arXiv:1601.07195 [quant-ph]*, May 2016, arXiv: 1601.07195. [Online]. Available: http://arxiv.org/abs/1601.07195

[85] D. B. Trieu, *Large-scale simulations of error-prone quantum computation devices*, ser. Schriften des Forschungszentrums Jülich IAS Series. Jülich: Forschungszentrum Jülich, 2010, no. 2, oCLC: 839432524.

[86] N. Khammassi, I. Ashraf, X. Fu, C. Almudever, and K. Bertels, “QX: A high-performance quantum computer simulation platform,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 464–469.

[87] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, “Open quantum assembly language,” 2017. [Online]. Available: https://arxiv.org/abs/1707.03429

[88] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heidel, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, and B. R. Johnson, “OpenQASM 3: A Broader and Deeper Quantum Assembly Language,” *ACM Transactions on Quantum Computing*, vol. 3, no. 3, p. 1–50, Sep. 2022. [Online]. Available: http://dx.doi.org/10.1145/3505636

[89] D. Wecker and K. M. Svore, “Liqui—¿: A software design architecture and domain-specific language for quantum computing,” 2014. [Online]. Available: https://arxiv.org/abs/1402.4467

[90] A. Kelly, “Simulating Quantum Computers Using OpenCL,” *arXiv:1805.00988 [quant-ph]*, May 2018, arXiv: 1805.00988. [Online]. Available: http://arxiv.org/abs/1805.00988

[91] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, “Quantum computing with Qiskit,” 2024.

[92] D. S. Steiger, T. Häner, and M. Troyer, “ProjectQ: An Open Source Software Framework for Quantum Computing,” *Quantum*, vol. 2, p. 49, Jan. 2018, arXiv: 1612.08091. [Online]. Available: http://arxiv.org/abs/1612.08091

[93] W. Vanderbauwhede and K. Benkrid, Eds., *High-Performance Computing Using FPGAs*. New York, NY: Springer New York, 2013. [Online]. Available: https://link.springer.com/10.1007/978-1-4614-1791-0

[94] H. R. Zohouri, "High Performance Computing with FPGAs and OpenCL," p. 130, 2018.

[95] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli, "FPGA-accelerated Information Retrieval: High-efficiency document filtering," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 417–422.

[96] C. Belady, "In the data center, power and cooling costs more than the it equipment it supports," *Electron Cool*, vol. 13, 01 2007.

[97] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, May 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7106407/

[98] O. Melikoglu, O. Ergin, B. Salami, J. Pavon, O. Unsal, and A. Cristal, "A Novel FPGA-Based High Throughput Accelerator For Binary Search Trees," in *2019 International Conference on High Performance Computing and Simulation (HPCS)*. Dublin, Ireland: IEEE, Jul. 2019, pp. 612–619. [Online]. Available: https://ieeexplore.ieee.org/document/9188158/

[99] M. C. Smith, J. S. Vetter, and S. R. Alam, "Scientific computing beyond CPUs: FPGA implementations of common scientific kernels," 2005.

[100] J. Gonzalez and R. C. Núñez, "Lapackrc: Fast linear algebra kernels/solvers for fpga accelerators," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012042, jul 2009. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/180/1/012042

[101] R. Dorrance, F. Ren, and D. Marković, "A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs." New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2554688.2554785

[102] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE Computer Society Annual Symposium on VLSI*, 2010, pp. 288–293.

[103] J. Davis, C. Thacker, and C. Chang, “Bee3: Revitalizing computer architecture research,” Tech. Rep. MSR-TR-2009-45, April 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/bee3-revitalizing-computer-architecture-research/

[104] T. De Matteis, J. d. F. Licht, and T. Hoefler, “FBLAS: Streaming Linear Algebra on FPGA,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–13, arXiv:1907.07929 [cs]. [Online]. Available: http://arxiv.org/abs/1907.07929

[105] A. Khalid, Z. Zilic, and K. Radecka, “FPGA emulation of quantum circuits,” in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* San Jose, CA, USA: IEEE, 2004, pp. 310–315. [Online]. Available: http://ieeexplore.ieee.org/document/1347938/

[106] M. Aminian, M. Saeedi, M. S. Zamani, and M. Sedighi, “FPGA-Based Circuit Model Emulation of Quantum Algorithms,” in *2008 IEEE Computer Society Annual Symposium on VLSI.* Montpellier, France: IEEE, 2008, pp. 399–404. [Online]. Available: http://ieeexplore.ieee.org/document/4556828/

[107] J. Pilch and J. Dlugopolski, “An FPGA-based real quantum computer emulator,” *Journal of Computational Electronics*, vol. 18, pp. 329–342, 2019.

[108] M. P. Frank, L. Oniciuc, U. H. Meyer-Baese, and I. Chiorescu, “A space-efficient quantum computer simulator suitable for high-speed FPGA implementation,” E. J. Donkor, A. R. Pirich, and H. E. Brandt, Eds., Orlando, Florida, USA, May 2009, p. 734203. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.817924

[109] E. Bernstein and U. Vazirani, “Quantum complexity theory,” *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1411–1473, 1997. [Online]. Available: https://doi.org/10.1137/S0097539796300921

[110] C. Conceicao and R. Reis, “Efficient emulation of quantum circuits on classical hardware,” in *2015 IEEE 6th Latin American Symposium on Circuits and Systems (LASCAS).* Montevideo, Uruguay: IEEE, Feb. 2015, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/7250404/

[111] Y. Lee, M. Khalil-Hani, and M. Marsono, “An FPGA-based quantum computing emulation framework based on serial-parallel architecture,” *International Journal of Reconfigurable Computing*, vol. 2016, p. 18, 2016.

[112] N. Mahmud and E. El-Araby, "A Scalable High-Precision and High-Throughput Architecture for Emulation of Quantum Algorithms," in *2018 31st IEEE International System-on-Chip Conference (SOCC)*. Arlington, VA: IEEE, Sep. 2018, pp. 206–212. [Online]. Available: https://ieeexplore.ieee.org/document/8618545/

[113] ——, "Dimension reduction using Quantum Wavelet Transform on a high-performance reconfigurable computer," *International Journal of Reconfigurable Computing*, vol. 2019, p. 14, 2019.

[114] N. Mahmud, B. Haase-Divine, A. MacGillivray, B. Srimoungchanh, A. Kuhnke, N. Blankenau, A. Rai, and E. El-Araby, "Modifying quantum Grover's algorithm for dynamic multi-pattern search on reconfigurable hardware," *Journal of Computational Electronics*, vol. 19, no. 3, pp. 1215–1231, Sep. 2020. [Online]. Available: https://link.springer.com/10.1007/s10825-020-01489-3

[115] M. Khalid, U. Mujahid, A. Jafri, H. Choi, and N. Muhammad, "An FPGA-based hardware abstraction of quantum computing systems," *Journal of Computational Electronics*, vol. 20, pp. 2001–2018, 2021.

[116] T. Bonny and A. Haq, "Emulation of high-performance correlation-based quantum clustering algorithm for two-dimensional data on FPGA," *Quantum Information Processing*, vol. 19, p. 179, 2020.

[117] E. Aïmeur, G. Brassard, and S. Gambs, "Quantum clustering algorithms," in *Proceedings of the 24th international conference on Machine learning - ICML '07*. Corvalis, Oregon: ACM Press, 2007, pp. 1–8. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1273496.1273497

[118] Kuk-Hyun Han and Jong-Hwan Kim, "Quantum-inspired evolutionary algorithm for a class of combinatorial optimization," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 6, pp. 580–593, Dec. 2002. [Online]. Available: http://ieeexplore.ieee.org/document/1134125/

[119] M. Aramon, G. Rosenberg, E. Valiante, T. Miyazawa, H. Tamura, and H. G. Katzgraber, "Physics-Inspired Optimization for Quadratic Unconstrained Problems Using a Digital Annealer," *Frontiers in Physics*, vol. 7, p. 48, Apr. 2019. [Online]. Available: https://www.frontiersin.org/article/10.3389/fphy.2019.00048/full

[120] H. M. Waidyasooriya, H. Oshiyama, Y. Kurebayashi, M. Hariyama, and M. Ohzeki, "A Scalable Emulator for Quantum Fourier Transform Using Multiple-FPGAs With High-Bandwidth-Memory," *IEEE Access*, vol. 10, pp. 65 103–65 117, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9798809/

[121] N. Mahmud, B. Haase-Divine, A. Kuhnke, A. Rai, A. MacGillivray, and E. El-Araby, "Efficient computation techniques and hardware architectures for unitary transformations in support of quantum algorithm emulation," *Journal of Signal Processing Systems*, vol. 92, no. 9, pp. 1017–1037, 2020.

[122] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "An Introduction to Quantum Programming in Quipper," *arXiv:1304.5485 [quant-ph]*, vol. 7948, pp. 110–124, 2013, arXiv: 1304.5485. [Online]. Available: http://arxiv.org/abs/1304.5485

[123] J. Hooyberghs, "Q# Language Overview and the Quantum Simulator," in *Introducing Microsoft Quantum Computing for Developers*. Berkeley, CA: Apress, 2022, pp. 121–167. [Online]. Available: https://link.springer.com/10.1007/978-1-4842-7246-6_6

[124] Y. Moawad, W. Vanderbauwhede, and R. Steijl, "Transformations for accelerator-based quantum circuit simulation in Haskell," 2022. [Online]. Available: https://arxiv.org/abs/2210.12703

[125] ——, "Quantum Circuit-Width Reduction through Parameterisation and Specialisation," *Algorithms*, vol. 16, no. 5, p. 241, May 2023. [Online]. Available: https://www.mdpi.com/1999-4893/16/5/241

[126] "Intel® FPGA SDK for OpenCL™ best practices guide," 2017, accessed: 01/10/2024. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/705111?fileName=aocl-best-practices-guide-17-1-683521-705111.pdf

[127] "Geforce GTX TITAN Black — Specifications," 2014, accessed: 01/10/2024. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/geforce-gtx-titan-black/specifications/

[128] "Stratix V Device Overview," 2020, accessed: 01/10/2024. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/670815?fileName=stx5_51001-683258-670815.pdf

[129] "Stratix® 10 GX/SX Product Table," 2020, accessed: 01/10/2024. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/652478?fileName=stratix-10-product-table.pdf

[130] O. Segal, N. Nasiri, M. Margala, and W. Vanderbauwhede, "High level programming of FPGAs for HPC and data centric applications," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–3.

[131] R. Steijl, "Quantum algorithms for nonlinear equations in fluid mechanics. in: Zhao, Y. (ed.) Quantum Computing and Communications. IntechOpen:London. ISBN 9781839681332 (doi: 10.5772/intechopen.95023)," 2022.