Imperial College London
Department of Computing

Improving Low Latency Applications for Reconfigurable Devices

Stewart Denholm

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, November 2022

Statement of Originality

The work presented in this thesis is derived from my own, original research. The work and collaboration of other authors is clearly acknowledged, and references to external literature cited.

Copyright Declaration

Abstract

This thesis seeks to improve low latency application performance via architectural improvements in reconfigurable devices. This is achieved by improving resource utilisation and access, and by exploiting the different environments within which reconfigurable devices are deployed.

Our first contribution leverages devices deployed at the network level to enable the low latency processing of financial market data feeds. Financial exchanges transmit messages via two identical data feeds to reduce the chance of message loss. We present an approach to arbitrate these redundant feeds at the network level using a Field-Programmable Gate Array (FPGA). With support for any messaging protocol, we evaluate our design using the NASDAQ TotalView-ITCH, OPRA, and ARCA data feed protocols, and provide two simultaneous outputs: one prioritising low latency, and one prioritising high reliability with three dynamically configurable windowing methods.

Our second contribution is a new ring-based architecture for low latency, parallel access to FPGA memory. Traditional FPGA memory is formed by grouping block memories (BRAMs) together and accessing them as a single device. Our architecture accesses these BRAMs independently and in parallel. Targeting memory-based computing, which stores pre-computed function results in memory, we benefit low latency applications that rely on: highly-complex functions; iterative computation; or many parallel accesses to a shared resource. We assess square root, power, trigonometric, and hyperbolic functions within the FPGA, and provide a tool to convert Python functions to our new architecture.

Our third contribution extends the ring-based architecture to support any FPGA processing element. We unify E heterogeneous processing elements within compute pools, with each element implementing the same function, and the pool serving D parallel function calls. Our implementation-agnostic approach supports processing elements with different latencies, implementations, and pipeline lengths, as well as non-deterministic latencies. Compute pools evenly balance access to processing elements across the entire application, and are evaluated by implementing eight different neural network activation functions within an FPGA.

Acknowledgements

I would first like to thank my supervisor, Prof. Wayne Luk for all his help and guidance with carrying out my research and in writing this thesis. His encouragement and support has made pursuing my PhD at Imperial College a much more productive and enjoyable experience.

I would like to thank Amani El-Kholy and everyone at Imperial College for their help and support, both personally and professionally. I would also like to thank my past and present colleagues for all of their invaluable help and contributions: Hiroaki Inoue, James Arram, Kuen Hung Tsoi, Maciej Kurek, Takashi Takenaka, Thomas Chau, Timothy Todman, and Tobias Becker.

Finally, I would like to thank my parents and family for their continued love and support over all these years. I wouldn't be here without you.

Contents

Statement of Originality **3**

Copyright Declaration **5**

Abstract **7**

Acknowledgements **9**

Contents **11**

List of Tables **17**

List of Figures **19**

Glossary **23**

1 Introduction **27**

1.1 Motivation 27

1.2 Identifying Challenges 29

1.3 Research Contributions 35

1.4 Publications 39

2 Background **40**

2.1 Field-Programmable Gate Arrays . . . 40

2.1.1 Development Comparison . . . 41

2.1.2 FPGA Resources . . . 42

2.1.3 FPGA Deployment . . . 49

2.2 FPGA Application Design and Development . . . 52

2.2.1 Design Flows . . . 52

2.2.2 Application Description and Development . . . 54

2.3 Coarse-grained FPGA Architectures . . . 59

2.3.1 Functional Units . . . 60

2.3.2 The Interconnect . . . 62

2.3.3 Application Development and Tools . . . 63

2.3.4 Summary . . . 64

2.4 Memory-based Computing . . . 64

2.4.1 Mapping Function Inputs . . . 66

2.4.2 Data Storage and Representation . . . 67

2.4.3 Memory Architectures . . . 70

2.4.4 Performance Improvements . . . 73

2.5 Market Data Feed Processing . . . 76

2.5.1 Reliability . . . 77

2.5.2 Multi-stream Arbitration . . . 77

2.5.3 Data Protocols . . . 79

2.5.4 Related Works 80

2.6 Neural Network Activation Functions 81

2.6.1 Neural Networks 82

2.6.2 Activation Functions 84

2.6.3 Training and Inference 86

2.7 Summary 89

3 Network-level FPGA Acceleration of Low Latency Market Data Feed Arbitration 92

3.1 Overview 92

3.2 Introduction 93

3.3 Flexible A/B Line Arbitration 94

3.3.1 High Reliability Modes 96

3.3.2 Low Latency Mode 98

3.3.3 Network-level Operation 99

3.3.4 Customisation and Extensibility 100

3.4 Performance Model 101

3.5 High-performance Architecture 102

3.5.1 Deterministic, Multiple-input Memory Access 102

3.5.2 Optimising Packet Accesses 104

3.5.3 Improving Throughput and Latency 105

3.5.4 Cycle-accurate Testing 106

3.6 Implementation 106

3.7 Results 108

3.7.1 Real-world Performance . . . 109

3.7.2 Latency . . . 111

3.7.3 Saturation Estimation . . . 113

3.8 Summary . . . 114

4 Maximising Parallel Memory Access for Low Latency FPGA Designs 116

4.1 Overview . . . 116

4.2 Introduction . . . 117

4.3 Targeting Low Latency Applications . . . 119

4.3.1 Function Choice . . . 119

4.3.2 Memory Type . . . 121

4.3.3 Sub-memory Interconnect . . . 124

4.4 Ring-based Architecture . . . 127

4.4.1 Configuration and Operation . . . 128

4.4.2 Latency . . . 130

4.5 Automated Tool . . . 131

4.5.1 Function Conversion . . . 132

4.5.2 Hybrid Computation . . . 133

4.6 Implementation . . . 134

4.6.1 Data Width and Resource Scaling . . . 135

4.6.2 Resource Use and Parallelism . . . 136

4.7 Results . . . 137

4.7.1 Pre-computed Values and Data Precision . . . 137

4.7.2 Latency Reduction of Complex Functions . . . 139

4.8 Summary . . . 141

5 A Unified Approach for Managing Heterogeneous Processing Elements on FPGAs 143

5.1 Overview . . . 143

5.2 Introduction . . . 144

5.3 Motivation and Challenges . . . 147

5.3.1 Evenly Sharing Processing Elements . . . 147

5.3.2 Utilise and Support FPGA Resources . . . 148

5.3.3 A Unified View of Processing Elements . . . 149

5.3.4 Assessment and Evaluation . . . 150

5.3.5 Summary . . . 151

5.4 The Compute Pool Model . . . 153

5.4.1 Rotating Architecture . . . 153

5.4.2 Minimising Routing While Supporting Non-determinism . . . 154

5.5 Classifying Processing Elements . . . 156

5.6 Heterogeneous Processing Elements . . . 158

5.6.1 Supporting Different Processing Element Implementations . . . 158

5.6.2 Supporting Different Processing Element Connections . . . 160

5.6.3 Latency-centric Model of Processing Elements . . . 161

5.7 Multi-pool Routing and Performance . . . 163

5.7.1 Inter-pool Routing . . . 164

5.7.2 Non-deterministic Latencies . . . 165

5.8 Implementation 169

5.9 Architecture Evaluation 170

5.9.1 Resource Scaling and Arbitration 170

5.9.2 Higher Data-path Parallelism with Fewer Elements 172

5.10 Framework Performance Evaluation 173

5.10.1 Utilisation 173

5.10.2 Overlay Throughput 174

5.10.3 Processing Elements 176

5.11 Application Evaluation 176

5.11.1 Functions and Methodology 177

5.11.2 Throughput 178

5.11.3 Latency 179

5.11.4 Fractional Processing Elements 180

5.12 Summary 183

6 Conclusion **185**

6.1 Summary of Achievements 185

6.2 Future Work 191

6.2.1 FPGA Processing at the Infrastructure Level 192

6.2.2 Optimising Parallel Access to Shared Resources 194

6.2.3 The Compute Pool Model 196

6.3 Final Thoughts 199

Bibliography **201**

List of Tables

2.1 Sigmoid activation functions. 85

2.2 ReLU-based activation functions. 86

3.1 Description of symbols for Eqn. 3.5. 102

3.2 Meta-data cache contents. 105

3.3 Packet protocol specifications. 107

4.1 Comparison of memory interconnects with D data-paths and N sub-memories. . 125

4.2 Our ring-based architecture's resource use for any 16-bit function. 136

4.3 Our approach vs. common FPGA functions in past works. 140

5.1 Compute pool framework resource use and operating frequencies on the Alveo U280. 171

5.2 Throughput comparison to past overlays and frameworks. 175

5.3 Computational intensity of functions (ops/byte). 177

List of Figures

1.1 Comparison of different processing devices. 28

2.1 FPGA resource columnar layout. 43

2.2 Xilinx UltraScale+ Virtex FPGA resources. Source: Xilinx [40]. 44

2.3 Intel's Infrastructure Processing Unit model. Source: Intel [63]. 51

2.4 Simple code example. 53

2.5 The HDL design flow. 55

2.6 An example of a coarse-grained architecture interconnect. 62

2.7 General approach for representing the function $y = fn(x)$ in memory. 65

2.8 Forming a user-defined memory by grouping many, smaller memories together. . 71

2.9 Example Gaussian probability density function with different standard deviations. 75

2.10 An example network linking a financial exchange with its members via the internet, a dedicated microwave network, and co-location. 76

2.11 Shares trades monthly on the NASDAQ, NYSE, and CBOE exchanges. Source: CBOE [105]. 78

2.12 An example of a fully connected neural network with three hidden layers. 82

3.1 The layout of our A/B line arbitration design. 95

3.2 High reliability time. 96

3.3 High reliability count. 97

3.4 High reliability time & count. 98

3.5 The address generator automatically splitting a large memory. 103

3.6 The cycle-accurate testing framework. 106

3.7 The layout of our arbitrator module within the FPGA. 108

3.8 Resource usage for the three messaging protocols. 109

3.9 Market data feed throughput during the day. 110

3.10 Worst-case time delay as an indicator of arbitrator saturation. 113

4.1 Four memory interconnect topologies. 125

4.2 Examples of ring-based architecture configurations. 128

4.3 An example of our ring-based architecture processing four 8-bit inputs. 129

4.4 Examples of Python function and configuration files for our automated tool. . . 132

4.5 Example hybrid function with a 2-bit selector, 12-bit input, and 16-bit output. . 134

4.6 Relationship between function input and output widths and LUT utilisation. . . 135

4.7 Resource use and best-fit linear regression predictions for the Alveo U280. 137

4.8 Comparing 8-bit input functions when outputting with 8-bit and 16-bit data widths. 138

5.1 Serving 4 data-paths with different numbers and types of processing elements. . 154

5.2 Three 32-bit DSP-based processing elements and one 24-bit fractional element serving 4 data-paths. 157

5.3 A call-and-response compute pool with 3 elements, serving 4 parallel data-paths. 159

5.4 Connecting processing elements within a pool. 160

5.5 Cascading routing between three function pools. 165

5.6 Latency improvements when de-synchronising multiple compute pools. 166

5.7 Relative latency reduction when increasing the input space each element can serve. 168

5.8 Regression analysis for compute pool framework resource use on the Alveo U280. 171

5.9 Activation function resource use on the Alveo U280. 172

5.10 Throughput for pools with 8-1024 processing elements. 178

5.11 Latency for pools with 8-1024 processing elements. 179

5.12 Adding fractional elements to compute pools with different initial occupancies. . 181

Glossary

ALM Adaptive Logic Module. 31, 45, 174

ALU Arithmetic Logic Unit. 60

ARCA Archipelago Exchange. 36, 79, 80, 94, 100, 106, 108, 109, 111, 115, 186, 187

ASIC Application-Specific Integrated Circuit. 27, 28, 40–42, 47

AXI Advanced eXtensible Interface. 55

BRAM Block Random Access Memory. 30, 31, 36, 37, 43–46, 48, 59–61, 70, 71, 73–75, 90, 103, 104, 109, 116–118, 122–124, 130, 135–137, 140–142, 144, 147, 149, 150, 153, 154, 156, 157, 174–176, 187–189, 191, 195

CAM Content-Addressable Memory. 102

CBOE Chicago Board Options Exchange. 77

CNN Convolutional Neural Network. 83, 84

CPU Central Processing Unit. 27, 30, 32, 34, 40–42, 45, 47, 49–58, 60, 61, 63, 84, 87, 88, 90, 93, 100, 120, 138, 151, 160, 173, 176, 182, 185, 187, 191, 193, 194, 198

DDR Dynamic Data Rate. 46, 48, 49, 70, 102, 122, 187, 195

DNN Deep Neural Network. 83, 84, 151

DSP Digital Signal Processor. 30, 43–47, 58–61, 65, 72, 88, 120, 123, 139, 144, 147–151, 153, 154, 156, 157, 161, 173, 174, 176

ELU Exponential Linear Unit. 86, 146, 171

FPGA Field-Programmable Gate Array. 27–38, 40–52, 54–64, 70–73, 78, 80, 81, 84, 87–90, 92–94, 99–101, 103, 105–109, 115–124, 131, 132, 138, 139, 142–151, 153, 157, 158, 161, 167, 169, 170, 172–178, 180, 182–187, 189–191, 193–195, 197–200

GELU Gaussian Error Linear Unit. 86, 146, 172, 173, 178, 179

GPU Graphics Processing Unit. 27, 30, 41, 49, 50, 52, 56–58, 84, 87, 88, 100, 136, 191

HBM High Bandwidth Memory. 31, 49, 70, 102, 122, 147, 195, 196

HDL Hardware Description Language. 54–59, 63

HFT High-Frequency Trading. 76

HLS High-Level Synthesis. 56–59, 197, 200

ICAP Internal Configuration Access Port. 74, 75

IoT Internet of Things. 32, 41, 193

LReLU Leaky Rectified Linear Unit. 86, 146, 171, 178, 179

LUT Look-up Table. 30, 31, 43–47, 59–61, 65, 70, 88, 109, 117, 122, 123, 136, 137, 140, 142, 144, 147–149, 151, 153, 154, 156, 157, 161, 171, 173, 174, 176, 188

MAC Multiply-Accumulate. 72, 84

NASDAQ National Association of Securities Dealers Automated Quotations. 36, 77, 79, 81, 94

NIC Network Interface Card. 35, 51, 80, 99, 107, 193

NUMA Non-Uniform Memory Access. 65, 195

NYSE New York Stock Exchange. 77, 79

OPRA Options Price Reporting Authority. 36, 79–81, 94, 95, 100, 106, 108, 109, 111, 112, 115, 186, 187

PCIe Peripheral Component Interconnect Express. 48, 49, 80, 107

ReLU Rectified Linear Unit. 84–86, 146, 178, 183, 190

RNN Recurrent Neural Network. 83–85

RTL Register-Transfer Level. 54, 56, 58

SBE Simple Binary Encoding. 80

SIMD Single Instruction, Multiple Data. 58

SoC System On Chip. 47, 49

SSI Stacked Silicon Interconnect. 74

TCP Transmission Control Protocol. 77, 80

TPU Tensor Processing Unit. 41

UDP User Datagram Protocol. 80, 95, 99, 108, 150

VHDL Very High Speed Integrated Circuits Hardware Description Language. 54, 55, 58, 59

VLIW Very Long Instruction Word. 60, 61

Chapter 1

Introduction

1.1 Motivation

Field-Programmable Gate Arrays (FPGAs) are reconfigurable computing devices composed of a number of digital logic resources. These resources may be general-purpose, or highly specialised blocks for data processing, storage, or routing. FPGAs can be programmed to implement a desired function by configuring and connecting together these blocks to form processing elements and data processing pipelines.

In a simplified comparison to other processing platforms, Fig. 1.1 shows that FPGAs serve as a middle ground between the pure hardware implementations of Application-Specific Integrated Circuits (ASICs), and the often slower, but easier to develop software implementations based around Central Processing Units (CPUs) and Graphics Processing Units (GPUs).

ASICs offer an optimised hardware solution tailored to either an application domain or a single application implementation. As a result, they provide better performance and lower power designs compared to FPGAs or software-based implementations. Alternatively, software solutions for CPU and GPU platforms are designed to run on general-purpose processors and so are much simpler and faster to develop. Software can easily be updated or improved should

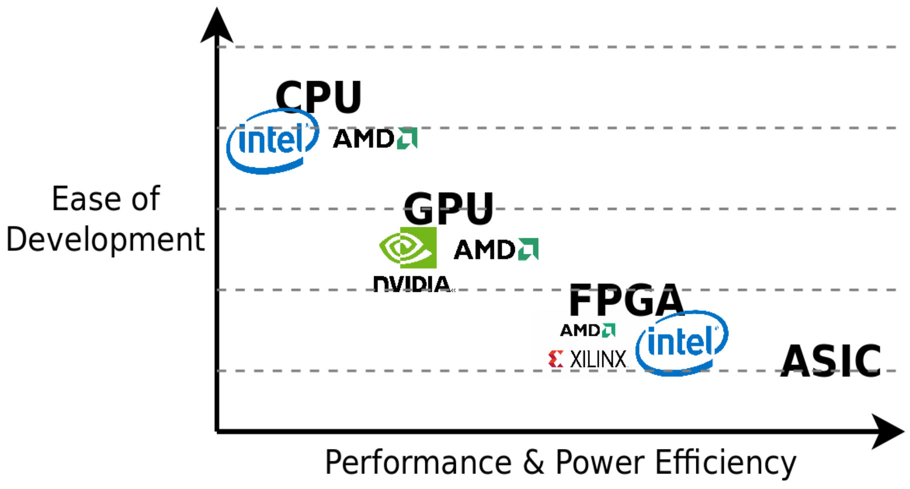


Figure 1.1: Comparison of different processing devices.

errors emerge in the design, or new application or implementation improvements be developed. We discuss the composition of FPGAs and their application development process in more detail in Sections 2.1 and 2.2.

Whether we view FPGAs as either a more flexible, reconfigurable version of an ASIC, or as a higher-performing, lower-power platform for software-based designs, FPGAs bring their own benefits and challenges. Modern FPGAs offer a large number and wide range of resources with which to construct and run applications. How then do we simplify and streamline development of this fundamentally hardware-oriented platform, and how do we leverage the unique computing opportunities that FPGAs provide?

This thesis seeks to improve the operation of low latency applications via new architectural frameworks and optimisations. Rather than simply optimising the design of just a single application or domain, we look to improve resource use within the FPGA itself. Through the creation and examination of these new low-level architectures, we seek to address a range of higher-level application requirements.

Low latency focused designs prioritise reducing the end-to-end processing time of their application. Within the domains of automated driving [1] [2] and robotics [3] we face strict safety requirements on the time taken to process, analyse and react to new data. High frequency financial trading [4] [5] [6] is another area that prioritises low latencies, but here latency is directly tied to the profit generating ability of a trading strategy. Real-time video and audio processing technologies—such as those in online gaming [7] or new 5G networks [8]—is another area of interest that instead relies on lower latencies to provide a good user experience, or to accurately portray the continuously changing conditions of a shared, virtual environment.

FPGAs are an attractive platform for such applications as they offer low-level control of computing hardware, thus allowing for the creation of more optimised and customised designs. However, our original challenges remain: facilitating the creation of a high-performing, bespoke design, while simplifying the traditionally complex and time consuming development process.

The above examples of low latency applications rely on and *prioritise* low latency, but as real-world applications they will have additional requirements that must be met. This may be a minimum level of data processing accuracy or throughput, or to work within the more common limitations of available resources and costs. If our aim is to create architectures that offer improvements to a wide range of low latency applications, we will need a general framework that prioritises latency, but is also able to account for, and support, these other factors.

1.2 Identifying Challenges

When examining low latency applications within FPGAs, this thesis looks at challenges stemming from a few key areas: **(a)** exploiting the different environments within which FPGAs may be tasked with serving low latency applications—whether this be at the network level, infrastructure level, or when serving as the primary processing device within a system; **(b)** the heterogeneity present within FPGA devices themselves, due to their many different types of processing elements; and **(c)** the high degrees of parallelism present, and increasingly demanded, within FPGA applications.

From an examination of these key areas we will identify 3 challenges to motivate our 3 contributions, described in Section 1.3.

Within a data centre, the edge computing model [9] seeks to more efficiently meet the processing needs of applications by placing processing elements closer to their sources of data. This sees many small processing elements placed within the infrastructure layer. These form part of the communication network, storage system, or some other subsystem or service offered by the data centre.

For our first key area **(a)**, FPGAs placed at the network level of a data centre provide us the opportunity to process and act upon incoming data streams with much lower latencies than if using traditional processing models. In addition, in-network processing allows us to meet the processing needs of multiple downstream applications. For example, should incoming data require an initial processing step—such as verification, normalisation, etc—then *each* application that uses this data must perform this step. In-network processing lets us remove this redundant functionality from applications, simplifying development, and provide a dedicated and optimised solution to this shared problem.

In-network processing also provides the opportunity to permanently move processing away from traditional computing nodes. With automated trading systems, for example, the application will look for a signal within a data feed—say a stock price crossing some threshold value—to which they will quickly react by placing a buy or sell order with a broker. As well as removing this workload from the data centre's dedicated computing nodes, we also reduce our response and network transmission times by performing this function, in its entirety, at the "edge" of the data centre.

When tackling key area **(b)**, it is the heterogeneous *nature* of FPGAs themselves we must look to address. As well as Look-up Tables (LUTs), Digital Signal Processors (DSPs) and Block memories (BRAMs), which together serve as the more general and configurable backbone of FPGA designs, newer FPGAs include an increasing number of additional resources.

From Intel and Xilinx, examples of embedded cores within or accompanying FPGAs include CPUs [10] [11], GPUs [12], as well as domain-specific digital signal processing [13] and cores targeting Artificial Intelligence (AI) [14]. Additional types of *existing* resources are also being added. New memory storage, like UltraRAM [15] and High Bandwidth Memory (HBM) [16] [17] offer greater storage, but less scope for configuration and parallelism over BRAMs.

LUTs are also advancing between generations. Older FPGAs from Xilinx made use of 4-input LUTs [18], while newer devices use 6-input LUTs [19]. Different LUT configurations are also possible. The Adaptive Logic Modules (ALMs) [20] within Intel FPGAs have 8 inputs and can implement two 4-input LUTs, one 6-input LUT, or some overlapping combination of 3, 4, 5, and 6 -input LUTs if they share one or more inputs.

This device heterogeneity complicates and extends development time. Developers must seek to both integrate and manage all of these elements in some high-level, general manner, while also fully utilising each resource's unique strengths. Managing heterogeneity is not only a problem for FPGAs, but also for the research and development of many-core systems in general [21].

High degrees of parallelism within FPGA designs is the final key area **(c)** as this is one of the main design patterns employed when developing for FPGAs. A processing core is created to perform some function or operation, then simply duplicated to provide additional, parallel data-paths. Bottlenecking issues then arise when these parallel data-paths seek to access or make use of a limited or finite resource. Within highly parallel designs, the performance penalty from routing alone can be significant [22]. In general, resource bottleneck issues are not limited to FPGAs, however, their fine-grained, granular construction of processing elements provides new opportunities to explore this topic.

This thesis aims to improve the performance of low latency applications within reconfigurable devices through improved architectures and a more optimised use of resources. Given the issues described above, we can now formally identify the challenges we will look to address and categorise them into three main areas:

- Low latency processing of financial data in edge computing environments
- Shared, parallel access to memory resources
- Shared, parallel access to all the heterogeneous FPGA processing elements

C1: Low latency processing of financial data in edge computing environments

We first look at exploiting the different areas within a system, or data centre, where an FPGA may be deployed. When placed at the infrastructure level, particularly the network level, FPGAs and their applications can serve as an initial, low latency data processing or filtering step to other applications.

As we discussed previously with edge computing, FPGAs may be placed at the data entry point of a single computing node, or as part of a larger system that serves an entire computing cluster. This provides us the opportunity to remove common functionality from the downstream, latency-sensitive applications we serve, group these together, and perform these functions at the network level.

We choose to focus on the processing of financial data due to its high requirements for **(a)** low latency processing, **(b)** data reliability, and **(c)** predictable and consistent performance. Specifically, we look to the processing of data feeds from financial exchanges. CPU-based approaches can see non-deterministic processing times, suggesting the dataflow model of FPGA processing would be of benefit. Existing works also tend to be highly application-specific, or support a single message communication protocol. We discuss these issues further in Sections 2.2.1 and 2.5. When employed at the network level, a more general and flexible framework is needed.

Edge processing has seen success in many areas [23] including the Internet of Things (IoT) [24] [25], deep learning [26]—for both inference and training—and mobile networks [27]. However, when we seek to reduce the data processing workloads of multiple, downstream applications, simply de-duplicating the common functionality is not sufficient.

For a robust, general-purpose framework we must look to not only improve performance through the creation of a lower latency design, but also support the different processing requirements of different downstream applications. The operation of these in-network functions may be tied to the downstream applications themselves, thus those same applications must be provided some means of controlling the in-network processing at run-time.

C2: Shared, parallel access to memory resources

For applications that make use of high degrees of parallelism, a key challenge is to reduce latency when multiple, parallel accesses are required to a shared resource. This can be multiple, independent applications seeking access to a limited resource, or just one application which itself contains multiple, parallel data-paths. Neural networks are a prime example of this latter application. A single neural network may make use of thousands of parallel neurons, all potentially performing the same function [28] [29].

We identify memory access as a key component in lowering latency. This applies not only to traditional memory reads and writes, but also to memory-based computing. Memory-based computing pre-computes the values of a given function and stores these results in memory to be read out at run-time. This effectively lowers the processing latency of any deterministic function to that of a single memory read operation. Compute-bound operations are then converted to memory or I/O-bound ones.

We describe memory-based computing in greater detail in Section 2.4. It is shown to be of particular importance to low latency applications whose operation or optimisations are typically improved by latency costly means, such as via iterative operations. Improving the latency of parallel memory accesses can thus benefit both memory and processing.

Schemes for sharing access to a shared resource have been explored using time-multiplexing and dynamic reconfiguration [30], as well as through context switching within more coarse-grained FPGA frameworks [31]. Techniques for the sharing of individual FPGA resources have also been introduced [32] [33]. The choice of connection topology is also a key consideration. We examine this further in Section 4.3.3.

The main challenges when reducing latencies for parallel accesses to a shared resource lie in scaling our framework to both: **(a)** large numbers of parallel accessing data-paths, and **(b)** large numbers of accessed resources, in this case, memory.

C3: Shared, parallel access to all the heterogeneous FPGA processing elements

A natural progression from managing and optimising parallel accesses to shared memory is to extend this framework to cover all processing elements within the FPGA. However, simply replacing our memory resources with "processing elements" will not be sufficient.

FPGAs contain a large number of different resources from which we can form any number of different processing elements. Even when designed to perform the same function, these different processing elements may have different implementations, latencies, pipeline depths, or operating frequencies.

Different processing elements may also have different data connection schemes, or inherent levels of parallelism that we must take into account. For example, some memory resources may have more than one read port, while embedded CPUs may offer multi-threaded operations. The individual benefits these heterogeneous elements provide must not be lost when incorporated into a larger, more general framework.

Past approaches often take a broader view, seeking to unify FPGA development with that of CPUs. Examples include ReconOS [34], which uses a multi-threading programming model with software threads and hardware threads, while FUSE [35] provides an abstract view of processing resources to CPU-based applications.

Past works that promote abstraction *within* the FPGA, such as overlays and coarse-grained architectures, are discussed and compared in greater detail in Section 2.3. These group together FPGA resources in order to define new processing units, and thus create a more homogeneous layout. FPGA applications are then built atop this new framework. This sees issues with low utilisation when an application's resource requirements do not exactly match the composition of these new processing units. Entirely new application development tools and toolchains may also be needed. These problems should be minimised or, ideally, eliminated.

A unified means to model the processing performance of different resources will also be required. This general model would classify resources using a standard metric, allowing us to quantify their performance, as well as compare and trade-off *any* type of FPGA resource. We could therefore support both the wide range of resources within an FPGA, as well as new resources included in future FPGAs

1.3 Research Contributions

Our aim is to improve the performance of low latency FPGA applications through improved architectures and an optimised use of resources. When considering the challenges we identify above, the three main contributions of this thesis are:

Low latency, network-level processing of financial data feeds

To capitalise on the increasing use of FPGAs operating at the network level, the first contribution of this thesis (**Chapter 3**) examines lowering the latency of multiple, downstream applications by pre-processing a shared, incoming data feed. For this we focus on processing the market data feeds provided by financial exchanges. These message feeds are used in time-critical automated trading applications, and two identical feeds (A and B feeds) are provided in order to reduce the risk of message loss. A key challenge is to support A/B line arbitration. This unifies the A and B lines into a single, consistent feed able to compensate for missing, duplicate, and out-of-order packets.

As every trading application that relies on this data feed must perform this arbitration, it is only natural to consolidate this shared functionality. The FPGA forms part of the Network Interface Card (NIC), so our arbitration forms a natural extension to the existing packet-processing pipeline. The challenge of non-deterministic processing times can then be solved by employing the FPGA's more data-driven, dataflow model of processing.

As well as lowering the latency of A/B line arbitration, we provide support for any messaging protocol, and output two simultaneous modes of operation: one prioritising low latency, and one prioritising high reliability with three dynamically configurable windowing methods. The latter windowing modes can be altered by downstream applications in real-time to account for the ever-changing market conditions.

This addresses the issues of providing a network-level service that is both **(a)** flexible and supporting of different communication protocols, while also **(b)** able to provide low latency, application-specific functionality.

We implement and assess the NASDAQ TotalView-ITCH, OPRA, and ARCA market data feed protocols on the Xilinx Virtex-5 and Virtex-6 FPGAs, reporting deterministic processing times of 1 cycle for our low latency output and 7 cycles for the high reliability output. This corresponds to latencies of $6ns$ and $42ns$ for the TotalView-ITCH data feed, and $5.25ns$ and $36.75ns$ for OPRA and ARCA, respectively.

Reduce memory access latencies for highly parallel, latency-sensitive functions

The second contribution of this thesis (**Chapter 4**) is a novel ring-based architecture to reduce latency when multiple, parallel accesses are required to FPGA memory. We target low latency designs that are implemented using memory-based computing. Here, the results for a given function are pre-computed and stored in FPGA memory. The function is then "called" at run-time by reading results from memory.

Memory is formed within FPGAs by grouping together multiple BRAMs then accessing them as a single, monolithic device. Our architecture accesses these constituent BRAMs independently, and can thus benefit low latency applications that rely on: highly-complex functions; numerical precision via iterative computation; or high degrees of parallelism, where many parallel data-paths access a shared memory resource. An automated tool is developed to convert any deterministic function or existing memory structure to our new, memory-based computing architecture.

We are able to reduce *any* 16-bit function to a 1 cycle memory read. High degrees of parallelism are also supported, with 1024 parallel function calls able to be processed at 300MHz on the Xilinx Alveo U280 accelerator card. Large numbers of parallel accesses now play to our advantage as we can reduce the average access time further to 0.5 cycles per function call by using the BRAM's dual read ports.

In contrast to the $O(N^2)$ scaling of point-to-point and crossbar schemes, or the $O(NlogN)$ scaling of multi-stage interconnects, our ring-based architecture's resources and number of links scale linearly, $O(N)$. This applies both to higher degrees of parallelism, i.e., more calls to the implemented function, and the number of BRAMs accessed in parallel. Our access times also scale linearly, while our interconnection scheme requires little logic and routing, and automatically arbitrates parallel accesses between the multiple function calls and BRAMs.

The Xilinx Alveo U280 accelerator card is used to evaluate and assess our approach using square root, power, trigonometric, and hyperbolic functions.

Centralised compute pools for shared access to FPGA processing elements

The third contribution of this thesis (**Chapter 5**) is to extend our ring-based architecture beyond memory to support any FPGA processing element. Our new architecture unifies heterogeneous processing elements into compute pools formed of E processing elements, each implementing the same function. A compute pool is then able to serve D parallel calls to its implemented function.

Through a call-and-response approach to computation, we develop a model based solely on the processing latency of each element in the pool. This addresses the challenge of forming a resource and implementation agnostic view of processing elements. A compute pool can support processing elements with different: implementations, connections, pipeline lengths, latencies, and non-deterministic processing times. Through this latency-oriented model we allow the plug-and-play of mismatched processing elements, and are able to easily add or remove processing elements from the compute pool for a known performance cost.

We focus on unifying *existing* implementations of processing elements, rather than follow the model of coarse-grained architectures and define our own. We therefore place no limits on the composition of processing elements, greatly improving resource utilisation, where we demonstrate 96.2% utilisation for 1024 parallel data-paths.

This approach also solves the problem of requiring new development tools and toolchains for our new architecture. The use of existing processing elements mean we support the use of all existing application design, development, and analysis tools.

We also explore new avenues of performance improvements. A centralised pool of processing elements evenly balances the use of all FPGA resources across the entire application. Our data routing scheme is able to reduce the bottleneck associated with multiple data-paths accessing a centralised pool, and almost eliminates it entirely for designs with known latencies. Additionally, as individual processing elements no longer have the sole responsibility for serving the entire range of possible function calls, new, specialised processing elements can be added to a compute pool to improve performance.

Rather than serve every possible input, these new *fractional* processing elements target a set range of possible function inputs—such as individual sections of a function curve—and can therefore be both smaller and more optimised towards their given operation. Fractional processing elements **collectively** serve the entire range of function inputs, so can **individually** be formed of fewer resources, providing a more fine-grained means of forming FPGA resources into processing elements.

We explore this new architecture and the formation of processing elements through an examination of neural networks. We assess 8 neural network activation functions with different levels of computational complexity, demonstrating an average 21% improvement in throughput, and 29% for latency when implemented on the Xilinx Alveo U280 accelerator card.

1.4 Publications

- S. Denholm and W. Luk, ‘A Unified Approach for Managing Heterogeneous Processing Elements on FPGAs’, in 2022 32nd International Conference on Field Programmable Logic and Applications (FPL), August 2022.

- S. Denholm and W. Luk, ‘Mixed-Resource Parallel Processing on FPGAs’, in 2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2022.

- S. Denholm and W. Luk, ‘Maximising Parallel Memory Access for Low Latency Designs’, in IEEE International Symposium on Circuits and Systems (ISCAS), May 2022.

- M. Kurek, T. Becker, C. Guo, S. Denholm, A.-I. Funie, M. Salmon, T. Todman, and W. Luk, ‘Self-aware hardware acceleration of financial applications on a heterogeneous cluster’, in Self-aware computing systems - an engineering approach, P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, Eds. Springer, 2016, pp. 241-260

- S. Denholm, H. Inoue, T. Takenaka, T. Becker, and W. Luk, ‘Network-level FPGA acceleration of low latency market data feed arbitration’, IEICE Trans. Inf. Syst., vol. 98-D, no. 2, pp. 288-297, 2015, doi: 10.1587/transinf.2014RCP0011.

- S. Denholm, H. Inoue, T. Takenaka, T. Becker, and W. Luk, ‘Low latency FPGA acceleration of market data feed arbitration’, in 2014 IEEE 25th international conference on application-specific systems, architectures and processors (ASAP), Los Alamitos, CA, USA, Jun. 2014, pp. 36-40. doi: 10.1109/ASAP.2014.6868628.

- S. Denholm, H. Inoue, T. Takenaka, and W. Luk, ‘Application-specific Customisation of Market Data Feed Arbitration’, in 2013 International Conference on Field-Programmable Technology (FPT), Dec. 2013, pp. 322-325. doi: 10.1109/FPT.2013.6718377.

Chapter 2

Background

This chapter gives a brief overview of the background work that forms the basis of this thesis. Section 2.1 describes FPGAs, their resources, and the computer systems in which they are deployed. The design and development of FPGA applications is explored in Section 2.2, then Section 2.3 discusses the use of coarse-grained FPGA architectures.

Section 2.4 describes memory-based computing, as this means of processing is one of the focal points of our new FPGA memory interconnect, introduced in Chapter 4. Finally, we describe two application areas within which we assess our work. Section 2.5 describes the processing of financial market data feed messages, and Section 2.6 details the layout and operations of neural networks, with a focus on their activation functions.

2.1 Field-Programmable Gate Arrays

FPGAs are reconfigurable semiconductor devices. In contrast to ASICs, whose functionality is defined at fabrication, FPGAs instead contain a mix of fine and coarse grained programmable logic elements. These elements can be configured, and reconfigured, after manufacture to implement arbitrary hardware circuits.

When categorising their performance and functionality, FPGAs are typically seen as sitting between ASICs and CPUs. Indeed, one of first use cases of FPGAs was for the design, testing, emulation, and verification of ASIC designs. However, the performance and flexibility of FPGAs have made them a recognised computing platform in their own right.

ASICs offer a dedicated hardware solution. This may be for a single problem, or within some domain—like audio, video, networking, etc. At the other end of the scale, CPUs instead enable software solutions by providing a defined set of instructions upon which applications can be built and run.

Rather than regarding FPGAs as the "third option", sitting between hardware-oriented ASICs and software-oriented CPUs, FPGAs should be viewed as an extension of each domain: hardware-augmented software, or software-augmented hardware. For hardware, FPGAs trade-off performance to provide more flexible designs, deliverable within a shorter time frame. While for software, an increase in development time and hardware requirements for FPGA applications can ultimately lead to higher performing, lower power applications, particularly when deployed at scale.

No single computing platform, CPU, FPGA, or ASIC, offers a universal solution to maximising performance. The *diversity* of application requirements—i.e., choosing to prioritise latency, throughput, power, area, etc—combined with the *diversity* of computing environments—from small IoT devices, to phones, laptops, and data centres—require a corresponding *diversity* in hardware. ASICs, CPUs, GPUs, FPGAs, Tensor Processing Units (TPUs) [36], and others all have roles to play.

2.1.1 Development Comparison

When it comes to application development, FPGAs boast a faster and cheaper development process compared to ASICs. ASIC designs must be extensively tested and simulated before final deployment as, once the chip has been fabricated, its functionality is unchangeable. FPGA development does not bear the cost of fabricating the design onto a dedicated chip.

The FPGA development process will instead see applications designed and tested with the same hardware, and in the same environment, in which they will ultimately be deployed. In this way, the FPGA development process aims to more closely align itself with the simpler design flow used by CPUs.

Such issues are of particular concern for ASICs when bugs or security vulnerabilities are identified. Spectre [37] is a recent example of a side-channel attack that exploits branch prediction mechanisms in Intel, AMD, and ARM microprocessors to read the private memory of other processes. As the development life cycle of CPU hardware is typically measured in years, addressing such hardware vulnerabilities can be very costly and time consuming.

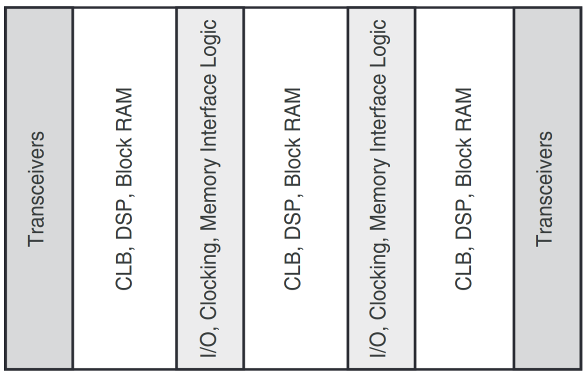
One of the main strengths of CPUs lie in their application development process. A well-defined and understood instruction set, ubiquity within the larger market, and a long history of development have resulted in a large volume and range of programming models, languages, and development tools. In comparison, FPGA development is less mature, and applications take longer to develop. Many different tools and approaches exist, and we discuss these in greater detail in Section 2.2.

The CPU's general-purpose approach to application design offers a greater range of application support and flexibility, but this comes at the cost of application-specific optimisations. For example, 16 or 32-bit applications running on a 64-bit processor do not require 64-bit arithmetic and logic operations, 64-bit registers, or the 64-bit data-paths these processors provide.

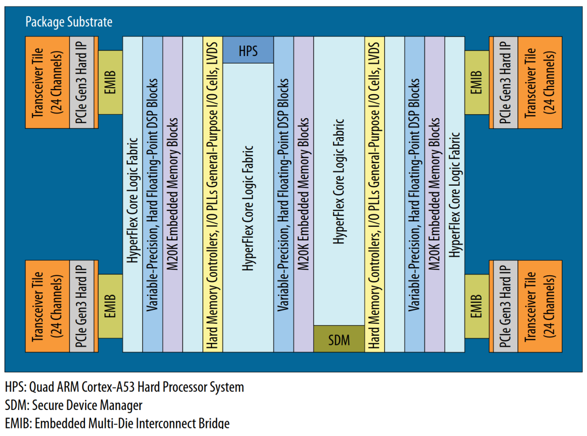
We desire both high performing applications, and a simple application development process. However, FPGA performance stems from customisation, while providing a simpler development process comes through increased abstraction. Resolving these conflicting desires is a large and active area of research.

2.1.2 FPGA Resources

There is diversity between computing platforms—FPGAs, ASICs, CPUs, etc—but also diversity *within* the computing devices themselves. This is perhaps most apparent with FPGAs. A single FPGA contains many different processing elements, memories, and interfaces, as well as a configurable interconnect that allows for a range of different routing schemes.



(a) Xilinx FPGAs [38].


Package Substrate
Transceiver Tile (24 Channels)
PCIe Gen3 Hard IP
EMIB
HyperFlex Core Logic Fabric
Variable-Precision, Hard Floating-Point DSP Blocks
M20K Embedded Memory Blocks
Hard Memory Controllers, I/O PLLs General-Purpose I/O Cells, LVDS
HPS
HyperFlex Core Logic Fabric
SDM
HPS: Quad ARM Cortex-A53 Hard Processor System
SDM: Secure Device Manager
EMIB: Embedded Multi-Die Interconnect Bridge

(b) Intel FPGAs [39].

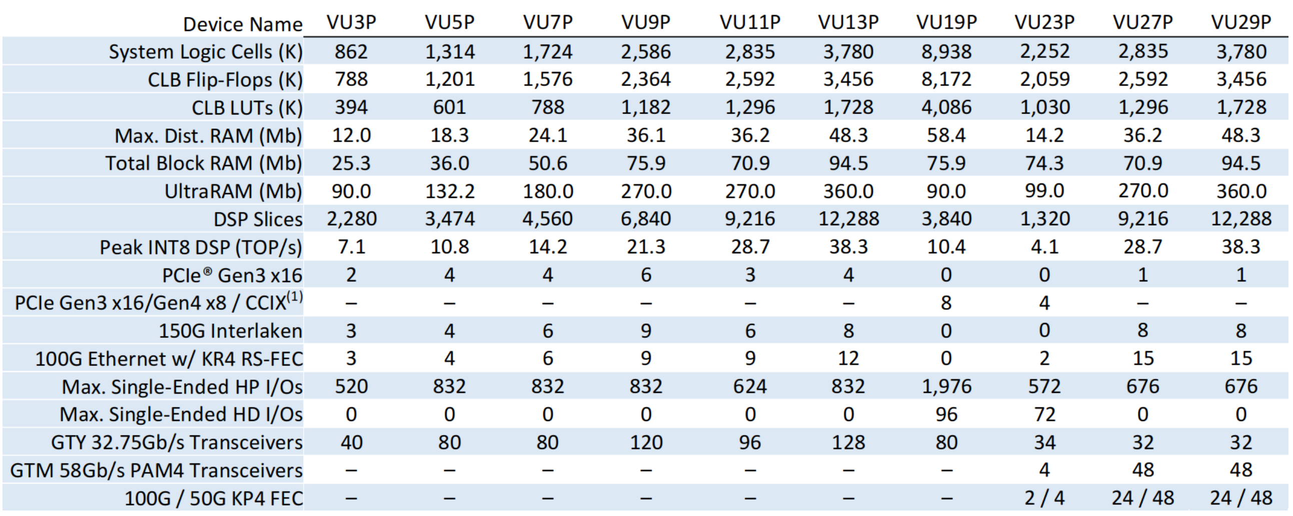
Figure 2.1: FPGA resource columnar layout.

This thesis looks to improve low latency applications within reconfigurable devices. As the fastest path through a system is not necessarily the shortest, an examination of processing elements is needed to better understand their benefits, drawbacks, and the functionality trade-offs we make when designing FPGA applications.

The two main FPGA manufacturers, Xilinx [38] and Intel [39], lay out resources within the FPGA using a columnar architecture. Fig. 2.1 shows examples of such a layout from both Xilinx and Intel. We can then identify 3 general sources of dataflow and processing heterogeneity: **(1)** the high volume, general-purpose reconfigurable fabric composed of elements such as LUTs, BRAMs, and DSPs; **(2)** the specialised, embedded cores and processor blocks that offer more rigid functionality, but with higher performance; and **(3)** the many dedicated I/O connections, and accompanying control logic that routes data into and out of the FPGA.

There is also heterogeneity between and within generations of FPGAs. Every new generation sees the release of multiple devices from FPGA manufacturers. Devices within the same FPGA family can offer widely differing types and quantities of processing elements.

Fig. 2.2 shows a selection of the resources present within Xilinx's 14nm UltraScale+ Virtex [40] range of FPGAs. Individual devices typically target specific markets—science [41], robotics [42], networking [43], etc—or specific domains—signal processing, low latency, high throughput, low power, etc. FPGA resources and interfaces are then included within the device to best address these requirements. This is another example of heterogeneity being a well-entrenched feature of FPGAs.



Device Name	VU3P	VU5P	VU7P	VU9P	VU11P	VU13P	VU19P	VU23P	VU27P	VU29P
System Logic Cells (K)	862	1,314	1,724	2,586	2,835	3,780	8,938	2,252	2,835	3,780
CLB Flip-Flops (K)	788	1,201	1,576	2,364	2,592	3,456	8,172	2,059	2,592	3,456
CLB LUTs (K)	394	601	788	1,182	1,296	1,728	4,086	1,030	1,296	1,728
Max. Dist. RAM (Mb)	12.0	18.3	24.1	36.1	36.2	48.3	58.4	14.2	36.2	48.3
Total Block RAM (Mb)	25.3	36.0	50.6	75.9	70.9	94.5	75.9	74.3	70.9	94.5
UltraRAM (Mb)	90.0	132.2	180.0	270.0	270.0	360.0	90.0	99.0	270.0	360.0
DSP Slices	2,280	3,474	4,560	6,840	9,216	12,288	3,840	1,320	9,216	12,288
Peak INT8 DSP (TOP/s)	7.1	10.8	14.2	21.3	28.7	38.3	10.4	4.1	28.7	38.3
PCIe® Gen3 x16	2	4	4	6	3	4	0	0	1	1
PCIe Gen3 x16/Gen4 x8 / CCIX(1)	–	–	–	–	–	–	8	4	–	–
150G Interlaken	3	4	6	9	6	8	0	0	8	8
100G Ethernet w/ KR4 RS-FEC	3	4	6	9	9	12	0	2	15	15
Max. Single-Ended HP I/Os	520	832	832	832	624	832	1,976	572	676	676
Max. Single-Ended HD I/Os	0	0	0	0	0	0	96	72	0	0
GTY 32.75Gb/s Transceivers	40	80	80	120	96	128	80	34	32	32
GTM 58Gb/s PAM4 Transceivers	–	–	–	–	–	–	–	4	48	48
100G / 50G KP4 FEC	–	–	–	–	–	–	–	2 / 4	24 / 48	24 / 48

Figure 2.2: Xilinx UltraScale+ Virtex FPGA resources. Source: Xilinx [40].

A more subtle difference lies not just in the resources present, but in the relative *ratio* of these resources. For example, Fig. 2.2 shows that the VU3P FPGA has about 6 DSPs per 1000 LUTs (5.79:1000), while the VU19P has fewer than 1 DSP per 1000 LUTs (0.94:1000). Resource ratios are an important consideration.

Ultimately, the FPGA will host and run an application, and this application will make use of one or many FPGA resources. The required ratios of these resources is highly application and implementation specific, and may not be known until the final design is coded and tested. Should an application require a 1:1 ratio of DSPs to BRAMs, for example, this leads to unused resources if this resource ratio is not exactly matched by the target FPGA. This results in lower performance, or may mean a design cannot be built at all should it require more resources than are present on the FPGA.

Problems then lie in both resource *heterogeneity*, and in *balancing* resource use across larger applications. Solutions have been proposed for transforming functionality between different FPGA resources. Moving functionality between LUTs and DSPs can be trivial for simple operations, while memory-based computing—explored in Section 2.4—can convert any function to a memory look-up. More complex function approximations can even see memory look-ups converted to DSPs [44].

Others have looked to tackle this problem at the development stage, and also within the FPGA itself via dedicated frameworks. We discuss this further in Section 2.2, and propose our own approaches in Chapters 4 and 5.

In this section we examine the FPGA resources themselves. We describe their functions and performance strengths, and also identify problems that may arise when looking to group and manage these disparate elements within a singular, unifying framework.

1: The Common Reconfigurable Fabric

Basic, configurable logic blocks form the backbone of most FPGA devices. In contrast to larger FPGA resources, such as embedded CPU cores, these resources are small, highly configurable and appear throughout the FPGA device in large numbers. There are 3 main types of common resource: **(a)** LUTs, **(b)** memories, and **(c)** DSPs.

As briefly described earlier, LUTs **(a)** are Look-up Tables that function as small memories, mapping an n-bit input to an m-bit output. When followed by a flip-flop to preserve this value, we are able to create synchronous circuits.

Current LUTs within Xilinx [19] and Intel [20] devices see the use of a 6-input LUT, for which a 6-bit input produces a 1-bit output. Other configurations are also possible, with multiple LUTs acting together within a larger configurable logic block able to produce a wider range of outputs. Intel's configurable ALMs have 8 inputs and can implement two 4-input LUTs, one 6-input LUT, or some overlapping combination of 3, 4, 5, and 6 -input LUTs if they share one or more inputs.

The exact mapping of a function or operation to LUTs is typically performed by compilers and build-time tools. While possible with low-level FPGA development flows, developers do not usually design applications with this level of specificity. LUTs typically implement control flow operations, like comparators, as well as simple arithmetic and logic functions.

Memory **(b)** exists within the common reconfigurable fabric as BRAMs and distributed memory. The latter repurposes LUTs as a memory resource, and is therefore faster than BRAM, but offers much less storage. Instead BRAMs are the more commonly used memory resource.

A BRAM's performance and flexibility come from their configurable width and depth. For example, Xilinx's 18kbit BRAMs [45] can store 512x36-bit data down to 16384x1-bit data.

Dual read ports are also a standard feature of BRAMs. These offer true, independent read and write access to data with single-cycle latencies. Such is the degree of customisation, developers are even able to specify the read and write policy should the same address be read from and written to memory in the same cycle [45].

Total BRAM capacities are typically small compared to external memory, such as DDR. Xilinx BRAMs offer 18kbits of storage, while Intel's M20K BRAMs [46] provide a comparable 20kbits. When larger on-chip memory capacities are needed, multiple BRAMs are connected together. In Chapter 4 we propose a new architecture for multiple parallel accesses to these many, small FPGA memories.

DSPs **(c)** are data processing blocks composed of a sequence of multipliers and adders. They offer high-performance implementations of one or more arithmetic operations, such as multiplication, multiply-accumulate, comparison, and concatenation. Data widths are configurable, but maximum widths exist for each operation. For example, Xilinx DSP48 blocks [47] see limits of 27x18-bits for multiplication and 48-bits for simpler logic operations.

The optional use of internal registers at different processing stages allows for the creation of high frequency, pipelined data-paths. While it is possible to replicate the functionality of DSPs with LUTs, DSPs are a dedicated hardware solution. They are therefore more power efficient, require less area on the FPGA, and can operate with higher clock frequencies. When allowed by the application, DSPs are then the preferred choice of processing element.

Finally, although not a processing element, the routing required to connect these elements should also be considered a resource. This is implemented as a 2D lattice within the FPGA. Horizontal and vertical wires are used to connect different FPGA resources, with switch boxes located at wire crossover points. These switches are configured as part of the final build process.

The relative placement of resources and the routing of data between them are two major factors in determining a design's attainable clock frequency. Smaller, less congested designs are therefore more likely to offer lower latencies and shorter build-times. In this case, higher performance stems purely from the fact that such designs allow greater choice of resource placement and data routing.

2: Specialised Embedded Cores

If the resources in the general reconfigurable fabric are able to implement the arithmetic, logic, memory, and control flow operations of applications, why then would we need additional specialised cores within an FPGA? The inclusion of these embedded cores within the FPGA serves two functions: to provide performance improvements within the FPGA, and to facilitate larger, external system design goals.

In the same way that DSPs improve on LUT implementations of functions, dedicated embedded cores offer higher performing, and more power and area efficient implementations of a function, domain, or entire application. Some functionality may be *required* within the FPGA, and it therefore makes sense to include this as a dedicated core. For example, the command, control, and processing logic for external interfaces, or the hash functions and/or error correction needed for data storage and high-speed communication.

From an external, system design perspective, the drive to provide a single System On Chip (SoC) solution naturally means the conglomeration of many heterogeneous elements. The general reconfigurable fabric of LUTs, memory, and DSPs will then serve these central elements, providing peripheral functionality such as control flow system, glue logic, or as simple FIFO buffers or registers.

If viewing FPGAs as a middle ground between ASICs and CPUs, then the inclusion of embedded CPU cores in FPGAs is a natural step. CPU applications can be more easily ported to FPGAs, and their functionality incrementally extended with new instructions implemented using FPGA resources.

Communications and signal analysis have been a long-term beneficiary of dedicated cores. These dedicated cores may be present to simply facilitate communication and control for Ethernet [48], DDR memory [16], or Peripheral Component Interconnect Express (PCIe) [49]. Cores can also tackle more specific applications like multimedia processing [12] or telecommunications [13].

More specialised cores are becoming common. The increasing popularity of machine learning is seeing dedicated cores included in FPGAs to better serve this domain. Xilinx's new Versal AI cores [50] are one such example, reporting throughputs of 3-17 TFLOP/s.

3: Dedicated I/O Connections

The final source of FPGA resource and operational heterogeneity we will examine concerns the many different types of connections into and out of the FPGA device itself. In the previous section we saw how FPGA device specialisation is seeing the inclusion of dedicated embedded cores. The processing of radio frequency or network data, for example, will then require the corresponding interfaces to be present on the FPGA.

As well as being used for their described purpose, the inclusion of these connections allows for new use cases, applications, and the re-imagining of existing resources and interfaces. Alachiotis [51] uses Ethernet, rather than the traditional PCIe interface to communicate between an FPGA and its host computer. In a similar vein, Gong [52] develops a new PCIe communication standard for FPGA-host communication, and provides it as an open-source library.

One example we explore with our new interconnect and memory architecture in Chapter 4 is the expansion of memory-based computing. This sees memory used as a processing element, rather than for data storage, and is examined in greater detail in Section 2.4. Memory-based computing designs have been explored in past works, such as table-based methods for function approximation [53], or when used as part of new computing strategies, such as distributed arithmetic [54]. We instead look to tackle the case of many parallel accesses being required of memory, for which memory interfaces can form a key hurdle.

The I/O connections play an important role in memory-based computing as we desire low latency access to memory. Our approach looks to BRAMs to solve this, but their low storage capacity can be a limitation. Larger memories are typically based on DDR-RAM, and therefore operate best when performing sequential reads.

Memory-based computing instead prioritises random read performance, and is much more sensitive to access latencies. Memory-based computing solutions will ideally place no conditions on the spatial or temporal relationship of read addresses, i.e., our "memory processing element" input data.

Multiple memory ports further increase access to stored data, improving parallel processing performance. The memory interconnect also plays an important role in determining performance, and in defining the practical limits of adapting memory to serve as a processing element.

Recently introduced High Bandwidth Memory (HBM) [16] [17] provides a new interface to memory, adapting the multi-bank architecture of DDR memory with an interconnect able to access each memory bank independently. Although promoted as a high throughput solution, this interconnect strategy brings a higher capacity for parallelism and opens the door for an expansion of memory-based computing approaches.

From this we can see that the number and type of dedicated I/O connections on the FPGA is an important factor in application design, and will continue to be so as new interfaces are added, and existing interfaces updated.

2.1.3 FPGA Deployment

We now look at how FPGAs are actually used within computing systems. They may serve a role as the primary means of processing within a system—as we saw earlier with SoC solutions—or FPGAs may function as application or domain specific accelerators, sitting alongside other compute devices like CPUs or GPUs.

Modern FPGAs are typically seen within **dedicated cards** or accelerators. These are connected to host computers via PCIe, and commonly have external network interfaces, such as Ethernet or InfiniBand. After configuration, FPGA processing may then be driven by data received via these external connections, or host-driven, with workloads determined by the system CPU. The FPGA is then just one processing element within the host computer, sitting alongside the CPU, and potentially a GPU or other accelerator.

In **cloud and computer cluster** environments, FPGAs are represented as compute nodes [55], with the same FPGA card configuration described above. This FPGA node is present in the larger cloud alongside CPU-centric and GPU-centric nodes. FPGA cloud services are available from a number of vendors, including Amazon [56], Alibaba [57], and Microsoft [58].

Direct communication with the FPGA may be possible, but this is typically achieved via the host computer. Dedicated, FPGA-only communication networks have also been explored. In our past work we find customised, inter-FPGA communication networks can result in lower latencies and reduced transmission protocol overheads [59] [60].

Within these clouds composed of CPU, GPU, and FPGA nodes, applications can make use of any or all of these computing resources. Models have been proposed to optimally partition workloads across these heterogeneous computing devices based on desired trade-offs between cost and performance [61]. Although adopting a multi-device architecture may seem like an overly complex design choice, it is in fact quite well suited to the growing trend of micro-service use within large cloud computing environments.

With a micro-service model, applications make use of existing, turn-key software solutions to common application problems. Examples include data storage and processing, user management, and network and infrastructure configuration. Forming a larger application from these micro-service building blocks means the services themselves can run on different processing devices, each chosen and optimised for that service.

This then brings us to the next area where FPGAs are deployed, the **infrastructure level**. More commonly known as edge computing [62], this decentralised approach places computing elements closer to their sources of data. This may be a database, memory, networking or some other subsystem or service.

Our work in Chapter 3 looks to exploit in-network processing as a means of serving multiple downstream applications. By pre-processing data as it arrives on the network, then forwarding this to one or more applications, we remove the need for this processing in the multiple, receiving applications. Additionally, by choosing to directly process network data, the FPGA design is

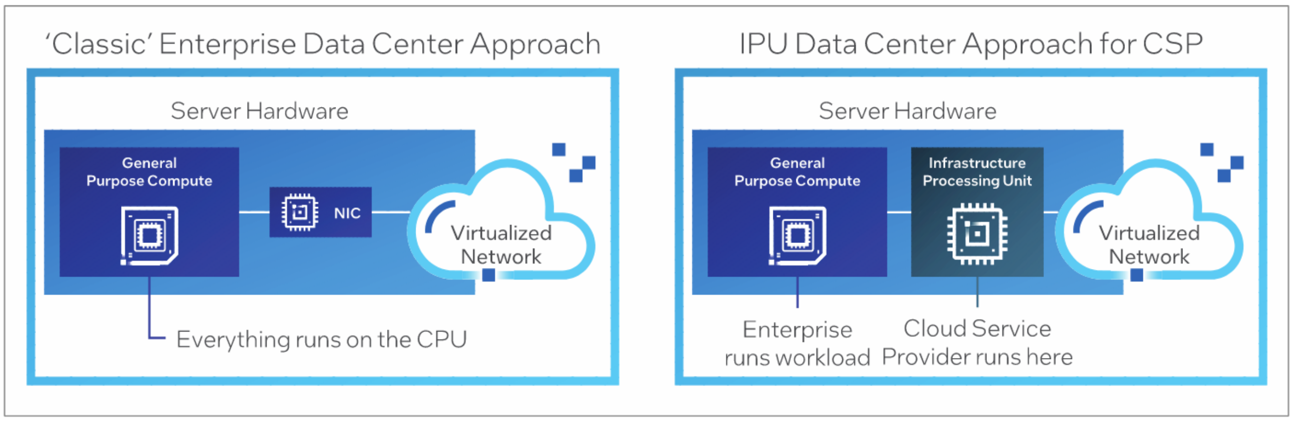

'Classic' Enterprise Data Center Approach
Server Hardware
General Purpose Compute
NIC
Virtualized Network
Everything runs on the CPU
IPU Data Center Approach for CSP
Server Hardware
General Purpose Compute
Infrastructure Processing Unit
Virtualized Network
Enterprise runs workload
Cloud Service Provider runs here

Figure 2.3: Intel's Infrastructure Processing Unit model. Source: Intel [63].

itself more efficient. This is because the FPGA's dataflow model of processing is an excellent fit to the existing pipeline flow of network data.

Such edge computing FPGA devices are common, with smart-NICs present in both data centres and clusters. Intel is also placing a growing focus on their own Infrastructure Processing Units (IPUs) [64]. Shown in Fig. 2.3, these CPU or FPGA based processors aim to **(a)** offload processing from traditional compute nodes, **(b)** provide lower latency application solutions, as well as **(c)** serve as a means of controlling data flow, and managing the complex infrastructure of large cloud applications.

When looking to improve low latency applications, the placement of FPGAs at the infrastructure level provides an excellent opportunity for quickly accessing incoming data. In addition, this placement provides a shorter transmission path should the device also output data, rather than simply routing it within the cluster.

The inclusion of additional, dedicated processing hardware means *all* computing devices will see less competition for their resources. Having to share processing time and resources between applications can lead to unknown, non-deterministic processing times. This creates difficulties when looking to return a result within a given time limit, or if we simply aim to achieve the lowest possible processing time. We explore an application for such in-network processing with the arbitration of financial market data feeds in Chapter 3.

2.2 FPGA Application Design and Development

In this thesis we target the improvement of low latency applications. This means not only improving the *latency* aspect of application performance, but also helping to streamline the overall development process. Even when prioritising low latency, applications must also tackle other application considerations, such as managing high degrees of parallelism, guaranteeing a given level of data accuracy or precision, and/or ensuring a functional, error-free design.

In Chapter 4 we describe an interconnect for low latency, multiple parallel accesses to memory. Chapter 5 extends this architecture to support any processing element with the creation of compute pools. Each pool can be accessed by multiple data-paths in parallel, with each pool performing a single function, e.g., trigonometric, probability, scientific, or some user-defined function.

How these new architectures are used in practice, and how they can be integrated into application designs, is an important consideration. This section looks at the programming models and tools used for developing FPGA applications. We seek to understand where application development problems may lie, and what form current and future solutions may take.

2.2.1 Design Flows

When creating a program for a general-purpose or domain-specific processor—such as a CPU, or GPU—we typically make use of a control flow computing model. With this model, the flow of data through the system is determined by the program itself. The processor will provide a set of instructions and a computer program is then formed as a list of instructions to be carried out.

For FPGA applications—defined as circuits in hardware—their decentralised nature, and support for high degrees of parallelism, mean a dataflow model is the more natural fit. The dataflow model is that of a directed graph, where nodes represent some processing element or operation to be performed, while edges represent the routing of data between these processing nodes.

```
y = A(x) + B(x);
if (y > 0)
  return C(y);
else
  return D(y);
```

Figure 2.4: Simple code example.

Colloquially, dataflow can be thought of as resembling an assembly line: processing elements are fixed and sequential, with the application data flowing from one processing stage to the next. Instructions are therefore inherent, i.e., if there is data present at a processing element then it should be processed by that element.

This assembly line approach still allows for branches in the code: given some condition, one pipeline is chosen over another. In the control flow model used by CPU applications, branches in the code come at a cost of wasted **time**: we don't know which branch will be taken so must either **(a)** delay work on the following operations, or **(b)** work on many possible future operations, discarding incorrect operations, and wasting CPU cycles.

In contrast, code branches in the dataflow model incur costs measured in wasted **space**, i.e., processing elements go unused when data are not routed down that path. This model can therefore be of benefit to applications that place a premium on latency: data processing costs traditionally paid in time can instead be paid in unused resources.

Given the simple code example in Fig. 2.4, for the control flow model, the functions A, B, C, and D are either provided directly by the processor, or are themselves composed of multiple processor instructions. This also applies to the branching comparison $y > 0$. For a dataflow approach there is no centralised processor, and each function is instead performed by an individual processing element.

The CPU provides a set of instructions, but not all available instructions may be used by an application. With the dataflow model, only the functions present in the application are present in the hardware, i.e., only A, B, C, D, and the branching comparator in Fig. 2.4. The downside of such a model is then the added complexity in developing applications: the developer must implement all functionality themselves, compared to it already being present within the CPU.

For general application development, this provides the freedom to include: **(a)** *only* the operations our application requires; **(b)** those operations *not* provided by the CPU; as well as **(c)** the opportunity to *customise* an operation's data precision, parallelism, latency, or any other parameter we value. However, this places a larger burden on the developer, as they must, again, define all operations themselves.

The use of libraries for common functions can help mitigate this, but the newly added emphasis on freedom of design, and support for customised hardware implementations, mean bespoke designs tend to be the rule rather than the exception.

2.2.2 Application Description and Development

A number of design tools and practices have been created to aid in FPGA development, both for control flow and dataflow FPGA designs. In this section we look at how programming languages and models aim to simplify the application creation process, as well as examining the FPGA design process itself.

In Section 2.1 we describe FPGAs as, essentially, a collection of resources. These resources can be configured and connected together to form processing elements. The granularity at which we define these elements—and with them, our application—differs depending on the development process. This begins with the programming language.

Hardware Description Languages

Hardware Description Languages (HDLs) operate at the Register-Transfer Level (RTL). Here, data are stored in registers, with data processing and control flow between registers defined using low-level, logical operations. The most commonly used languages are Verilog and the Very High Speed Integrated Circuits Hardware Description Language (VHDL).

From an electrical engineering perspective, HDLs may be considered a higher-level language since operations are defined at the logic level, rather than at the transistor level. However, when considered from the perspective of computer science, HDLs are very low-level languages,

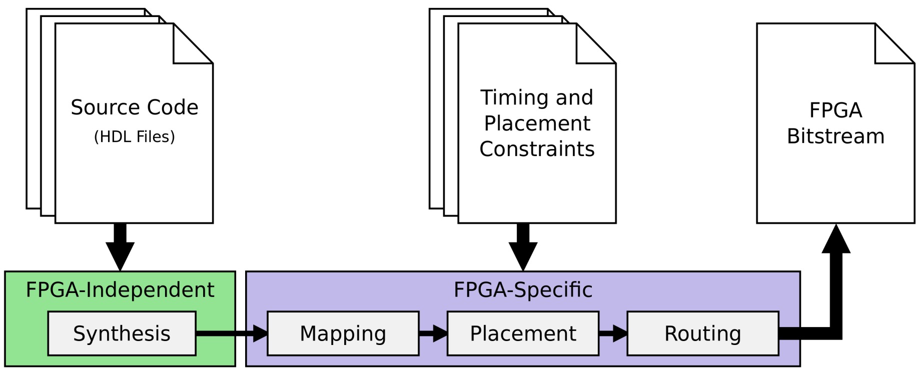


Figure 2.5: The HDL design flow.

operating well below the levels of abstraction of languages typically considered low-level, such as C. For example in VHDL, representing data more than 1 bit in length requires the use of the *std_logic_vector* data type. This must be imported from a common library as it is not a data type native to VHDL. In fact, arrays of any type require some degree of manual definition.

At this lower level we have more control when defining functions and managing data flow. This added control can, by itself, lead to performance improvements. For example, a logical shift of data by N places may take 1 or more cycles on the CPU. However, when defined in hardware, this same shift will take 0 cycles, requiring only that we connect the input wires to the output wires with an offset of N, setting the remaining output wires to zero.

This greater degree of control comes at the cost of a much longer and more complex development process. The inherently parallel nature of hardware-described designs can also make them very difficult to simulate and debug. Therefore, the current trends for HDLs promote code reuse, well established libraries of processing elements and functions, and for connection schemes using common interconnects, such as the Advanced eXtensible Interface (AXI) [65].

Tools that build FPGA applications, i.e., generate the final bitstream to configure the FPGA, typically require an HDL definition of an application. As such, higher-level compilers and code generators for FPGAs almost universally output HDL files, which then follow the existing build process.

Fig. 2.5 shows how the final FPGA bitstream is built from one or more HDL source files. Combined with timing and placement constraints, the logic-level application description is mapped onto the available FPGA resources, a physical placement for it chosen on the target device, then the routing between these resources is configured.

In general, when developing applications, the HDL development approach can lead to more efficient, higher performing designs, as well as a more comprehensive insight into the cycle-level flow of data within the application. This latter consideration is a critically important factor in the development of low latency applications.

However, the difficulty when expressing higher-level functions and design patterns can lead to not only longer development times, but also an increased risk of introducing bugs into the application. For financial trading applications in particular, this can be incredibly costly [66].

High-Level Synthesis

High-Level Synthesis (HLS) builds upon the HDL approach by providing a development flow that either: **(a)** translates existing, CPU-style code to an FPGA-compatible implementation; or **(b)** uses a new or existing programming framework to make FPGA development more in line with current CPU or GPU programming trends.

McFarland [67] presents a survey and discussion on the translation-based **(a)** approach, showing how data and control flow tasks are to be identified, broken down, and mapped into hardware. SPARK [68] and AutoPilot [69] are examples of HLS frameworks and languages that make use of this approach. They translate C code to lower-level, RTL code, in some cases needing minimal alteration of the initial, CPU-based design.

While code alteration is likely required for larger, more complex applications, the main challenge with a translation-based approach arises due to fundamental differences between higher-level and FPGA programming models. Higher-level programming models are typically sequential, while FPGAs are instead inherently parallel computing devices. It can be difficult to automatically adapt existing sequential code to a parallel model without loss of performance, or requiring some degree of additional, manual configuration.

To tackle this, translation-based languages, and most HLS languages in general, provide mechanisms for developers to specify areas of sequential and parallel operation. This helps compilers convert from the imperative, control flow model of the source design to the more parallel and dataflow approach of an FPGA application.

Rather than the creation of optimal, high performing designs, the goal of such translation-focused approaches is to help ease the transition of existing CPU-based applications to operate within the FPGA. Once a legacy application is operational within the FPGA, developers would use the newly provided pragmas and FPGA-specific processing elements to make the application's performance approach that of a native FPGA application. This would occur through iterative improvements, restructuring, and more specific code optimisations.

Instead of this translation-based approach, current development tools and trends focus on providing more general programming frameworks **(b)**. Here, an application is defined using a common language, such as C or Python, then its workload distributed among one or many processors. With this model, FPGAs are just one possible processing device, sitting alongside CPUs, and sometimes GPUs.

Examples include the widely used Open Computing Language (OpenCL) [70] [71] which originally targeted CPUs and GPUs, and has expanded to include FPGAs. There are also tools from the two main FPGA manufacturers: the Vitis [72] and Vivado [73] development suites from Xilinx, and Intel's Quartus Prime [74] and HLS Compiler [75]. Both manufacturers support OpenCL and HDL based development, along with tools to simulate applications, and perform power, area, and performance analysis.

Third-party companies either extend the existing manufacturer processes—such as the cloud-based FPGA platforms from Amazon [56] and Microsoft [58]—while others introduce their own development flows for their FPGA platforms, such as Maxeler's Java-based MaxCompiler [76].

When following this more general programming framework model, the FPGA is viewed as a more abstract processing device. Instead of defining processing elements explicitly, the language charges the device with performing a given function with some defined degree of parallelism. A vector multiplication of N elements by some constant value, for example, may be defined as a simple *for* loop in C, but realised on the FPGA with N parallel operations. If the same code were instead targeted towards a GPU, the compiler would make use of the GPU's SIMD, multi-core architecture.

For FPGAs, this approach sees the higher-level language—aided by the accompanying pragmas—being converted into lower-level RTL code to run on the FPGA. This code is typically Verilog or VHDL. Higher-level development tools and toolchains can then make use of the lower-level HDL development flow described in Section 2.2.2.

This allows for the use of existing HDL simulation and code analysis tools. More importantly, these higher-level approaches are then able to make use of existing, low-level libraries of common processing elements. Our above vector multiplication example could then simply be defined within the FPGA as a series of N DSPs. The alternative would see the HLS provider defining an entirely new processing element themselves which they must continually update and support.

HLS approaches still face the original underlying problem of high-level FPGA application development: that FPGA performance stems from customisation, while ease of application development arises from increasing abstraction. An easier development process therefore comes at the cost of a lower performing design.

BDTI [77] find that the conversion process from C to lower-level HDL can be highly efficient, but studies by Winterstein [78] and Nane [79] highlight the superior performance of manually generated HDL code. Nane notes that this is a function of both the tools and developers, concluding: "*software engineers need to take into account that optimizations that are necessary to realize high performance in hardware (e.g., enabling loop pipelining and removing control flow) differ significantly from software-oriented ones (e.g., data reorganization for cache locality).*"

As perfection is an unrealistic goal, we must instead strive for an optimal *balance* of development considerations. Combining higher-level function definitions with additional lower-level pragmas to boost performance, has shown to be both a practical and effective choice. With a growing developer community, a greater adoption of combined CPU/GPU/FPGA design flows, and new improvements to compilers and common libraries, we are seeing the continued improvement and simplification of the FPGA development process.

Development Tools Within This Work

As part of our work, we make use of Xilinx's Vitis and Vivado design tools. Vitis is primarily an HLS development environment, while Vivado focuses on HDL applications written in Verilog and VHDL. Vivado is our main development environment, as we create low-level architectures and therefore need greater, fine-grained control of the design. The built-in simulator and library of IP cores also aid in the overall development process.

In Chapters 4 and 5 we develop architectures that are designed to be used by higher-level tools and processes. Chapter 4, sees the creation of an interconnect facilitating many parallel access to FPGA BRAMs, and provides an automated tool to open up this new architecture to other developers.

This tool translates a Python function into a memory-based computing module that employs our new architecture and interconnect. In line with the general programming frameworks **(b)** described above, we provide an additional configuration file for the developer to describe their desired data types, widths, and precision.

After processing and converting the given input code, our automated tool, like most HLS, outputs HDL code in the form of Verilog files. This makes it a simple matter to incorporate this common format into existing HLS toolchains. We describe this tool further in Section 4.5.

2.3 Coarse-grained FPGA Architectures

Section 2.1.2 described the many different types of resources present in an FPGA: **(a)** the common, reconfigurable LUT, BRAM, and DSP resources; **(b)** the specialised processing cores and memories; and **(c)** the configurable interconnect for routing data between them. Rather than dealing with these resources individually, a more coarse-grained architectural approach is also possible.

When used within an FPGA, coarse-grained architectures [80] group multiple FPGA resources together into larger, higher-level processing elements. These architectures may be referred to as overlays, frameworks, or sometimes virtual FPGAs. The central idea of this approach is to define new functional units—i.e., higher-level processing elements—using this coarse-grained methodology, and use these as the fundamental building blocks upon which to build an FPGA application. The newly defined functional units—or sometimes, groups of these units—now serve as our quantum of processing.

In addition to these new functional units, a simplified, common interconnect is developed to link them together. Such schemes include the establishment of communication protocols and topologies, and also define the width and format of communicated and processed data. In the same way functional units build upon the basic FPGA resources—LUTs, BRAMs, DSPs, etc–to define a new processing element, these new communication and interconnect operations will similarly use the FPGA's built-in routing framework to create the new interconnect.

The simplest example of such a coarse-grained architecture is that of a CPU soft core. This seeks to build a traditional control flow CPU using the FPGA's LUT, BRAM, and DSP resources. The new functional unit of this architecture will then be the CPU itself, or perhaps the individual execution units that make up the ALU. This new platform, aka the CPU core, then processes data according to its established parameters. For example, with a standard 16-bit or 32-bit data width, or using a Very Long Instruction Word (VLIW) instruction set. Applications designed for this new CPU core will then also be radically different to that of the traditional FPGA development process. We discuss this further in Section 2.3.3.

2.3.1 Functional Units

Within a coarse-grained architecture, the newly defined functional units may be composed of one or many FPGA resources. Past works have seen functional units constructed around a single resource type. ZUMA [81] makes use of LUTs with the aim of developing applications using a newly defined, higher-level LUT. DeCO [82], as well as the overlay in [83], instead choose the DSP block as the focal point of their functional units.

Another option sees functional units containing a mix of common resources—LUTs, BRAMs, and DSPs—in some pre-determined ratio and connection configuration, such as FPCA [84].

TILT [85] uses these elements to form functional units with simple processing cores that implement a VLIW instruction set. Application-specific functional units are also explored [86].

Rarely are functional units formed from resources other than LUTs, BRAMs, and DSPs. These resources are both highly configurable and present in large numbers on FPGAs, so this is a logical choice. However, such coarse-grained architectures will struggle to make use of all FPGA resources. If specialised, embedded processing elements—such as embedded CPU cores or domain-specific processors—are available in the FPGA, but not included in the functional units, then these processing elements are wasted.

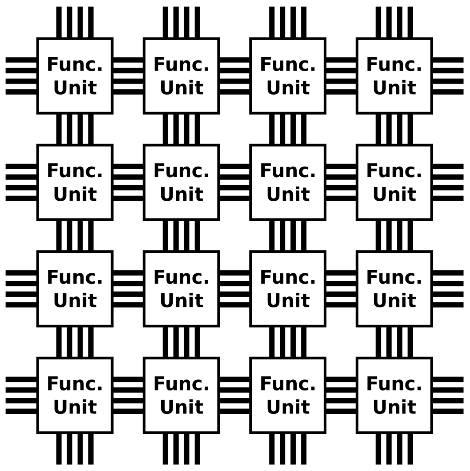
Resource utilisation *within* functional units is also an issue. For example, if an FPGA application built upon a new coarse-grained architecture requires 10 LUTs for every DSP, i.e., a 10:1 ratio, then this will result in wasted LUTs when functional units have a 20:1 ratio of LUTs to DSPs. Similarly, a 5:1 ratio of resources within the functional units will see wasted DSPs.

The functional unit's design can also place limits on functionality. The DSP-centric units in DeCO [82] fix data widths at 16-bits, while FPCA [84] fixes this at 32-bits. We then lose the ability to exploit the FPGA's support for arbitrary data width storage, routing, and processing.

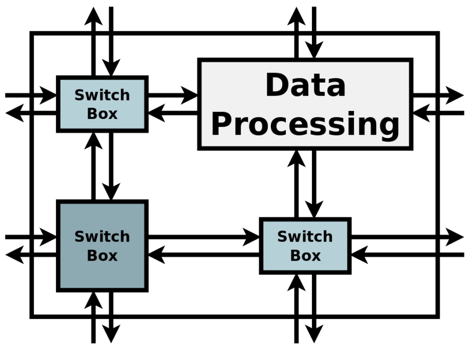
In Chapter 5 we introduce our own coarse-grained architectural approach with new, function-specific compute pools. Each pool may contain multiple processing elements, similar to that of a functional unit, but our units are homogeneous in *function*, not resource composition. A multiplication compute pool, for example, will only carry out the multiplication function. Within the pool, the one or more processing elements that perform this function may be composed of LUTs, BRAMs, DSPs, specialised cores, or any other FPGA resource.

More than one multiplication compute pool could be defined within the FPGA. However, if defining just a single pool, our minimal data routing scheme reduces the bottleneck associated with multiple data-paths accessing a single, centralised pool, and can almost eliminate it entirely for designs with known latencies.

With this approach we aim to overcome the limitation of rigidly defined functional units, while still offering a higher-level view of application development.



(a) 2D grid connecting functional units.



(b) Internal functional unit connections.

Figure 2.6: An example of a coarse-grained architecture interconnect.

2.3.2 The Interconnect

A new coarse-grained architecture also requires the creation of an interconnect to route data between the newly defined functional units, or groups of functional units. This interconnect will typically be of an island, or grid configuration, with neighbouring nodes connected via switch boxes. Connections within the functional unit may have different topologies or degrees of complexity, depending on the resources present. An example of a simplified coarse-grained interconnect is shown in Fig. 2.6.

When building an FPGA application, finding the best paths for routing data between resources can be costly in both time and processing power. The use of a higher-level, pre-built and routed interconnect can greatly reduce these costs. For combined place and route times, intermediate fabrics [87] averaged a 554x speed-up across 12 case studies, with a maximum speed-up of 2407x, but at the cost of a 30-40% area overhead.

Grid topologies, like that in the example in Fig. 2.6a, are common as they mimic the existing crossbar routing of the FPGA. However, the use of a higher-level fixed topology will create problems for applications that do not match its configuration. A 2D grid or ring topology is ill-suited to more centralised applications or communication patterns. A star topology will be similarly inefficient within a purely sequential, dataflow pipeline.

Again, we see the use of a more rigid framework comes at the cost of design customisations and performance. It is then important that the correct coarse-grained architecture is chosen to match the operation and performance goals of the target application.

2.3.3 Application Development and Tools

The use of a new FPGA architecture necessitates the creation of new development tools and processes. Existing development, simulation, and performance analysis tools may be incompatible, so developers of any new coarse-grained architecture must provide these tools. Some architectures, such as TILT [85], are designed with existing development and programming models in mind. In the case of TILT, which implements simple CPU-like cores, they follow the model of OpenCL where data workloads occur across many parallel threads. Existing tools and design flows can then be reused.

One advantage of adopting a higher-level architectural approach is the improved ability to simulate and test FPGA applications. With HDLs, application testing is done using waveforms, which track the bit-level movement of data across the entire FPGA. This can be very computationally intensive, and also a time-consuming task for the developer. Higher degrees of abstraction mean fewer processing elements to simulate, reducing the computational burden on the simulator. This also hides low-level data movements from the developer, making the design process more akin to software development rather than that of digital hardware, lowering the barrier of entry for FPGA application development.

A major disadvantage however is the need to fundamentally rewrite existing applications for these new coarse-grained architectures. This can include even the most basic, common library functions. Bespoke solutions that address application-specific problems may also no longer be possible. Examples of these include table look-up methods that combine memory and processing elements, like distributed arithmetic [54], or communication protocols and topologies that optimise for specific network traffic patterns, or for the access of sparse memory structures.

An advantage of coarse-grained architecture development is that many aspects of the initial application development process have already been carried out. The coarse-grained architecture can provide a common communication protocol and interconnect, and may offer guarantees of data processing throughput or latency. The new functional units themselves may also offer new higher-level, optimised, or domain-specific operations.

2.3.4 Summary

In grouping together FPGA resources into new functional units, coarse-grained architectures provide a higher-level, abstracted view of the FPGA. While this can reduce or remove some of the customisation provided by the FPGA, these types of architectures offer a new approach to the high-level design and development of FPGA applications.

Application development within coarse-grained architectures can see boosts from higher-level implementation and simulation. This reduces the need for more low-level and manual processes. However, we must depend on the architecture creator providing these tools and functionality.

When part of a full and more established ecosystem, development times can be improved, but, because of our dependence on the architecture provider, we may find that less used, or more bespoke, architectures may suffer from lack of support. Application developers would then need to spend additional time converting designs to operate within the new coarse-grained architecture.

From our examination of coarse-grained architectures, we find the main aims for new, higher-level architectures are then: **(a)** a higher-level view of processing elements within the FPGA; **(b)** minimal constraints and limitations on resource use, i.e., allow the developer to make use of all FPGA resources; and **(c)** the ability to integrate the new architecture into existing development flows and toolchains. We look to tackle these issues with our new interconnect and compute pool architecture, outlined in Chapter 5.

2.4 Memory-based Computing

Rather than using conventional processing elements, memory-based computing implements functions through the use of memory look-ups. A function, $y = f(x)$, is represented in memory

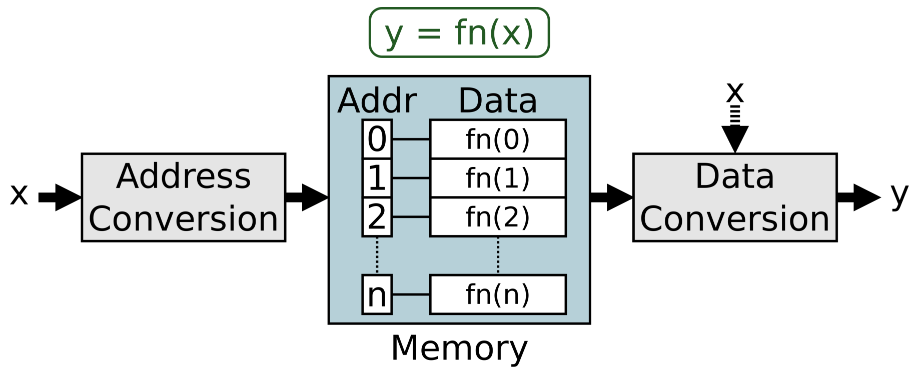


Figure 2.7: General approach for representing the function $y = fn(x)$ in memory.

by converting the function input, $\boldsymbol{x}$, to a memory address, then storing the pre-computed function result, $\boldsymbol{y}$, at the corresponding memory location. A user is then able to “call” this function by reading out the result from memory.

The goals of memory-based computing are similar to function approximation schemes, but with a greater focus on exploiting the features and opportunities present in memory. These include: multiple, parallel access ports; Non-uniform Memory Access (NUMA) hierarchies; and end-to-end processing latencies determined by memory access times, not function complexity.

Converting memory to act as a processing element can also make use of otherwise unused memory resources. This is of particular benefit to compute-bound operations. Alternatively, employing a memory-based computing architecture can free up conventional processing resources—such as LUTs, DSPs, or embedded cores—for use in other tasks.

A generalised example of a memory-computing architecture is shown in Fig. 2.7. From the figure we see that, compared to traditional computation, improvements and optimisations arise from 3 areas:

- how we map function inputs to memory addresses (Section 2.4.1);
- the way we represent the function result in memory, and the time and logic required to restore this data to its original format (Section 2.4.2);
- the composition, implementation, and hierarchy of the memory itself (Section 2.4.3).

2.4.1 Mapping Function Inputs

Given one or more inputs to a function, memory-based computing must first convert this function input into a memory address. The scheme used can be a straightforward, direct-mapping of function inputs to addresses, or we may choose to employ any number of range reduction methods. Reducing the range of possible inputs is desirable as this subsequently reduces the number of function results we must store in memory.

This initial step may go further and return results for simple or known inputs. For example, when one input to a multiplication operation is zero the output will also be zero, so no memory look-up is needed. This may increase routing complexity in hardware, but can see significant memory storage savings for functions with a wide range of inputs, but a small range of outputs.

In this section we give a brief overview of input mapping approaches.

Direct-mapping

Direct-mapping is the most straightforward and computationally efficient approach to function input mapping. Here, we simply present the n-bit function input to the memory as an n-bit integer address. Should there be multiple function inputs, i.e., $y = f(x_0, x_1, ..., x_n)$, all inputs are simply concatenated.

This method is the easiest to implement in hardware and also the fastest, requiring 0 cycles to map input values. While other schemes may require 1 or more cycles to check or convert inputs, direct-mapping will simply route input data through as a memory address.

Custom Mapping

More complicated, bespoke mapping schemes are also possible. The goal remains the same: to map a function input to a memory address. However, a more in-depth understanding of the target function may provide range reduction optimisations, reducing the amount of memory required to store function results.

Other than limitations due to data width, it is typically the operation and limitations of the function—rather than the memory hardware—that will play the biggest role in determining the efficacy and efficiency of the input mapping.

One example is the case where different inputs produce the same output data, such as arithmetical multiplication, where $X \times Y = Y \times X$. For such 2-input functions we can almost half the amount of memory required.

Periodicity within the data can also be exploited. A simple example is $sin(x)$, which is periodic about $[0,\ 2\pi)$. We would then only need to store results for inputs within this range. Input values outside this range would be converted by reducing them modulo 2π.

Conventional hashing functions are typically not suitable. These are primarily designed to map arbitrarily sized data to a fixed data width. While they aim to minimise collisions, a 1:1 mapping is not guaranteed, even when the input and output data widths are the same. For input mapping within memory-based computing, this means different function inputs will map to the same address, returning an incorrect result.

Our choice of mapping function must also consider that it must be implemented in hardware. This mapping must be performed every time we address memory, thus requiring both resources and extra computing cycles. While a robust mapping step can reduce the number of function results we must store in memory, the latency it adds will reduce the performance gained from a memory-based computing approach.

2.4.2 Data Storage and Representation

The way we represent function results in memory also offers chances for performance optimisation. Function results for all possible inputs are pre-computed at build-time, then stored in memory. This stored data can be of any type or format. For example, while a function may operate on 16-bit floating-point numbers, we have no obligation to store data in memory using this format.

As we are not actually performing mathematical operations, just memory reads, the memory address (function input) and memory data (function output) formats can be selected independently.

Data compression is an obvious first step. This helps mitigate one of the main downsides of memory-based computing: the large amount of memory required to store large functions. The data conversion unit in Fig. 2.7 would serve this role. Like the address conversion step mentioned previously, this data conversion operation will require additional resources and processing cycles. A reduction in memory storage then comes at a cost of this reduced performance.

The data conversion stage can also make use of the input $\boldsymbol{x}$ to modify the retrieved data, providing a lightweight means of exploiting data patterns. For example, with $X \times Y$, the final result can be negated should one of the two inputs be negative. Such mirroring will be present in periodic functions, and also within certain ranges of non-periodic functions.

In addition to compression and exploiting patterns in data, the use of different data *formats* is another interesting proposition. Instead of simply reducing our memory footprint, we can exploit arbitrary data formats to enhance the precision and operation of the function itself.

Arbitrary Data Formats

We can hard-code our desired data format during the pre-computing step, i.e., when we initially calculate all the function results to be stored in memory. This is a build-time operation, so additional analysis and problem-solving steps are also possible. Conventional computing methods must recognise and handle problem inputs and edge cases at run-time, but we can address these during the pre-computing stage.

A common problem with floating-point numbers is recognising and accounting for invalid numbers. FloPoCo [88] uses 2 extra bits to encode for exception numbers—zero, infinities, and Not a Numbers (NaNs). This is used to reduce the logic burden when identifying these cases at run-time. Memory-based computing lets us identify these situations at build-time, without the run-time performance penalty, or the overhead from the additional bits.

In addition, for floating-point numbers, their validity and precision can be guaranteed. With memory-based computing, the run-time computation of such numbers mean we will return consistent and faithfully rounded results [89].

Arbitrary data formats also let us introduce control flow operations into memory look-ups. As an example, consider converting the Gaussian function to a memory-based implementation. Image processing Gaussian filters [90] [91], LaPlace of Gaussian, and Difference of Gaussians in object recognition [92] take only a handful of precise values for the standard deviation.

$$\text{Gaussian}(\mu, \sigma, x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{(-\frac{(x-\mu)^2}{2\sigma^2})} \tag{2.1}$$

The Gaussian probability density function, defined in Eqn. 2.1, takes three inputs. The first of these, the mean, μ, and standard deviation, σ, define the shape of the distribution, while the final variable, x, is the value we are interested in finding within this distribution. If our application uses only $\sigma \in \{1, 2, 3, 4\}$, for example, then we would only store results for these distributions, rather than all possible values of σ.

For this example, the standard deviation can then be replaced by a 2-bit toggle, t, for selecting which of the pre-computed standard deviation value to use. In this way, control flow is optimised and represented entirely within the memory itself.

In a similar manner, discontinuous functions benefit from this combined operation/look-up table setup. For such functions we can hard-code the function's responses to discontinuities, such as those of jumps or divisions by zero.

Such hard-coded responses can then be application specific. While the result of a division by zero is undefined, the zero-value input to the function may originate from a sensor. From our understanding of this particular system, we may know, for example, that such values instead represent "a small value" rather than a literal zero, and can select the appropriate output value accordingly.

2.4.3 Memory Architectures

The memory architecture we use when storing function results in memory provides another area to explore for performance improvements. We have seen that implementing a function in memory involves mapping function inputs to an address, then returning the function results from memory. Function “performance” is therefore independent of our implemented function, and dependent on the performance of the memory itself.

In this section we look at how memories are formed within the FPGA, the role memory read and write patterns can play in enhancing performance, and how memories can aid in computation by working alongside traditional processing elements.

Memory Composition Within an FPGA

Within an FPGA, LUTs are memories that translate a given n-bit input into a m-bit output. These small memories serve as the backbone of the FPGA’s reconfigurable fabric, and are bundled into logic blocks alongside other resources. Due to their limited storage capacity and close-knit integration with other resources, we will instead only be considering the larger, dedicated memory resources.

FPGAs contain a number of memory resources, with the two main vendors, Xilinx and Intel, offering similar types of memory. Xilinx provide LUT-based, distributed memory [93], BRAM [45], UltraRAM [15], and HBM [17]. Intel devices have similar LUT-based, distributed memory [46], BRAM [46], and HBM [16] resources. Direct interfaces to DDR memory are also commonly available from both vendors.

A user-defined memory is created within the FPGA by grouping together one or more of these memory resources. BRAMs are the best memory type for the task as they: **(a)** are the most versatile, having configurable address and data widths; **(b)** are present in large numbers and spread throughout the FPGA; and **(c)** do not have the operational overhead of dynamic memory, which must periodically refresh its memory contents.

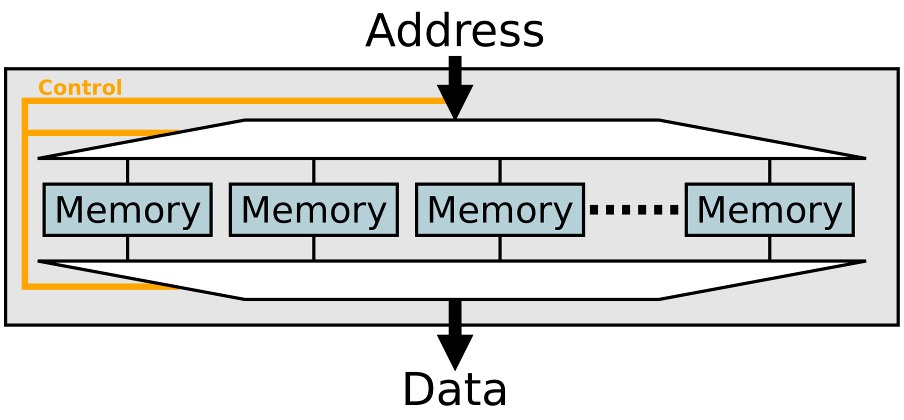


Figure 2.8: Forming a user-defined memory by grouping many, smaller memories together.

Fig. 2.8 shows an example of forming a user-defined memory by grouping together a collection of memory blocks. For a user memory with a capacity of C bits, formed of B-bit memory blocks, we will require $\lceil \frac{C}{B} \rceil$ memory blocks. For example, using Intel's M20K BRAMs with 20kbit storage, a user memory of C=1Mbit thus needs $\lceil \frac{1M}{20k} \rceil = 52$ BRAMs.

While before we had B individual memories, the newly constructed user memory is now accessed as one single unit. This offers a standardised, straightforward approach to working with FPGA memory, and helps simplify application development. However, we replace the previous B memory interfaces with a single interface, reducing opportunities for parallelism.

While understandable as a general approach, this configuration removes the option for application-specific optimisations and performance improvements. In Chapter 4 we introduce a new architecture for forming larger memories from BRAMs while maintaining their independent, parallel accesses. We target memory-based computing, and demonstrate improved performance for highly parallel, low latency applications.

Data Coherency

Memory-based computing "calls" a function by reading out a pre-computed function result at run-time. Therefore, only memory read operations are needed at run-time. Memory is written once, during the initial configuration, when function results are loaded into memory. We therefore avoid issues relating to Read After Write (RAW) and Write and Read (WAR) dependencies. When using multiple memories as part of some larger memory architecture or hierarchy, we also do not face issues with caching stale data.

As data are only read, never written, memory-based computing therefore requires very few management, arbitration, or control mechanisms. For example, there is no need to support or enforce write-through or write-back policies within memory hierarchies. Local or distributed caching schemes can also be implemented without the cache line flags or controllers that ensure cache coherency.

Hybrid Architectures

When adapting functions to a memory-based computing approach, we do not need to convert all of the function operations to memory. Functions whose computation are based both on traditional processing elements and memory-based look-ups, have seen a lot of success.

Distributed arithmetic [54] is a good example of such a hybrid design. For Multiply-Accumulate (MAC) operations where one of the input vectors is a constant, distributed arithmetic sees the operation performed bit-wise, rather than element-wise. Multiplication is transformed and performed using memory look-ups, with a final add and shift operation carried out using traditional processing elements.

More important than the elimination of the cycle and area costly multiplication operation is the change in how the MAC scales with larger inputs. With the hybrid memory approach of distributed arithmetic, the MAC operation now scales with the *width* of the input data, rather than the *number* of N elements in the input vector.

Conventional MACs with N elements require $N-1$ operations to compute the result, assuming a combined multiplication and addition operation (FPGA DSPs provide such functions). By scaling with data width, distributed arithmetic on 16-bit data requires 16 cycles for N elements, while 32-bit data would require 32 cycles for N elements. Distributed arithmetic then provides a faster choice when the data width is less than the number of N input vector elements.

Table-based methods are another example of this hybrid approach. In recent work, Barbone [94] uses cubic spline polynomial interpolation to implement functions on FPGAs using a mixture of processing and memory resources, reporting orders of magnitude reductions in resource utilisation compared to direct implementations.

Architecture Summary

The composition, layout, and application-specific use of memory can play an important role in improving performance. Converting a function, or aspects of a function, to memory presents new opportunities for optimisations and can also act to *change* the nature of the problem we face. We explore this further in Chapter 4 with the introduction of our new memory architecture.

2.4.4 Performance Improvements

The final aspect of memory-based computing we examine concerns performance improvements that arise from memory operations and configurations. While we touch on some of these areas in this thesis, this section serves mainly as an examination of future improvements that may be applied to memory-based computing architectures.

Multi-pumping

Within an FPGA, the maximum operating frequency of a memory resource will typically be higher than that of the full application. This is a common occurrence as the full FPGA application must contend with many different processing element, routing, interface, and design considerations. Larger designs in particular are therefore unlikely to reach the maximum operational frequencies achievable by even the slowest FPGA memory. This is typically BRAM, with Xilinx block memories having a maximum operating frequency of 450MHz, while Intel's M20K achieve 600MHz.

One solution for utilising the higher operating frequency of memory is multi-pumping. This method first isolates memory into its own clock domain, operated at the memory's maximum clock frequency. A memory controller then sits at the new clock boundary and manages the movement of data—often asynchronously—between memory and the application logic.

Data may face a latency penalty when crossing clock boundaries due to the different clock characteristics in each domain. Solutions exist to hide domain crossing latency by pre-computing state machine transitions in designs with unknown clock relationships [95]. However, this can be reduced, or eliminated entirely, when using clocks **(a)** derived from the same source, and **(b)** with frequencies that are rational multiples of each other [96].

In a scenario where user logic operates at 200MHz, and a dedicated memory clock domain operates at 400MHz or more, the ideal case would see memory able to serve 2 requests within the space of a single 200MHz cycle. From the perspective of the user logic, we therefore effectively double the number of memory access ports. In addition to reducing latency, increasing the number of memory ports is one of the primary means of improving memory-based computing performance.

Dynamic Reconfiguration

We discuss in Section 2.4.3 that, at run-time, memory-based computing is a read-only operation. The memory is only written once, during the initial configuration step. While run-time writing of memory is not performed with memory-based computing, it is still possible. Such operations are then analogous to dynamic reconfiguration, that is, altering the application's functionality at run-time.

Memory-based dynamic reconfiguration can be faster than conventional approaches. The Xilinx Internal Configuration Access Port (ICAP) interface is 32-bits wide and operates at 200 MHz for older devices, and 125MHz for newer devices using Stacked Silicon Interconnect (SSI) technology [97]. This leads to maximum theoretical throughputs of 800 MB/s and 500 MB/s respectively. The latter, slower interface is in use by Xilinx's current UltraScale+, and previous UltraScale architectures [98].

In contrast to conventional dynamic reconfiguration interfaces, the memory-based approach is limited by the maximum performance of memory. For example, Xilinx's BRAM with a maxi-

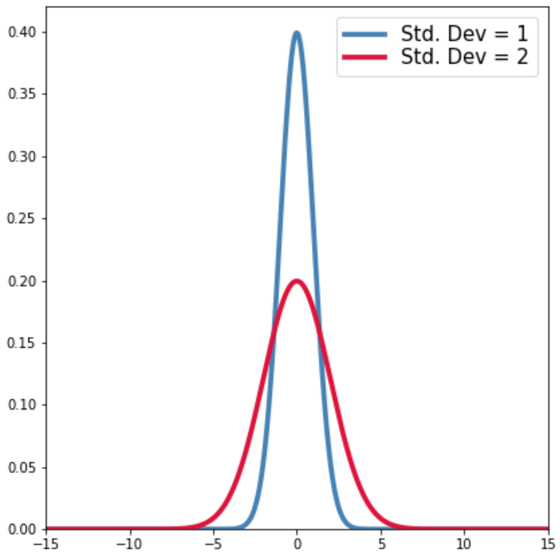


Figure 2.9: Example Gaussian probability density function with different standard deviations.

mum data width of 36-bits, 2 parallel access ports, and operating at 450MHz, has a maximum throughput of $36 \times 2 \times 450M = 4.05GB/s$. On the latest Xilinx Alveo U280 accelerator card, this memory-based approach translates to an 8.1x speed-up over the ICAP interface.

It is difficult to directly compare approaches, as performance depends on the relative size of computation logic vs. the amount of stored data. The same BRAM performance for a 16-bit function, rather than for the theoretical 36-bit one, results in a throughput of 1.8GB/s. However, when considering configuration interfaces, memory-based computing will tend to see noticeably higher throughput.

Memory-based reconfiguration can also offer much finer-grained control of the reconfigured functionality, as we can reconfigure at the sub-function level. For example, consider the Gaussian probability density function defined in Eqn. 2.1. Here, very large and small values of the input variable, x, result in the function output tending to zero. The example in Fig. 2.9 shows that if we reconfigure the hard-coded Gaussian(μ, σ=2, x) function to become Gaussian(μ, σ=1, x), then the new function will share the same results for very large and small inputs. These memory entries will then not need to be updated, reducing reconfiguration time.

A number of dynamic and partial reconfiguration methods have been explored [99]. Memory-based methods could see similar benefits.

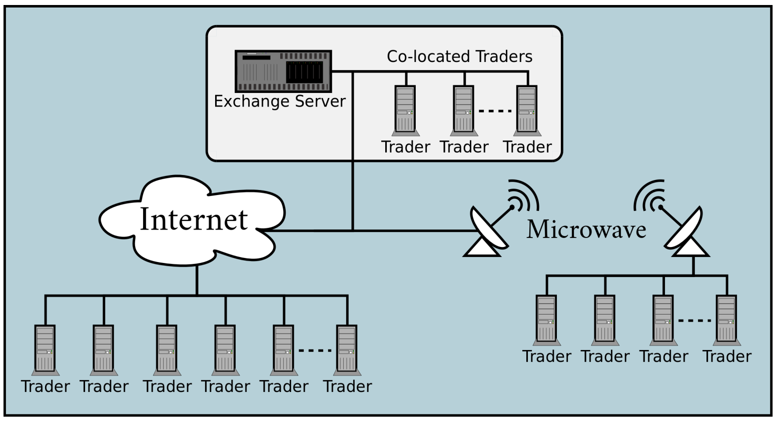


Figure 2.10: An example network linking a financial exchange with its members via the internet, a dedicated microwave network, and co-location.

2.5 Market Data Feed Processing

One area of low latency applications we will be exploring is the processing of financial market data feeds. These feeds are transmitted by financial exchanges as a stream of messages, informing recipients about the current state of the market. Messages describe market events, such as available and completed trades, as well as information about the state of the exchange itself [100].

Financial institutions subscribe to these feeds to stay informed about current market prices and events. These help inform the actions of traders and trading applications, and are therefore both important, and highly time-sensitive.

This latter time element is a critical component of High-Frequency Trading (HFT) applications [101]. These seek data receipt, analysis, and trade execution all within sub-millisecond time frames. Computing hardware is therefore often located as physically close to financial exchanges as possible to minimise latency. This co-location sees the trading computers of clients located within the same premises as the exchange itself.

Where this is not viable, dedicated transmission networks may be used. Multiple line-of-sight microwave networks exist to transmit low latency messages between financial exchanges in Chicago and New York [102], exploiting the fact that light travels faster through air, compared to the glass of fibre-optic cables. Within this domain, low latency is the primary consideration.

Fig. 2.10 gives an example of a simplified network linking a financial exchange with its clients using the schemes described above. Future trends will see the creation of new low latency transmission networks using space-based communications, provided by companies such as SpaceX [103] and OneWeb [104].

2.5.1 Reliability

Due to errors during transmission, or differences in the paths taken by packets when travelling through the network, a recipient may fail to receive a message, or messages may arrive out of order. Reliable communication protocols, like the Transmission Control Protocol (TCP), provide a means of retransmitting lost packets to ensure a reliable message stream. However, retransmissions lead to higher latencies, as well as degrading overall throughput.

The time-critical nature of automated trading in particular makes both missing and out-of-order packets a costly issue to overcome. Since exchanges send multiple messages per packet, any error during transmission that results in the loss of a packet will result in the loss of all packet messages. Having more messages per packet increases both the packet's size and informational value. Larger packets have a higher chance of experiencing a bit error, and a greater amount of information is lost with each lost packet.

Fig. 2.11 shows the number of monthly trades on three major stock exchanges: the NASDAQ, the New York Stock Exchange (NYSE), and the Chicago Board Options Exchange (CBOE). Financial exchanges are processing an ever increasing number of trades, and this does not include the many additions, removals, and updates each client may make to their positions within the order book. The actual number of messages processed and relayed by the exchange will be much greater. An *up-to-date* and *reliable* view of the market is essential.

2.5.2 Multi-stream Arbitration

One method to improve reliability is achieved by duplicating packets and sending them via multiple communication channels. Multi-path communication is an established method to

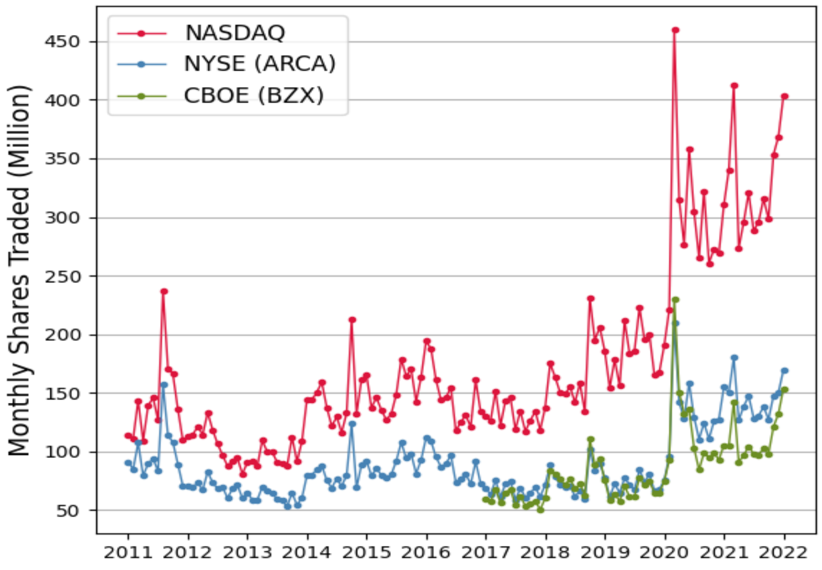


Figure 2.11: Shares trades monthly on the NASDAQ, NYSE, and CBOE exchanges. Source: CBOE [105].

provide reliable communication for time-critical applications [106]. It has been demonstrated that dual-path communication with FPGA-based duplication and merging can maintain higher bandwidths with lower retransmission rates than a single-path solution [107]. Previous work in the area of multi-path communication typically then focuses on reducing retransmission rates, maintaining high bandwidths, or providing protection against complete link failure.

Financial exchanges seek to address potential packet loss and reordering by providing two identical data feeds, termed the A and B streams. Members subscribe to both streams to reduce the likelihood of message loss, but must now merge and order the two streams, a process termed A/B line arbitration.

While multi-path communication and subsequent A/B line arbitration cannot eliminate packet loss, it provides a means for client-side applications to accommodate for missing packets. Indeed, for financial applications, a missing packet may be defined as a packet that fails to arrive within a given time frame, rather than a packet that fails to arrive at all. This time frame is dependent on the given application, and may even fluctuate in real-time corresponding to current market conditions. A separate and independent transmission path is therefore an excellent means to diversify our communication channels and reduce the risk of packet loss.

2.5.3 Data Protocols

The protocol and data format used by exchanges to transmit market information is also of importance. Examples include NASDAQ TotalView-ITCH 4.1 [108], OPRA [109], and ARCA [110].

TotalView-ITCH is a data feed provided by NASDAQ, delivering a range of market data in variable length messages. The messages include: order book information reflecting the interest of buyers and sellers in a particular financial instrument; trade messages; administrative messages, such as paused trading on a security; and event controls, such as start and end of day.

The NYSE ARCA data feed contains similar market information on: depth of book, trades, order imbalance data, and security status messages. Similarly, OPRA provides information about transactions in the options markets via its data feed. Variable length packets are used, so packets may contain different numbers of messages.

Packets transmitted across the network may contain multiple market data messages. While packets with more messages will take longer to process and route through the system, the per-message latency may actually be lower for larger packets, as the size of the header data relative to the message data is reduced. An additional complication is that different messages may have different lengths—data messages vs. text-based status messages, for example—and so may have different processing latencies.

The order and "freshness" of data is represented by a unique sequence number. This may be attached to individual messages, or defined at the packet level for multiple messages. In the latter case, messages will still be given a sequence number. The first message in a packet is typically assigned the packet sequence number, with subsequent messages given sequence numbers counting up from this value.

Technical implementations differ between protocols. This reflects both the different types of financial instruments traded on the exchange, as well as the design priorities of the protocol itself. Some protocols make use of small, compressed packet formats to reduce processing latency, such as Simple Binary Encoding (SBE) [111]. Others may include broader market information, and so prioritise information completeness over saving space, such as those based on sentiment analysis or keyword identification in news feeds and social media [112] [113].

OPRA and ARCA operate on top of the User Datagram Protocol (UDP), while TotalView-ITCH uses a UDP variant called moldUDP64 [114]. As UDP is a connectionless communication protocol, it offers no guarantee that recipients will receive a transmitted packet, or that packets will arrive in order. The use of an unreliable transmission protocol may seem counter-intuitive; however, as such a high premium is placed on reducing latency, UDP's minimal communication overhead offers a significant latency advantage compared to alternative protocols, like TCP.

2.5.4 Related Works

In Chapter 3 of this work, we implement an FPGA solution to A/B line arbitration at the network level. We support any messaging protocol and output two simultaneous modes of operation: one prioritising low latency, and one prioritising high reliability with three dynamically configurable windowing methods. The latter windowing modes can be altered by downstream applications in real-time.

Morris [115] uses a Celoxica board to process financial messages, achieving a throughput of 3.5M messages per second, and hardware latency of $4\mu s$. Their trading platform is one of the few including line arbitration, but no details of its performance are given. It uses a single, simple windowing system similar to the high reliability count mode in our work and supports only the OPRA FAST format. The windowing thresholds are not discussed and cannot be changed.

Most stand-alone A/B arbitrators are commercial and their implementation details are usually not presented. They tend to operate within a NIC and communicate with the host via PCIe.

One such arbitrator from Solarflare [116] uses an Altera Stratix V FPGA. It supports either a low latency mode or a maximum reliability mode; the latter being similar to the high reliability time & count mode in our work. Multiple message protocols are supported, but no processing latency figures are available. Another platform from Enyx [117], also using the Altera Stratix V, does not give any details regarding the windowing method used or possible configuration options. It is non-deterministic, with packet processing latencies ranging from $1050 - 3080ns$ based on 1500 byte packets. Some protocols, like TotalView-ITCH 4.1, specify 9000 byte packets must be supported, so it is unclear how this latency will scale with larger packets.

A number of FPGA-based feed processors have been proposed, however the majority do not mention A/B line arbitration, such as the OPRA FAST feed decoder from Leber [118], and the NASDAQ data feed handler by Pottathuparambil [119]. Other works describe arbitration, but do not implement it, like the high frequency trading IP library from Lockwood [120]. This is an unfortunate omission since line arbitration is an integral part of message feed processing as it increases the amount of available information and actively prevents message loss.

Platforms incorporating some aspects of data feed processing and trading within an FPGA are limited in the range of functions they provide, making it difficult to customise desired features. The flexibility to support applications with different data requirements and different time scales is not present in past works. Single trading platforms are therefore unlikely to be deployed within financial organisations unless the design features exactly meet the needs of the organisation, including the market data feed protocol used.

2.6 Neural Network Activation Functions

Neural network activation functions are used in this thesis to assess the efficacy of our new compute pool architecture, introduced in Chapter 5. As part of our evaluation, we seek to determine the computational complexity a function must have to benefit from our approach.

Activation functions were chosen as they are real-world functions, currently used by a number of applications, and span a wide range of complexities. Additionally, they are a good fit for the FPGA's dataflow processing model and, apart from differences in complexity, all other function aspects—like number and type of inputs, outputs, etc—are the same.

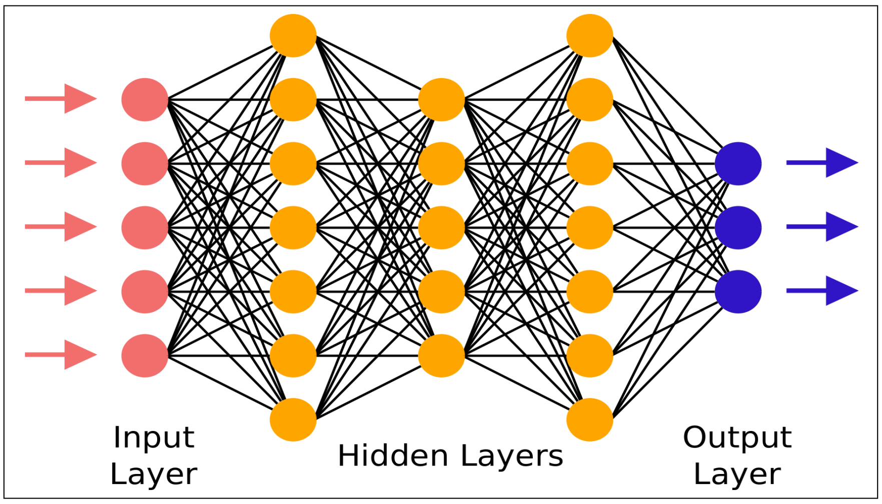


Figure 2.12: An example of a fully connected neural network with three hidden layers.

This section gives a general overview of neural networks, describes the different types of activation functions they may employ, and examines areas where neural networks are deployed and how they may benefit low latency applications.

2.6.1 Neural Networks

Neural networks are a branch of machine learning that take inspiration from the biological neural networks present in the brain. A neural network is formed of many neurons. Each neuron calculates the weighted sum of its inputs, passes this value through an activation function, then outputs a result.

Neurons are arranged into layers, as shown in Fig. 2.12. Each layer may be fully connected, i.e., connected to all the neurons in the following and preceding layers, or connections between neurons may be restricted to enhance known patterns in the data.

Before a neural network is able to perform a task it must first be trained. Example input data are used to teach the layers of neurons how they should react to a given set of data. For example, an image recognition task would use a series of labelled images to train the network to identify one or more objects in an image. Once trained to perform this task, the network is then used to make predictions on, usually new, unseen data. We discuss these two, training and prediction, stages in greater detail in Section 2.6.3.

When presented with an input, such as an image or some set of data, the network processes this data and produces some final output result. Networks may output a **discrete** classification, such as determining if one or more types of object are present in an image, or it may output a **continuous** regression value, for example predicting the price of a house given information on its location, age, number of bedrooms, etc.

Confusingly, the discrete classification may also be a continuous value, but in this case it represents the certainty level for a particular classification, e.g., an object is an 80% match to an apple, 40% match to an orange, 5% match to a banana, etc. There is also considerable overlap in the methods and goals of classification and regression. While regression seeks to find the line of best fit through the data, classification can utilise this same method, but use the resulting line as a boundary between different classes.

The use, training, neuron connections, and choice of activation function within neural networks can carry a great deal of nuance, and many different types of neural networks have been proposed. We give a brief description of the three most popular types of neural network here.

Deep Neural Networks (DNNs) [121] are a class of neural network that contain multiple hidden layers. “Very” deep neural networks are then simply those with many, often 10 or more, hidden layers [121]. More layers do not guarantee a deeper insight into the data. The aim instead is for the network to be of sufficient depth and complexity to represent the desired task or function.

Convolutional Neural Networks (CNNs) [122] primarily target image processing tasks. Neurons perform a convolution on the incoming data, rather than a weighted sum; the result is then passed to the activation function. Each layer of neurons therefore builds up a set of features. As this feature-space is smaller than the original input data, CNNs then also reduce the number of connections between neurons, allowing for sparser neural networks.

Recurrent Neural Networks (RNNs) [121] are designed to work with time-series data. DNNs and CNNs are feed-forward networks, i.e., data only travels in one direction, from the input to the output. In contrast RNNs maintain some internal state, so past data can inform decisions alongside current data. Some implementations [123] also see future data used to make decisions about current data. This reflects patterns in languages where, for example, the words at the end of a sentence affect the meaning of earlier words.

2.6.2 Activation Functions

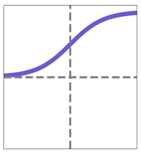
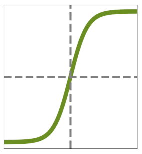
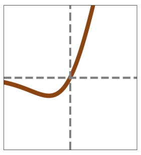
The initial processing step of neurons may be different for different types of neural network, and also for different layers within the same network. DNNs perform a weighted sum of the inputs, CNNs can instead use convolution, and RNNs incorporate data from some internal memory. However, for all networks the final step is to pass this data to an activation function, whose output determines the final output of the neuron.

Despite their differences, these initial processing steps all tend to be highly parallelisable. The neuron inputs are typically independent, and the final result achieved by a series of MAC operations. As MACs are a common function within applications, dedicated hardware and instructions for its use are seen in CPUs [124], GPUs [125], and FPGAs [47] [126].

On the other hand, the activation functions may be more complex. The Rectified Linear Unit (ReLU) is the most popular activation function used today [127], and is constructed of a very simple $max(x, 0)$ operation. However, as we describe below, this function is not always suitable. Larger, more complex functions may be needed, involving floating-point arithmetic, irrational numbers, or requiring iterative computation.

Given the wide range of possible activation functions, there are rarely common hardware-accelerated operations or instructions to perform them. This thesis does not focus on neural networks, but their range of activation functions, with their differing complexities, present excellent candidates to test our new compute pool architecture in Chapter 5. We examine two types of activation function: sigmoid functions (Table 2.1), and those based on ReLU (Table 2.2).

Table 2.1: Sigmoid activation functions.

	$sigmoid(x) = \frac{1}{1+\exp(-x)}$		$tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
	$swish(x) = \frac{x}{1+\exp(-\beta x)}$		

Sigmoid Activation Functions

Sigmoid and tanh both have long histories as activation functions [128]. They typically see use in shallower neural networks as the vanishing gradient problem causes issues when training large networks [129]. This is because the error value used to adjust neuron weights during the training stage becomes smaller, the longer it is backpropagated through the network. Neurons in the early layers of the network are then never updated.

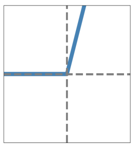
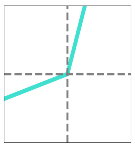
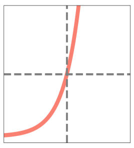
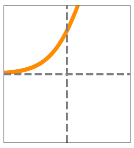
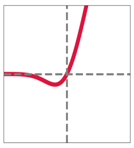
These functions have however seen wider use in RNNs. Their required inclusion in Long Short Term Memory (LSTM) [130] and Gated Recurrent Units (GRUs) [131]—used to boost internal memory performance and broaden the time frames of relevant data—has led to increased interest in their implementation and optimisation.

Developed by Google, Swish [132] is a more recently introduced function. Rather than simply creating a sigmoid response to input data, it seeks to combine the benefits of sigmoid and ReLU functions via a trainable parameter, β. This added parameter can also help alleviate the vanishing gradient problem [132].

ReLU-based Activation Functions

ReLU [133] is one of the most popular activation functions due to its computational simplicity and ability to solve the vanishing gradient problem. However, it also faces its own challenges, such as the dying ReLU problem in the training phase [134]. As ReLU outputs zero for negative inputs (Table 2.2), a large number of neurons with zero outputs can flatten the error gradient,

Table 2.2: ReLU-based activation functions.

	$ReLU(x) = \max(x, 0)$		$LReLU(x) = \max(x, \alpha x)$
	$ELU(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha(\exp(x) - 1) & \text{otherwise} \end{cases}$		
	$softplus(x) = \frac{1}{\beta} \times \log(1 + \exp(\beta x))$		
	$GELU(x) = \frac{x}{2} \times (1 + tanh(\sqrt{\frac{2}{\pi}} \times (x + 0.044715x^3)))$		

making it difficult for neurons to adapt to new inputs. Some tasks and neural networks therefore require more nuanced activation functions than the simple ReLU function.

Leaky ReLU (LReLU) [135], Exponential Linear Unit (ELU) [136], softplus [137], and the Gaussian Error Linear Unit (GELU) [138] functions aim to improve on ReLU. As ReLU has such a simple implementation, the added performance of these new functions will come at the cost of additional computational time and complexity. Determining the stage at which this becomes beneficial is the same complexity vs. performance trade-off we seek to measure within our new compute pool architecture in Chapter 5. Our work could therefore help guide these design choices.

2.6.3 Training and Inference

The neural network's training stage, and its final inference stage where it is deployed, have different computational and processing requirements. These include different: **(a)** numbers of processing resources; **(b)** time frames of operation, with training requiring many orders of magnitude longer than inference; and **(c)** deployment on different computing platforms, environments, and hardware. This section further explores these training and inference stages.

Training

Before a neural network can be used to solve a specific task it must first be trained. This process sees the network presented with an example input, which propagates through the layers of neurons to produce some final output. With little or no previous training, these early outputs are essentially random. By comparing the neural network's output to the correct, expected output we generate an error value. From this error we are able to update the weights and biases of each neuron. Given enough training, the network should, ideally, converge on a configuration that produces the correct output for any given input.

This type of learning is referred to as supervised learning [139]. Here, input data are labelled with a correct output, and we are therefore able to direct the neural network from its current state into one able to produce the correct result. Alternatively, unsupervised learning models use unlabelled data and rely on the training stage to find its own categories or lines of best fit through the data. This task of finding patterns in the data may itself be the ultimate goal of the neural network. Finally, reinforcement learning views the output of the network as a reaction to a given input. A reward is given for correct responses, from which an approach, similar to supervised learning, is used to update the neural network towards a correct output.

No matter which learning method is chosen, the training stage can be a very time consuming and processing intensive task. Data must be processed through networks potentially containing thousands of neurons [29], then each of those neurons updated should the output be incorrect. This is then repeated many thousands, or millions, of times [140].

Large, high performance computer systems are typically used for this training stage. CPUs and GPUs are the most common platforms, but FPGAs can offer comparable or better performance, particularly when it comes to power efficiency [141] [142].

Weight and bias data must also be stored in the network for each neuron, with large networks then requiring large amounts of storage. As this data are used and potentially updated during every neuron interaction, they must always be available to the neuron. This is one benefit of FPGA-based networks, as their small, distributed memories can hold this data close to each neuron's logic. Control flow designs, such as those using CPUs or GPUs, can face a memory bottleneck if they must constantly move large amounts of data to and from memory.

Within FPGAs, the training stage would see the use of larger devices containing a range of processing elements: LUTs, DSPs, embedded CPU cores, and dedicated AI cores [14]. Making use of different processing elements, all working together and performing the same function, is one of the goals of our new compute pool architecture in Chapter 5.

Inference

Once trained, a neural network is deployed like any other application: it is fed input data, for which it produces a final output result in line with the task for which it was trained. The network will no longer be updated. The stage where a trained neural network predicts outputs given new input data is referred to as inference.

Some training methods actively seek to induce sparsity in neural network designs [143]. This means the layout of the inference stage may have changed from its initial configuration during training. Connections between neurons, and neurons themselves, may have been removed. The final inference design may then have fewer processing stages and connections, increasing its throughput and latency performance.

While this stage is much less computationally intensive than the training stage, data must still be processed through hundreds, perhaps thousands of neurons to produce an output. Depending on our throughput and latency goals, we may use few processing elements, or need many elements and high degrees of parallelism. The final neural network application may prioritise: **speed** for low latency applications, such as robotics or automated driving; **accuracy**, to more correctly predict or infer patterns in data; or **resources** when deployed in smaller, lower power devices, like those used in edge computing [144].

Our new compute pool approach is such that an application can be designed without specifically defining numbers or types of processing elements. A neural network—or any application—can then scale throughput and latency performance up and down on an FPGA by adding or removing processing elements from the compute pool. Pools will then automatically arbitrate the mismatched number of parallel data-paths and processing elements, and support both performance-oriented and power-oriented processing elements.

2.7 Summary

In this chapter we first describe what FPGAs are, how they fit into the wider field of computing platforms, and the many different processing, memory, and routing resources they include. We see that problems arise in both accounting for resource heterogeneity and in balancing resource use across larger applications. For this, higher-level approaches, such as those we introduce in Chapters 4 and 5, are needed.

The areas where FPGAs are deployed is also explored. The inclusion of FPGAs at the infrastructure level in particular is shown to offer excellent opportunities for the dataflow processing model used by FPGAs. We seek to capitalise on this at the network level with our A/B line arbitrator in Chapter 3, and for memory accesses with our low latency memory interconnect in Chapter 4.

Next, the FPGA application development process is examined. We show that benefiting low latency applications on FPGAs occurs by both improving their performance, and simplifying the application design process itself. Our A/B line arbitrator in Chapter 3 operates at the network level, seeking to remove common functionality from downstream applications. Future applications will therefore not need to provide their own implementations of this function.

The examination of the development process demonstrates the benefits of higher-level approaches, such as those adopted by our work in Chapters 4 and 5. The automated tool we develop in Chapter 4 is also highlighted to show how it can be used within existing development tools and toolchains.

With our focus on higher-level approaches to FPGA application design, we next describe coarse-grained FPGA architectures. These look to form higher-level processing elements, called functional units, by grouping together multiple FPGA resources. We examine the benefits and disadvantages of such architectures and introduce our new compute pool architecture as a more flexible, high-level approach. We describe how grouping processing elements by function, rather than via a rigid set of pre-defined FPGA resources: **(a)** allows for the inclusion of all FPGA resources; **(b)** improves the utilisation of FPGA resources; **(c)** does not limit routing and interconnection topologies; and **(d)** maintains the high-level application design process.

Chapter 4 focusses on improving low latency applications using an architecture targeted to memory-based computing. We therefore describe the workings of memory-based computing, and examine its benefits and drawbacks. We detail how functions are mapped to, and stored in, memory. FPGA BRAMs are shown as the best type of memory for low latency applications that employ memory-based computing. The small storage capacity of BRAMs mean many will be needed, providing the opportunity for many parallel accesses to memory. Our architecture in Chapter 4 is designed to maximise this parallelism, benefiting applications that require highly parallel, low latency access to this shared memory resource.

The processing of financial market data feeds is described next. In particular, the reliability of these data feeds, and the *need* for reliability, is examined. Arbitrating between the two redundant A and B data feeds to account for missing messages is then a critical task. We address this with our network-level arbitration of data feeds in Chapter 3. Past works are also explored. We show that there is a need for different, customisable modes of arbitration, and for supporting different data feed protocols, both of which we address with our arbitrator. We also highlight the benefits of deterministic processing times achievable through an FPGA, rather than CPU, implementation

The final area we discuss is that of neural network activation functions. These are used by our compute pool architecture (Chapter 5) to determine the level of computational complexity a function must possess to see benefits with our approach. This general overview and examination of neural networks and their activation functions, gives real-world examples of applications for which our new architecture can provide performance benefits. These benefits arise from not only the improvement of throughput, latency, and resource use, but also via the customisable nature of the new compute pools themselves.

A neural network application—or any application—can add processing elements to a centralised pool to increase performance, or remove elements to reduce resource use. This is particularly applicable to neural networks which see processing requirements change between the training and inference stages, and also due to the different environments within which they will be deployed. The variation in types and tasks of neural networks, the range of possible functions, and the need for high degrees of parallelism, make activation functions an ideal test-bed for our new compute pool architecture.

Chapter 3

Network-level FPGA Acceleration of Low Latency Market Data Feed Arbitration

3.1 Overview

This chapter seeks to address the challenge of processing financial data feeds in edge computing environments. Using an FPGA deployed at the network level, we arbitrate between the redundant data feeds that financial exchanges use to transmit market information. Data feed arbitration—described in Section 2.5—must be performed by all downstream applications, so we consolidate this at the network level. Our low latency processing of these data feeds **(a)** promote deterministic processing times, **(b)** support any data messaging protocol, and **(c)** provide downstream applications with real-time control of our in-network processing.

With our real-time controls we demonstrate that more general-purpose, network-level processing can accommodate changing, and application-specific operations. Further, we support different types of applications by providing two simultaneous output streams: one stream prioritising low latency, able to process messages in 1 cycle; and another stream prioritising high reliability, processing messages in 7 cycles.

3.2 Introduction

To allow for market data feed arbitration, financial exchanges provide two redundant data feeds (A and B) to help guard against message loss. The downstream trading applications that subscribe to these feeds may find messages are delayed or missing on one, or both, of these incoming streams. All downstream applications must then arbitrate between the A and B data feeds to ensure a full and correct view of the market. We look to consolidate this duplicated work at the network level with a low latency, high performance FPGA implementation.

Financial applications use the messages received on these data feeds to **(a)** determine the current state of the market and the institution's risk, and **(b)** as signals for buying or selling financial instruments. For **(b)**, time-critical decisions have to be made based on the received messages, often involving the analysis of patterns within the data.

This decision making is a critical element for electronic trading and hence, it is vital that messages are received quickly and in the correct order. Failure to do so will result in the loss of profit-generating opportunities, provide competitors with an advantage, and create a false image of the current state of the market, increasing risk.

When considering CPU-based systems, these general-purpose architectures separate data acquisition and processing, leading to latency penalties when processing external data. The pipelined and parallel nature of A/B arbitration instead provides an opportunity for hardware acceleration of time-critical processing before the resulting data are passed to the CPU.

While FPGAs are capable of performing this task, achieving a high bandwidth, low latency solution requires careful consideration of design choices. This is especially true when dealing with the data-path widths necessary to support the ever-increasing demands on latency and throughput. The choice, placement and, often times, duplication of processing resources play a pivotal role in meeting these constraints.

In this work we present an architecture for low latency A/B arbitration, supporting all market data feed protocols. We provide two simultaneous outputs: one prioritising low latency, and one prioritising high data reliability.

These features allow support for both a wide range of downstream applications, and for individual applications to make trading decisions across different time frames. An application might act quickly based on incomplete data, such as when a packet is delayed, and then update or undo a previous action at a later time when the delayed packet has arrived.

We also support dynamically configurable windowing methods so downstream applications can alter the arbitrator to respond to changing requirements in real time.

The main contributions of this chapter are as follows:

- A new hardware-accelerated, low latency A/B line arbitrator which runs two packet windowing modes simultaneously. We support three dynamic windowing methods, any market data protocol, independent memory allocation per input, and configurable datapath widths (Section 3.3).
- Performance models and examinations of the design considerations necessary to perform low latency processing of data from multiple inputs. This includes: critical path analysis, guaranteeing deterministic memory access, and the creation of a low latency testing framework (Sections 3.4 and 3.5).
- Implementation and evaluation of A/B line arbitration using the NASDAQ TotalView-ITCH [108], OPRA [109], and ARCA [110] market data feed protocols on a Xilinx Virtex-6 FPGA. TotalView-ITCH is also implemented on a Xilinx Virtex-5 FPGA within a network interface card, where we use our new low latency testing framework and real market data to measure its performance (Sections 3.6 and 3.7).

3.3 Flexible A/B Line Arbitration

When merging and arbitrating between the A and B data streams, uncertainty regarding the presence and order of messages gives rise to four possibilities. A message may: (**1**) arrive on both streams; (**2**) be missing from one stream; (**3**) be missing from both streams; or (**4**) arrive out-of-order. For the first and second cases we should pass through the earliest message we encounter, and have no expectation of seeing it again. However, the third and fourth cases

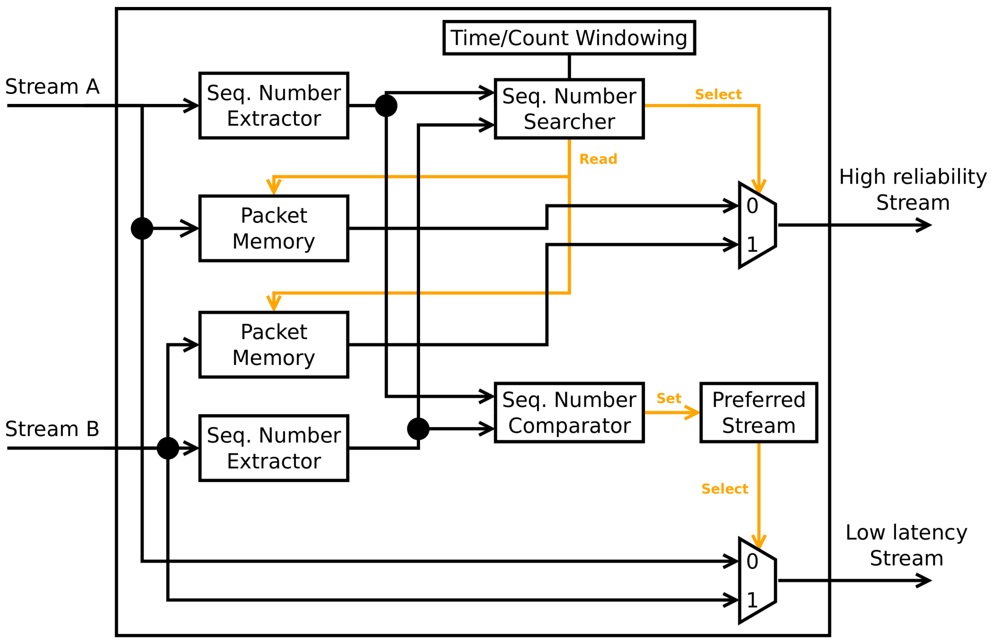


Figure 3.1: The layout of our A/B line arbitration design.

illustrate the need for an expectation regarding future messages. We require a centralised system to monitor the streams and share state information.

It is important at this stage to distinguish between market data messages and packets. Exchanges send UDP packets containing one or more messages. This can be viewed as a continuous block of messages, all correctly ordered by sequence number, with no missing messages. Therefore, when a packet is missing we are in fact dealing with a block of missing messages.

This means packets, rather than messages, are the smallest unit of data we process and store. In the case where market data protocols do not issue packet numbers—such as OPRA, where sequence numbers are assigned to messages—we use the sequence number of the first message in the packet to identify that packet. The next expected packet is then:

$$SN_{pkt+1} = SN_{pkt} + M_{pkt} \tag{3.1}$$

where SN_{pkt} is the sequence number of the current packet and M_{pkt} is the number of messages it contains.

Fig. 3.1 gives our design layout, showing the processing and output of the high reliability and low latency processing modes. The windowing module supports three high reliability modes of operation, for which the windowing thresholds can be set at run-time. An operator or monitoring function can adjust these thresholds to meet application or data feed requirements.

3.3.1 High Reliability Modes

The high reliability output sacrifices some latency performance in order to account and correct for packets missing or arriving out of order. An acceptable threshold for this reduction in latency is application-specific, and indeed may change in real-time.

When we encounter a packet with a sequence number larger than the next expected sequence number, it has arrived out of order. The missing packet, or packets, may be late or never arrive. The high reliability mode stores these early packets and waits for the missing packets, stalling the output.

We decide how long to wait for missing packets using a windowing system, based on either: the amount of time we have stalled the output, the number of messages delayed, or a hybrid of both time and message count. Within this window we store any new packets we receive while waiting for the missing packets. Whatever arbitration or windowing system we use, we must ensure not to delay a valid, expected packet as this is the most likely case.

Count-based windowing is used by Morris [115], time & count by Solar Flare [116], while Enyx [117] does not detail their windowing approach. Our work is the first to support all three windowing methods, provide low latency, application-specific parametrisation, and output a high reliability and low latency stream simultaneously. Furthermore, we offer dynamic configuration of the three windowing modes so that downstream applications can modify the arbitration method in real time, a feature not previously available.

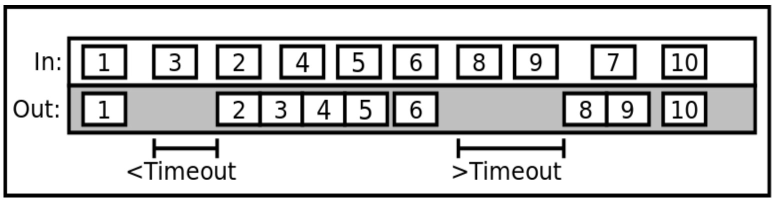


Figure 3.2: High reliability time.

High reliability - time: A time-based windowing approach is good when we want to set a hard limit on possible delays and define the maximum processing time of packets.

When we delay a packet, P, we assign it a timeout value, T, the maximum number of clock cycles we will delay it. T is decremented each clock cycle and when it reaches zero we discard any missing packets and output P. An example with a single input is given in Fig. 3.2, where packet P_2 is late, but arrives before P_3's timeout reaches zero and is thus able to be output. P_7 however is too late, so the delayed packets P_8 and P_9 are output, and P_7 is discarded.

Assigning the maximum timeout value to a packet then decrementing it is more beneficial than incrementing from zero. The timeout check is then simply a zero equality check. The number of remaining cycles may also be used to predict this condition and pre-compute future data values, such as the expected number of buffered messages in the next cycle.

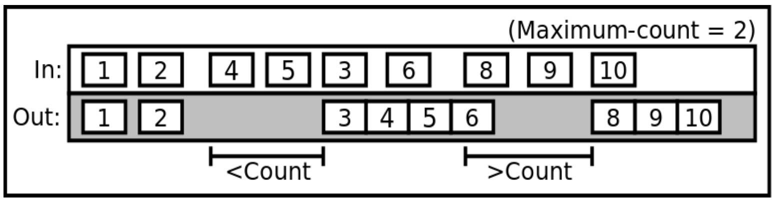


Figure 3.3: High reliability count.

High reliability - count: Time-based windowing sets a packet timeout regardless of how many messages a packet contains. Counting delayed messages—not packets—more accurately represents processing delay, as the number of messages per packet varies during the day. This time-independent approach better matches the pace of incoming data.

We output a delayed packet when either: the missing packet or packets arrive, or the number of stored messages exceeds the maximum-count threshold. Two examples of this are shown in Fig. 3.3's single input example. Packet P_3 arrives before we exceed maximum-count = 2 buffered messages, so P_3 and the stored packets P_4 and P_5 are output. Packet P_7 does not arrive, so when we receive P_{10} there are now more than maximum-count = 2 messages buffered, so we discard P_7 and output the stored packets in order.

For simplicity, our example has just one message per packet. When packets contain multiple messages, it is possible for our maximum-count limit to be exceeded within a single packet. This again illustrates the real world disconnect between the number of *messages* updating the market state and number of *packets* that are received via the network. Our multiple windowing methods, and real-time control of windowing parameters, are then important tools when processing financial messages.

One issue with count-based windowing occurs at the end of the day. With receipt of the final packet there will be no more input packets to process, so we cannot output any stored packets. This windowing is used in [115], but residual packets are not addressed. It is solved in this work either by use of the hybrid time & count method's time limit, or by dynamically altering the maximum-count threshold.

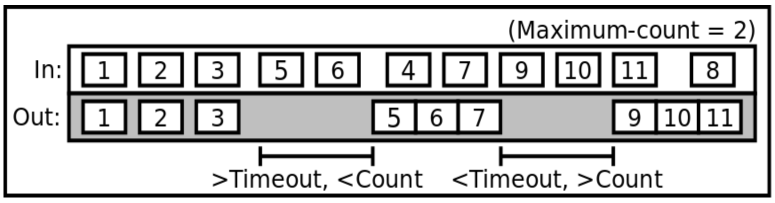


Figure 3.4: High reliability time & count.

High reliability - time & count: Combining the time and count based high reliability modes provide the most robust solution for processing out-of-order packets. We can utilise the count threshold's time-independent ability to follow the incoming packet rate as it fluctuates during the day, whilst still establishing an upper limit on delay times.

In Fig. 3.4's single input example, both the time and count windowing thresholds are used to determine if a stored packet should be output. Packet P_4 takes too long to arrive, therefore exceeding P_5's timeout and resulting in P_4 being discarded. Later, P_8 is also late, but whilst waiting for P_9's timeout, the number of buffered messages exceeds maximum-count = 2, and P_8 is discarded.

3.3.2 Low Latency Mode

In contrast to the above high reliability mode, the low latency mode prioritises the receipt of *new packets*, rather than *all packets*. Our design layout in Fig. 3.1 shows that data from the A and B streams are not buffered or changed in any way before being output on the low latency stream. This is arbitration in its simplest form: outputting a stream of unique, ordered packets.

We treat an input packet as valid based solely on whether its sequence number is larger than or equal to the next expected sequence number. We do not wait for missing packets and hence, do not require resources for packet storage. Our goal is to minimise transmission latencies.

The Ethernet, IP and UDP packet headers pose a problem when trying to minimise the arbitration latency. The packet's sequence number is only visible after we process these headers, which may take a number of cycles. We solve this by assuming a packet is valid and immediately output it. When we encounter the sequence number and it is not valid—i.e., less than the next expected sequence number—we register an output error, causing the packet to be discarded.

Similarly, when packets arrive on both input streams simultaneously, we must make the choice of which packet we should output without any information on either packet's contents. There is no method that can guarantee a priori which stream to select, so we instead select the last stream on which we encountered a valid packet. This differs from the previous high reliability modes where we have additional cycles available to process and compare the sequence number.

With these simple operations we reduce arbitration to a single cycle. The user does not need to choose between the low latency and high reliability modes. Due to the low latency mode's minimal resource requirements, it can be output simultaneously with our high reliability mode.

3.3.3 Network-level Operation

By choosing to arbitrate between message streams at the network level we remove the need for each downstream application to arbitrate between the streams themselves, eliminating this processing redundancy. However, when processing at the network level, rather than within a computing node, we must take an active role in routing non-market data packets.

Even within a dedicated market data feed network, routers and NICs will transmit information requests to other nodes. We must reserve FPGA resources to route these non-market feed packets. Network identification packets are typically only hundreds of bits, requiring little storage space, and are processed at the lowest possible priority to minimise interference with market packets.

Past works [115] [116] [117] focus on processing market data feeds on FPGAs situated within computing nodes rather than at the network level. Data are then passed to the CPU or GPU via low latency DMA transfers. This scales poorly if further nodes are needed as each will need an FPGA for data feed processing. Our packet-based, network-level arbitrator consolidates the node-independent arbitration operations. Only the low latency DMA transfers need be implemented within nodes for us to create a newly scalable system with the same functionality as past works.

3.3.4 Customisation and Extensibility

As our arbitrator deals with the initial stages of storing, processing, and identifying market feed packets and messages, it is a simple matter to extend our system to provide additional functionality within the FPGA. We support the following customisations:

The windowing threshold values can be reconfigured at run-time, as discussed above. The user, or a monitoring function, is able to tailor the time and number of messages delayed in order to meet both the changing needs of the market, and those of the downstream applications.

Any physical connection for input and output ports can be used. Our arbitrator can connect to any commercial or customised network by translating the connection's interface, e.g., Ethernet or InfiniBand, to a standard FPGA interface. The arbitrator may also be used within a computing node, rather than at the network level.

The size and number of packets stored can be configured at compile time, for both market and non-market packets. The size of the pipeline is adjusted accordingly.

Any market data feed protocol can be adopted, not just those of TotalView-ITCH, OPRA, and ARCA. The only protocol-specific information required is the maximum packet size, and the location and bit-width of the sequence number within the packet.

3.4 Performance Model

Processing FPGA data has many difficulties when aiming to minimise latency. The performance requirements of A/B line arbitration specifically, pose a number of challenges. In this section we present a performance model of our high performance, low latency arbitration approach.

Any decision regarding a new packet is dependent on its sequence number, and therefore the number of cycles we must wait to process it. Given the packet byte position where the sequence number begins, Pos_{sq}, its length in bytes, Len_{sq}, and the width of our data-path in bytes, W, the number of cycles we must wait to encounter the sequence number is then:

$$C_{sq} = \left\lfloor \frac{\mathrm{Pos}_{sq} + \mathrm{Len}_{sq} - 1}{W} \right\rfloor \tag{3.2}$$

The packet processing latency's lower bound is achieved when we encounter expected packets. No further processing is needed, but due to the delay from Eqn. 3.2 the high reliability mode must still begin storing it in memory and then read it out over C_{read} cycles. The number of cycles for an expected packet is then:

$$C_{exp} = C_{sq} + C_{read} \tag{3.3}$$

The windowing delay, C_{win}, is the worst-case time delay we may experience when the windowing system delays a packet, and depends on the chosen windowing method and its configuration. With C_{write} as the number of cycles taken to store a packet in memory, we can now define the upper bound cycle delay for a packet as:

$$C_{max} = C_{exp} + C_{write} + C_{win} \tag{3.4}$$

For the time mode, the windowing delay C_{win} is simply the timeout, while for time & count it is the lower of the time and count delays. The high reliability count mode is designed to be time-independent. Here, C_{win} is potentially infinite, however, in practice this only occurs when there are no further packets to process, and can be resolved via run-time alteration of the maximum-count threshold.

Table 3.1: Description of symbols for Eqn. 3.5.

Symbol	**Description**
MC	maximum-count value
n	maximum number of packets that can be stored in the buffer
$\log_2(n)$	cycles to find the stored packet with the lowest sequence number
λ_p	number of packets per input, per cycle
λ_m	number of messages per packet
I	number of inputs

In finding C_{win} for the count mode we must take into account the time taken to receive sufficient messages—not packets—to exceed our maximum-count threshold. We define this based on the factors described in Table 3.1, from which C_{win} for the count mode is then:

$$C_{win} = \log_2(n) + \frac{MC}{I \times \lambda_p \times \lambda_m} \tag{3.5}$$

3.5 High-performance Architecture

Achieving fast, low latency processing in our arbitrator requires careful development of the overall hardware architecture. From the models described in Section 3.4 we can make a number of recommendations for arbitrator designs. In this section, we present several architectural considerations to achieve low latency processing of data from multiple input streams.

3.5.1 Deterministic, Multiple-input Memory Access

Our first recommendation deals with memory access. As all windowing modes rely on reading and writing packets to memory, low latency memory access (C_{read} and C_{write}) is essential. Storing data in DDR memory is unsuitable for low latency applications due its high access times. In addition, DDR memory locations must be refreshed periodically to maintain their state, leading to non-deterministic access patterns. High Bandwidth Memory (HBM) follows a similar design to DDR memory, and thus shares its latency disadvantages.

Other memory types, such as Content-Addressable Memory (CAM) or specialised flash memory, are not widely available in commercial systems and their access latencies are still higher than

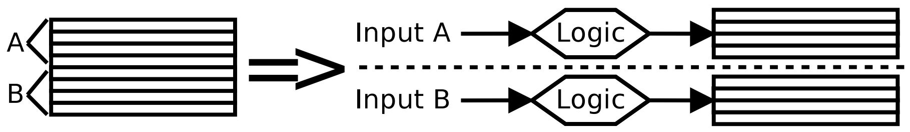


Figure 3.5: The address generator automatically splitting a large memory.

those required by high performance designs. The internal FPGA BRAMs are the best option to ensure fast and predictable access times. Their constant-latency random access is of particular benefit as, when arriving out-of-order, packets will be read from memory in a different order from which they were written.

When considering multiple input sources and just a single, centralised memory within an FPGA, we see the number of writers to memory will exceed the number of available memory ports. In reality, a large memory is formed on the FPGA by connecting together many, small BRAMs. Many memory ports are then available to us, but micromanaging memory on such a scale requires a dedicated architecture, such as the one we introduce in Chapter 4. For A/B line arbitration we instead tackle this problem at the algorithmic level, through the use of memory address generators.

For our system with multiple inputs, we create a single logical memory that provides the address of a free memory location to the logic at each input. The simplest case of two inputs is shown in Fig. 3.5, where the address range of input A covers the first half of the memory, and input B, the second. When built within the actual FPGA, the two halves of memory will be independent and so will be mapped close to their respective accessors.

To tackle storage inefficiencies, i.e., when data does not appear uniformly across all inputs, we must: **(a)** minimise data duplicated across inputs; **(b)** ensure the initial memory allocation reflects the percentage of total data originating at each input; and **(c)** make sure data are removed from each memory in proportion to that memory's occupancy.

Our approach in Chapter 4 describes a more general solution to the problem of facilitating multiple, low latency accesses to memory. However, that is not required here. For **(a)**, we have no duplicated data in memory as we guarantee packets are globally unique by checking packet sequence numbers.

Given the nature of our duplicated market data feeds, **(b)** is tackled by evenly allocating memory to each input. If packets were instead more likely to appear on feed A rather than feed B, for example, then the proportion of memory allocated to feed A could be increased accordingly.

Finally, for storage inefficiencies arising from **(c)**, arbitration provides for a well-defined, ordered removal of packets, for which our packet windowing methods establish an upper bound.

3.5.2 Optimising Packet Accesses

With multiple packets stored in memory, we select a packet to output by finding the smallest sequence number using a binary search, requiring $\log_2(n)$ cycles for n packets. Binary search is pipelined, and uses a BRAM for storing packet numbers. Sequence number comparison is performed with a simple comparator and a register storing the search key.

The search index calculation uses two registers for the upper and lower search index, an adder and bit-shift for the midpoint calculation, and a comparator. We choose this method, rather than sorting packets before writing them to memory, as the $\log_2(n)$ cycles required for a binary insertion is the same as for a binary search, but does not scale well with multiple inputs.

Now that data access is predictable and can be easily scaled for multiple writes, we must deal with read latency. Our design makes use of an additional buffering stage, meaning BRAMs require two clock cycles for their data to be available. This allows us to maintain a high clock rate and support our extended 128-bit data-path, but makes packet comparison in memory very costly, especially if both packets are stored within a single BRAM. A small meta-data cache in registers will instead allow immediate access to packet-specific information, reducing latency.

The meta-data cache is shown in Table 3.2, with example data widths for the larger, TotalView-ITCH protocol. To be effective the cache must be small, fast, and contain all necessary packet information. A cache line is 128 bits wide and utilises dual-port distributed RAM so it can be written and read in the same clock cycle.

Table 3.2: Meta-data cache contents.

Name	**Bits**	**Description**
Number of Data	10	Size of packet/data-path width
Sequence Number	64	Packet's unique sequence number
Cycles Remaining	32	Cycles until this packet times out
Number of Messages	16	Total messages in this packet
Final Byte Enable	4	Enable signal for final packet bytes
Packet Being Input	1	Is this packet still being input
Cache Line Free	1	Is cache line available or occupied
Total	**128**	

3.5.3 Improving Throughput and Latency

Improving the throughput and latency of a design can be accomplished by widening the data-path or increasing the clock frequency. A wider data-path requires more resources and routing at each stage of the design, but fewer clock cycles are needed to process packets (see Eqn. 3.2).

The critical path is defined by operations with the largest combined logic (T_l) and routing (T_r) delays. The maximum clock frequency is then:

$$F_{max} = \frac{1}{T_l + T_r} \tag{3.6}$$

Logic delays are reduced by increasing parallelism, and using multi-staged pipelines to spread processing over multiple cycles. Our $\log_2(n)$ binary packet searcher and multi-stage input packet processing are targeted at minimising this delay.

Routing delays are reduced by using fewer resources, placing interconnecting resources physically closer together, and using additional data buffering stages. Routing delays are harder to reduce and depend heavily on the FPGA utilisation and the design's interconnections.

We tackle this by separating data processing streams and treating them as independent data-paths. As shown in Fig 3.1, the logic and memory for each stream do not interact, and can therefore be more freely placed and routed within the FPGA.

Our design's critical path comes from packet storage and sequence number comparisons. Any resource/frequency trade-offs to improve throughput and latency must then be made by modifying these design elements: either storing fewer packets or comparing shorter sequence numbers.

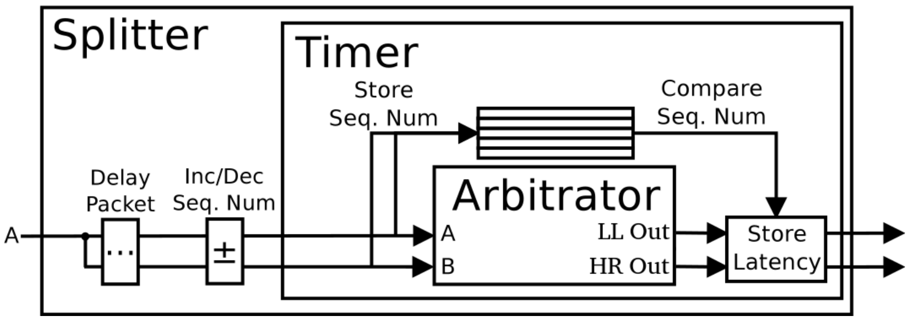


Figure 3.6: The cycle-accurate testing framework.

3.5.4 Cycle-accurate Testing

The difficulty in testing low latency designs is that many of the corner cases occur within nanosecond time frames. Missing packets can simply be modelled by the sender skipping sequence numbers. However, it becomes difficult to arrange specific packets, from multiple feeds, to arrive at the arbitrator at precise times. We therefore create a testing framework within the FPGA using two wrappers around our design, as shown in Fig. 3.6.

The outer, splitter wrapper takes a packet from one input and mirrors it on the others, either in the current cycle or delayed one or two cycles. The splitter can increment or decrement the mirrored sequence number so it appears as a new packet. The inner, timer wrapper notes incoming packet sequence numbers, counts the cycles until it appears on the output, and writes this latency into the packet header.

The testing framework only requires the packet's sequence number location, and is otherwise application and protocol independent. We do not interfere with the arbitrator's operation or affect the critical path as all measurements are performed outside of the arbitration module.

3.6 Implementation

We verify our proposed design and low latency architecture by implementing an A/B line arbitrator for each of the TotalView-ITCH, OPRA, and ARCA market data feed protocols. For each protocol we require knowledge of the maximum packet size, the sequence number width, and the byte position of the sequence number in the packet. This is determined by their specifications, and is given in Table 3.3.

Table 3.3: Packet protocol specifications.

Protocol	**Max Packet Size (bytes)**	**Sequence Number**	
		Width (bits)	**Position**
ITCH	9000	64	53
OPRA	1000	31	47
ARCA	1400	32	46

To reduce latency we make use of a wide 128-bit data-path, double the 64-bit width commonly used. This 64-bit convention comes about due to the 64-bit width multiplied by the commonly used 156.25MHz reference frequency for processing 10Gbps Ethernet data. Our wider 128-bit data-path has the disadvantage that it may negatively affect the routing delay, but with our low latency architecture we achieve latencies an order of magnitude lower than Enyx [117] while maintaining at least 20Gbps throughput.

Our wider, 128-bit data-paths allow us to support higher Ethernet line rates, or to arbitrate data feeds originating from other interfaces. This could be another type of network, such as InfiniBand, or data coming from the local computing node itself, such as via PCIe.

We verify and test our design in two ways. First, we implement an arbitrator for each of our chosen protocols within a Xilinx Virtex-6 LX365T FPGA on an Alpha Data ADM-XRC-6T1 card. As our processing rate is greater than the 10Gbps Ethernet connections used by each protocol, we transfer data via PCIe. We configure each arbitrator for their respective protocols by entering the values from Table 3.3 into our configuration file. Adopting a new protocol in the future requires only that we indicate where the equivalent fields are located within the new packet format.

For our second design verification we implement our arbitrator on a Xilinx Virtex-5 LX330T FPGA within an iD ID-XV5-PE20G NIC. This card receives a duplicated data feed over two 10Gbps Ethernet connections, more closely matching the configuration of real-world trading systems. The high reliability and low latency outputs are transmitted to the host via PCIe, with the layout given in Fig. 3.7. The TotalView-ITCH protocol is used to test this design as it is the most resource and processing intensive. Trading messages from 9 September 2012 are used to test the system.

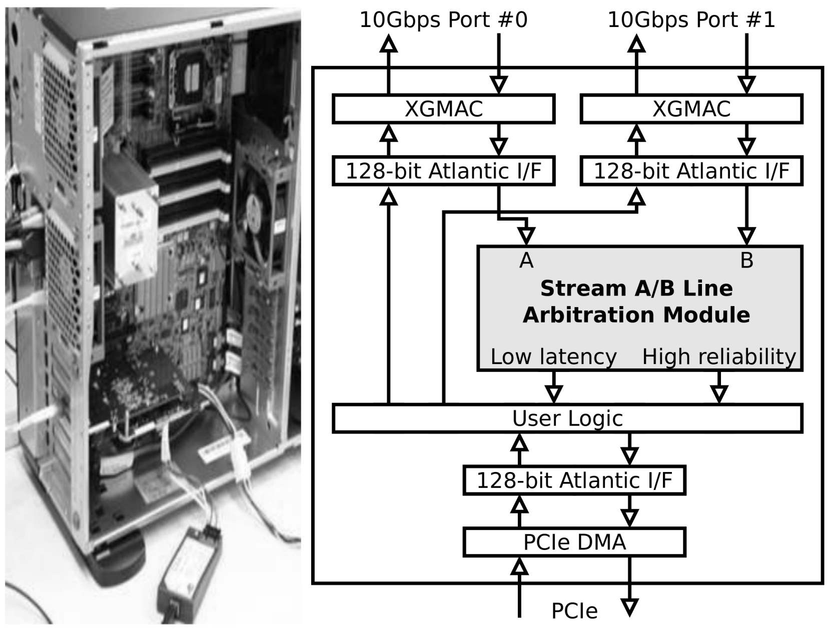


Figure 3.7: The layout of our arbitrator module within the FPGA.

OPRA and ARCA operate on top of UDP, while TotalView-ITCH uses a UDP variant called moldUDP64 [114]. Our design stores 8 packets, each with sufficient space for the Ethernet (14 bytes), IP (20 bytes), and UDP (8 bytes) headers, as well as the packet payloads from Table 3.3. Each packet has an associated meta-data cache entry, for which the entire cache will require only 4 FPGA slices.

With our deterministic, multiple-input memory architecture we can simultaneously write packets from each input feed and read packets out. The high-level nature of the architecture also makes it simple to expand or contract the memory size at compile time to suit our specific market protocol.

3.7 Results

Our Virtex-6 implementation found TotalView-ITCH, with its 64-bit sequence numbers, achieved a single cycle latency of $6ns$. OPRA and ARCA both achieved $5.25ns$ with sequence number widths half that of TotalView-ITCH. Section 3.5.3 and Eqn. 3.2 describe how sequence number comparisons are one source of our critical path, suggesting that artificially truncating the sequence numbers of packets can benefit arbitration. This would come at the cost of additional logic to deal with packets that straddle the new sequence number boundary.

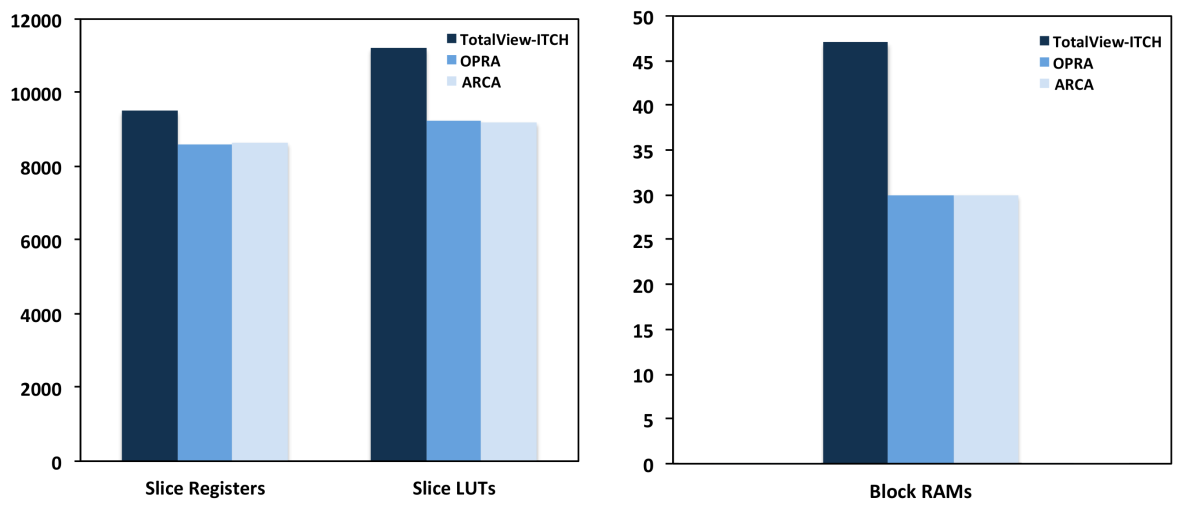


Figure 3.8: Resource usage for the three messaging protocols.

TotalView-ITCH's $6ns$ latency results in a 166MHz FPGA design with a maximum throughput of 21.3Gbps, while OPRA and ARCA's $5.25ns$ latency means a 190MHz design and a maximum throughput of 24.3Gbps. Both designs are fast enough to satisfy our targeted 20Gbps processing.

With financial markets making greater use of higher throughput connections, our design will be well placed to capitalise on this increased throughput capacity. Indeed, the TotalView-ITCH message feed is already available via both 10Gbps and 40Gbps connections. However, only a fraction of this throughput is currently used by the message feed.

TotalView-ITCH's requirement for 9000 byte packets is multiple times that of OPRA (1000 bytes) or ARCA (1400 bytes). Fig. 3.8 shows TotalView-ITCH's resource usage within our arbitrator does not increase in proportion to this requirement. For example, when processing TotalView-ITCH data, we do not need 9x the registers, LUTs, or BRAMs of OPRA, or 6.4x that of ARCA. This is mainly due to buffering host communications. Buffering plays a larger role in our second verification design, as it is implemented within a network interface card and must process two bi-directional 10Gbps Ethernet connections.

3.7.1 Real-world Performance

Our second design, implemented within a network interface card, serves as an important test of real world performance. When analysing the TotalView-ITCH messages from the two redundant

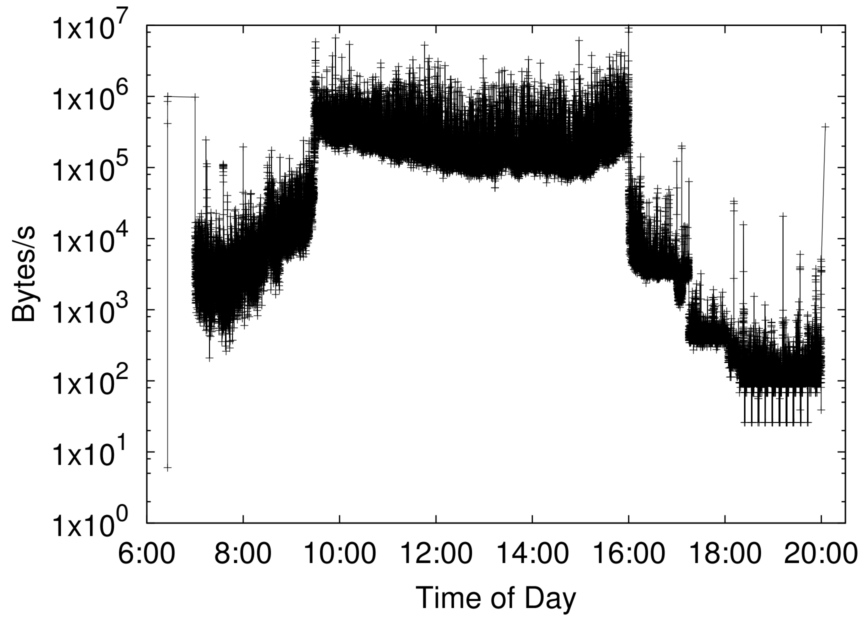


Figure 3.9: Market data feed throughput during the day.

10Gbps Ethernet links, we find we process about 322 million messages throughout the day. This would require 29 bits for message sequence numbers, demonstrating that it is possible to safely truncate TotalView-ITCH sequence numbers without affecting performance.

When describing market data feed processing in Section 2.5, Fig. 2.11 shows that the number of trades in 2022 is nearly 4x those processed in our example data. This would then mean 31 bit sequence numbers are needed, still well below the full 64 bits required by the TotalView-ITCH standard. Later, in Section 3.7.3, we demonstrate this increased throughput is also well within the processing capacity of our arbitrator.

This second design also allows us to verify our latency calculations by making use of our cycle-accurate testing framework. By inspecting packets on the host after they have been arbitrated we can easily read out the number of cycles it took for each packet to be processed. For the high reliability mode we find it takes 7 cycles to process expected packets, i.e., packets not needing to be buffered. For the low latency mode we find packets are processed in 1 cycle. Both of these results match those determined during simulation. These real-world latency measurements also succeed in demonstrating the resolution of our testing framework as we are able to measure processes that occur within one cycle.

Fig. 3.9 illustrates the market activity for TotalView-ITCH at different times of the day, showing the total throughput per second. The rate of incoming messages changes throughout the day, illustrating the need for run-time configuration of the high reliability mode thresholds. For example, the acceptable delay between the busy *9.30-16.00* market hours will be less than in the pre and post market hours.

At peak times we find that we must process 73Mbps (or 1000 packets per second). Our implementation is more than capable of meeting existing demand, so we must focus on shortening our processing latency, i.e., the time taken to react to packets and messages. Previous works often emphasise the fact that their designs operate at the 10Gbps Ethernet line rate, but this is not required to meet the rate of current market data feed messages.

3.7.2 Latency

We measure our packet processing latency by analysing our implemented design and making use of the formulae we derive in Section 3.4.

Our implementation takes 2 cycles to write to the packet buffer (C_{write}): one cycle to trigger a write operation, and one cycle to write to the memory. Memory reads (C_{read}) take 4 cycles: one cycle to trigger an output operation, one cycle to specify the memory location, one cycle for the data to appear on the memory output, and one cycle to output the data. Each packet is broken up into smaller segments equal to the data-path width, W, and subsequent packet segment reads are pipelined.

For each of the three protocols—TotalView-ITCH, OPRA, and ARCA—we plug in the values for the position (Pos_{sq}) and length (Len_{sq}) of their sequence numbers from Table 3.3 into Eqn. 3.2, along with the width (W) of our 128-bit (16 byte) data-path. From this we find the number of cycles to encounter the sequence number for each protocol to be:

$$\begin{aligned} &\text{TotalView-ITCH:} \quad && C_{sq} = \left\lfloor \frac{53+8-1}{16} \right\rfloor = 3 \\ &\text{OPRA:} && C_{sq} = \left\lfloor \frac{47+4-1}{16} \right\rfloor = 3 \\ &\text{ARCA:} && C_{sq} = \left\lfloor \frac{46+4-1}{16} \right\rfloor = 3 \end{aligned}$$

Despite having different message formats, the sequence numbers of all three protocols happen to be visible in the same cycle. This is surprising as we make use of a comparatively wide data-path compared to other, current works. This suggests it is the format of the market data feed protocols themselves that lead to this additional processing latency. From Eqn. 3.2, an optimal format would see the sequence numbers: **(1)** present closer to the beginning of a packet, and **(2)** use as few bits as possible.

The lower bound packet processing latency for all three message protocols is then found as $C_{exp} = 3 + 4 = 7$ cycles.

We can compare this to a software arbitrator using the IBM PowerEN processor [145], with out-of-order packets stored in L2 cache and using a time-based windowing mechanism similar to the high reliability time mode in this work. Here, arbitration is performed using only the OPRA protocol and has a best-case performance of $150ns$, compared to 7 cycles $\times$ $5.25ns = 36.75ns$ in our design: a 4.1x latency reduction.

The upper bound latency, C_{max}, requires knowledge of the windowing method's worst-case time delay, C_{win}. For the high reliability time mode, C_{win} is the user-defined timeout value for a stored packet, therefore, $C_{max} = 3 + 2 + C_{win} = 5 + C_{win}$. To delay a packet for a maximum of 50 cycles, for example, we set the timeout to $50 - 5 = 45$ cycles.

As the high reliability count mode is time-independent, C_{max} serves as a poor measure of upper bound latency. As an example, lets us consider operations at the peak performance for TotalView-ITCH messages, where we receive around 1000 messages per second. From this we obtain: $n = 8$, $I = 2$, $\lambda_p = 6.5 \times 10^{-6}$ packets per cycle and $\lambda_m = \frac{9000}{6} = 1500$ messages per packet. Here, λ_m assumes the worst-case scenario of 100% packet utilisation, modelling a fully saturated market feed.

Using Eqn. 3.5 and setting the maximum-count threshold $MC = 10$ messages, we find $C_{win} = \log_2(8) + \frac{10}{2 \times 6.5 \times 10^{-6} \times 1500} = 516$ cycles. The upper bound latency is then $C_{max} = 3 + (2-1) + 516 = 520$ cycles, where $C_{write} = 1$ as one of the write cycles is performed in parallel. A worst-case delay of 520 cycles is a long time for TotalView-ITCH messages, but for OPRA we find an even

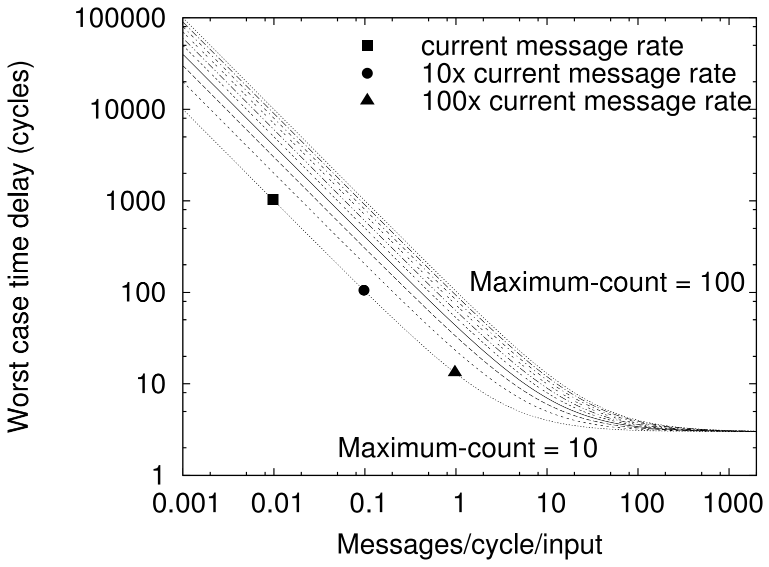


Figure 3.10: Worst-case time delay as an indicator of arbitrator saturation.

longer 10125 cycles. For $MC = 10$ it is possible for this threshold to be exceeded multiple time over within a single cycle, and indeed, this is the most likely scenario.

It is therefore best that C_{max} is not used as an indicator of upper bound latency, but rather as a measure of the incoming message rate. This suggests two possible uses for C_{max}: **(1)** a short-term measurement of message rates, or **(2)** as a measurement of throughput saturation within the arbitrator. We explore **(2)** in greater detail in Section 3.7.3.

For **(1)**, measurements of the current message rate align with traditional measurements of market trading activity performed by trading applications. Downstream applications could then use existing market analysis methods to set our arbitrator's windowing parameters in real-time, or conversely, the message rate, C_{max}, may itself be used to help inform trading decisions.

3.7.3 Saturation Estimation

To measure throughput saturation within our arbitrator, Fig. 3.10 uses C_{max} as a measure of the message rate to calculate the worst-case time delay, C_{win}. This is done for maximum-count values ranging between 10 and 100 messages. As the rate of incoming messages increases, C_{win} tends towards $\log_2(8) = 3$ cycles, indicating the arbitrator is becoming saturated. In this state, the high reliability mode effectively functions as a higher latency version of the low latency mode.

The value we calculated for C_{win} using our real market data implementation is plotted in Fig. 3.10 as a square symbol. From this we see that current market rates are well within the range that allows the high reliability modes to work effectively.

To model future message rates we take the rate of our test data—600 thousand messages per second—and increase it by one and two orders of magnitude. These are plotted as circle and triangle symbols, representing 6 million and 60 million messages per second, respectively.

We see that 6 million messages is over an order of magnitude away from saturation, meaning our arbitrator is well within its operational limits. The triangle, at 60 million messages per second, is closer to saturation, but has not yet crossed the inflection point. For such high message rates, the effectiveness of arbitration can then be increased using a higher maximum-count.

As even a message rate of 60 million messages per second is still less than the 10Gbps line rate of Ethernet, it is possible for our arbitrator to process 100 times the current market data rate using current technology.

3.8 Summary

In this chapter we outline an A/B line arbitrator for market data feeds, operating at the network level. We present an architecture that simultaneously produces a high reliability and a low latency output stream.

A key novelty in this work is the ability to dynamically configure the high reliability output stream's windowing methods. We can therefore adapt to the requirements of downstream financial applications in real time.

Our architecture is able to support any market data protocol, can be configured for different data-path widths, and can be deployed either at the network level, or within an individual computing node.

We present a model describing message and packet processing latencies, and discuss architectural considerations of efficient low latency designs. We discover latency measurements can indicate message feed saturation, providing a quantifiable method to indicate the point at which the low latency windowing mode becomes the optimal approach.

An implementation of our architecture on the Xilinx Virtex-6 FPGA demonstrates a lower bound latency of 7 cycles for high reliability processing, achieving latencies of $42ns$ for TotalView-ITCH message feeds, and $36.75ns$ for OPRA and ARCA feeds. Our new low latency mode performs simple arbitration within 1 cycle, corresponding to latencies of $6ns$ and $5.25ns$ respectively.

Finally, the most resource intensive protocol, TotalView-ITCH, is also implemented on a Xilinx Virtex-5 FPGA within a network interface card. We evaluate our design using real market data, and verify its operation using our new cycle-accurate testing scheme. Using our newly introduced models and performance estimators, we demonstrate the effectiveness of our design at message rates 100 times the current market level.

For the three messaging protocols examined, TotalView-ITCH, OPRA, and ARCA, we offer latencies 10x lower than an FPGA-based commercial design and 4.1x lower than the hardware-accelerated IBM PowerEN processor. Our arbitrator achieves throughputs of 21.3Gbps for TotalView-ITCH, and 24.3Gbps for OPRA and ARCA, more than double the 10Gbps network line rate.

Chapter 4

Maximising Parallel Memory Access for Low Latency FPGA Designs

4.1 Overview

In this chapter we look to reduce latency when one or more applications require parallel access to FPGA memory. When addressing this challenge, we tackle the issue of scaling to higher degrees of parallelism with the introduction of our new ring-based architecture. This scales linearly with both higher numbers of parallel accesses, and higher numbers of the FPGA BRAMs that collectively form the larger memory unit. Our interconnect requires little logic and routing, and automatically arbitrates parallel accesses to all BRAMs.

With a focus on memory-based computing—described in Chapter 2.4—our approach can reduce the latency of any 16-bit function to that of a 1 cycle memory read. With support for 1024 parallel function calls, the average latency becomes 0.5 cycles per function call by making use of the BRAM's dual read ports. An open-source tool is also provided to convert any deterministic Python function to our new architecture.

4.2 Introduction

In general, when designing applications, latency may be sacrificed for data precision in iterative operations—such as CORDIC-based implementations [146]—or traded for throughput by increasing the number of pipeline stages to allow for higher operating frequencies [147]. Applications that depend on low latencies are then at a disadvantage if they employ operations that are typically improved by these methods, or require higher degrees of numerical precision that can only come from latency-costly computation.

Such latency-dependent designs play a key role in real-time applications, like teleoperations and automation, as well consumer and business applications, from robotics to high frequency trading. These functions may be: formed of a chain of multiple, elementary operations; involve floating-point, multi-cycle, or other computationally expensive operations; or bespoke, user-defined functions for which no standard approximation will apply. For brevity, we will term these simply as "complex functions", and quantify them further during our implementation.

Traditional function optimisation, evaluation, and approximation can offer latency improvements, such as those based on distributed arithmetic [148], polynomial evaluation [149], or table-based methods [53]. However, pre-computing function values and storing them in low latency FPGA BRAMs is a much lower latency approach.

While costly in terms of memory resources, this method can be applied reliably to any deterministic function. BRAM also offers a dedicated memory solution, rather than the LUT-based approach taken by distributed memory. BRAMs can therefore boast both a high operating frequency, and a 1 cycle latency when returning the result of a given function input.

This approach can suffer when: **(a)** scaled to larger numbers of BRAMs; **(b)** we require higher degrees of data-path parallelism; or **(c)** there are limited options for parallel accesses to memory. For parallel access in particular, the unbalanced nature of available memory ports not matching the required number of parallel accesses also becomes an issue.

We describe in Section 2.4.3 how a large memory is traditionally formed within the FPGA by grouping together N_{SM} smaller sub-memory units. This new, larger memory is then accessed as a single, monolithic structure. With this scheme, independent access to the constituent BRAMs is no longer possible. Storing pre-computed values in look-up tables is not new, but little work has been done to leverage parallel accesses to the underlying BRAMs to benefit low latency designs.

To address the second challenge area we identify in Section 1.2, this chapter introduces a novel approach for highly parallel function evaluation that exploits the capability and parallelism of BRAMs on recent FPGAs. Our memory-compute core is able to serve multiple, parallel calls to its implemented function by independently accessing the BRAMs that are traditionally accessed as a single memory unit.

Our architecture's resource use and access times scale linearly to both higher degrees of parallelism, i.e., more calls to the implemented function, and the number of BRAMs accessed in parallel. The interconnection scheme requires little logic and routing, and automatically arbitrates parallel accesses between the multiple incoming function calls and their target BRAMs. Our approach decouples latency from the complexity of the implemented function, allowing for higher clock speeds and improved throughput.

In summary, the main contributions of this chapter are:

- a novel, minimally routed and interconnected architecture that independently accesses the multiple BRAMs that collectively serve as a function look-up table;
- an architecture that scales linearly to both larger numbers of incoming, parallel accesses, and to larger numbers of target BRAMs without the resource, routing, and latency costs seen by traditional approaches;
- a ring-based topology facilitating multiple, parallel calls to the implemented function, decoupled from the number of underlying BRAMs and available memory ports;
- a simple, tool-based design flow for implementing arbitrary functions as memory-based compute elements, along with an implementation and evaluation of square root, Gaussian, trigonometric and hyperbolic functions on an FPGA, showing enhanced performance, scalability, and resource utilisation over existing designs.

4.3 Targeting Low Latency Applications

We seek to benefit low latency applications through the use of a new architecture that leverages parallel accesses to FPGA memory. To achieve this, we break down the challenges we face into 3 areas: **(1)** which latency-sensitive functions within the application are best suited for memory-based computing (Section 4.3.1); **(2)** what type of memory should be used with such an architecture (Section 4.3.2); and **(3)** what form of memory interconnect should we use to meet our need for many parallel, low latency accesses (Section 4.3.3).

4.3.1 Function Choice

To benefit from our approach, designs require one or more of the following: **(A)** complex, multi-cycle functions; **(B)** high levels of data precision, but lacking the latency budget for iterative solutions; or **(C)** high levels of parallelism.

A: Complex, Multi-cycle Functions

The first group of functions that will benefit from our approach are those we termed complex functions in Section 4.2. These are formed of: a chain of multiple operations; involve computationally expensive operations; or are bespoke, user-defined functions. We later quantify this complexity when implementing and assessing our approach in Section 4.7.2.

With our approach we reduce the latency of such multi-cycle operations to that of a single memory read operation. When using fast, low latency memory accesses, this becomes the optimal design for low latency applications. However, a question remains in how we should manage the many, parallel accesses now required of memory. This is the challenge we seek to address with our architecture.

Our chosen memory-compute approach also see benefits when porting existing applications to the FPGA. Many library functions—such as those used by languages like C or Python, for example—may not have a corresponding FPGA implementation. If we pre-compute these function values in software, then load the results into FPGA memory, we remove the need for a new FPGA implementation, significantly reducing development time.

B: High Data Precision

The next set of functions that will benefit from our approach are those relying on high levels of data precision, but lacking the latency budget for iterative solutions. As well as requiring multiple cycles to arrive at an acceptable level of precision, such iterative solutions may also be non-deterministic. Instead, by using a pre-computed result stored in memory, we can guarantee a targeted level of both precision and time taken to return a result. The latter being a simple memory read.

Additionally, the repeated operations within an iterative solution may see the introduction of rounding errors. To reduce resource use, a traditional FPGA-based implementation—using DSPs, for example—may employ processes with lower levels of numerical precision compared to CPUs [150], compounding this error.

If we obtain pre-computed results from the higher-precision CPU implementation, then access these results from FPGA memory at run-time, we achieve more precise, reliable, and consistent results for FPGA applications. We examine and measure the benefits for data-precision with our approach in Section 4.7.1.

C: High Degrees of Parallelism

Finally, with our highly parallel memory-compute architecture, we look to benefit applications that require high degrees of parallelism. This includes scenarios in which many application instances run concurrently and independently—like multi-sensor robotics or monitoring systems [91] [92]—or where a single application may itself comprise many parallel computations, like neural network inference's [151] non-linear activation functions. In both cases, a single function is called many times in parallel, whether originating from one or many instances of an application.

One approach to scale logic-based designs to higher degrees of parallelism is to create a single data processing pipeline, then instantiate this pipeline multiple times. The availability of resources then becomes the limiting factor. Alternatively, multiple data-paths may share common resources, minimising resource use, but increasing latency and requiring the inclusion of arbitration logic.

Both are standard approaches supported by existing tools and compilers, such as FloPoCo [152]. Our architecture offers a different approach. Using independent, parallel accesses of sub-memories, and employing a greatly simplified control and routing scheme, our architecture scales much more favourably to higher degrees of parallelism. We explore this in greater depth in Section 4.3.3.

Addressing the challenges of parallel, memory-based computations outlined in Section 4.2, our approach therefore:

- uses less logic and memory, scaling more favourably to higher degrees of parallelism compared to the simple approach of duplicating processing pipelines;
- requires substantially less arbitration, routing, and control logic than that typically used to sequentialise accesses to a shared resource;
- minimises the latency penalty incurred when routing multiple data-paths to and from a shared resource, with the potential to eliminate it entirely.

4.3.2 Memory Type

When looking to optimise highly parallel, low latency access to memory, the type of memory we employ is an important factor. Different types of memory provide and prioritise different storage, access, and resource use requirements. As FPGAs offer a variety of memories, we review and assess their suitability for our approach. The important features to consider are:

- **low latency access**: minimising the number of clock cycles required to read a result from memory;

- **configurable interfaces**: implementing arbitrary functions requires the memory interface support a range of address and data widths;

- **available read ports**: more memory access ports means more parallel function calls can be performed per cycle, increasing performance, resource utilisation, and overall efficiency;

- **maximum operating frequency**: as well as faster access times, we can use techniques such as multi-pumping (Section 2.4.4) to increase the number of memory accesses per cycle.

Memory within an FPGA can be divided into two categories: latency-oriented, and bandwidth-oriented memories. LUT-based distributed memory, as well as block memories—like Xilinx's BRAMs and Intel's and M20K memory—would be classified as latency-oriented memories. Xilinx's UltraRAMs would also fall into this category. While the new HBM, along with conventional DDR memory, would be considered bandwidth-oriented.

Random read accesses suffer a high latency cost within bandwidth-oriented memories. Latency reduction is then more commonly explored for these memories via new interconnects and optimised multi-bank accesses. Within the FPGA, both HBM and DDR memory experience prohibitively long latencies, making them unsuitable for our goal of highly parallel, low latency accesses. Shuhai [153] finds HBM access to take 48+ cycles (106.7ns) and DDR4 22+ cycles (73.3ns) on the Alveo U280 accelerator card, both of which are non-deterministic due to possible bank conflicts and additional cycles needed for data refreshing, column/row switching, etc.

BRAMs have an operating frequency of 450MHz compared to 900MHz for HBM. However, when accessing the HBM, user logic is limited to 450MHz [154], so this benefit is reduced. HBM offers up to 32 parallel memory banks for 16GB memory, while BRAMs offer 2 read ports for their 18kbit (2.25KB) memory. This translates to 1 port per 4Gbits for HBM, and 1 port per 9kbits for BRAMs. This large comparative difference reflects the different goals of these memory types.

BRAMs are better aligned with our goals of maximising memory access parallelism and minimising latency, and are therefore the best choice for memory-based computation. BRAMs offer low latency, decentralised storage, and the ability to customise their address and data port widths. While the 288kbit UltraRAM has 16x the storage of BRAMs, it offers much less flexibility due to its fixed address and data widths. They are also available in fewer numbers on the FPGA itself, outnumbered by BRAM 4-to-1 on larger UltraScale+ devices, and 15-to-1 on smaller devices [40].

BRAMs are widely distributed throughout the device, allowing them to better serve as their own isolated processing elements. This improved localisation helps lower the routing cost when memory is used to supplement or replace existing LUT or DSP -based logic.

Forming Sub-memories

As we describe in Section 2.4, memory-based computing pre-computes all possible results of a given function then stores these values in FPGA memory. To map function inputs to memory addresses, our approach adopts the direct-mapping conversion method described in Section 2.4.1. Function inputs are concatenated into a single bit sequence of width W_{ip}, while at the output, the final function result has a width of W_{op}. To implement this function in memory using FPGA sub-memories of size S_{SM} bits, the number of sub-memories N_{SM} required is given by:

$$N_{\mathrm{SM}} = \left\lceil \frac{2^{W_{ip}} \times W_{op}}{S_{\mathrm{SM}}} \right\rceil \tag{4.1}$$

For example, a Xilinx BRAM holds 18kbits—which we will simplify as 2^{14} for this example—so a function with two 8-bit inputs and a 16-bit output will need: $N_{\mathrm{SM}} = \left\lceil \frac{2^{8+8} \times 16}{2^{14}} \right\rceil = 64$ sub-memories.

Assuming a 1-cycle read latency for BRAMs, D data-paths accessing a single memory unit would normally require D cycles to serve all of the data-paths. However, by independently interrogating each of the constituent N_{SM} BRAMs in parallel—each holding $\frac{1}{N_{\mathrm{SM}}}$th of the possible function results—it now takes D cycles in the worst case, and $\lceil \frac{D}{N_{\mathrm{SM}}} \rceil$ in the best. When the number of N_{SM} BRAMs is larger than or equal to the number of incoming D data-paths, it is even possible—with ideal data alignment—to serve all D data-paths in a single cycle.

Sub-memories serve as our quantum of memory, but the actual hardware definition of this varies between platforms. We primarily consider Xilinx's 18kbit block memories [45] here, but Intel's M20K [46] range of memory is equally applicable, and is available on the newest Stratix 10 series [155] of FPGAs. Intel's M9K [156] would provide twice the level of granularity, but this memory is only offered in lower numbers on smaller, low power Cyclone 10 devices, reducing this benefit.

4.3.3 Sub-memory Interconnect

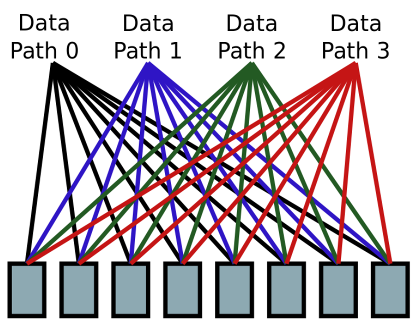
To connect D incoming memory reads with N_{SM} sub-memories, we build upon a uni-directional ring topology. The exact operation of our *ring-based architecture* is described in Section 4.4, but we consider its connections and behaviour here, and compare it to other common topologies.

Point-to-point, crossbar, multi-stage, and our ring-based topology are shown in Fig. 4.1. Each seeks to connect D=4 data-paths with N_{SM}=8 sub-memories. The exact connection scheme of multi-stage networks is not shown as this differs across different implementations, such as Clos [157], Benes [158], and Omega [159] networks. Such topologies traditionally connect $N \times N$ networks, so an asymmetric, $D \times N_{\text{SM}}$, interconnect leads to additional challenges when configuring and balancing links.

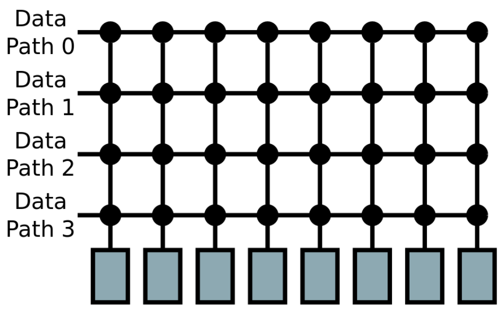
Table 4.1 highlights the three key areas we seek to minimise in regards to memory interconnect topologies: **(a)** the number of links; **(b)** the latency when traversing the interconnect; and **(c)** the need to arbitrate and manage memory accesses.

Links

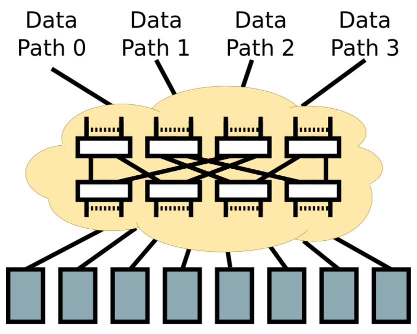
The number of links **(a)** determines how our interconnect will scale to larger numbers of sub-memories. A link is defined here as a connection between resources, for example a direct



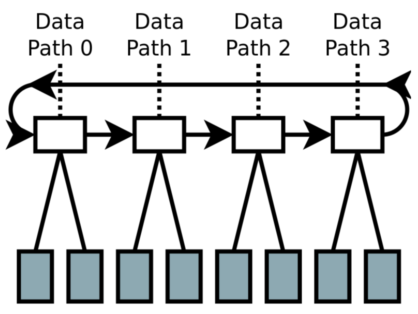
(a) Point-to-point.



(b) Crossbar.



(c) Multi-stage.



(d) Ring-based (this work).

Figure 4.1: Four memory interconnect topologies.

Table 4.1: Comparison of memory interconnects with D data-paths and N sub-memories.

Interconnect	**Links**	**Latency**		**Arbitration Logic**
		Best	**Worst**	
Point-to-Point	$O(ND)$	1	$O(D)$	$O(N)$
Crossbar	$O(ND)$	1	$O(D)$	$O(ND)$
Multi-Stage	$O(N \log D)$	$O(\log D)$	$O(\log D)$	$O(N \log D)$
Ring-based (this work)	$O(D)$	1	$O(D)$	$O(1)$

connection between logic and memory, or an intermediate connection between switch boxes. From Eqn. 4.1 we see that for memory-based computing, the number of sub-memories required scales exponentially with the width of the function's input data. Moving from an 8-bit input function to a 16-bit input function, for example, requires $2^{16-8} = 256$x the storage. Our requirements for large numbers of links mean point-to-point and crossbar are not viable due to their $O(N^2)$ link scaling.

We show in Section 4.4 that our ring-based architecture connects one-to-one with the incoming D data-paths. We therefore scale linearly with the D data-paths, and independently of the number of N_{SM} sub-memories when $D > N_{\text{SM}}$.

Latency

Regarding latency **(b)**, i.e., the number of cycles required to traverse the interconnect, the best case latencies will occur when all D incoming reads are served immediately and independently by the N=N_{SM} sub-memories. The worst latencies are seen when all D memory reads target the same sub-memory.

Interconnect latencies occur both when addressing memory and returning the result. As latency scales with the number of N_{SM} sub-memories, *not* the number of D parallel data-paths, this added minimum level of latency can be significant in practice. This is the primary motivation for not basing our architecture on traditional multi-stage interconnects.

Multi-stage interconnects can have lower latencies [160], but these never fall below $O(logN)$, even in the ideal case where incoming reads and the sub-memories they target are perfectly aligned. The topology of our ring-based architecture matches the best and worst-case latencies of point-to-point and crossbar networks, leaving the possibility open for correctly aligned data to provide optimal application performance.

Arbitration

Arbitration **(c)** relates to the need for an interconnect to manage the physical connections of different, possibly conflicting, routes parallel data will take through the network. For point-to-point this can be via a global controller, or on a peer-to-peer basis. Crossbars typically see routing decisions made at the individual switch points. The complexity of these schemes scales with both D data-paths and N_{SM} sub-memories. Multi-stage networks use a pipelined approach to route data, increasing latency but greatly simplifying arbitration.

The ring-based topology we employ rotates data past all N_{SM} sub-memories. While the length of the ring increases with the number of D data-paths, the amount of arbitration is constant. As each N_{SM} sub-memory is connected to a single D data-path, multiple data-paths cannot access the same sub-memory. There is therefore no possibility for conflicting memory accesses, greatly reducing our use of arbitration logic.

In summary, our ring-based architecture has low latencies, does not suffer from prohibitive scaling and routing requirements for N_{SM} sub-memories, and does not require arbitration to manage the D parallel data reads traversing the interconnect.

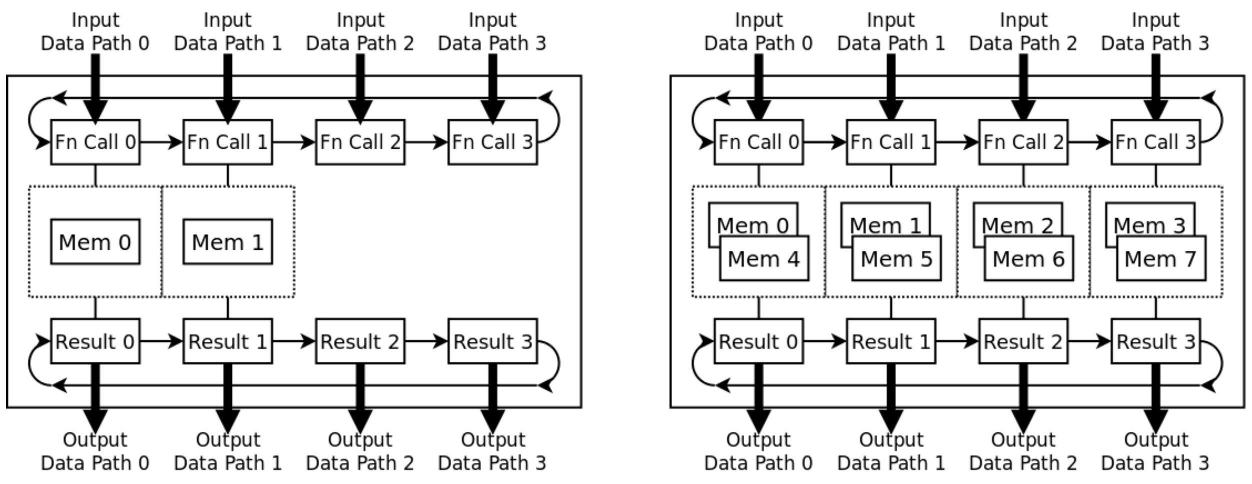
4.4 Ring-based Architecture

Our proposed architecture is based on a unidirectional, ring-based topology. It connects every parallel data-path, i.e., each call to the memory-based function, to a set of sub-memories in one cycle, and then a different set of sub-memories in the next cycle. All D parallel function calls are initially loaded into a circular shift register, and data are rotated from one data-path to the next every cycle.

We employ two circular shift registers, one at the input for the memory addresses, and one at the output for results (Fig. 4.2). These two registers are synchronised and rotate together, one data-path per cycle. The data in the input shift register, and the result in the output shift register, are then always aligned.

With our architecture, memory and logic connections are fixed to specific input and output registers, and the data themselves rotate around them. This enables concurrent lookup in all sub-memories, maximizing parallelism while reducing latency. This one-to-one connection of data-paths and sub-memories means there is no connection fan-in and fan-out to memory. We require no arbitration logic, and only a small amount of control logic for the two circular shift registers. This significantly reduces the logic and routing overhead traditionally seen when controlling parallel accesses to a shared resource.

With this uni-directional ring topology we provide additional "capacity" for parallelism. Additional sub-memories can be added and more data-paths can be served using fewer resources compared to the traditional approach of simply duplicating existing cores or pipelines. Since we decouple the number of available sub-memories from the number of data-paths, we can alter each independently.



(a) Processing 4 function calls with 2 sub-memories. (b) Processing 4 function calls with 8 sub-memories.

Figure 4.2: Examples of ring-based architecture configurations.

4.4.1 Configuration and Operation

Fig. 4.2 gives two example configurations of our ring-based architecture, each with D=4 parallel data-paths sharing a memory-compute unit. Function calls not initially served by sub-memories, such as *Fn Call 2* and *Fn Call 3* in Fig. 4.2a, are rotated past these memories in future cycles. Where sub-memories outnumber calls, like Fig. 4.2b, multiple sub-memories serve each call. The maximum latency is then bounded by the number of data-paths, rather than the number of sub-memories. In the extreme case, all sub-memories can be connected to a single data-path.

An example memory-compute unit is given in Fig. 4.3, containing N_{SM}=4 sub-memories, and processing D=4, 8-bit function calls. Each sub-memory then holds the full result for $\frac{1}{N_{\text{SM}}} = \frac{1}{4}$th of the possible inputs. Each cycle the sub-memories interrogate the data in their connected register. When data in a register targets the sub-memory connected to that register—i.e., its address is within the range the sub memory can serve—, we indicate this in the figure with a green tick.

In the initial cycle, cycle 0, the data in register *R2* targets the connected sub-memory, and the corresponding result is read out on the next cycle. The data in the remaining registers do not target their currently connected sub-memory, and so are not served. Each cycle the function call data are shifted one place to the right, so all data can ultimately be presented to each sub-memory.

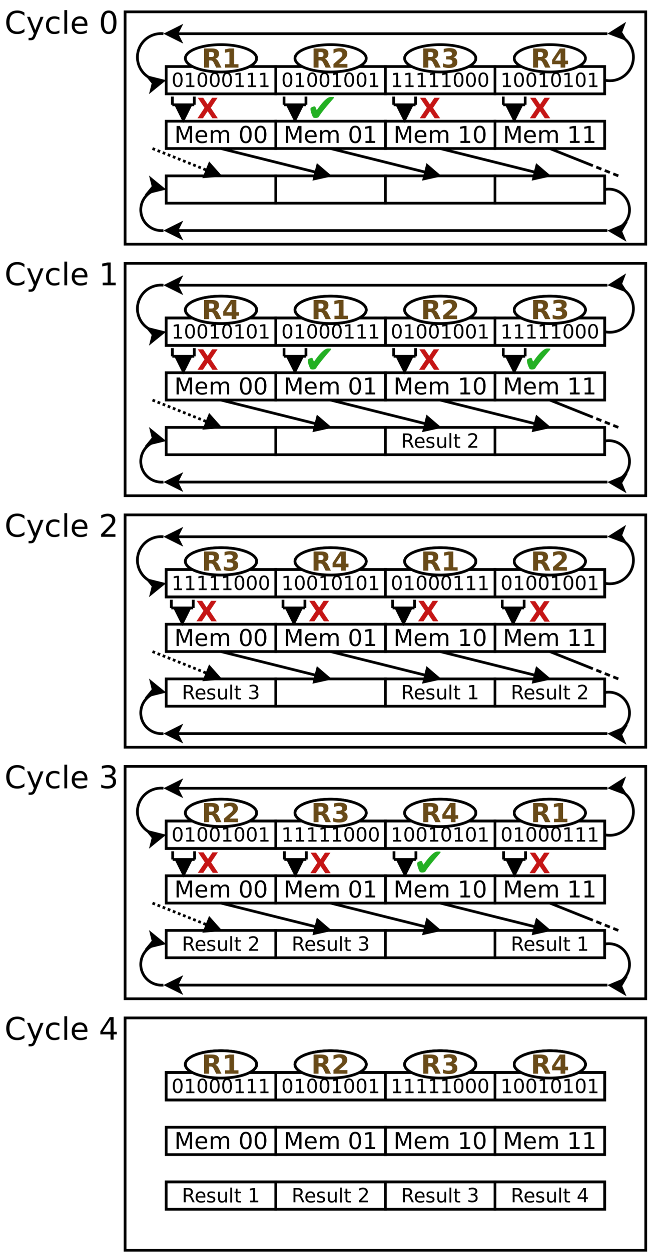


Figure 4.3: An example of our ring-based architecture processing four 8-bit inputs.

In cycles 1 and 2 respectively we see multiple function calls, or none, can be served each cycle. The memory-compute unit outputs the final result on the requesting data-path as soon as it is available. When all function calls have been fulfilled, requiring a maximum of D=4 cycles in this example, a new batch of D inputs are loaded into the input ring-register. This simplifies input routing, allows results to be output as soon as possible, and ensures the input pipelines remain synchronised.

4.4.2 Latency

We now look to determine the latency cost of our ring-based architecture. For traditional, pipelined architectures accessing a shared resource, the latency to process D data-paths is:

$$L_{\text{PIPE}} = D + N_{\text{ST}} - 1 \tag{4.2}$$

where N_{ST} is the number of pipeline stages. BRAMs can support multi-stage pipelines through the use of internal registers. These longer pipelines allow for higher frequency designs and improve throughput, but can be detrimental to designs that: require low latencies; access a shared resource; or deal with non-streaming, irregular data.

With our ring-based architecture and decision to implement 1-cycle read latencies, the worst-case latency is D cycles for D function calls. As we can serve multiple calls at once, the lower-bound latency for *any* degree of parallelism approaches the 1-cycle sub-memory access.

This opens up a new opportunity for latency optimisation. By ordering parallel function calls within the input shift register to align, or be more likely to align, with their targeted sub-memories, then results will be present within the first sub-memories we check. This lowers expected latency when serving the D parallel function calls. Future work will explore methods for achieving latency reductions based on the distribution and access patterns of the incoming function calls.

As discussed in Section 4.3.2, the memory architecture itself plays a key role in improving latency and resource utilisation. Additional read ports for each sub-memory effectively increases the number of sub-memories. That the contents of this memory are already present in our architecture does not result in redundant accesses. When interrogating each sub-memory in turn—i.e., when rotating data within the input and output ring-registers—our step-size is then just equal to the number of available memory read ports, P. Thus, the worst-case latency for our architecture when processing D parallel calls to a function is found to be:

$$L_{\text{MEM}} = \left\lceil \frac{D}{P} \right\rceil + 2 \tag{4.3}$$

The additional 2 cycles of latency are a result of architectural overhead (not present in the Fig. 4.3 example), and become an increasingly negligible factor at higher levels of D parallelism.

4.5 Automated Tool

A tool [1] was developed to convert any arbitrary function or existing memory structure to our new architecture. To support a wide range of high-level functions we opt to convert functions written in Python. Python's use is particularly important due to its wide range of existing libraries. Packages such as NumPy [161], Scikit-learn [162], and SciPy [163] offer tried and tested, industry-standard implementations of common functions.

Porting these library functions to FPGAs has been explored before [164] [165], often necessitating the transformation or approximation of the underlying function. This is also the approach used by tools such as FloPoCo.

As mentioned in Section 4.3.2, our architecture adopts the direct-mapping conversion method to map function inputs to memory addresses. Our tool must therefore account for every possible combination of inputs when converting a function, so we opt for a brute-force approach to conversion. This method places no requirements on the function itself, such as being continuous or periodic, and does not limit the format or numerical precision of data.

[1] github.com/sdenholm/memory-compute

```python
def mult(x, y):
  return x * y
```

```yaml
---
vivado-project-directory: "/path/to/vivado_project_dir"
function:
  name: "mult"
  arguments:
    - name:  x
      type:  uint
      width: 4
    - name:  y
      type:  uint
      width: 4
  output-width: 8
  output-type:  uint
```

(a) 4-bit unsigned integer multiplier.

```python
def tanh(x):
  import numpy as np
  xNumpyFloat = np.float16(x)
  return np.tanh(xNumpyFloat)
```

```yaml
---
vivado-project-directory: "/path/to/vivado_project_dir"
function:
  name: "tanh"
  arguments:
    - name:      x
      type:      float
      width:     16
      precision: 11
  output-width:     16
  output-type:      float
  output-precision: 11
```

(b) 16-bit floating-point implementation of NumPy's tanh.

Figure 4.4: Examples of Python function and configuration files for our automated tool.

We therefore **(a)** guarantee accurate and consistent results, **(b)** provide a high degree of function compatibility, and **(c)** lower the barriers for entry when porting new or modified versions of existing functions to FPGAs.

4.5.1 Function Conversion

A configuration file is used to describe the number of function inputs, as well as the width, type, and precision of all inputs and outputs. We support any degree of precision for floating-point numbers, not just those of IEEE 754 [166].

Example functions and their corresponding configuration files are shown in Fig. 4.4. In Fig. 4.4a we show a simple multiplier, taking two 4-bit unsigned integer inputs and producing an 8-bit unsigned integer result. Fig. 4.4b converts NumPy's implementation of tanh to our memory-computing architecture, using 16-bit, half-precision floating-point numbers.

We place no limitations on the implementation of the Python function. Developers are able to make use of any Python libraries, functions, and data structures, as well as connect to external resources, such as databases. We require no new or proprietary interfaces or external tools, so any existing Python function is compatible with our tool.

The tool converts a function into a mapping of input to output values, storing the results in memory configuration files targeting Xilinx's Vivado 2019.2 Design Suite. A Verilog module is created to group the N_{SM} sub-memories together and connect them using our new ring-based architecture. The user sets the number of required D parallel function calls using the instantiation parameters of the Verilog module. As Verilog is a commonly supported language, and the memory configuration files use a simple comma-separated-values (CSV) structure, it is straightforward to integrate our tool into other development tools and toolchains.

There are multiple ways of arranging sub-memories within the architecture. In the case of fewer data-paths than sub-memories ($D < N_{\mathrm{SM}}$) we must determine how many sub-memories to connect to each incoming D data-path. When sub-memories outnumber data-paths ($D > N_{\mathrm{SM}}$) the spacing between sub-memories is our main concern, i.e., which data-paths are not connected to any sub-memory.

Our tool automatically makes an optimised arrangement. We evenly distribute and space out sub-memories across all available data-paths, with the goal of maximising the chance a function call will be served, thus lowering the overall expected latency. When sub-memories outnumber data-paths, we first populate every data-path with a sub-memory, then evenly space out the remaining sub-memories, overlapping those placed earlier.

4.5.2 Hybrid Computation

For memory-based computing, "computation" is now defined simply as reading a value from memory. The original function's internal structure and control flow have no effect on the latency, resources, or overall performance of our new computation process.

Fig. 4.5 gives an example hybrid function we can convert with our automated tool. The function has two inputs: a 2-bit selector and a 12-bit floating-point variable. The function uses the 2-bit selector to choose both the operation to perform, and which pre-defined constant to use.

With such a configuration we can control the use of different function configurations at run-time. For example, selecting the use of different operations, filter parameters, input and output data formatting, etc.

```python
def fn(select, value):
  import numpy as np

  if select == 0:
    return np.sin(value)
  elif select == 1:
    return np.cos(value)
  elif select == 2:
    someConstant = 112358
    return np.tan(value) + someConstant
  else:
    someOtherConstant = 314159
    return np.tan(value) + someOtherConstant
```

```yaml
---
vivado-project-directory: "/path/to/vivado_project_dir"
function:
  name: "fn"
  arguments:
    - name:  select
      type:  uint
      width: 2
    - name:  value
      type:  float
      width: 12
      precision: 7
  output-width:     16
  output-type:      float
  output-precision: 11
```

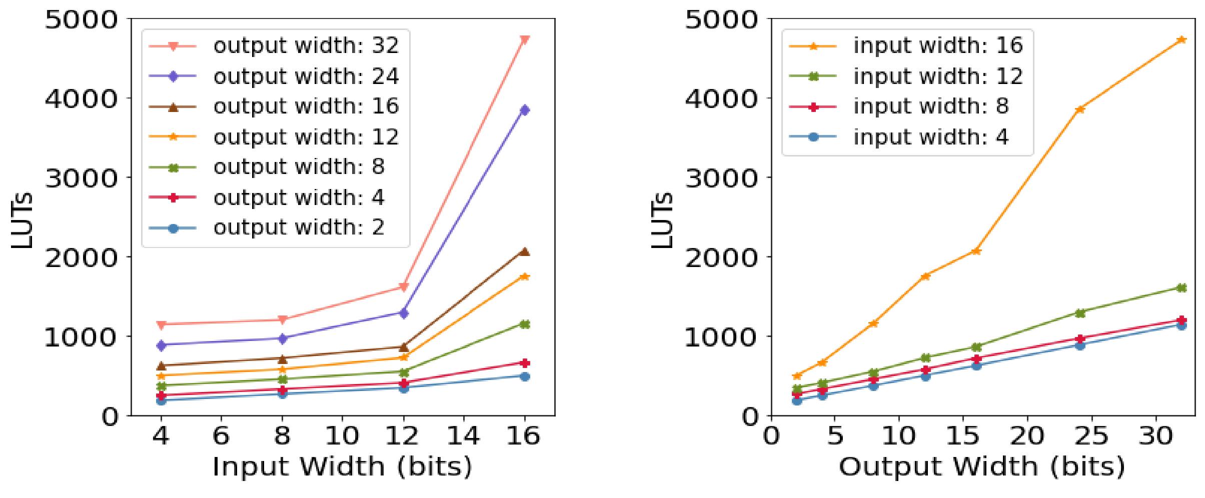
Figure 4.5: Example hybrid function with a 2-bit selector, 12-bit input, and 16-bit output.

The example function's input is a non-standard 12-bit float with 7 bits of precision, while its output is a 16-bit float with 11 bits of precision—the IEEE 754 standard for half-precision floating-point numbers. By setting such a configuration using our tool, and through the use of pre-computed values, we effectively define a function that creates an additional 4 bits of precision at the output. Mixing and matching data types can be accomplished without compromising the precision or accuracy of the original calculation. We explore this idea further when analysing our results in Section 4.7.1.

4.6 Implementation

To assess our memory-based architecture we implement a range of design configurations on the Xilinx Alveo U280 accelerator card. Designs are built using the automated tool we describe in Section 4.5, and developed using the Xilinx Vitis and Vivado 2019.2 design tools.

We test functions possessing the features identified in Section 4.3.1: high function complexity, requiring high levels of data precision, and used within applications employing high levels of



(a) Varying input data widths. (b) Varying output data widths.

Figure 4.6: Relationship between function input and output widths and LUT utilisation.

parallelism. Tests are performed for D=2 to D=1024 parallel data-paths, with all designs operating at 300MHz.

We use Xilinx BRAMs for memory, but for Intel devices the M10K and M20K [46] block memories are also suitable. Our approach is extendable to any uniform-latency memory.

4.6.1 Data Width and Resource Scaling

Fig. 4.6 plots resource use versus data width. From Eqn. 4.1 we expect the required memory to scale exponentially with the data width of the function's input, and linearly with data width of its output. While the function's output data width should not be discounted, we expect input width to be the dominant factor in determining our overall resource use, and this is reflected in our results.

We see an inflection point at 12-bit inputs. Below this we use only a single BRAM to store function values so cannot leverage our ring-based architecture's parallel accesses. Functions with data widths less than this will therefore see little or no resource reduction with our approach. 16-bit input and output functions do not require significantly more resources than 12-bit, and as a commonly used data width is the more practical choice to consider when adapting functions to our architecture.

Given the data width budget, the use of univariate functions is recommended to maximise data precision per function argument. Multivariate functions are supported as long as total data width is below our target limit, e.g., one 16-bit input is equivalent to two 8-bit inputs.

INT8—for 2-input functions—and 16-bit are common formats in GPU processing and machine learning [167] [168]. Gupta [151] finds 16 bits sufficient for training some deep neural networks, while Chu [169] shows that progressively decreasing data widths within a neural network has little effect on performance due to the corresponding decrease in the range of feature sets.

4.6.2 Resource Use and Parallelism

As the most practical choice, we now focus on supporting 16-bit functions. Table 4.2 shows the LUTs required by our architecture increases from 889 to 75,080 when moving from supporting D=2 to D=1024 parallel function calls. A 512x increase in parallelism sees only a 85x increase in LUTs, while the number of required 18kbit BRAMs remains constant. Our approach becomes more resource efficient as the required degree of parallelism increases, maintaining its 300MHz operating frequency up to D=1024 parallel function calls.

Linear regression analysis was performed on our LUT and register use on the Alveo U280, given in Table 4.2. As we scale to higher numbers of D function calls we find:

$$LUTs = 72.3 \times \mathrm{D} + 1320 \qquad \text{P-value} < 0.001$$
$$REGs = 66.6 \times \mathrm{D} + 2690 \qquad \text{P-value} < 0.001$$

We plot the results from Table 4.2 in Fig. 4.7 and overlay the best-fit lines from our linear regression analysis. The X and Y axes are scaled logarithmically to better highlight the areas where our resource use deviates from that predicted by the regression model.

Table 4.2: Our ring-based architecture's resource use for any 16-bit function.

	Data-paths									
	2	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
LUTs	889	1,152	1,450	2,072	3,412	7,231	11,000	20,506	38,398	75,080
Registers	2,419	2,749	3,020	3,639	4,170	7,296	11,898	20,279	37,246	70,487
18kbit BRAMs	64	64	64	64	64	64	64	64	64	64

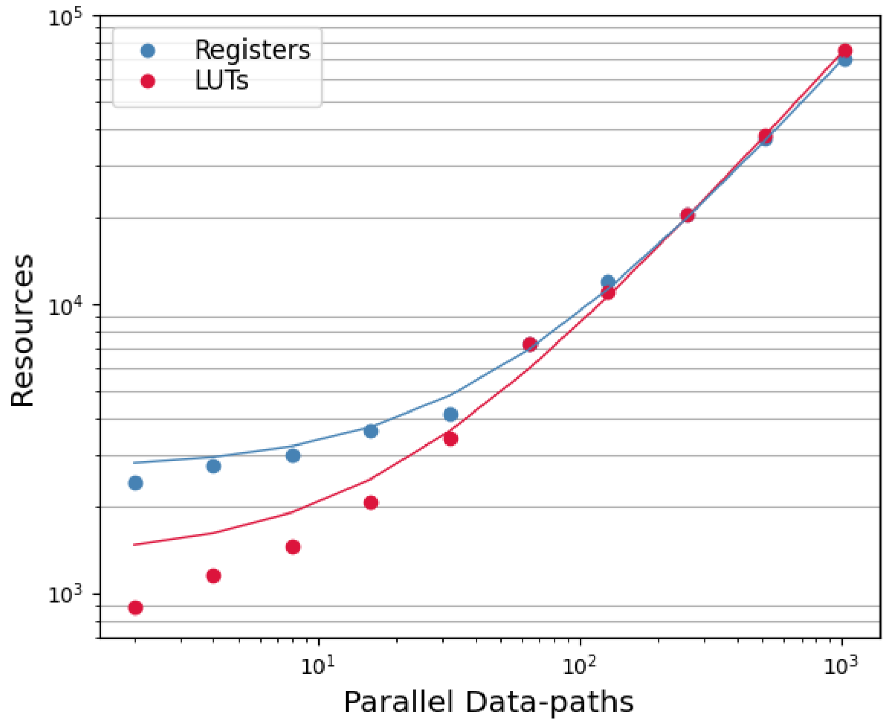


Figure 4.7: Resource use and best-fit linear regression predictions for the Alveo U280.

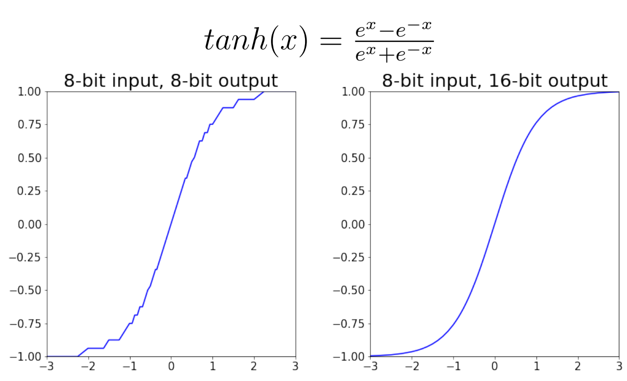
Register use closely matches predictions at all degrees of parallelism, but the model overestimates LUT use for $D < 32$ parallel data-paths. Below this level, resource use scales much less favourably, with the inflection point occurring around $D = N_{\text{SM}}$.

From both our implementation and Eqn. 4.1, we note that 16-bit function implementations require $N_{\text{SM}} = \left\lceil \frac{2^{16}\times 16}{2^{14}} \right\rceil = 64$ BRAMs. Our approach targets functions with high degrees of parallelism, i.e., many D parallel function calls and/or data-paths. Based on these results, we can more formally quantify our definition of "high degree of parallelism" to be $D \geq N_{\text{SM}}$.

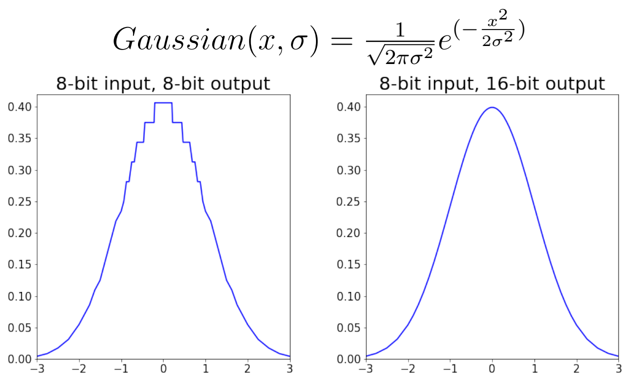
4.7 Results

4.7.1 Pre-computed Values and Data Precision

Memory-based computing converts function inputs to memory addresses, and function outputs to data stored in memory. We therefore represent functions at the function level, rather



(a) $tanh(x)$ function.



(b) $Gaussian(x, 1)$ function.

Figure 4.8: Comparing 8-bit input functions when outputting with 8-bit and 16-bit data widths.

than as a collection of constituent computations. This is important when considering the complex functions we discuss in Section 4.3.1. Such functions may use a number of intermediate calculations—multiplication, divisions, square roots, etc—that are performed before returning the final result.

Should we wish to use 16-bit floating-point representations on the FPGA, for example, the traditional computing approaches see us perform all these intermediate calculations with this 16-bit data width and its 11 bits of precision at run-time. With our approach we can instead achieve a higher level of accuracy by pre-computing function values on the CPU with arbitrary precision, then converting **only the final result** to a 16-bit representation.

Consider that Eqn. 4.1 and Fig. 4.6 show our architecture scales exponentially with a function's input data width, but linearly with its output data width. As discussed in Section 4.5.2, our tool supports the use of different data formats for input and output data. Our approach is therefore able to decouple the data widths of a function's input and output.

With wider output data widths we increase the fidelity of our result. Fig. 4.8 shows the $tanh(x)$ and $Gaussian(x, 1)$ functions with 8-bit inputs and their corresponding 8-bit and 16-bit outputs. The outputs are derived using 128-bit floating-point representations, then the final results are converted down to 8-bit or 16-bit respectively. This simple example shows that even with a limited input range, a wider output width has a noticeable effect on our ability to accurately represent our underlying function.

For some functions this additional fidelity is achievable with other approaches, such as CORDIC using iterative calculations to increase precision. However, only with pre-computed results can this opportunity be made available to every function.

Differences in input and output data widths are also useful when converting between data types, such as calculating the square root of an integer to produce a precise, floating-point result. Alternatively, when used at the terminal stage of a neural network, it could directly feed quantised INT8 data into later, de-quantised stages of an application pipeline.

4.7.2 Latency Reduction of Complex Functions

By following a generalised roofline model for FPGAs [170], we determine function complexity using computational intensity, i.e., the number of operations per byte. The more operations a function performs on each datum, the more cycles it is likely to require. In the context of an FPGA, this metric sees an operation occur every cycle, which is then normalised by dividing by the width of the data-path.

By averaging 2 function calls per cycle, i.e., 0.5 cycles per function call—described below—our architecture offers a uniform computational intensity of 0.25 ops/byte for any 16-bit function, doubling to 0.5 ops/byte for 8-bit functions. This sets the minimum boundary for performance improvement.

Our approach is function agnostic, but simple functions like integer arithmetic (0.17 ops/byte) or more "fundamental" operations supported by DSPs—like multiply-accumulate (0.33 ops/byte)—see little benefit from memory-based computing in general. The complex, low latency functions we aim to improve with our architecture are then those with higher computational intensity, such as square root, sinh, cosh, tanh, or any user-defined, multi-cycle function. With our approach, these will see their computational intensity reduced, and latency lowered to just 1 cycle.

Table 4.3 compares past works to a parallel, 16-bit implementation of our memory-compute architecture on the Xilinx Alveo U280. Assessed against interpolation [165], function approxi-

Table 4.3: Our approach vs. common FPGA functions in past works.

	Comp. Inten.	**Width (bits)**	**Per Function Call**					**Platform**	**Process**	**Latency Improv.**
			LUTs	**DSPs**	**Memory (kbits)**	**Cycles**	**Latency**			
square root										
CORDIC [171]	4.0	16	247	0	0	16@330MHz	48.48ns	UltraScale+	16nm	**29.0x**
sin, cos										
CORDIC [172]	3.5	16	697	0	0	14@421MHz	33.32ns	Cyclone II	90nm	**20.0x**
CORDIC [171]	5.0	16	995	0	0	20@720MHz	27.78ns	UltraScale+	16nm	**16.6x**
FloPoCo [152]	1.0	16	297	5	0	4@298MHz	13.42ns	Virtex-5	65nm	**8.0x**
CORDIC [152]	1.0	16	713	2	0	4@238MHz	16.81ns	Virtex-5	65nm	**10.1x**
CORDIC [152]	3.3	16	719	2	0	13@368MHz	35.33ns	Virtex-5	65nm	**21.2x**
sinh, cosh										
CORDIC [171]	5.0	16	1,143	0	0	20@720MHz	27.78ns	UltraScale+	16nm	**16.6x**
tanh										
Fn. Approx. [164]	1.5	13	137	0	-	5@436MHz	11.47ns	Virtex-4	90nm	**6.9x**
DCT Interp. [165]	0.5	16	129	0	1250	2@264MHz	7.60ns	Virtex-7	28nm	**4.6x**
CORDIC [171]	5.3	16	550	0	0	21@720MHz	29.17ns	UltraScale+	16nm	**17.5x**
atan2										
Taylor [173]	1.3	16	329	6	72	5@220MHz	22.73ns	Virtex-6	40nm	**13.6x**
Taylor [173]	1.3	16	331	6	72	1@135MHz	7.41ns	Virtex-6	40nm	**4.4x**
CORDIC [173]	2.0	16	796	0	0	8@389MHz	20.57ns	Virtex-6	40nm	**12.3x**
CORDIC [173]	2.0	16	816	0	0	2@191MHz	10.47ns	Virtex-6	40nm	**6.3x**
Gaussian										
FloPoCo [174]	2.8	16	770	0	32	11@300MHz	36.67ns	UltraScale+	16nm	**22.0x**
any function										
This Work	**0.25**	**16**	**81**	**0**	**1152**	**0.5@300MHz**	**1.67ns**	**UltraScale+**	**16nm**	

mation [164], CORDIC [152] [172] [173] and FloPoCo [152] [173] [174], we see that it is difficult for other function implementations to match the 1-cycle latency of memory look-ups.

Memory choice—described in Section 4.3.2—plays an important role in our results. By using the 2 read ports of BRAM we are able to further lower our latency, averaging just 0.5 cycles per function call (Eqn. 4.3).

We achieve up to 29x latency reduction, as well as a 37-92% reduction in LUTs per function call using our new architecture. Our resource use in Table 4.2 and linear regression results from Section 4.6.2 show we scale favourably to higher degrees of parallelism.

Highly pipelined applications—typically identified in Table 4.3 by their higher cycle count and operating frequency—are able to approach our *per cycle* latency. However, their long pipelines result in a higher overall latency and, as discussed in Section 4.4.2, also incur higher resource costs.

Memory use itself can also be improved compared to past works, as our ring-based architecture facilitates more efficient and streamlined access to shared memory resources. Abdelsalam [165] uses 1250kbits of memory to store approximations for a single tanh function, while we use 1152kbits to support the entire function. Their memory use can be reduced at the cost of data precision.

The improvements with our approach do not come just from the inherent benefit of memory. A lowered computational intensity makes functions more sensitive to I/O bandwidth limitations. Our choice of a simplified uni-directional ring interconnect, described in Section 4.3.3, ensures we maintain a high 300MHz operating frequency up to D=1024 parallel data-paths. This is the key benefit to our approach: a 0.5-cycle average latency and a high, sustained 300MHz operating frequency at higher degrees of parallelism for *any* 16-bit function.

4.8 Summary

In this chapter we present a novel architecture to improve highly parallel memory access for low latency applications. Using pre-computed function values and memory-based look-ups, we use a new, ring-based topology to arbitrate independent access to the N_{SM} individual BRAMs that are traditionally grouped together and accessed as a single, monolithic memory unit.

Our interconnect requires less logic and routing compared to point-to-point, crossbar, and multi-stage networks. We scale linearly with both higher D degrees of parallelism, i.e., more calls to the implemented function, as well as the number of BRAMs accessed in parallel. For such an interconnect, we find this scaling is most efficient when the number of D parallel function calls exceed the number of N_{SM} sub-memories.

We target low latency applications with: **(a)** complex, multi-cycle functions, **(b)** high data precision requirements, and/or **(c)** high levels of parallelism. We determine that functions require a computational intensity—i.e., the number of operations per byte—of at least 0.25 ops/byte for 16-bit functions, or 0.5 ops/byte for 8-bit functions to see benefits from our approach.

An automated tool is introduced to convert existing Python functions to our new architecture. No function transformations or approximations are used, and we support both integer and arbitrary precision floating-point numbers. This guarantees **(a)** accurate and consistent results,

(b) a high degree of function compatibility, and **(c)** low barriers for entry when porting new or existing functions to our FPGA architecture.

We implement our new architecture on the Xilinx Alveo U280 accelerator card, finding 12-bit functions to be the most resource efficient, but 16-bit designs are preferred due to their alignment to standard data formats. When assessed against square root, Gaussian, trigonometric, and hyperbolic functions, we see a 37-92% reduction in LUTs per parallel function call, and a 4.4x to 29x reduction in latency.

A key benefit of our approach is support for *any* 16-bit function, offering a 1-cycle latency and a sustained 300MHz operating frequency for up to 1024 parallel function calls. Using the 2 read ports of Xilinx BRAMs we further lower latency, averaging just 0.5 cycles per function call.

Chapter 5

A Unified Approach for Managing Heterogeneous Processing Elements on FPGAs

5.1 Overview

In this chapter we extend our ring-based architecture from Chapter 4 to support shared, parallel access to *all* of the FPGA's heterogeneous processing elements. A key challenge lies in supporting processing elements with different implementations, connections, pipeline lengths, and latencies. We address this with our implementation-agnostic, call-and-response approach to computation. We also define a latency-centric performance model that is able to quantify and compare the performance of current, and future, processing elements.

Where past works unified access to processing elements by defining new "functional units" composed of FPGA resources, we unify processing elements according to their function, *not* their resource composition. We group elements that implement the same function into compute pools, with each pool able to serve multiple, parallel function calls. We see greatly improved resource utilisation, demonstrating 96.2% utilisation for 1024 parallel data-paths accessing our new compute pool architecture.

Our function-oriented approach also means we do not require the creation of new development tools or design flows. All existing design, development, and analysis tools are compatible with our architecture.

When evaluating applications using our architecture, we see an average 21% improvement in throughput and 29% in latency for 8 neural network activation functions, when compared to the traditional dataflow processing model. We also explore new avenues of performance improvements with our fractional processing elements, which we describe more detail in Section 5.5.

5.2 Introduction

To meet the challenge of supporting processing elements with different implementations, connections, pipeline lengths, and latencies, we must first examine the processing elements themselves. Modern FPGAs increasingly see the addition of specialised processing elements targeting specific domains, such as audio, video, and communications [8], and more recently, artificial intelligence, machine learning, and autonomous robotics [50]. These processing elements take the form of hard-coded, embedded cores and are present alongside the traditional LUT, DSP, and BRAM resources.

Developers will not typically utilise all available FPGA processing elements when creating applications. When making use of many different processing elements, additional work is required to manage their different: operating frequencies, end-to-end latencies, pipelining capabilities, and individual customisation options.

If we adopt a simple dataflow approach to manage heterogeneous processing elements, we face difficulties balancing parallel pipelines if elements have different numbers of pipeline stages, or stages have different degrees of internal complexity. A more robust and unified approach is needed to balance and manage these different processing elements.

A common, high-level approach to unify and manage processing elements is to adopt a more coarse-grained approach to FPGA development. For this, new overlays or virtual FPGAs may be developed, which we describe in greater detail in Section 2.3. Such architectures create a more homogeneous layout within the FPGA by defining their own functional units composed of multiple FPGA resources. User applications are then defined using these new functional units.

However, when such frameworks are configured with a user application [31] [175] we see wasted resources, as not all of the resources within each functional unit are used. Similarly, as functional units are now the *only* available processing element, FPGA resources not included within functional units cannot be used. These excluded resources are typically newer, or less ubiquitous FPGA processing elements, or specialised, high performance resources, such as memories [16] [17], or domain-specific cores [176].

Instead, we introduce a novel architecture to group heterogeneous processing elements into a unified compute pool. Rather than defining new functional units, we provide a unified means of managing *existing* implementations of processing elements.

The E elements in a pool will each implement the same function—such as tanh, for example—with a compute pool serving D parallel function calls to the pool's implemented function. The number of D parallel function calls is then independent of the number of E processing elements.

This approach follows the recommendations of past works [177], which show the aim of coarse-grained architectures should be to maximise logical functions compared to their associated routing. Our compute pool architecture is therefore designed to de-couple the number of E processing elements from the D accessing function calls, letting us scale them independently.

We deal with processing elements based solely on their total, end-to-end latency by employing a call-and-response approach to computation. This creates a resource and implementation agnostic view of processing elements, simplifying development. It also allows any type of processing element to be easily added to the pool, even those with non-deterministic latencies.

The D parallel function calls—or data-paths—will often outnumber the available E processing elements, requiring the use of an interconnect to route and manage accesses. This asymmetry is typically due to **(1)** designers limiting the number of processing elements to save resources, and/or **(2)** the application demanding high levels of parallelism.

Neural network activation functions are an ideal candidate and test-bed for such an architecture. As described in Section 2.6, neural networks are formed of many interconnected neurons, typically grouped into layers. All neurons in a layer implement the same activation function, and layers can be repeated multiple times within a single network. This leads to high levels of data parallelism, with each neuron calling the same activation function. We then collect together the different FPGA processing elements that implement this activation function into a compute pool, and manage access for all the connected neurons.

Our compute pools are able to support a range of processing element implementations. A centralised pool also provides a simple means for designers to add processing elements to improve performance, or remove them to reduce resource use. Our data routing scheme—described in Section 5.4.2—is able to reduce the bottleneck associated with multiple data-paths accessing a centralised pool, and almost eliminates it entirely for designs with known latencies.

Our compute pools are then logically, rather than physically, centralised. This allows for more *decentralised* designs and routing, while maintaining *centralised* arbitration of incoming function calls. We can therefore balance the use of all processing elements across the entire application.

The main contributions of this chapter are as follows:

- a new architecture where compute pools of E heterogeneous processing elements, each implementing the same function, serve D parallel function calls to their given function, thus evenly sharing and balancing access to all FPGA resources;
- a call-and-response approach to computation to support processing elements: with different implementations; formed of different resources; operating with different latencies; and those elements targeting only a narrow range of all possible inputs, such as fast function approximations;
- an efficient and scalable interconnection framework supporting high degrees of parallelism, and flexible enough to implement applications with different design latency, throughput, and resource use goals;
- new opportunities to trade-off performance with resource use, described with a generalised model, and assessed on an FPGA using ReLU, LReLU, ELU, GELU, sigmoid, swish, softplus, and tanh functions.

5.3 Motivation and Challenges

For simplicity, we can classify our general-purpose compute pool architecture as performing three functions:

- evenly share E processing elements within a compute pool among D accessing data-paths;
- support different types, implementations, and functionality of processing elements within an FPGA;
- define compute pool elements by function, rather than resource composition, creating a higher-level view to unify heterogeneous processing elements.

Here we look at past works to see how they address these challenges and to identify areas where our new architecture can provide improvements to performance and resource use.

5.3.1 Evenly Sharing Processing Elements

FPGAs contain a large number of processing resources, such as LUTs, DSPs, BRAM, and embedded processors. These resources may be grouped together to collectively form a processing element. Alternatively, some resources, such as embedded processors, can serve as stand-alone processing elements.

When using a dataflow model of computation, processing elements will serve one or more incoming data streams. Matching the D incoming data-paths with the E available processing elements is one of the problems we seek to solve with our centralised compute pool approach.

Past works that deal with multiple parallel accesses to a shared resource include HBM [16] [17] interconnects [153], and packet switching and time-multiplexing within FPGA overlay networks [178]. Multiple-Input, Multiple-Output (MIMO) queues also focus on this issue [179], both in the context of data routing and variation tolerant processing.

When assessing these different topologies we find three areas to *minimise* when creating our new interconnect: **(a)** the processing element **access latency**, achieved by reducing the number of links, i.e., hops, required for data to pass from the incoming data path to an available processing element; **(b)** the number of **connections** as we scale to more D data-paths and E processing elements in the compute pool; and **(c)** the **arbitration logic** used when managing the many parallel data movements within the interconnect.

We seek to address these points with our new compute pools and interconnect.

5.3.2 Utilise and Support FPGA Resources

In Section 2.3 we describe how coarse-grained architectures define new functional units within the FPGA. They do this by grouping together FPGA resources into new, homogeneous functional units. This approach sees wasted resources when an application built atop this new architecture has resource requirements that differ from those within these new functional units. For example, the application may require a higher ratio of LUTs to DSPs than are available in a single functional unit.

There is no universal resource ratio that will satisfy every application. This utilisation penalty is then an important metric within coarse-grained architectures, with utilisations of 22% [83] or lower being reported. This is accepted because the goal of these past works is typically not maximising performance, but simplifying development and speeding up device configuration. The latter being of particular importance in fields benefiting from fast configuration times.

Additionally, at higher levels, coarse-grained architectures tend to be formed of $M \times N$ grids [82] [180] of functional units, or groups of units. This limits the granularity of the overlay or framework when applied to an FPGA. If there is only space for $N - 1$ additional functional units, then an additional row/column cannot be added.

We therefore see the need to create a scalable and flexible architecture to support large, and potentially non-uniform configurations of heterogeneous processing elements. Our approach is not to define new functional units, but bring together **existing** implementations of processing elements, whether formed of one or many FPGA resources.

Our new compute pools contain different implementations of processing elements, but each element performs the same function. A simple example would be a compute pool that performs multiplication. It would include one or more DSP-based, LUT-based, and/or BRAM-based processing elements, all accessed as a single pool of processing elements. Each element would perform the multiplication function, but perhaps with different implementations and/or latencies.

Our compute pools not only tackle the problem of supporting processing elements with different implementations and latencies, but also those with different **functionality**. Our approach allows the inclusion of new *fractional* processing elements: elements that only serve a limited range of function inputs.

We describe these new processing elements in more detail in Section 5.5. Such fractional elements—with limited functionality—will be common if they are formed from surplus or unused FPGA resources, or if the resources themselves offer reduced functionality. For example, using a single Xilinx DSP multiplier—which is limited to 18-bit operands—as part of an otherwise 32-bit design. The DSP could then only carry out multiplications of smaller numbers.

Such configurations provide a way for traditionally unused resources, such as memory in compute-bound applications, to be used to improve performance. Additional arbitration and routing logic must be provided to support such a system. This is achieved with our new interconnect, described further in Section 5.4.

5.3.3 A Unified View of Processing Elements

Rather than form new functional units from FPGA resources, we bring together existing implementations of processing elements into centralised compute pools. Given their different implementations and connection schemes, how then do we unify access to heterogeneous processing elements?

A scheme or protocol to handle access to all of these elements must consider that processing elements formed of different resources will naturally operate best under different conditions. Within Xilinx UltraScale+ devices, for example, DSPs achieve over 590MHz when fully pipelined [181], while a BRAM's maximum operating frequency is 450MHz [45]. However, BRAMs produce results in one or two cycles, while DSPs require multiple cycles at higher frequencies. The full functionality of both resource types must be supported.

Coarse-grained architectures typically avoid this issue as their functional units provide a more homogeneous environment and routing structure. This is one of the main benefits of their approach, and can greatly reduce build-time [87].

Due to the heterogeneous nature of the processing elements we support, we must instead look to higher-level solutions. Examples include communication protocols used by cluster/distributed systems, such as MQTT [182], or lower, transport layer schemes like UDP [183]. We take inspiration from these higher-level protocols to create a new communication model for lower, architecture-level processing.

Finally, within the compute pool itself, we must consider the optimal connection scheme when positioning and routing E processing elements in a pool with D incoming/outgoing data-paths. When $E \neq D$, how spread-out or clustered should our processing elements be in relation to the accessing data-paths? What role does element sparsity play—i.e., when $E \ll D$—in determining overall application performance? A high-level performance model must be developed so different pool configurations can be planned and tested, and their performance determined at build-time.

5.3.4 Assessment and Evaluation

Neural network activation functions have been chosen to help evaluate our new architecture, and are described in greater detail in Section 2.6. While this work does not focus on neural networks, their activation functions provide an ideal test-bed for fully utilising all resources on an FPGA.

Activation functions are compatible with our dataflow model of processing, and cover a wide range of different: computational complexities; implementations; and mathematical operations.

Our compute pool model supports high degrees of parallelism and aims to decouple the number of D data-paths from the available E processing elements. Activation functions are again an ideal candidate to evaluate our new architecture as: **(1)** they are called by many neurons in parallel, requiring high degrees of parallelism; and **(2)** the desired parallelism may be so high that parallel data-paths may need to share FPGA resources. High parallelism is not an uncommon requirement with neural networks. Indeed, the "deep" aspect of DNNs refers to the number of layers, with ResNet [29] originally demonstrating up to 1202 layers.

In regards to processing elements within an FPGA, activation functions may be implemented using newer AI cores [176], CPU hard and soft cores [184] [185], or conventional LUTs and DSPs. A single, unified means of utilising all of these resources is an ongoing challenge. In this regard, FPGAs have high processing *potential*, but require additional work to account for different processing elements implementations, latencies, and physical placement within the FPGA. Our higher, function level approach can meet this need.

5.3.5 Summary

Given the above examination, we can now formally identify the challenges we seek to address in this chapter. Create a unified compute pool of E processing elements, each performing the same function, with the compute pool able to serve D parallel function calls such that we:

- **C1**: scale favourably with high degrees of D parallel calls to the compute pool;
- **C2**: support processing elements with different implementations, latencies, and connections, as well as those elements targeting only a limited range of possible inputs;
- **C3**: allow the plug-and-play of mismatched processing elements, easily adding or removing elements for a known performance cost;
- **C4**: are flexible enough to implement functions with different resource use, latency, or throughput priorities.

We address challenge **C1** in Section 5.4, where we describe our interconnect and compute pool model as a means of serving D data-paths with E processing elements. We then begin exploring challenge **C2** in Section 5.5, where we describe and classify different processing elements. This section also outlines the use of new fractional processing elements. These small, resource-efficient elements that target a limited range of function inputs, require additional arbitration that our compute pool model can automatically provide.

Having now presented our new architecture and more formally defined different processing element types, we continue addressing challenge **C2** and progress to challenge **C3** in Section 5.6. This section describes our call-and-response approach to computation, supporting processing elements with non-deterministic latencies, thus facilitating latency-insensitive computation.

Section 5.6 also presents a general-purpose performance model for optimally adding/removing processing elements to/from a compute pool. We earlier describe, in Section 5.4, how each processing element is connected to just one data-path. This newly introduced performance model can now tell us how best to position processing elements within a pool when presented with a mismatched number of processing elements and data-paths, i.e., $D \neq E$.

The behaviour and performance of multiple, successive compute pools is then explored in Section 5.7. This expands on our solution to challenges **C1** and **C2**, and looks at the new opportunities for performance improvements that our architecture opens up.

The final sections evaluate our new architecture, addressing challenge **C4** and proving the efficacy of our overall design by implementing it on the Xilinx Alveo U280.

We look at the resource use and scaling performance of our interconnect and rotating scheduler in Section 5.9; compare our approach to past works in Section 5.10; and examine the latency, throughput, and resource use of functions implemented using our compute pool model in Section 5.11.

5.4 The Compute Pool Model

Incorporating different types of processing element into a design can be difficult if they have different implementations, end-to-end latencies, or connections. Additional development time and FPGA logic must go towards managing access to these different elements. We must also contend with the demands of the user application. For highly parallel applications, the number of available processing elements E may not match the required level of D parallel calls to the compute pool.

5.4.1 Rotating Architecture

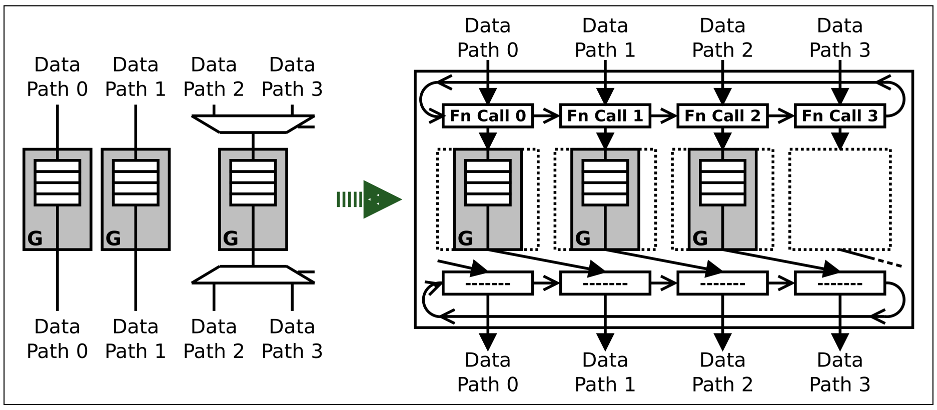
We propose the use of a centralised compute pool to bring together and manage the different, heterogeneous processing elements within an FPGA. These new compute pools make use of the ring-based interconnect we introduce in Chapter 4. The previous chapter has already explored this interconnect, including its: linear scaling; low latency performance; and minimal requirements when arbitrating multiple, parallel accesses to a shared resource.

In the same manner as our interconnect in Chapter 4, data from the D data-paths enter the pool and are stored in one of D registers. In each subsequent cycle, data are moved one position to the right within the input and output rings, connecting them to the next processing element. We now build upon this interconnect architecture to facilitate shared access to the many heterogeneous processing elements within a compute pool.

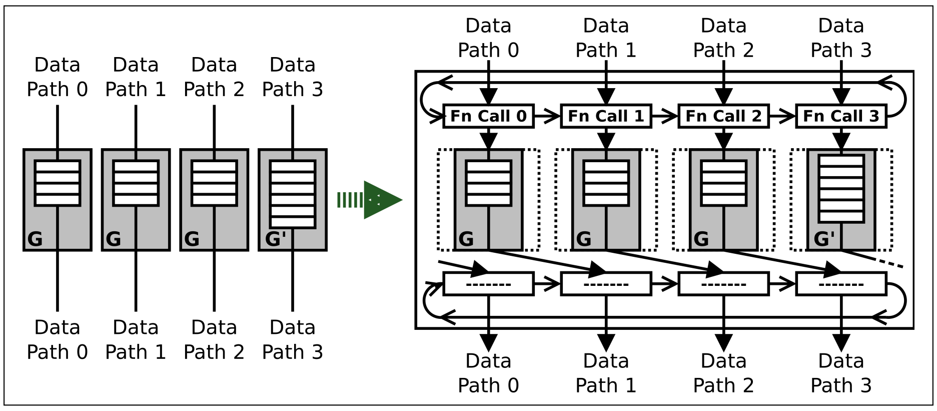
For a simplified example of our new compute pool architecture, suppose we wish to implement the neural network activation function $\boldsymbol{G}$, composed of DSPs. The function will be accessed by D=4 neurons, therefore requiring 4 parallel data-paths and E=4 copies of $\boldsymbol{G}$. However, the number of DSPs in our FPGA limit us to only 3 copies of $\boldsymbol{G}$.

This is typically solved in one of two ways—illustrated by the left-hand side of Figs. 5.1a and 5.1b: **(1)** schedule the 4 parallel data-paths to share the 3 processing elements; or **(2)** create a 4th processing element $\boldsymbol{G'}$ using available resources: LUTs and BRAMs in this case.

For the first case we encounter a latency penalty when sharing our 3 elements between the 4 data-paths; data-path 2 or 3 must wait 1 cycle to be served. In the second case, the pipeline



(a) Sharing three DSP-based processing elements ($\boldsymbol{G}$) across 4 data-paths.



(b) Three DSP-based processing elements ($\boldsymbol{G}$) and one LUT-based element ($\boldsymbol{G'}$) serving 4 data-paths.

Figure 5.1: Serving 4 data-paths with different numbers and types of processing elements.

has become unbalanced as the new LUT/BRAM processing element has more pipeline stages compared to the existing DSP elements.

We propose the creation of a centralised compute pool and novel rotating architecture to automatically schedule D data-paths accessing E elements. Each processing element is connected to a single data-path, but now the data can be rotated through all the data-paths using a repeated circular shift. In the first case in Fig. 5.1a, our rotating architecture acts as a simple scheduler, allowing data-path 3 to be served by data-path 0's processing element after a 1 cycle delay. When the inputs have all been served, the results are restored back to their original data-path.

5.4.2 Minimising Routing While Supporting Non-determinism

For the second case, given in Fig. 5.1b, we solve the problem of unbalanced pipelines through the use of a call-and-response protocol. This places no limitations or expectations on the processing latency of elements, and we describe this in greater detail Section 5.6.1. However, for the *interconnect* itself, the routing of data and connection of elements within the pool still sees challenges from non-deterministic latencies. We solve this using our rotating scheduler.

Our scheduler shifts input data from one data-path to the next until it is served by a processing element. After every input has been served, and the results stored in the output shift register, all data must then be shifted back to their original data-path. For a deterministic processing latency, L, we will know at build-time to simply shift all data back by $L \mod D$ places, requiring a simple 1-to-1 routing scheme for this final "corrective shift".

However, for non-deterministic L latencies, this final shifting distance will vary, and can only be known at run-time. Routing all D data-paths to *each* possible output becomes costly for even moderate degrees of parallelism, but this is not the case for our rotating scheduler.

Our key benefit is that the order of parallel data-paths is maintained as we shift data between registers. Therefore, after N cycles, **all results** must be shifted N positions to find their original path. Final pool results can then be shifted bitwise. For example, with D=16 data-paths, a final shift of 12 (1100_b) places is achieved with $\log_2 16 = 4$ shifts:

$$\begin{aligned} 1 &: 2^0 \times 0 = 0 \text{ positions} \\ 2 &: 2^1 \times 0 = 0 \text{ positions} \\ 3 &: 2^2 \times 1 = 4 \text{ positions} \\ 4 &: 2^3 \times 1 = 8 \text{ positions} \end{aligned}$$

Each D result now has 2 connections at each stage, rather than D. By making use of the variable shifting distance of barrel shifters, we can increase to 4 connections at each stage, and therefore reduce the number of required shifting stages by half. Using 8 connections further reduces the number of stages required to shift all data back to their original data-paths. This number of connections adds minimal logic overhead, and thus does not affect our critical path.

For D=1024 data-paths, using 8-way routing, our architecture can now support non-deterministic processing elements with at most $\lceil \log_8 1024 \rceil = 4$ additional cycles of latency.

5.5 Classifying Processing Elements

Looking again at Fig. 5.1b, rather than create a new element ***G'***, it is actually faster to delay data-path 3 so it can be served by data-path 0's processing element. However, this may not be the case if the 4 processing elements serve more than 4 data-paths, or the elements had non-deterministic latencies, or were unable to be pipelined.

We also cannot assume the inclusion of a processing element will automatically improve performance. Conversely, we cannot assume such a determination will apply to *all* applications and degrees of parallelism. In this section we look to more accurately define and categorise processing elements and so begin to address the challenge of supporting different processing element implementations, identified in Section 5.3.

We begin by classifying processing elements as falling into one of three categories: full processing elements, fractional processing elements, and targeted processing elements.

A **full processing element** is simply the full implementation of a function, whether a simple multiplier or a larger, bespoke function. This element provides an output given one or more inputs, and serves the entire range of possible inputs to its given function.

In contrast to a full processing element, a **fractional processing element** serves only a limited range of all possible inputs. Following on from our previous example in Fig. 5.1, a third **(3)** solution to our problem of insufficient DSP resources is illustrated in Fig. 5.2. Here we use LUTs and BRAMs to form a fractional processing element that replicates some, but not all, of the original processing element's functionality.

The new fractional element ***G"*** uses fewer resources as it only carries out 24-bit operations, compared to the 32-bit operations of the other ***G*** elements. In this example, should the upper 8 bits of data-path 3's input be zero, then we can process it using this new fractional processing element. The obvious limitation being that we are unable to serve the full range of all possible inputs: if the upper 8 bits are not zero, then data-path 3 cannot proceed.

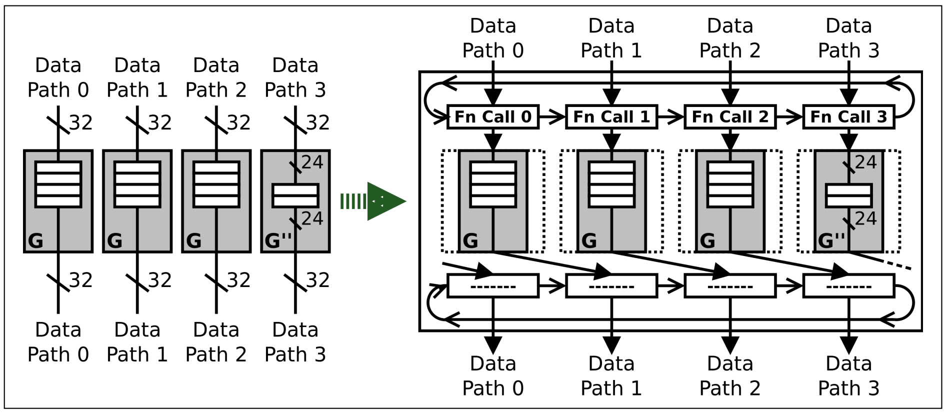


Figure 5.2: Three 32-bit DSP-based processing elements and one 24-bit fractional element serving 4 data-paths.

This uncertainty is the key problem facing fractional processing elements, however, our new architecture is automatically able to address this issue. If the new processing element ***G"*** is unable to serve data-path 3 then, with our rotating scheduler, data-path 3's data must only wait 1 cycle to be served by data-path 0's processing element.

Assuming uniform random data, this example gives a $\frac{1}{256}$th chance for the processing time to be reduced by 1 cycle compared to the traditional approach in Fig. 5.1a and case **(1)** above. While small, this improvement is "free", as the resources were otherwise unable to be used to create a full processing element. This situation is common within applications as resource requirements—LUTs, DSPs, memory, etc—are unlikely to match the exact number and ratio of available resources of a specific FPGA device [186].

We later show in Section 5.11.4 that the benefits of these smaller, fractional elements become more pronounced when the number of parallel, accessing data-paths outnumber the available processing elements.

The final processing element classification are those we term **targeted processing elements**. These are not processing elements in the traditional sense, but instead leverage the strength of individual FPGA resources to improve application performance. For example, unused BRAMs could act as caches, serving function calls based on past results, or LUTs may form simple comparators to serve function calls where the input is invalid—like floating-point infinities or NaNs—or if the input has a simple, known solution.

Examples include compute-in-memory [187], table-based look-ups [53], and piece-wise function curve approximations [188]. These can provide significant performance improvements, but may only be applicable to certain function inputs or data widths. Caches in particular will target and exploit recurring data patterns, and so are unable to serve unexpected function inputs.

In the case of piece-wise function approximations, multiple processing elements are tasked with each approximating one section of a function curve. Here, N processing elements will each only be able to serve $\frac{1}{N}$ inputs. Processing element uncertainty, unbalanced pipelines, and non-deterministic latencies are therefore an inherent part of such a solution. Our approach is designed to make such solutions both resource-efficient and applicable to highly parallel designs.

With fractional processing elements, and targeted elements in particular, we are able to make full use of all resources available on the FPGA

5.6 Heterogeneous Processing Elements

Now that we have a general outline of both the compute pool model and what constitutes a processing element, we now describe how our compute pool architecture: unifies access to processing elements with different implementations, latencies, connections, etc; manages and connects elements within a compute pool; and optimises the parallel accessing of elements within a pool.

5.6.1 Supporting Different Processing Element Implementations

Each processing element in a pool performs the same function, e.g., the *tanh(x)* activation function. Functions should be chosen such that their processing elements are stateless as we cannot guarantee if, and in what order, processing elements will serve incoming data. To operate within our new compute pool architecture, stateful functions must then either be modified, or broken down into more fine-grained, stateless operations.

If processing elements are formed from different resources, then elements with different latencies will be the norm, so we must support different or unknown processing latencies. For this we adopt a call-and-response approach to computation.

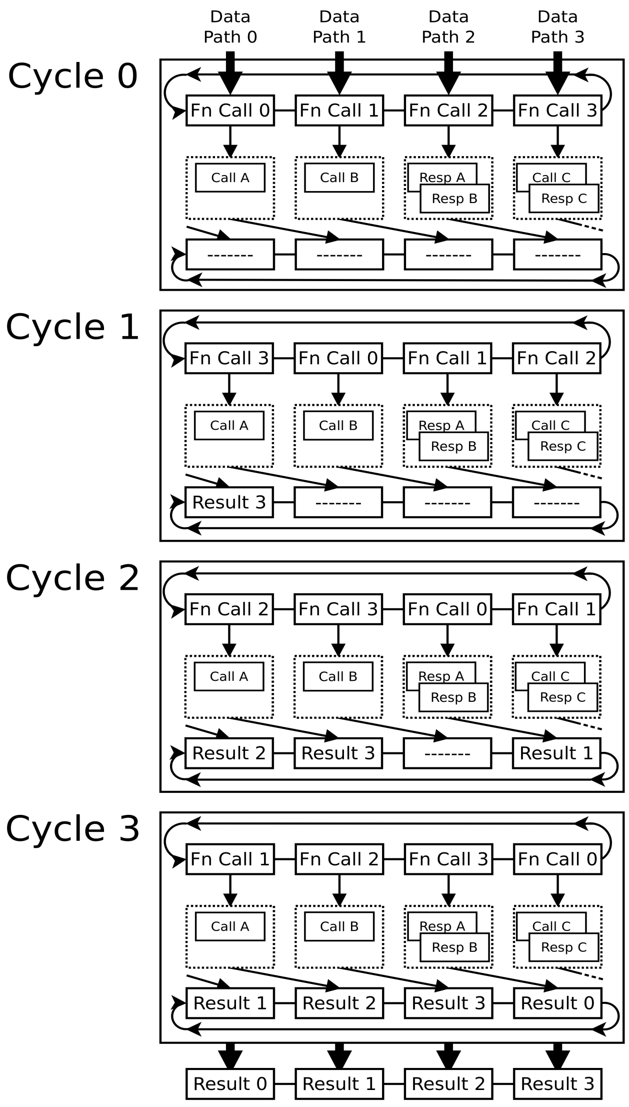
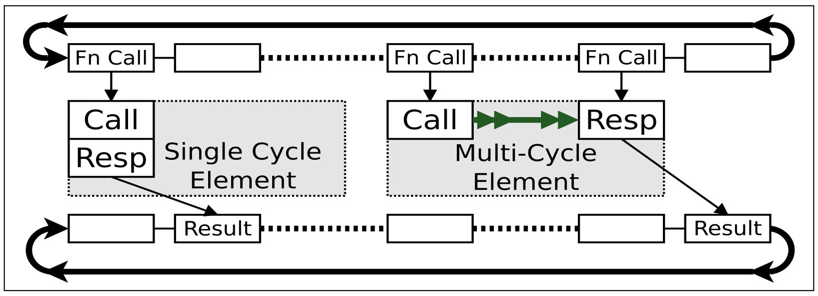


Figure 5.3: A call-and-response compute pool with 3 elements, serving 4 parallel data-paths.

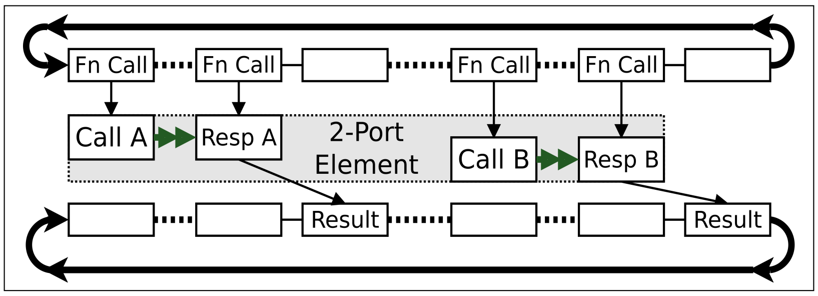
Each processing element is represented in a pool by two units: a call unit and a response unit. A function call is made to the call unit in one cycle, and the result provided by the response unit L cycles later.

Fig. 5.3 shows an example of a pool serving 4 parallel function calls to 3 processing elements, A, B, and C. We will look at only full processing elements, with each element able to serve any function call. The end-to-end latency for element A is L=3 cycles, L=2 for element B, and L=1 for element C.

For simplicity, just the connections to each element's call and response units are shown: *Call A* and *Resp A* for element A, and so on. Since multiple call and response units can be connected to a single data-path, the dotted boxes show how units are grouped together at each position.



(a) Processing elements that have single or multi-cycle latencies.



(b) A 2-port processing element, with a multi-cycle latency.

Figure 5.4: Connecting processing elements within a pool.

The call unit accepts function input data, and the processing element begins processing it as normal. In Fig. 5.3, element A serves *Fn Call 0* in the first cycle, but the result is not written to the output until it passes A's response unit in cycle 3. Element C on the other hand serves *Fn Call 3* in the first cycle and the result is written on the next cycle.

We handle the different latencies of processing elements by the spacing of their call and response units within the pool. Multiple elements can serve a single data-path's function call, as shown by the response units for elements A and B in Fig. 5.3. We therefore have no issue with conflicting latencies causing elements to overlap. Elements with non-deterministic processing times must then only cache their results and wait for the requesting function call to be rotated past their response unit.

5.6.2 Supporting Different Processing Element Connections

Fig. 5.4 shows how different types of processing element can be connected into a common pool. Some processing elements, such as embedded CPUs or multi-port memories, may accept more than one data-path in parallel. All available inputs to a processing element can therefore be independently connected within a pool.

Elements can have any number of call and response connections into a pool, as shown in Fig. 5.4b. These units are best connected as far apart as possible within the pool. If placed side-by-side, it is possible for the left-most call unit to constantly starve the right-most unit. We discuss the optimal placement of elements in Section 5.6.3.

Call-and-response uses a standard valid/ready handshake, in which elements inform the pool if they can serve a given input. This is true for call and response units, but call units do not produce results. If a response unit indicates it can serve the incoming data, it must produce a result on the next cycle.

The position of call and response units in the pool is defined at build-time by the designer. A pool serving D data-paths will have D possible connection positions, with any number of call or response units able to be connected to each pool position. For example, in the extreme case, a pool with D=1 data-path can have all E processing elements connected to its data-path.

5.6.3 Latency-centric Model of Processing Elements

Like DSPs, LUTs, embedded cores, or any other FPGA component, *routing* is also a finite resource. Designs will fail if there are insufficient paths to route data, or may see a reduction in the operating frequency, slowing the entire application. If our goal is to minimise routing, then a mismatched number of D data-paths and E processing elements should be considered the standard case.

To deal with this asymmetry we introduce a general-purpose performance model for optimally adding or removing elements from a compute pool. Within a pool containing E processing elements, each with their own latency L_e, we see that the expected latency $\bar{L}_e$ is simply:

$$\bar{L}_e = \frac{1}{E}\sum_{n=1}^{E} L_{e_n} \tag{5.1}$$

Processing elements are connected to data-paths. In a pool with D data-paths there are D positions for a processing element to be connected. If pools have fewer processing elements than incoming function calls, i.e., $E < D$, then some data-paths will encounter a "missing" processing element. If this occurs, the incoming data on this data-path must wait L_{wait} cycles for our architecture to rotate it past an actual processing element. This waiting latency is an important consideration.

When data enters a pool, the best case waiting latency occurs when it immediately encounters a processing element, i.e, $L_{wait} = 0$ cycles. In a pool with E evenly spaced processing elements, the worst case waiting latency occurs when the data must rotate for $L_{wait} = \frac{D}{E} - 1$ cycles within the compute pool until it encounters an element.

As D parallel data-paths enter the pool at once, each data-path will experience a different waiting latency. With the E processing elements evenly spaced within a pool, the D data-paths are now essentially formed into E queues. If a data-path in this queue must wait between 0 and $x = \frac{D}{E} - 1$ cycles, then the average waiting latency for a data-path is found to be:

$$\begin{aligned}
\bar{L}_{wait} &= \frac{1}{x}\sum_{n=0}^{x} n \\
&= \frac{1}{x} \times \frac{x(x+1)}{2} \\
&= \frac{x+1}{2} \\
&= \frac{(\frac{D}{E} - 1) + 1}{2} \qquad \text{(substituting } x = \frac{D}{E} - 1) \\
&= \frac{D}{2E}
\end{aligned} \tag{5.2}$$

The latency required for a data-path to be served within a pool, L_{fn}, i.e., a function call, is found by considering these two forms of latency: the latency of the processing elements, L_e, and the latency to encounter an element within the pool, L_{wait}. Taken together we find the latency of a function call to be:

$$L_{fn} = \begin{cases} L_e & \text{if element present,} \\ \bar{L}_e + \bar{L}_{wait} & \text{if no element present} \end{cases} \tag{5.3}$$

Removing a processing element from the pool can then be viewed as simply replacing its latency with the pool's average latency, $\bar{L}_e$, and the average number of cycles to rotate data to another element, $\bar{L}_{wait}$. The placement of processing elements within a pool is the primary means of reducing this latter factor.

Within a compute pool with D parallel data-paths, there is no global starting position for input data. Data on data-path 0 is first stored in position 0, whilst data-path 1's data are stored in position 1, and so on. Rotation direction determines the order that processing elements are likely to be accessed.

We rotate data from left to right, so must therefore place elements in ascending order of latency from left to right. We evenly space "missing" elements, i.e., gaps in the compute pool, so as to minimise the average distance data must travel to encounter a processing element.

Following this, when adding new processing elements to a pool, we can now determine that new elements must have a latency of no more than $\bar{L}_e + \bar{L}_{wait}$ cycles to improve pool performance. If the latency of an *existing* element is greater than this, then **its removal would improve average latency**.

The example in Fig. 5.3 does not follow these recommendations. A closer look shows processing element A is redundant, as element B could also have served *Fn Call 0* in cycle 3. With just elements B and C, the maximum allowed latency is $\bar{L}_e + \frac{D}{2E} = (\frac{1+2}{2}) + \frac{4}{2*2} = 2.5$ cycles, to element A's 3 cycles.

5.7 Multi-pool Routing and Performance

Having defined compute pools in Section 5.4, processing elements in Section 5.5, and the means by which we connect elements within pools in Section 5.6, we now look at routing schemes and dataflow between compute pools. Our ability to arbitrate D data-paths being served by $E < D$ processing elements, along with our support for non-deterministic processing latencies, provides opportunities to improve application performance via our architectural choices.

5.7.1 Inter-pool Routing

In order to support processing elements with different implementations and formed of different resources, we deal with processing elements based solely on their latency. When considering the compute pool as a whole, Eqn. 5.3 sees the latency of a function call as arising from both the latency of the E processing elements **and** the time required to route data to the next element within the pool. This second, newly introduced factor provides an additional way to improve performance: by reducing this “waiting latency”.

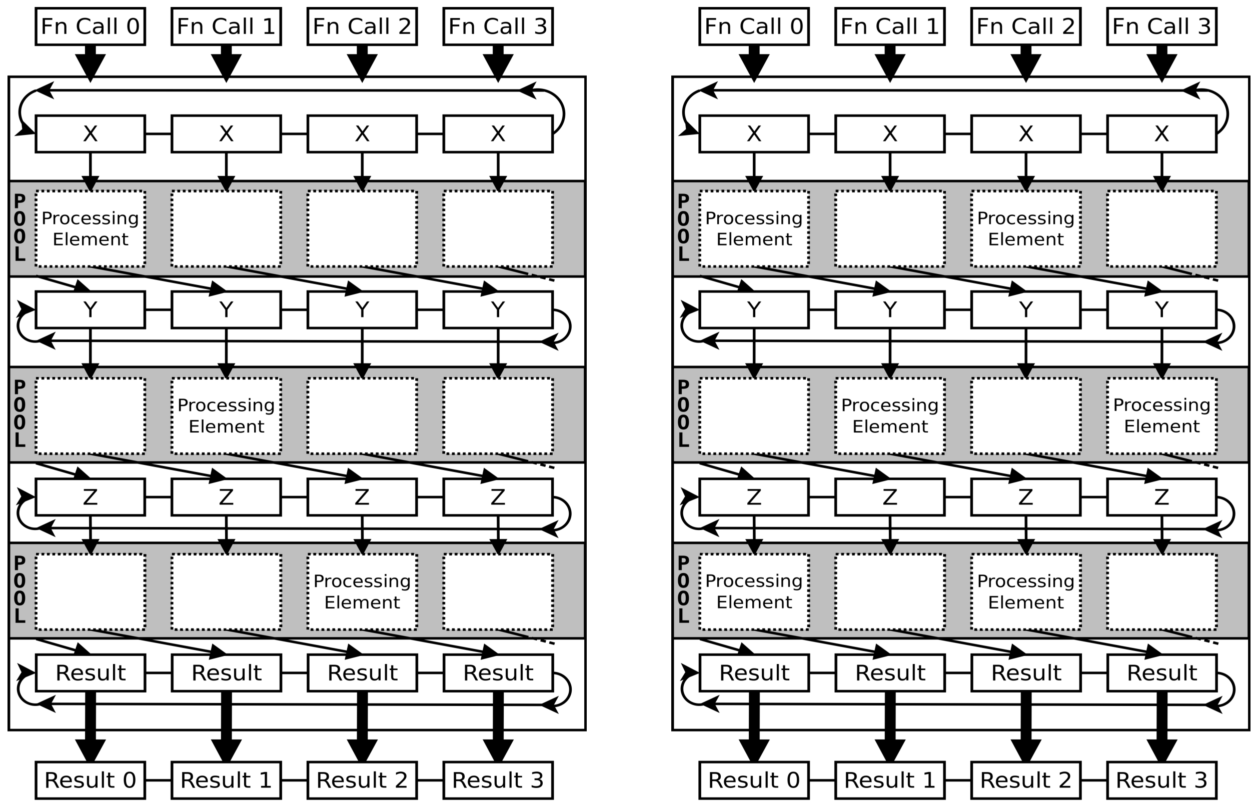
When an application uses multiple, successive compute pools, one method of improving performance is to allow the D data-paths between successive compute pools to become out-of-sync. Rather than each successive pool waiting for all D data-paths to be served before advancing the pipeline, results from pool P_n immediately feed through to pool P_{n+1}.

We are then effectively combining successive rotating schedulers, and so also combining the average post-processing wait time for pool P_n with the average pre-processing wait time for pool P_{n+1}. After passing through N pools, the data-paths will meet a barrier point and be synchronised again. For P pools, and an average latency of all pool elements $\bar{L}_{P_e}$, the average latency for *each* pool to serve all of their D data-paths is then:

$$\bar{L}_P = \bar{L}_{P_e} + \frac{D-E}{EP} \tag{5.4}$$

Eqn. 5.4 assumes the same number of D data-paths and E processing elements across the successive pools. Should this not be the case, then the latter factor, $\frac{D-E}{EP}$, should instead be calculated on a per-pool basis, rather than using global values for D and E.

If all processing elements have known latencies then the layout of each pool can be optimised at build-time by cascading the result from pool P_n directly into a processing element in pool P_{n+1}. Fig. 5.5 shows examples of two cascading configurations, each with three compute pools. They contain one and two processing elements respectively, and for simplicity have a latency of $L_e = 1$ cycles.



(a) One processing element per pool.

(b) Two processing elements per pool.

Figure 5.5: Cascading routing between three function pools.

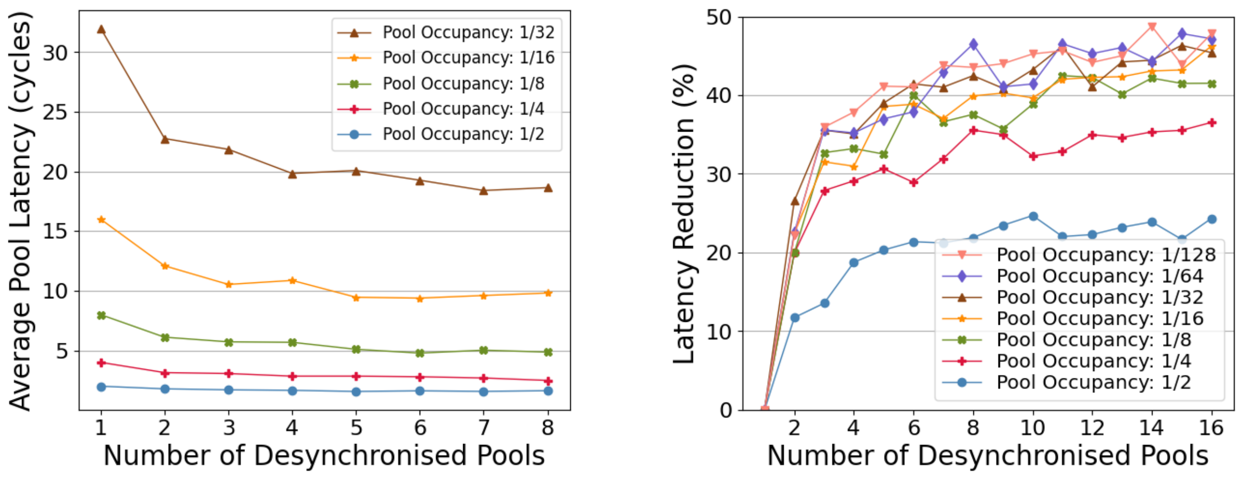
With a latency—and initiation interval—equal to 1 cycle, we would normally expect a pool with E=1 processing element to take 4 cycles to serve the D=4 data-paths: 1 cycle for processing, 3 cycles for waiting pre/post -processing. However, in Fig. 5.5a we find that with our cascading routing it takes each pool $\bar{L}_P = 1 + \frac{4-1}{1\times 3} = 2$ cycles to serve the 4 data-paths.

This improvement is achieved **solely** through architectural choices. Without affecting data or processing elements, we reduce the total latency across the three pools from $3 \times 4 = 12$ cycles to $3 \times 2 = 6$ cycles.

5.7.2 Non-deterministic Latencies

The above scheme of chaining together the output of a processing element in pool P_n with the input of an element in pool P_{n+1} requires knowledge of P_n's processing latency. A more general configuration is required to support non-deterministic latencies.

Unknown latencies introduce a degree of randomness into our overall processing time. This



(a) Serving 32 data-paths.

(b) Normalised.

Figure 5.6: Latency improvements when de-synchronising multiple compute pools.

arises from two sources: **(1)** the compute pool containing fewer E elements than there are D data-paths, i.e., a data-path may have to wait to access an element; and **(2)** processing elements, like fractional elements, may not serve the given function input. A cycle-accurate simulator was developed in Python to examine these effects, and we explore them here.

Probability 1: Fewer Elements than Data-paths

When D data-paths share access to $E < D$ processing elements, each D data-path faces a random chance of being the last path served by the pool. However, it is unlikely the *same* path will be delayed in subsequent compute pools. This follows the central limit theorem [189], where multiple, independent random events tend towards a Normal distribution. More extreme events, such as the same data-path facing successive worst-case delays, become increasingly unlikely.

To reduce the risk of a data-path facing multiple worst-case delays as it passes through successive compute pools, we must simply **de-synchronise more compute pools**. We will therefore gradually tend towards the optimised, cascading design, improving the average performance of all compute pools.

Fig. 5.6a shows the results of de-synchronising multiple pools, each with D=32 data-paths and a function latency $L_{fn} = 1$. With 32 data-paths a pool occupancy of $\frac{1}{8}$ means there are E=4 processing elements per pool, while an occupancy of $\frac{1}{16}$ translates to E=2 processing elements, and so on. When tested with uniform random input data, Fig. 5.6a shows the average pool latency tends towards ~4.5 cycles, rather than the expected $\frac{32}{4} = 8$ cycles. Like the optimised cascading approach, this latency reduction comes from effectively combining the post and pre processing wait times of successive compute pools.

Fig. 5.6b shows a normalised comparison for D=128 data-paths with different pool occupancies. Here, a pool with $\frac{1}{2}$ occupancy has E=64 processing elements, while $\frac{1}{4}$ has E=32 elements, and so on. For pools with lower occupancy rates, i.e., fewer elements, de-synchronisation has a much greater effect. In the extreme case of E=1 element serving all D=128 data-paths, the average pool latency improves 22% by de-synchronising just 2 successive pools, and climbs to a 41% improvement with 5 pools.

Probability 2: Elements Able to Serve a Function Call

The second source of randomness we identified—that a processing element in the pool may be unable to serve a given function input—comes from our support for fractional and targeted processing elements. The inclusion of such elements can either provide additional processing performance by making use of otherwise unused FPGA resources, or may in fact be required as part of the function solution. For example, a design with multiple processing elements, each of which are optimised to serve a set range of function inputs.

The distribution of these random accesses depends on: **(a)** the likelihood of a function input being seen; and **(b)** the relative availability of different fractional processing elements within the pool, i.e., how many processing elements are able to serve a given function input. Despite this randomness being entirely application-dependent, our compute pools are still able to offer performance and resource benefits.

With our rotating scheduler, individual processing elements do not have the sole responsibility for serving the entire input space of their connected data-path. Function calls from all parallel data-paths are rotated past each element in a compute pool, meaning elements can specialise in serving specific aspects of a function call without fear of starving the non-supported inputs.

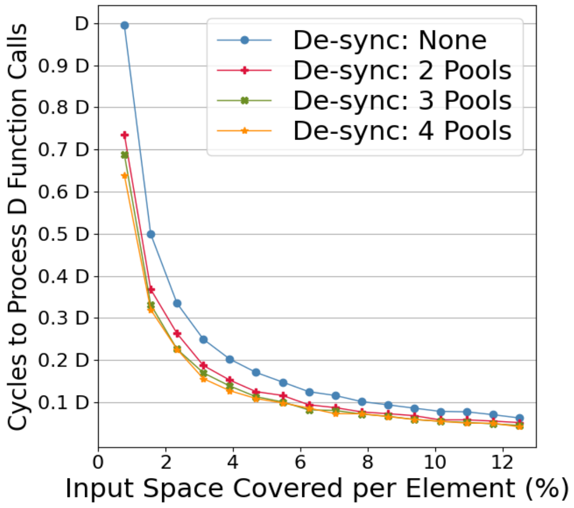


Figure 5.7: Relative latency reduction when increasing the input space each element can serve.

This processing element specialisation is a standard approach taken by some function approximation methods. Here, functions are broken down into smaller ranges or sub-functions that collectively serve the whole input space, such as representing small sections of a curve with linear approximations. This solution-by-parts implementation can provide better performance and use fewer resources than a full function implementation, and so is ideally placed to make use of fractional compute elements and our compute pool architecture.

To model such configurations, Fig. 5.7 shows the normalised case where D data-paths feed into a single compute pool of D=E processing elements. In this configuration, each element starts out serving a different $\frac{1}{D}$ fraction of the possible input space. In a pool with D=32 data-paths each element would serve $\frac{1}{32}$th of the input space, while for a pool with D=128 data-paths each element would serve $\frac{1}{128}$th, and so on.

Tested using uniform random input data, the time taken to process the D data-paths falls as we increase the percentage of the input space each processing element can serve. This is achieved by either **(a)** the processing elements serving a greater range of inputs, or by **(b)** adding more fractional elements to the compute pool. As with Fig. 5.6, we see in Fig. 5.7 that de-synchronising just 2 pools has a noticeable effect on the processing latency.

Rather than E=1 full processing element with 100% input coverage serving D=128 data-paths, we now have E=128 smaller, fractional elements. While there is a small chance any single element will serve a specific data-path, we are now performing 128 checks at once. We may now **serve multiple paths in parallel**, therefore reducing the average latency. Fig. 5.7 shows increasing the input space coverage by only ~1% per element can significantly reduce latency.

As these improvements are averaged across one or more compute pools, it is possible to see some worst-case spikes in latency, creating problems for latency-sensitive applications. With our approach, strategically spacing one or more full-function processing elements in a pool thus **guarantees** a known upper-bound latency for such applications, ensuring correct performance.

5.8 Implementation

We implement and evaluate our architecture using the Xilinx Alveo U280 accelerator card. As described in Section 5.3.4, we evaluate our approach using a range of neural network activation functions. A compute pool design is created for each activation function. Each pool is accessed by D=1024 parallel neurons, using 32-bit data-widths. Development and testing are done with the Vitis and Vivado 2019.2 design tools.

Pipelined, floating-point processing elements are constructed using FloPoCo [174]. While these processing elements are not optimal, they provide a fair baseline from which to measure the efficacy of our architecture, rather than the function implementations themselves.

Our evaluation is broken down into 3 sections. Section 5.9 evaluates how our interconnect and rotating scheduler scale with different D data-paths and E processing elements. Section 5.10 compares our architecture to current and past works, examining it as a general-purpose means of unifying heterogeneous processing elements and its ability to maximise resource utilisation within an FPGA. Finally, Section 5.11 examines the performance of applications implemented using our compute pool model, specifically throughput, latency, and the effect fractional processing elements have on performance and overall design choices.

Development and Toolchains

As we discuss in Section 2.3.3, new coarse-grained architectures, such as overlay frameworks and virtual FPGAs, require entirely new development tools and processes. Many, like ZUMA [81] provide open source compilers, but additional analysis, modelling, and development tools for new frameworks are rarely seen. Wijtvliet [80] highlights this in their review of coarse-grained architectures over the past 25 years. They specifically note the lack of new design space exploration tools and tool-flows for designers to make use of the new architectures that researchers introduce.

Our compute pools unify *existing* elements, without the need for a new underlying, more coarse-grained framework. Our new architecture therefore does not require the creation of new development platforms, tools, or toolchains. We support all existing methods for the creation, analysis, and design space exploration of processing elements.

While our implementation uses FloPoCo to create processing elements, we equally support existing soft IP cores, embedded processing cores, or memory-based processing. Indeed, a virtual FPGA itself would be a valid processing element within our compute pools.

5.9 Architecture Evaluation

In this section we assess how well our architecture and interconnection scheme **(A)** scales with increasing D data-paths and E processing elements, and **(B)** balances different degrees of parallelism for the D data-paths and E processing elements.

5.9.1 Resource Scaling and Arbitration

The resource use and operating frequencies of our architecture, without processing elements, is given in Table 5.1. Regression analysis of our resource use is reported in Fig. 5.8. From these results we see resources scale linearly with the number of successive P compute pools and D data-paths.

Table 5.1: Compute pool framework resource use and operating frequencies on the Alveo U280.

		Data-paths (32-bit)						
	Pools	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
LUTs	1	2,146	3,314	5,003	7,427	14,139	27,316	49,592
	2	2,817	4,542	7,534	12,564	23,820	47,879	88,721
	3	3,478	5,837	10,042	17,529	33,695	67,745	126,510
	4	4,125	7,105	12,560	22,515	43,681	87,686	167,563
Regs	1	4,339	5,978	8,292	12,909	22,034	42,568	81,430
	2	5,542	8,294	12,966	22,402	41,516	79,585	157,225
	3	6,749	10,753	17,794	31,798	60,605	117,298	232,524
	4	7,968	13,158	22,559	41,413	79,590	156,197	308,318
Freq	1	412	350	350	295	290	256	227
(MHz)	2+	350	300	275	280	255	214	201

$$LUTs = D(38.5P + 8.37) + 2040 \qquad \text{P-value} < 0.001$$
$$REGs = D(73.9P + 2.24) + 3280 \qquad \text{P-value} < 0.001$$

Figure 5.8: Regression analysis for compute pool framework resource use on the Alveo U280.

For example, when moving from pools balancing and arbitrating D=16 data-paths to D=1024 data-paths, we achieve a 64x increase in parallelism at a cost of only 23.8x the resources. This D=1024 data-path framework uses just 3.8% and 3.12% of available LUTs and registers on the Alveo U280, respectively.

We now populate our compute pools with processing elements; one compute pool implementation for each of the 8 activation functions. Fig. 5.9 shows the resource use for each activation function when D=1024 parallel data-paths access a pool comprised of E=1024 down to E=8 processing elements. For reference, the horizontal, dotted lines show the available resources within the Alveo U50, U200, U250, and U280 accelerator cards.

If using a simplified dataflow approach we would require 1 processing element per neuron, D=E. Any other approach requires some form of additional arbitration overhead. For all but the smaller LReLU and ELU functions, the D=E=1024 configurations use more LUTs than are available on the Alveo U280. The standard, non-arbitrated dataflow approach is therefore unable to support 1024 neurons if each requires its own independent pipeline. For such applications, a management and arbitration architecture is required.

Our rotating scheduler is modelled on a uni-directional ring topology, with data passed between data-paths using shift registers (Fig. 5.3). If, instead, we had chosen point-to-point or crossbar

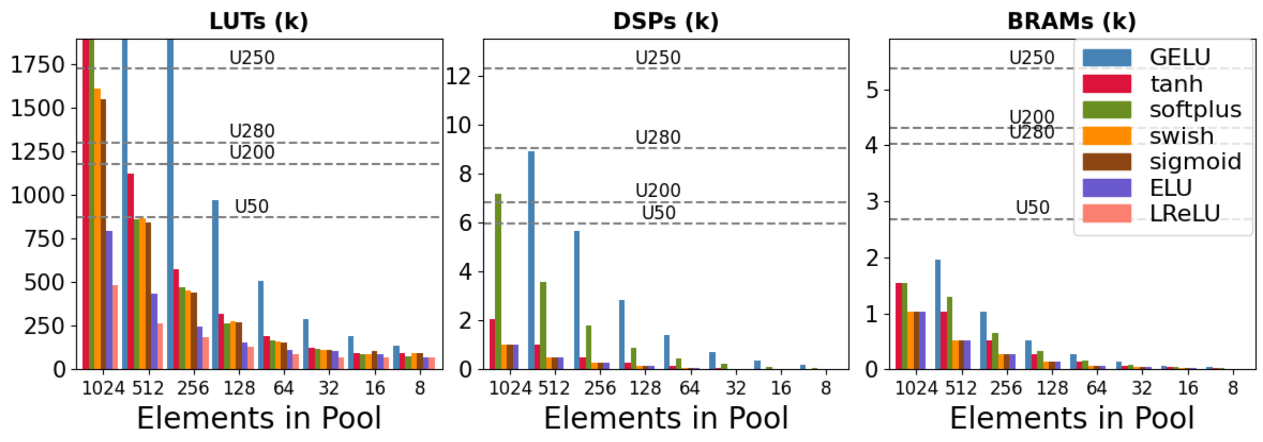


Figure 5.9: Activation function resource use on the Alveo U280.

topologies, the number of required connections and switches would scale with $O(N^2)$. This would create a large number of switching positions, i.e., points of possible collision, requiring arbitration. This makes such topologies unsuitable for high degrees of either D data-path or E processing element parallelism.

Another option, multi-stage topologies, like Clos [157], Benes [158], and Omega [159] scale more favourably at $O(NlogN)$ and require much less arbitration logic. However, this is less than our chosen ring-topology's $O(N)$ scaling, and greater than our required arbitration logic. Additionally, multi-stage topologies traditionally connect $N \times N$ networks, so an asymmetric $D \times E$ interconnect leads to additional challenges when configuring and balancing links.

5.9.2 Higher Data-path Parallelism with Fewer Elements

With our compute pool approach we decouple the number of E processing elements from the number of D data-paths they must serve. It is therefore a simple matter of supporting arbitrary degrees of parallelism using the processing elements available within our target FPGA.

For example, by reducing to E=512 processing elements, Fig. 5.9 shows all but GELU and tanh will now fit on the Alveo U280. The D=1024 data-paths will automatically be balanced across the remaining elements. Section 5.6.3's performance model lets us predetermine the 1 cycle latency cost this incurs.

While maintaining our D=1024 data-paths, we can further reduce our number of processing elements to E=256. This drastically lowers the resource footprint, putting all functions but GELU within the capabilities of the smaller and cheaper Alveo U50 card. Following this method, a neural network could be trained using a large number of processing elements, then deployed on a smaller FPGA by reducing the number of processing elements.

5.10 Framework Performance Evaluation

Providing a general comparison to past overlays, coarse-grained architectures, and virtual FPGA can be difficult. This is due to both the different designs and design goals of these past works, as well as our ability to normalise the reported results to provide an equal comparison. We therefore look to compare our work to past frameworks according to three criteria: **(A)** resource utilisation, **(B)** throughput, and **(C)** processing element extensibility and support.

5.10.1 Utilisation

Overlays and virtual FPGAs provide new functional units composed of one or more FPGA resources. This more coarse-grained approach suffers wasted resources when the applications that are built atop these new frameworks do not use 100% of the resources within the new functional units. These unused resources—DSPs, LUTs, CPU cores, etc—are locked away in the functional units.

FPCA [84] reports 61% utilisation within their design. The overlay in [83] reports 22% utilisation, but this approach prioritises faster run-time reconfiguration times, for which it achieves a 1000x speed-up over conventional methods. DeCO [82] reports 40-90% utilisation, depending on the benchmark.

With a focus on maximising the use of FPGA resources, the framework for our compute pool architecture is constructed with just LUTs and registers, as reported in Table 5.1. This leaves the remaining FPGA resources available to be formed into processing elements for use within our new compute pool structure.

As we standardise processing elements based on their end-to-end processing latency, not their implementation, we therefore do not need to define our own functional units. Based on our reported architectural overhead, we find an effective utilisation of 96.2% for D=1024 data-paths, rising to 99.8% if instead choosing fewer, D=16 data-paths.

5.10.2 Overlay Throughput

Our compute pool's unified support of heterogeneous processing elements lets us make use of all the processing resources available on the FPGA. Combined with our rotating scheduler's efficient $O(N)$ scaling to high degrees of parallelism, we are able to achieve higher performance than past works.

Table 5.2 provides a normalised comparison to recent overlay frameworks and virtual FPGAs. We measure maximum framework throughput (independent of any implemented application) in billion operations per second (GOP/s). Some works report their throughput [82] [83] [180] [190], while for others we determine OP/s by the number of n-bit operations their functional units perform, then multiply this by the total size of the overlay.

Throughput is normalised based on LUTs or ALMs, rather than per unit area, as an area comparison may prove misleading. Past works define their functional units as collections of LUTs, DSPs, and/or BRAMs, whilst our "functional units" are function implementations. Different transistor technologies should be taken into account, however, a throughput per area metric would overly benefit our approach for simple functions—such as those requiring few LUTs—and those functions using specialised cores—such as Xilinx's new 7 nm Versal AI cores [50], which report throughputs of 3-17 TFLOP/s [193]. Our comparison therefore focuses on the throughput of the frameworks themselves.

Our 32-bit floating-point implementations see normalised throughputs matching and surpassing those of past 16-bit frameworks. When comparing to 32-bit frameworks we find a 10.9x-

Table 5.2: Throughput comparison to past overlays and frameworks.

Work	Data Type	Freq (MHz)	LUTs\ ALMs	DSPs	Throughput G(FL)OP/s Raw	Throughput G(FL)OP/s per k-LUTs
16-bit						
Zynq-7000 [180]	Int	338	28,000	128	64.90	2.32
Zynq-7000 [83]	Int	300	37,000	128	115.20	3.11
Virtex-7 [83]	Int	380	228,000	800	912.00	4.00
Zynq-7000 [82]	Int	395	6,156	90	28.44	4.62
Alveo U280[1]	Float	300	82,944	0	614.40	7.41
32-bit						
Stratix IV [191]	Int	355	162,376	204	60.40	0.37
simulator [190]	Int	757	-	-	177.20	-
Stratix IV [191]	Float	312	161,415	264	32.40	0.20
Stratix V [192]	Float	223	51,000	N/A	22.98	0.45
This Work (Alveo U280)						
1 Pool	Float	227	49,592	0	232.45	4.69
2 Pools	Float	201	88,721	0	411.65	4.64
3 Pools	Float	201	126,510	0	617.47	4.88
4 Pools	Float	201	167,563	0	823.30	4.91

24.4x increase in normalised throughput. For this comparison we limit our framework to just 4 compute pools, as this uses ~12% of available resources on the Alveo U280, leaving the remaining 88% for processing elements. When implementing simpler functions, fewer FPGA resources would be needed, and thus higher throughputs achieved by using more than 4 compute pools.

The 16-bit Alveo U280 framework we reference is our own past work, described in Chapter 4. In contrast to the work described in this chapter, the earlier version of our interconnect was limited to connecting BRAMs, and offered no support for non-deterministic latencies. Our current work sees an increase in raw throughput, but a drop in normalised throughput. This is due to the additional resource overhead of compute pools now supporting higher data widths, heterogeneous processing elements, and non-deterministic latencies.

Podobas [194] surveyed a wide range of 27 coarse-grained FPGA architectures. They report the highest framework throughputs as 100 GOP/s for 32-bit integer operations and 50 GFLOP/s for floating-point. Exact figures and methodologies for these results were not provided, so could not be individually included here.

[1]Our past work, described in Chapter 4.

5.10.3 Processing Elements

One of the recommendations from Wijtvliet's [80] survey of 35 coarse-grained FPGA architectures is that frameworks should aim to unify resource use and processing elements at a higher, function level. The functional units of past works can be rigid in terms of their supported resources and interconnections. Instead of forming functional units from LUTs, DSPs, BRAMs, etc, *functions* serve as the functional units within our compute pool architecture. This provides a more generalised, higher-level means of both defining and working with all types of FPGA resource.

In our handling of heterogeneous processing elements, our approach looks to mimic the collection of function-specific execution units within CPUs. Such architectures also allow one or more parallel data-paths to access and share a common pool of processing elements. At still higher levels, we see similarities to the workload sharing models of thread pools. We aim to combine these higher-level, abstract structures with the more customised, application-specific approach of dataflow computing.

When defining their processing elements, past works also see difficulty supporting data widths above 16 bits as DSP blocks typically form the processing backbone of functional units. Xilinx [47] and Intel [126] DSPs are limited to 18x27 bits so expanding beyond this incurs high utilisation and performance costs, as shown in Table 5.2. Our work does not face these limitations

5.11 Application Evaluation

We now look at how our architectural approach affects application performance. If we can easily add or remove processing elements, what performance improvement would this provide? We might expect that adding more elements would lead to improved throughput and latency, but this may not be the case with FPGAs. Placing and routing these additional elements can lower the maximum operating frequency, thereby slowing the entire design.

Our ability to add or remove processing elements from the pool then provides an additional resource to trade-off: data routing. Fewer processing elements mean fewer connections within the FPGA, resulting in less congested designs and faster build-times. When considering routing as a resource, **we maximise the routing performance by minimising its use**.

5.11.1 Functions and Methodology

As previously mentioned, we use neural network activation functions as a test-bed for our design, however, this work does not focus on neural networks or activation functions themselves. We instead look to fully utilise all available resources within an FPGA. Our new compute pools and rotating scheduler introduce additional resource overhead. This must be offset by gains in overall performance, therefore, not all functions are suitable for this architecture.

As we manage processing elements at a higher, function level our evaluation should therefore look to determine at what level a function is too simple for our approach. For this, we follow a generalised roofline model for FPGAs [170] to quantify functions based on their computational intensity. The more operations a function performs per datum, the greater its computational intensity. This metric is highly implementation-dependent, and is the main reason we have implemented our test functions using the general-purpose FloPoCo framework, thus ensuring a fair comparison.

Throughput (Section 5.11.2) and latency (Section 5.11.3) are evaluated by varying the number of E processing elements in a compute pool serving D=1024 data-paths. Table 5.3 shows the computational intensities for each of the activation functions we test. The functions cover a range of intensities, letting us not only evaluate throughput and latency, but also how complex or resource-intensive a function must be to benefit from our approach.

Table 5.3: Computational intensity of functions (ops/byte).

ReLU	**LReLU**	**ELU**	**softplus**	**sigmoid**	**swish**	**tanh**	**GELU**
0.25	1.00	2.75	5.50	6.75	6.75	6.75	10.50

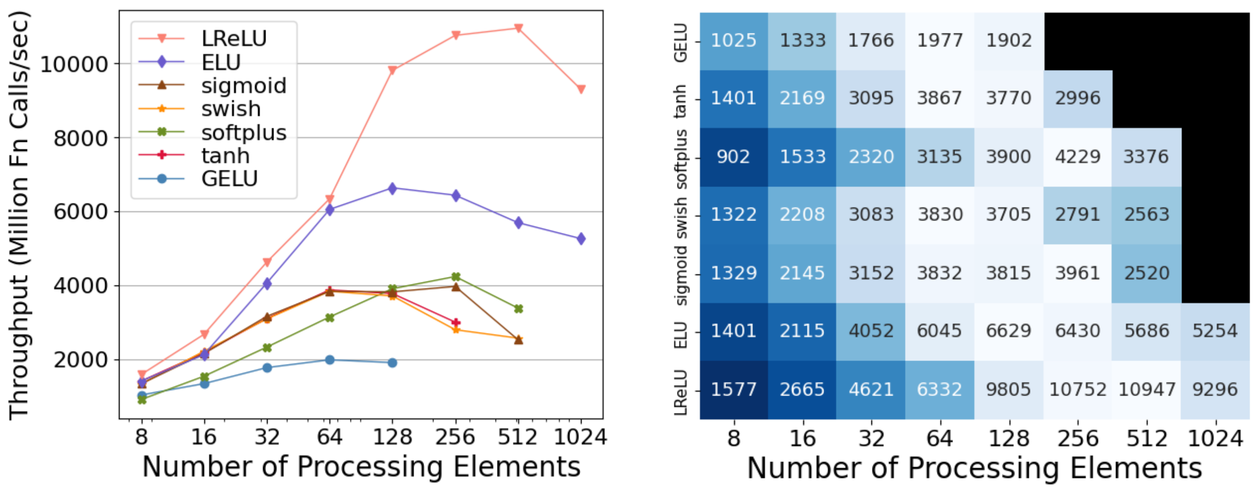


Figure 5.10: Throughput for pools with 8-1024 processing elements.

We find that basic functions, like ReLU, are too computationally simple to see many benefits with our architecture. Formed of just a 1-bit comparator, ReLU is a widely used activation function, although not suitable for all applications or types of neural network. We instead focus on the remaining 7 activation functions to evaluate the latency and throughput performance of our architecture.

5.11.2 Throughput

Fig. 5.10 shows the throughputs achieved for pools comprised of E=8 to E=1024 processing elements. The associated heat map gives exact throughputs for each function when E elements serve the D=1024 data-paths. The higher the throughput, the lighter the colour, with black squares being designs that failed to build due to resource constraints. Results are ordered by the function's computational intensity, with the lowest, LReLU, at the bottom, and largest, GELU, at the top.

The simplified, non-arbitrated dataflow approach sees one processing element per data-path, i.e., D=E=1024. However, for most functions we find maximum throughput occurs when the D=1024 data-paths are served by about E=64-128 processing elements, with improvements averaging 21% compared to the traditional dataflow case.

With just E=64-128 elements we are far from using all available FPGA resources. What then facilitates this increased performance? Eqn. 5.3 offers some insight. Removing an element from

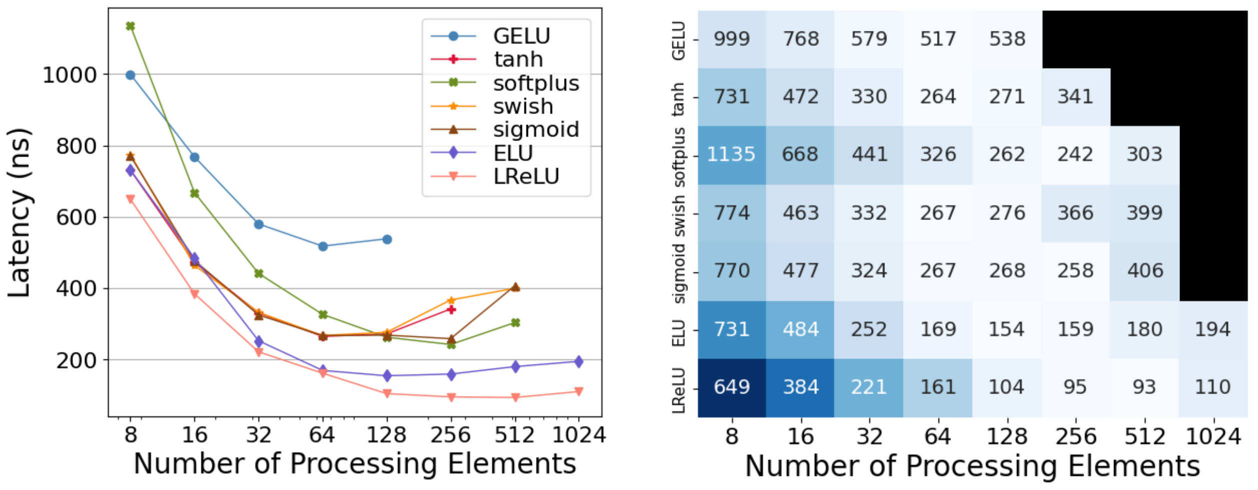


Figure 5.11: Latency for pools with 8-1024 processing elements.

the compute pool simply replaces its latency, L_e, with the pool's average processing latency plus the time taken for the data to be shifted to the next element: $\bar{L}_e + \bar{L}_{wait}$.

The increase in latency from removing a processing element is more than offset by the higher operating frequency achieved by the less congested design. Fig. 5.9 shows the effects of this congestion with rising resource costs, which will also result in a subsequent increase in interconnection and data routing.

5.11.3 Latency

Higher operating frequencies also benefit latency. While removing processing elements from the pool increases the total number of cycles to process the D data-paths, a higher operating frequency makes each cycle shorter. We might expect the benefits of such an improvement to be limited to the more computationally intensive functions, i.e., those requiring many cycles to produce a result, such as GELU and tanh. However, we see improvements **even for simple functions**, like LReLU.

Neural networks require all neuron outputs to be calculated before the network produces a final result. That is, latency is determined as the time taken to serve all data-paths, rather than at the level of any one individual data-path. For our compute pool approach, we therefore measure latency as the total time taken (ns) to process all D=1024 accessing data-paths.

Fig. 5.11 examines the latency in compute pools comprised of E=8 to E=1024 processing elements. Similar to throughput, latencies reach minimums when about E=64-256 processing elements serve the D=1024 data-paths. We average a 29% reduction in latency, with sigmoid achieving a 57% reduction, compared to the traditional D=E=1024 dataflow case.

5.11.4 Fractional Processing Elements

Section 5.5 described how fractional processing elements can provide additional computation by making use of otherwise unused resources. Of these new elements, targeted processing elements in particular can leverage the strength of the individual FPGA resources to improve application performance.

However, Sections 5.11.2 and 5.11.3 show that just maximising the number of processing elements is not the best way to achieve better performance. What role then can fractional processing elements play?

Individually, fractional elements serve only a small range of all possible inputs. The compute pool, *as a whole*, is tasked with serving all calls to the pool's implemented function. Numerous methods exist for decomposing and approximating a function into multiple fractional processing elements. Examples include cubic spline interpolation [195], least-squares approximation [196], and the Remez algorithm [197].

With a unified, compute pool approach we can not only decouple the number of D parallel data-paths from the number of E processing elements, but also the pool's implemented function from the individual processing elements. For example, a pool performing the tanh function can use cubic spline interpolation to assemble N processing elements, each tasked with serving $\frac{1}{N}$th of the function curve. This pool would then serve D parallel calls to the full tanh function.

To test the behaviour and efficacy of such a model we identify the primary drivers of performance as: **(1)** the percentage of the pool composed of fractional elements, i.e., $\frac{E}{D}$; and **(2)** the range of the input space covered by each fractional element. In performing our analysis we do not consider specific functions (such as tanh) or implementations (such as cubic spline interpo-

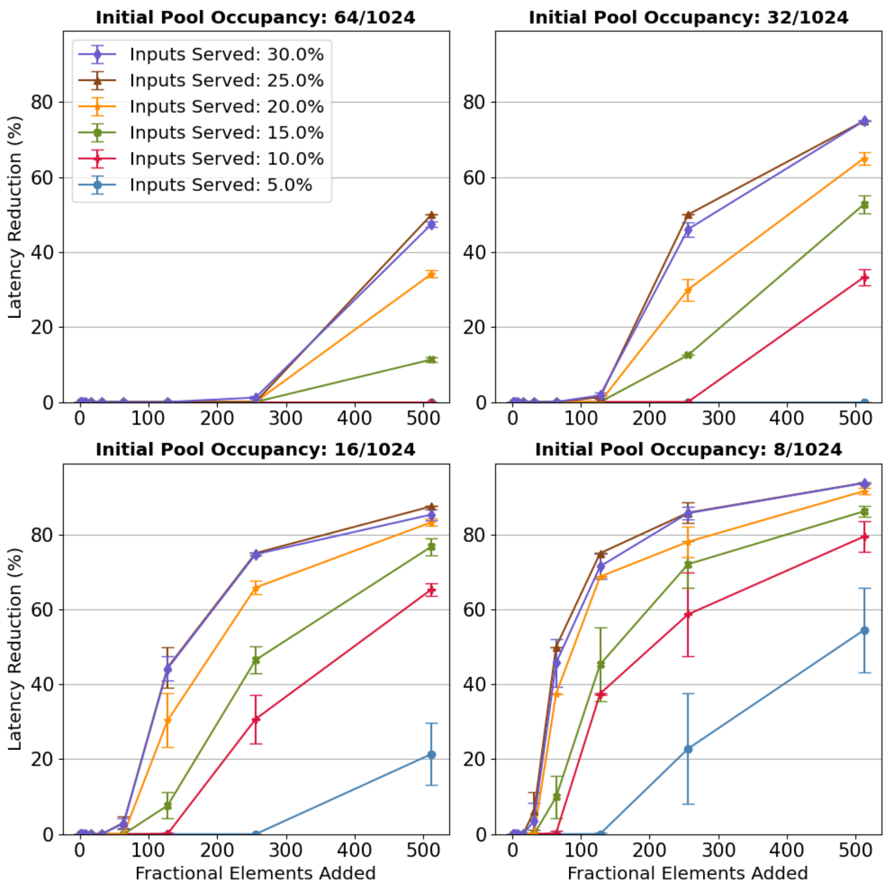


Figure 5.12: Adding fractional elements to compute pools with different initial occupancies.

lation). Instead we look at how our identified drivers affect performance using our approach. Tests are performed using both Gaussian and uniform random input data.

Fig. 5.12 shows the latency improvements when adding fractional elements to a compute pool. The pool is initially populated with processing elements that serve 100% of the input space, but not enough elements to simultaneously serve all the incoming D=1024 data-paths, i.e., $E < D$. Pool occupancy rates ($\frac{E}{D}$) of $\frac{64}{1024}$, $\frac{32}{1024}$, $\frac{16}{1024}$, $\frac{8}{1024}$ are tested. Tests are performed by adding increasing numbers of fractional processing elements to the pool, with each new element only able to serve some percentage of the total input space.

The greatest latency reductions are seen after adding 128-256 fractional elements to the pool, but the **input coverage** of each fractional element plays a greater role in reducing latency. This suggests performance driver **(2)** is more important than **(1)**, i.e., quality, rather than quantity of processing elements is a key factor. When forming new fractional elements within the FPGA, resources should then be combined to **serve more targeted operations** within a pool, rather than simply attached on an individual basis.

While adding 128 fractional elements to a pool may seem large, this is relative to the number of D data-paths. For D=32 data-paths, this translates to just 4 fractional elements.

When using fractional elements, pools with **lower occupancy** see the greatest improvement. Serving D=1024 data-paths with 256 fractional elements sees up to 50% reduction in latency when added to pools with E=32 full elements, 75% reduction for pools with E=16 full elements, and 85% reduction for pools with E=8 full elements.

Such configurations benefit smaller FPGAs, or those with a limited number of dedicated processing elements, like embedded CPUs or newer AI cores. These FPGAs can augment their limited number of full processing elements with many small, fractional processing elements, thus providing support for high degrees of data-path parallelism. Additionally, due to their smaller, specialised nature, the new fractional and targeted elements are likely to have lower latencies than their full element counterparts, further benefiting latency-bound applications.

One unusual feature we see when adding more fractional elements is that elements with 25% coverage of the input space outperform those elements with 30% coverage. This suggests **a limit to simply maximising coverage**, i.e., performance driver **(2)**. We observe this for pools with both high and low occupancy rates, suggesting this limiting factor is a product of the processing elements ability to serve a data-path, rather than due to a feature of our architecture.

When considering full processing elements, operating at 100% coverage of the input space, these elements appear well past the point of diminishing returns. This suggests compute pools formed solely of fractional elements may then be the more resource-optimal designs. Future work will explore this further.

5.12 Summary

In this chapter we present a novel architecture for incorporating processing elements with different implementations, latencies, and connections into a unified compute pool. Each heterogeneous processing element within a pool performs the same function, with a pool able to serve multiple, parallel calls to its implemented function.

Pools with 32-bit data-paths are implemented and evaluated on the Xilinx Alveo U280 accelerator card. We show that our new architecture and rotating scheduler scale linearly with parallelism. When scaling from 16 parallel function calls to 1024, we see a 64x increase in parallelism needing only 23.8x the resources.

Past coarse-grained overlays and virtual FPGAs see problems with low resource utilisation when applications are built atop these new frameworks. In contrast, we demonstrate 96.2% utilisation for 1024 parallel data-paths and 99.8% for 16 data-paths. Our maximum throughput for 32-bit floating-point implementations also sees a 10.9x-24.4x improvement over past approaches.

To determine application performance within our architecture, we evaluate 8 neural network activation functions, with 8-1024 processing elements serving 1024 parallel neurons. By classifying functions based on their computational intensity, i.e., number of operations per datum, we find basic functions like ReLU are too simple for our approach. However, all tested functions of higher computational intensity see improvement, with the sigmoid, swish, and tanh functions benefiting most.

Compared to a traditional, simplified dataflow approach, where each of the 1024 parallel data-paths require a dedicated processing element, we see an average 21% improvement in throughput and 29% for latency in pools with 64-256 processing elements serving 1024 data-paths.

With our approach, FPGA applications can scale performance up or down by adding/removing processing elements from compute pools. An application can be designed (or a neural network trained) on a larger FPGA, then deployed on smaller devices at a known performance cost, determined by our described performance model. We demonstrate that a mismatched number of data-paths and processing elements is both expected and required for optimal performance. We find maximising performance is achieved by *supporting access* to FPGA processing elements, not simply maximising resource use.

We demonstrate support for non-deterministic processing elements within our new compute pool architecture, and explore a new inter-pool routing scheme for successive compute pools. By allowing the parallel data-paths between pools to become temporarily out of sync, compute pools with few processing elements will see average latencies reduced by at least 11% for 2 successive pools, rising to over 40% for 5 pools.

As processing elements in a compute pool are *collectively* responsible for serving a function, we are able to provide support for new, fractional processing elements. These elements specialise in serving a set range of incoming function calls, with the limitation that an individual element may not be able to serve all possible function inputs.

In pools serving many parallel data-paths with few, full processing elements, the addition of these new fractional processing elements can see upwards of 85% reductions in expected latency. By tailoring both the number of small, fractional elements and the range of inputs they cover, we also provide a more fine-grained means of adding and removing FPGA resources to/from an application.

Chapter 6

Conclusion

In this thesis we look to improve the performance of low latency applications within reconfigurable devices through improved architectures and a more optimised use of resources. This chapter describes the contributions we make towards achieving this goal, discusses our results, and outlines possible areas of future work.

6.1 Summary of Achievements

Low latency applications implemented within FPGAs face a number of challenges. These stem from both traditional latency reduction challenges, as well as when deploying such applications within an FPGA. Although FPGA development can be slower and require more specialised, low-level design knowledge compared to CPU-based solutions, it offers a number of unique advantages and opportunities.

In Section 1.2 we identify 3 challenges for low latency applications within FPGAs: **(C1)** the low latency processing of data in edge computing environments, with a focus on financial data; **(C2)** facilitating low latency, shared access to memory resources; and **(C3)** a unified means of sharing access to all of the heterogeneous processing elements on an FPGA. In this section we describe the contributions we have made towards addressing these challenges.

Low latency, network-level processing of financial data feeds

The first challenge lies in exploring ways of leveraging FPGAs deployed at the network level. In Chapter 3 we describe an FPGA design for in-network arbitration of the redundant A/B data feeds provided by financial exchanges. This arbitration function must be carried out by all downstream applications, so we consolidate this functionality, and perform it only once at the network level.

Our key goals here were: **(a)** low latency, deterministic performance; **(b)** provide a general-purpose solution, supporting a wide range of communication protocols; and **(c)** allowing for application-specific optimisations and control in the more general-purpose, network level application.

For **(b)** and **(c)**, we support the different priorities of different downstream applications by providing two simultaneous output streams: one prioritising low latency and one prioritising high message reliability. An important addition to this is the ability for downstream applications to control, in real-time, the windowing method used by the high reliability output stream. These modes are based on timeouts, the number of messages received, or both.

Performing arbitration at the network level also allows a broader, less application-specific view of data streams. As part of our work we present a model to describe message and packet processing latencies. This can indicate data feed saturation within the windowing methods, i.e., the point at which network, communication, or traffic volume issues, begin to degrade arbitration performance. When analysing network traffic, the model provides a quantifiable means of identifying when use of the low latency output stream, rather than the high reliability stream, becomes the optimal approach.

In addressing goal **(a)**, we implement our arbitrator on a Xilinx Virtex-6 FPGA, demonstrating latencies of 7 cycles for the high reliability mode, while the low latency mode requires just 1 cycle for simple arbitration. High reliability therefore achieves latencies of $42ns$ for TotalView-ITCH data feeds, and $36.75ns$ for OPRA and ARCA feeds. The low latency mode then corresponds to latencies of $6ns$ and $5.25ns$ respectively. In addition, our dataflow design means latencies are deterministic. This improves on CPU-based designs that see uncertainty from caching issues, DDR memory operation latencies, and possible CPU scheduling conflicts in multi-user computing environments.

For the three messaging protocols examined, TotalView-ITCH, OPRA, and ARCA, we offer latencies 10x lower than an FPGA-based commercial design and 4.1x lower than the hardware-accelerated IBM PowerEN processor. Our arbitrator achieves throughputs of 21.3Gbps for TotalView-ITCH, and 24.3Gbps for OPRA and ARCA, more than double the specified 10Gbps network line rate.

The most resource intensive protocol, TotalView-ITCH, is also implemented on a Xilinx Virtex-5 FPGA within a network interface card, and tested using real market data. Through use of our model, performance estimators, and our new cycle-accurate FPGA testing framework, we demonstrate the effectiveness of our design at message rates 100 times their current level.

Reduce memory access latencies for highly parallel, latency-sensitive functions

The second challenge involves reducing latency when multiple, parallel accesses are required to a shared resource. This parallelism may come from many independent applications or from just one application which itself contains multiple, parallel data-paths, like a neural network.

Chapter 4 describes our focus on lowering memory access latencies and how we apply this to memory-based computing. Using pre-computed function values and memory-based look-ups, we use a new, ring-based topology to provide independent access to the N_{SM} individual BRAMs that are traditionally grouped together within the FPGA and accessed as a single, monolithic memory unit.

Here, our key goals were for: **(a)** the consistent, low latency performance of any function implemented with memory-based computing; **(b)** a high-level, automated tool for developers to make use of our approach; and **(c)** favourable performance, routing, and arbitration logic when scaling to both higher numbers of parallel function calls, and higher numbers of BRAMs.

For **(a)**, when looking to memory-based computing, we target low latency applications: employing complex, multi-cycle functions; requiring iterative computation, but lacking the latency budget for such solutions; and/or relying on high levels of parallelism.

As compute-bound functions will be converted to memory-bound look-ups, we see the latency of any function reduced to just a single cycle. This is of particular benefit to latency-sensitive applications which are improved by latency-costly methods, such as the higher degrees of numerical precision that come from iterative computation.

We find functions require a computational intensity—i.e., the number of operations per byte—of at least 0.25 ops/byte for 16-bit functions, or 0.5 ops/byte for 8-bit functions to see benefits from our approach. For goal **(b)**, an automated tool is provided to convert existing Python functions and memory structures to our new architecture. Both integer and arbitrary precision floating-point numbers are supported. As no function transformations or approximations are used we guarantee: accurate and consistent results; a high degree of function compatibility; and low barriers for entry when porting new or existing functions to our architecture.

Low latency performance and resource use are factors when scaling to increased levels of parallelism **(c)**. Our interconnect scales linearly with both the many D memory reads—i.e., function calls—as well as to the many N_{SM} BRAMs that collectively store the function data. We find this scaling is most efficient when the number of D parallel function calls exceed the number of N_{SM} sub-memories. Our interconnect is also found to require less logic, routing, and arbitration compared to point-to-point, crossbar, and multi-stage networks.

The width of the function's input data is shown to be the main limiting factor to a memory-based computing approach. When implementing our new architecture on the Xilinx Alveo U280 accelerator card we find 12-bit functions to be the most resource efficient. However, 16-bit designs are preferred due to their alignment to standard data formats.

When assessed against square root, Gaussian, trigonometric, and hyperbolic functions, we see a 37-92% reduction in LUTs per parallel function call. This demonstrates our approach can efficiently free up these resources when transitioning to a memory-based computing solution. For performance, we also show a 4.4x to 29x reduction in latency compared to past works.

The key benefit of our approach is the support for *any* 16-bit function, offering a 1-cycle latency and high, sustained 300MHz operating frequency for up to 1024 parallel function calls. By using the 2 read ports of Xilinx BRAMs we can further lower latency, averaging just 0.5 cycles per parallel function call.

Centralised compute pools for shared access to FPGA processing elements

As a natural extension of the previous challenge, the third challenge identifies a broader need for facilitating shared, parallel access to all FPGA resources, not just memory. This requires a much more robust, generalised framework able to support any type of processing element, while still scaling favourably to higher degrees of parallelism.

Our goals were to: **(a)** support all of the different types of processing element available on the FPGA; **(b)** unify them into a scalable, high-level architecture; and finally, when building applications with this new architecture we should **(c)** efficiently utilise all FPGA resources and **(d)** facilitate high throughput, low latency, and resource-efficient designs.

Chapter 5 introduces a latency-centric, call-and-response model that allows us a single, unified view of the many heterogeneous processing elements within an FPGA **(a)**. From this model we create a new compute pool architecture, where each compute pool contains many different types of processing element, each implementing the same function. Our approach unifies processing elements based on function, not implementation. A compute pool therefore supports processing elements with different implementations, connections, pipeline lengths, latencies, as well as non-deterministic latencies.

For **(b)**, we see that when implemented on the Xilinx Alveo U280 accelerator card, our new compute pool architecture and rotating scheduler scale linearly with parallelism. When scaling from 16 parallel function calls to 1024, using 32-bit data-paths, we can achieve a 64x increase in parallelism with only 23.8x the resources.

Past coarse-grained architectures within FPGAs can suffer from low resource utilisation when applications are built atop their new frameworks. In contrast, by supporting any processing element implementation we achieve goal **(c)** by demonstrating 96.2% utilisation for pools with 1024 parallel data-paths, and 99.8% for 16 data-paths. Our maximum throughput for 32-bit floating-point implementations also sees a 10.9x-24.4x improvement over past approaches.

Expanding on our model, we explore a new inter-pool routing scheme for designs employing multiple compute pools. By allowing the parallel data-paths between successive pools to become temporarily out of sync, compute pools with few processing elements serving many parallel paths see average latencies reduced by at least 11% for 2 successive pools, rising to over 40% for 5 pools.

To determine application performance within our architecture for goal **(d)**, we evaluate 8 neural network activation functions on the Xilinx Alveo U280. We find basic functions like ReLU—composed of just a 1-bit comparator—are too computationally simple for our approach. However, all other tested functions see improvement, with the sigmoid, swish, and tanh functions benefiting most.

When testing 8-1024 processing elements serving 1024 parallel neurons we see an average 21% improvement in throughput, and 29% for latency. A traditional, simplified dataflow approach would see 1024 processing elements serve the 1024 data-paths, but we see these improvements when only 64-256 processing elements serve the 1024 data-paths. We demonstrate that a mismatched number of data-paths and processing elements is both expected and required for optimal performance. Maximising performance is therefore achieved by *supporting access* to FPGA processing elements, not simply maximising resource use.

With our new architecture, the multiple processing elements within a compute pool are *collectively* responsible for serving the pool's implemented function. We explore this idea, and expand on goal **(d)**, by introducing support for new, fractional processing elements. These elements specialise in serving a set range of incoming function calls; however, individual elements may be unable to serve all function inputs. This model allows individual elements to become more specialised, opening the door to the use of more optimised and targeted processing elements.

Adding these new fractional elements to compute pools containing small numbers of conventional, full processing elements, sees upwards of 85% reductions in expected latencies. By tailoring both **(1)** the number of small, fractional elements and **(2)** the range of inputs they cover, this approach provides a more fine-grained means of adding/removing FPGA resources to/from an application.

6.2 Future Work

When describing our work in this thesis we have also looked to outline its limitations. Some of these may be addressed in future work, either through new solutions, new design choices, or via the use of a more general-purpose tool for implementing FPGA designs using our ideas.

In regards to in-network processing, our arbitrator is implemented as an application for bespoke clusters. It may therefore be unsuitable for deployment in a cloud computing environment without additional care taken to ensure we do not interfere with background network traffic.

We seek to improve low latency applications via memory-based computing and our new rotating scheduler and architecture, described in Chapter 4. While not only applicable to memory-based computing, we see promising results when it is used in this context. However, the small BRAM storage elements we use limit us to smaller data widths. This approach may then not be applicable to more general-purpose computation, such as those based on common 32-bit and 64-bit data widths.

In regards to the compute pools we introduce in Chapter 5, the definition and use if processing elements should be more strictly codified. How best do we include processing cores, such as those of CPUs, GPUs, and AI Engines, if they contain functionality—such as accumulators—that may be incompatible with a compute pool model?

In this section we will address these issues and discuss the areas where our existing work could be further developed or improved. We will also look to highlight some possible directions for future research.

6.2.1 FPGA Processing at the Infrastructure Level

Adoption of the Micro-service Model

The message data feed arbitrator we describe in Chapter 3 allows downstream applications to control aspects of the arbitrator's operation, specifically, the threshold values for the high reliability windowing modes. This should be expanded to allow **(a)** downstream applications a greater range of control, and **(b)** the arbitrator itself to act as a source of information.

For **(a)**, an increased range of control could see an expansion of our 3 windowing methods such that determinations are made based on message contents, rather than the message meta-data of timeouts and message counts. Meanwhile, for **(b)**, Section 3.7 shows message rates can act as an indicator of the arbitrator becoming saturated, leading to reduced performance. This insight can also be of benefit to downstream financial applications, or even to network monitoring and management tools.

Future applications should then look to follow the micro-service model [198], which sees common schemes and protocols for transaction-based—often stateless—communication between services. Alignment with this model ensures both compatibility with wider industry practices [199], and that there is some common mechanism for other applications to interact with, control, and access useful information within these new, in-network applications.

In-network Pre-processing of Data

In addition to financial data feed arbitration, we can consolidate other common financial functions at the network level. One example is on-the-fly calculation of financial indices. These track the price of a group of assets, normally as a weighted sum. The index price is updated when the price of any constituent asset changes, so index price calculations must be performed regularly. Common signal analyses, such as tracking the moving average price or volatility of some security, could also be consolidated at the network level.

Other financial data processing approaches might see these common calculations performed on dedicated compute nodes, stored in a central database, then later read out by interested applications. We can instead perform calculations on messages as they pass through the FPGA NIC, then append results to the end of the message packet itself. This lowers processing latency and reduces the total amount of network traffic.

The consolidation of common functions in other domains is also possible. For example, with distributed data storage, the integrity of data can be verified as it passes through the network. An appropriate error detection and correction scheme could see errors fixed automatically [200]. Data security and encryption policies can also be checked and enforced in real-time [201].

The inclusion of FPGAs as the network level processing device, rather than CPUs, brings additional benefits for such in-network processing. A CPU-based control flow model might see transmitted packets written to and read from a storage buffer. In contrast, the dataflow model of execution within FPGAs mean the contents of each packet will always be streamed through the FPGA. A deeper inspection of packets then comes at a minimal cost. The dataflow model can also help provide tighter bounds when guaranteeing deterministic behaviour due to the integrated nature of memory and processing resources.

Stand-alone In-network Applications

Rather than just acting to process data within a data centre environment, in-network processes can instead be classed as an application in their own right. The micro-service model sees applications formed from the coordinated effort of many, smaller services. However, for simple applications, the use of a single, small service may be sufficient.

The Internet of Things (IoT) refers to the growing number of small, internet-connected devices that serve a simple function. Examples include consumer devices, such as “smart” thermostats, refrigerators, doorbells, etc, as well as enterprise devices such as security, monitoring, and industrial control equipment. Such devices may depend on cloud computing environments for computing tasks, or for some additional form of coordination or control. These simple needs can be met by equally simple applications.

FPGAs contain many resources that can be formed into independent processing elements. The inherently parallel nature of FPGAs mean many small packet processing applications can be hosted on a single FPGA device. Multiple FPGA applications can then query the packet as it passes through the network pipeline. For comparison, the CPU-based control flow model is limited by the centralised nature of the processing elements, and faces a memory access bottleneck when multiple processes operate in parallel. FPGAs are the natural choice for such stand-alone in-network applications, and are the ideal platform for exploring such future work.

6.2.2 Optimising Parallel Access to Shared Resources

Improving the Alignment of Function Calls

When introducing our new low latency interconnect in Chapter 4, we show that the lowest latencies are achieved when each of the D parallel function calls are exactly aligned with the sub-memory they target. That is, when the function call first stored in register 0 targets sub-memory 0, the function call in register 1 targets sub-memory 1, and so on. As all function calls can then be served immediately, such a configuration would require only 1 cycle to process all of the D parallel function calls.

It is difficult to ensure this alignment with real-world data; however, even small efforts towards optimising alignment can see large improvements. For this, future work should explore the use of pre-sorting methods for the D parallel function calls to memory. As multiple function calls may target the same sub-memory, a complete sort of all parallel calls should not be our goal. We must instead look to reduce overall latency by rearranging function calls so that they align, or are more likely to align, with the correct sub-memory.

In Section 4.7 we evaluate our interconnect using D=1024 parallel data-paths. 1024 calls to a traditional, monolithic memory would typically require 1024 cycles. Using our new architecture, it is worth spending 10s or even 100s of cycles to align, or more favourably align, these function calls if it can lead to greater latency savings. The nature and performance of such sorting methods is an interesting avenue to explore for optimising the use of our new architecture.

Support for Different Types of Memory

The goal of the new interconnect we introduce in Chapter 4 is to facilitate many parallel, low latency accesses to memory. For choice of memory, we therefore focus on FPGA BRAMs due to: **(a)** the large number of BRAMs available on FPGAs, **(b)** their 2 independent access ports, and **(c)** their 1-cycle access times. However, the small storage capacity of BRAMs is a limiting factor in supporting higher data widths.

As part of our work, we looked to memory-based computing to offer memory access improvements to low latency, compute-bound applications. As storage constraints caused resource use to scale exponentially with data width, we chose to support up to 16-bit functions. Supporting higher data widths would make our architecture available to a greater range of functions, make it easier to transition existing designs to our architecture, and lower the level of computational complexity required to benefit from our approach. For this we must look beyond BRAMs.

DDR memory has greater storage capacity, but suffers due to the limited number of access ports and high latencies for random memory accesses. HBM is a promising candidate. Its storage is comparable to DDR memory, but offers independent access to its multiple memory banks. The Xilinx Alveo U280 accelerator card used to evaluate our work in Chapters 4 and 5 includes 8GB of HBM [202], with 32 independent access ports. The reduced number of access ports compared to BRAM becomes a limiting factor. From Eqn. 4.1 we find that a 16-bit function would use 64 BRAMs, and therefore possess 128 access ports. Another issue with HBM is that it uses DDR memory for storage, leading to the same high latencies for random memory accesses.

These factors, combined with the many different types of memory on the FPGA, mean a hybrid approach is the next logical step when expanding our use of memory. This opens up traditional FPGA memory resources—such as UltraRAMs, distributed memory, DDR, and HBM—but also network-attached storage. For such an architecture, a more general NUMA [203] model is needed. The interconnect we develop in Chapter 4 requires deterministic read latencies, however, we can make use of the framework developed in Chapter 5 to overcome this limitation.

Adopting a memory hierarchy approach would provide a sliding scale of performance. The storage requirements for 16-bit functions, for example, could be met entirely by BRAMs. Functions with wider data widths would require more storage. Thus, additional memory—

like HBM—would be needed to serve some function calls. Traditional memory improvements, like caching [204], can also aid in lowering access latencies. We discuss a number of different possible improvements in greater detail in Sections 2.4.3 and 2.4.4.

The evaluation of such a multi-memory architecture would seek to determine both the efficacy of the approach, and the stage at which the support for higher data widths begins to compromise our goal of low latency memory access.

6.2.3 The Compute Pool Model

Greater Focus on Processing Elements

The compute pools introduced in Chapter 5 support any number of heterogeneous processing elements. We place no limits on the implementation or behaviour of processing elements. If we extend the traditional definition of a "processing element" we can open up access to a more diverse range of current and future resources.

An extended processing element may be a gateway to some system or network level resource, such as a shared database, or an external hardware accelerator. The extended element may also simply be a communication channel to other compute devices; we would pass function input data to this processing element, wait L cycles, then retrieve the result.

For such a system, the latency-centric, call-and-response model we describe in Section 5.6, forms the heart of this new architecture. Its unified, implementation-agnostic view of processing elements means we are able to support *any* processing element. By using latency as our "common currency" for determining performance, low latency applications will see particular benefit from this increased processing element diversity.

Further Examination of Fractional Processing Elements

From Fig. 5.12, we see that the performance of fractional elements peak when serving only 25-30% of all possible function inputs. Serving a wider range of function inputs does not lead to higher throughputs or lower latencies. Large numbers of these smaller, more optimised processing elements may then be the optimal model in terms of both performance, power, and resource use.

Traditional, fully implemented processing elements may then be long past the point of diminishing returns. We only briefly explore this within the context of evaluating these new fractional elements in Section 5.11. An interesting research question remains in how effective these fractional elements will be in completely replacing traditional, full processing elements.

FPGA-agnostic Application Development

Our new compute pool architecture would benefit from a more formal application development process. This would both streamline application design, and improve our compatibility with HLS development flows.

A key benefit of our approach is that while the number of parallel data-paths connected to a pool is exactly defined, the number of processing elements within a pool is not. An application can be designed, simulated, built, and tested using a *minimal* design, i.e., just one processing element in each compute pool. We can then demonstrate the correctness of a design without the long build-times typically associated with FPGA application development.

As a final step, the number of processing elements in each pool is increased until optimal performance is reached for our target FPGA. If we generalise this approach, we can provide a tool for the automatic design space exploration of mixed-resource designs. Given a target FPGA and resource budget we could, for example, **(a)** promote the use of all the processing elements on large, heterogeneous FPGAs, or **(b)** create a high performing, minimalist design so the application can fit on smaller devices.

As we mention in Section 5.9, neural networks in particular would see benefits from such a design flow. Neural networks could be trained on a large FPGA with many processing elements, then deployed on a smaller FPGA by using fewer elements in the compute pools. Since activation functions have shown such promise with our new architecture, a more in-depth examination should be carried out.

With our implementation-agnostic compute pools, we could provide developers with parametric libraries of common neural network activation functions. Each function would be implemented on, and optimised for, multiple sets of FPGA processing elements. These new parametrised library functions can then automatically populate a compute pool with processing elements to fit a desired throughput, latency, or resource use goal. The low-level implementation and management of these functions is then fully abstracted away from the developer.

Existing libraries of processing elements can also be used without modification, along with the long established and well supported development and analysis tools developed by Xilinx [72], Intel [74], and third-party developers [205].

Run-time Control of Compute Pools: Performance

Our evaluation of compute pools in Chapter 5 looked at design space exploration for throughput, latency, and resource use, but not power. Compute pools provide an excellent mechanism for dynamic power control. A pool with N full processing elements can disable 1 or more elements to reduce power use. The application will naturally see a reduction in performance, the latency penalty for which can be determined by our model described in Section 5.6.

Another area where dynamically enabling or disabling pool elements can be of benefit is within elastic processing environments. Here, the number of processing elements in a compute pool is used to determine application performance. A user will configure their application or system for an initial level of performance, then when workloads increase, this performance can be scaled up in real-time by enabling additional processing elements within a compute pool.

Similar setups are used by cloud computing [56] [57] [58] regarding virtual CPUs, the number of which can be scaled up or down at run-time. With our new compute pool model we can extend this level of fine-grained control to FPGA-based applications.

Run-time Control of Compute Pools: Applications

Run-time control can also be applied to fault tolerance in safety-critical systems. Here, we can modify the level of processing *redundancy* within a compute pool. Instead of being served by 1 of N processing elements, each function call could be served by k of the available N elements. The k results would then be compared using a standard majority-rule to produce a single, final result. This level of k redundancy may then be time, application, or function specific, and can allow failing processing elements to be easily removed and replaced within the compute pool.

Run-time control of processing elements can also have wider-reaching application benefits. One example is helping protect against side-channel attacks on cryptographic functions. Processing time and power use are two of the main indicators that may leak information to potential attackers [206].

By randomly enabling/disabling processing elements in a pool that perform some cryptographic function, we make processing time and power use less predictable. Uncertainty is introduced both between repeated calls to the same function, as well as within the function itself. The heterogeneous nature of processing elements in the pool also adds uncertainty due to their different implementations, resources, latencies, and power use.

6.3 Final Thoughts

FPGAs provide a powerful and highly varied platform for computer processing, bringing together many different memory resources, processing elements, and architectural ideas. The inclusion of many different interconnected and configurable resources mean FPGA designs can often blur the line between hardware and software solutions.

This latter point means application development on FPGAs can tend to be more time consuming and requiring of specialised hardware knowledge. We must therefore ensure that the benefits and possibilities of FPGAs are not overshadowed by the often longer and more involved development process. For this goal, we see our higher-level, general-purpose approach to FPGA processing elements and application design as the way forward.

Our performance and resource models, described in Chapters 4 and 5, allow us to efficiently, and scalably, unify current and upcoming processing elements—like Xilinx's new AI Engines [14]—

while also allowing for the inclusion of future memory and processing resources. Unifying access to FPGA resources, streamlining development, and more closely aligning to existing HLS development flows, will mean the potential of FPGAs as a dedicated computing platform can be more fully realised.

With the introduction and examination of our new architectures and ideas, we hope this thesis can make progress towards this goal, and look forward to continuing this research into the future.

Bibliography

[1] Y. Lyu, L. Bai, and X. Huang, 'ChipNet: Real-Time LiDAR Processing for Drivable Region Segmentation on an FPGA', IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 66, no. 5, pp. 1769-1779, May 2019, doi: 10.1109/TCSI.2018.2881162.

[2] L. Bai, Y. Lyu, X. Xu, and X. Huang, 'PointNet on FPGA for Real-Time LiDAR Point Cloud Processing', in 2020 IEEE International Symposium on Circuits and Systems (ISCAS), Oct. 2020, pp. 1-5. doi: 10.1109/ISCAS45731.2020.9180841.

[3] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, 'A Survey of FPGA-Based Robotic Computing', IEEE Circuits and Systems Magazine, vol. 21, no. 2, pp. 48-74, 2021, doi: 10.1109/MCAS.2021.3071609.

[4] M. Belwal, M. Purnaprajna, and S. TSB, 'Enabling seamless execution on hybrid CPU/FPGA systems: Challenges amp; directions', in 2015 25th International Conference on Field Programmable Logic and Applications (FPL), Sep. 2015, pp. 1-8. doi: 10.1109/FPL.2015.7294022.

[5] Q. Tang, L. Jiang, M. Su, and Q. Dai, 'A pipelined market data processing architecture to overcome financial data dependency', in 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC), Dec. 2016, pp. 1-8. doi: 10.1109/PCCC.2016.7820632.

[6] A. Boutros, B. Grady, M. Abbas, and P. Chow, 'Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis', in 2017 International Conference on ReCon-

Figurable Computing and FPGAs (ReConFig), Dec. 2017, pp. 1-6. doi: 10.1109/RECONFIG.2017.8279781.

[7] S. Friston, A. Steed, S. Tilbury, and G. Gaydadjiev, 'Construction and Evaluation of an Ultra Low Latency Frameless Renderer for VR', IEEE Transactions on Visualization and Computer Graphics, vol. 22, no. 4, pp. 1377-1386, Apr. 2016, doi: 10.1109/TVCG.2016.2518079.

[8] Chamola, Vinay, Sambit Patra, Neeraj Kumar, and Mohsen Guizani. 'FPGA for 5G: Re-Configurable Hardware for Next Generation Communication'. IEEE Wireless Communications 27, no. 3 (June 2020): 140-47. https://doi.org/10.1109/MWC.001.1900359.

[9] M. Satyanarayanan, 'The Emergence of Edge Computing', Computer, vol. 50, no. 1, pp. 30-39, Jan. 2017, doi: 10.1109/MC.2017.9.

[10] Intel, 'Intel Agilex FPGA Product Brief'. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/agilex-fpga-product-brief.pdf

[11] Xilinx, 'Zynq-7000 SoC Data Sheet: Overview (DS190)'. 2018. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[12] Y. Gosain and A. Gupta, 'Xilinx Advanced Multimedia Solutions with Video Codec/Graphics Engines'. 2017. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/white_papers/wp497-multimedia.pdf

[13] Xilinx, 'Zynq UltraScale+ RFSoC Product Brief'. 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-rfsoc-product-brief.pdf

[14] Xilinx, 'Xilinx AI Engines and Their Applications'. Jul. 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/white_papers/wp506-ai-engine.pdf

[15] Xilinx, 'UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices'. Jun. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[16] S. Velagapudi, 'White Paper: Addressing Memory-Bandwidth and Compute-Intensive Challenges with Intel Agilex M-Series FPGAs'. Intel. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/memory-bandwidth-and-compute-intensive-with-agilex-m-series-white-paper.pdf

[17] M. Wissolik, D. Zacher, A. Torza, and B. Day, 'Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance'. 2019. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf

[18] Xilinx, 'Virtex-4 FPGA User Guide'. Dec. 2008. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[19] Xilinx, 'UltraScale Architecture Configurable Logic Block User Guide (UG574)'. Feb. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[20] Intel, 'FPGA Architecture'. Jul. 2006. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf

[21] D. Burke, R. A. Shafik, and A. Yakovlev, 'Challenges and opportunities in research and education of heterogeneous many-core applications', in 2016 11th European Workshop on Microelectronics Education (EWME), May 2016, pp. 1-6. doi: 10.1109/EWME.2016.7496480.

[22] A. Dehon, 'Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)', Jan. 1999, doi: 10.1145/296399.296431.

[23] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, 'Edge Computing: Vision and Challenges', IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637-646, Oct. 2016, doi: 10.1109/JIOT.2016.2579198.

[24] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, ‘A Survey on the Edge Computing for the Internet of Things’, IEEE Access, vol. 6, pp. 6900-6919, 2018, doi: 10.1109/ACCESS.2017.2778504.

[25] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, ‘Survey on Multi-Access Edge Computing for Internet of Things Realization’, IEEE Communications Surveys & Tutorials, vol. 20, no. 4, pp. 2961-2991, 2018, doi: 10.1109/COMST.2018.2849509.

[26] J. Chen and X. Ran, ‘Deep Learning With Edge Computing: A Review’, Proceedings of the IEEE, vol. 107, no. 8, pp. 1655-1674, Aug. 2019, doi: 10.1109/JPROC.2019.2921977.

[27] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, ‘A Survey on Mobile Edge Computing: The Communication Perspective’, IEEE Communications Surveys & Tutorials, vol. 19, no. 4, pp. 2322-2358, 2017, doi: 10.1109/COMST.2017.2745201.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ‘ImageNet classification with deep convolutional neural networks’, Commun. ACM, vol. 60, no. 6, pp. 84-90, May 2017, doi: 10.1145/3065386.

[29] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. ‘Deep Residual Learning for Image Recognition’. ArXiv:1512.03385 [Cs], 10 December 2015. `http://arxiv.org/abs/1512.03385`.

[30] M. Nguyen and J. C. Hoe, ‘Time-Shared Execution of Realtime Computer Vision Pipelines by Dynamic Partial Reconfiguration’, in 2018 28th International Conference on Field Programmable Logic and Applications (FPL), Aug. 2018, pp. 230-2304. doi: 10.1109/FPL.2018.00046.

[31] X. Li, A. K. Jain, D. L. Maskell, and S. A. Fahmy, ‘A time-multiplexed FPGA overlay with linear interconnect’, in 2018 Design, Automation Test in Europe Conference Exhibition (DATE), Mar. 2018, pp. 1075-1080. doi: 10.23919/DATE.2018.8342171.

[32] H. J. Yang, K. Fleming, M. Adler, and J. Emer, ‘LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories’, in 2014 IEEE 22nd Annual International

Symposium on Field-Programmable Custom Computing Machines, May 2014, pp. 117-124. doi: 10.1109/FCCM.2014.43.

[33] B. Ronak and S. A. Fahmy, ‘Improved resource sharing for FPGA DSP blocks’, in 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Aug. 2016, pp. 1-4. doi: 10.1109/FPL.2016.7577373.

[34] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, ‘ReconOS: An Operating System Approach for Reconfigurable Computing’, Micro, IEEE, vol. 34, pp. 60-71, Jan. 2014, doi: 10.1109/MM.2013.110.

[35] A. Ismail and L. Shannon, ‘FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators’, Jun. 2011, pp. 170-177. doi: 10.1109/FCCM.2011.48.

[36] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, ‘In-Datacenter Performance Analysis of a Tensor Processing Unit’, in Proceedings of the 44th Annual International Symposium on Computer Architecture, New York, NY, USA, Jun. 2017, pp. 1-12. doi: 10.1145/3079856.3080246.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, ‘Spectre Attacks: Exploiting Speculative Execution’, in 2019 IEEE Symposium on Security and Privacy (SP), May 2019, pp. 1-19. doi: 10.1109/SP.2019.00002.

[38] Xilinx, 'UltraScale Architecture and Product Data Sheet: Overview (DS890)'. May 2022. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds890-ultrascale-overview.pdf

[39] Intel, 'Intel Stratix 10 GX/SX Device Overview'. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683729/current/gx-sx-device-overview.html

[40] Xilinx, 'Ultrascale+ FPGA Product Selection Guide', 2021. https://www.xilinx.com/content/dam/xilinx/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (accessed Aug. 30, 2021).

[41] Xilinx, 'Medical Imaging with CT, MRI and PET', Xilinx, 2022. https://www.xilinx.com/applications/medical/medical-imaging-ct-mri-pet.html (accessed Nov. 21, 2022).

[42] Xilinx, 'Industrial Boards and Kits', Robotics, 2022. https://www.xilinx.com/products/boards-and-kits/market/nav-industrial.html (accessed Nov. 21, 2022).

[43] Xilinx, 'Virtex UltraScale+ 58G PAM4 FPGA', 2022. https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/xilinx-virtex-ultraScale-puls-58g-PAM4-FPGA-product-brief-rev-1_2.pdf (accessed Nov. 21, 2022).

[44] G. W. Morris, G. A. Constantinides, and P. Y. K. Cheung, 'ROM to DSP block transfer for resource constrained synthesis', IET Computers & Digital Techniques, vol. 1, no. 1, pp. 17-26(9), Jan. 2007.

[45] Xilinx, 'Block Memory Generator v8.4: LogiCORE IP Product Guide'. Xilinx, Dec. 19, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf

[46] Intel, 'Intel Stratix 10 Embedded Memory User Guide'. Accessed: Mar. 01, 2021. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-memory.pdf

[47] Xilinx, 'UltraScale Architecture DSP Slice User Guide'. 2021. [Online]. Available: `https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf`

[48] Xilinx, '40G/50G High Speed Ethernet Subsystem Product Guide (PG211)'. May 2022. [Online]. Available: `https://docs.xilinx.com/r/en-US/pg211-50g-ethernet/Introduction`

[49] Xilinx, 'UltraScale+ Devices Integrated Block for PCI Express Product Guide (PG213)'. May 2022. [Online]. Available: `https://docs.xilinx.com/r/en-US/pg213-pcie4-ultrascale-plus/Introduction`

[50] Xilinx. 'Versal Architecture and Product Data Sheet: Overview (DS950)', 2021. `https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf`.

[51] N. Alachiotis, S. A. Berger, and A. Stamatakis, 'Efficient PC-FPGA Communication over Gigabit Ethernet', in 2010 10th IEEE International Conference on Computer and Information Technology, Jun. 2010, pp. 1727-1734. doi: 10.1109/CIT.2010.302.

[52] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, 'An efficient and flexible host-FPGA PCIe communication library', in 2014 24th International Conference on Field Programmable Logic and Applications (FPL), Sep. 2014, pp. 1-6. doi: 10.1109/FPL.2014.6927459.

[53] J. Detrey and F. de Dinechin, 'Table-based polynomials for fast hardware function evaluation', in 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05), Jul. 2005, pp. 328-333. doi: 10.1109/ASAP.2005.61.

[54] S. A. White, 'Applications of distributed arithmetic to digital signal processing: a tutorial review', IEEE ASSP Magazine, vol. 6, no. 3, pp. 4-19, Jul. 1989, doi: 10.1109/53.29648.

[55] X. Wang, Y. Niu, F. Liu, and Z. Xu, 'When FPGA Meets Cloud: A First Look at Performance', IEEE Transactions on Cloud Computing, 2020, doi: 10.1109/TCC.2020.2992548.

[56] Amazon, 'Amazon EC2 F1 Instances', Amazon Web Services, Inc. `https://aws.amazon.com/ec2/instance-types/f1` (accessed May 30, 2022).

[57] Alibaba, 'What is FPGA as a Service?' `https://www.alibabacloud.com/help/en/fpga-based-ecs-instance/latest/what-are-fpga-accelerated-instances` (accessed May 30, 2022).

[58] Microsoft, 'Azure virtual machine sizes for field-programmable gate arrays (FPGA)'. `https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-field-programmable-gate-arrays` (accessed May 30, 2022).

[59] S. Denholm, K. H. Tsoi, P. Pietzuch, and W. Luk, 'CusComNet: A customisable network for reconfigurable heterogeneous clusters', in ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, Sep. 2011, pp. 9-16. doi: 10.1109/ASAP.2011.6043231.

[60] S. Denholm, K. H. Tsoi, P. Pietzuch, and W. Luk, 'Efficient Communication for FPGA Clusters', in Reconfigurable Computing: Architectures, Tools and Applications, Berlin, Heidelberg, 2012, pp. 335-341. doi: 10.1007/978-3-642-28365-9_29.

[61] G. Inggs, D. B. Thomas, G. Constantinides, and W. Luk, 'Seeing Shapes in Clouds: On the Performance-Cost trade-off for Heterogeneous Infrastructure-as-a-Service'. arXiv, Aug. 27, 2015. doi: 10.48550/arXiv.1506.06684.

[62] I. Sittón-Candanedo, R. S. Alonso, J. M. Corchado, S. Rodríguez-González, and R. Casado-Vara, 'A review of edge computing reference architectures and a new global edge proposal', Future Gener. Comput. Syst., vol. 99, no. C, pp. 278-294, Oct. 2019, doi: 10.1016/j.future.2019.04.016.

[63] A. Moore, 'IPU-Based Cloud Infrastructure: The Fulcrum for Digital Business'. 2021. [Online]. Available: `https://www.intel.co.uk/content/dam/www/central-libraries/us/en/documents/ipu-based-cloud-infrastructure-white-paper.pdf`

[64] Intel, 'Infrastructure Processing Units (IPUs)'. `https://www.intel.com/content/www/uk/en/products/network-io/smartnic.html` (accessed May 30, 2022).

[65] ARM, 'An introduction to AMBA AXI'. `https://developer.arm.com/documentation/102202/0200/AXI-protocol-overview` (accessed Jun. 02, 2022).

[66] A. Massoudi, 'Knight Capital glitch loss hits $461m', Financial Times. `https://www.ft.com/content/928a1528-1859-11e2-80e9-00144feabdc0` (accessed Jun. 02, 2022).

[67] M. C. McFarland, A. C. Parker, and R. Camposano, 'The high-level synthesis of digital systems', Proceedings of the IEEE, vol. 78, no. 2, pp. 301-318, Feb. 1990, doi: 10.1109/5.52214.

[68] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, 'SPARK: a high-level synthesis framework for applying parallelizing compiler transformations', in 16th International Conference on VLSI Design, 2003. Proceedings., Jan. 2003, pp. 461-466. doi: 10.1109/ICVD.2003.1183177.

[69] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, 'AutoPilot: A Platform-Based ESL Synthesis System', in High-Level Synthesis: From Algorithm to Digital Circuit, P. Coussy and A. Morawiec, Eds. Dordrecht: Springer Netherlands, 2008, pp. 99-112. doi: 10.1007/978-1-4020-8588-8_6.

[70] J. E. Stone, D. Gohara, and G. Shi, 'OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems', Computing in Science & Engineering, vol. 12, no. 3, pp. 66-73, May 2010, doi: 10.1109/MCSE.2010.69.

[71] OpenCL, 'Khronos OpenCL Registry - The Khronos Group Inc'. `https://www.khronos.org/registry/OpenCL/index.php` (accessed Jun. 02, 2022).

[72] Xilinx, 'Vitis Software Platform', Xilinx. `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html` (accessed Jun. 02, 2022).

[73] Xilinx, 'Vivado ML Overview', Xilinx. `https://www.xilinx.com/products/design-tools/vivado.html` (accessed Jun. 02, 2022).

[74] Intel, 'FPGA Design Software - Intel Quartus Prime', Intel. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html` (accessed Jun. 02, 2022).

[75] Intel, ‘High-Level Synthesis Compiler - Intel HLS Compiler’, Intel. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (accessed Jun. 02, 2022).

[76] Maxeler, ‘MaxCompiler — Maxeler Technologies’. https://www.maxeler.com/products/software/maxcompiler (accessed Jun. 02, 2022).

[77] BDTI, ‘The AutoESL AutoPilot High-Level Synthesis Tool’. 2010. [Online]. Available: https://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf

[78] F. Winterstein, S. Bayliss, and G. A. Constantinides, ‘High-level synthesis of dynamic data structures: A case study using Vivado HLS’, in 2013 International Conference on Field-Programmable Technology (FPT), Dec. 2013, pp. 362-365. doi: 10.1109/FPT.2013.6718388.

[79] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, ‘A Survey and Evaluation of FPGA High-Level Synthesis Tools’, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pp. 1591-1604, Oct. 2016, doi: 10.1109/TCAD.2015.2513673.

[80] M. Wijtvliet, L. Waeijen, and H. Corporaal, ‘Coarse grained reconfigurable architectures in the past 25 years: Overview and classification’, in 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), Jul. 2016, pp. 235-244. doi: 10.1109/SAMOS.2016.7818353.

[81] A. Brant and G. G. F. Lemieux, ‘ZUMA: An Open FPGA Overlay Architecture’, in 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, Apr. 2012, pp. 93-96. doi: 10.1109/FCCM.2012.25.

[82] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy, ‘DeCO: A DSP Block Based FPGA Accelerator Overlay with Low Overhead Interconnect’, in 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2016, pp. 1-8. doi: 10.1109/FCCM.2016.10.

[83] A. K. Jain, D. L. Maskell, and S. A. Fahmy, 'Throughput oriented FPGA overlays using DSP blocks', in 2016 Design, Automation Test in Europe Conference Exhibition (DATE), Mar. 2016, pp. 1628-1633.

[84] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, 'A Fully Pipelined and Dynamically Composable Architecture of CGRA', in 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, May 2014, pp. 9-16. doi: 10.1109/FCCM.2014.12.

[85] K. Ovtcharov, I. Tili, and J. G. Steffan, 'TILT: A multithreaded VLIW soft processor family', in 2013 23rd International Conference on Field programmable Logic and Applications, Sep. 2013, pp. 1-4. doi: 10.1109/FPL.2013.6645553.

[86] H. Chen, Y. Dai, R. Xue, K. Zhong, and T. Li, 'Towards Efficient Microarchitecture Design of Simultaneous Localization and Mapping in Augmented Reality Era', in 2018 IEEE 36th International Conference on Computer Design (ICCD), Oct. 2018, pp. 397-404. doi: 10.1109/ICCD.2018.00066.

[87] J. Coole and G. Stitt, 'Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing', in 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Oct. 2010, pp. 13-22. doi: 10.1145/1878961.1878966.

[88] 'FloPoCo user manual'. `http://flopoco.gforge.inria.fr/flopoco_user_manual.html` (accessed Mar. 27, 2021).

[89] M. Lange and S. M. Rump, 'Faithfully Rounded Floating-point Computations', ACM Trans. Math. Softw., vol. 46, no. 3, p. 21:1-21:20, Jul. 2020, doi: 10.1145/3290955.

[90] D. Marr, E. Hildreth, and S. Brenner, 'Theory of edge detection', Proceedings of the Royal Society of London. Series B. Biological Sciences, vol. 207, no. 1167, pp. 187-217, Feb. 1980, doi: 10.1098/rspb.1980.0020.

[91] F. Cabello, J. León, Y. Iano, and R. Arthur, 'Implementation of a fixed-point 2D Gaussian Filter for Image Processing based on FPGA', in 2015 Signal Processing: Algo-

rithms, Architectures, Arrangements, and Applications (SPA), Sep. 2015, pp. 28-33. doi: 10.1109/SPA.2015.7365108.

[92] D. G. Lowe, ‘Object recognition from local scale-invariant features’, in Proceedings of the Seventh IEEE International Conference on Computer Vision, Sep. 1999, vol. 2, pp. 1150-1157 vol.2. doi: 10.1109/ICCV.1999.790410.

[93] Xilinx, ‘LUTRAM • Versal ACAP Configurable Logic Block Architecture Manual (AM005)’. 2021. Accessed: Jul. 15, 2022. [Online]. Available: `https://docs.xilinx.com/r/en-US/am005-versal-clb/LUTRAM`

[94] M. Barbone, B. W. Kwaadgras, U. Oelfke, W. Luk, and G. Gaydadjiev, ‘Efficient Table-Based Polynomial on FPGA’, in 2021 IEEE 39th International Conference on Computer Design (ICCD), Oct. 2021, pp. 374-382. doi: 10.1109/ICCD53106.2021.00066.

[95] G. Tarawneh and A. Yakovlev, ‘An RTL method for hiding clock domain crossing latency’, in 2012 19th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2012), Dec. 2012, pp. 540-543. doi: 10.1109/ICECS.2012.6463557.

[96] A. Chakraborty and M. R. Greenstreet, ‘Efficient self-timed interfaces for crossing clock domains’, in Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings., May 2003, pp. 78-88. doi: 10.1109/ASYNC.2003.1199168.

[97] Xilinx, ‘Fast Partial Reconfiguration Over PCI Express’. 2019. [Online]. Available: `https://www.xilinx.com/support/documentation/application_notes/xapp1338-fast-partial-reconfiguration-pci-express.pdf`

[98] Xilinx, ‘UltraScale Architecture Configuration User Guide’. 2022. [Online]. Available: `https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf`

[99] V. Kizheppatt and S. A. Fahmy, ‘FPGA Dynamic and Partial Reconfiguration’, ACM Computing Surveys (CSUR), Jul. 2018, doi: 10.1145/3193827.

[100] London Stock Exchange, 'Market Data', LSEG. https://www.lseg.com/markets-products-and-services/market-information/market-data (accessed Feb. 18, 2022).

[101] M. Aquilina, E. Budish, and P. O'Neill, 'Quantifying the High-Frequency Trading "Arms Race": A Simple New Methodology and Estimates', SSRN Electronic Journal, Jan. 2020, doi: 10.2139/ssrn.3636323.

[102] G. Laughlin, A. Aguirre, and J. Grundfest, 'Information Transmission Between Financial Markets in Chicago and New York', arXiv:1302.5966 [q-fin], Feb. 2013, Accessed: Feb. 18, 2022. [Online]. Available: http://arxiv.org/abs/1302.5966

[103] M. Handley, 'Delay is Not an Option: Low Latency Routing in Space', in Proceedings of the 17th ACM Workshop on Hot Topics in Networks, New York, NY, USA, Nov. 2018, pp. 85-91. doi: 10.1145/3286062.3286075.

[104] M. Handley, 'Using ground relays for low-latency wide-area routing in megaconstellations', in Proceedings of the 18th ACM Workshop on Hot Topics in Networks, New York, NY, USA, Nov. 2019, pp. 125-132. doi: 10.1145/3365609.3365859.

[105] Cboe Global Markets, Inc. (Cboe), 'US Equities Historical Market Volume Data'. https://www.cboe.com/us/equities/market_statistics/historical_market_volume (accessed Feb. 18, 2022).

[106] P. Ramanathan and K. G. Shin, 'Delivery of time-critical messages using a multiple copy approach', ACM Trans. Comput. Syst., vol. 10, no. 2, pp. 144-166, May 1992, doi: 10.1145/128899.128902.

[107] Y. Kodama, T. Kudoh, and T. Shimizu, 'Dependable communication using multiple network paths on fast long-distance networks', Syst. Comput. Japan, vol. 38, no. 12, pp. 46-54, Nov. 2007.

[108] NASDAQ, 'NASDAQ TotalView-ITCH 4.1'. 2013. [Online]. Available: https://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTV-ITCH-V4_1.pdf

[109] OPRA, ‘OPRA Participant Interface Specification’. 2018. [Online]. Available: `https://uploads-ssl.webflow.com/5ba40927ac854d8c97bc92d7/5bf418ee4414ed1b4c397c39_opra_input_binary_part_spec.pdf`

[110] NYSE ARCA, ‘ArcaBook Client Specification’. Mar. 2022. Accessed: Jun. 18, 2022. [Online]. Available: `https://www.nyse.com/publicdocs/nyse/data/ArcaBook_Client_Specification.pdf`

[111] FIX Trading, ‘Simple Binary Encoding - Technical Specification - Final • FIX Trading Community’, FIX Trading Community, Nov. 2020. `https://www.fixtrading.org/packages/simple-binary-encoding-technical-specification-final` (accessed Feb. 18, 2022).

[112] E. Cambria, ‘Affective Computing and Sentiment Analysis’, IEEE Intelligent Systems, vol. 31, no. 2, pp. 102-107, Mar. 2016, doi: 10.1109/MIS.2016.31.

[113] K. Schouten and F. Frasincar, ‘Survey on Aspect-Level Sentiment Analysis’, IEEE Transactions on Knowledge and Data Engineering, vol. 28, no. 3, pp. 813-830, Mar. 2016, doi: 10.1109/TKDE.2015.2485209.

[114] ‘MoldUDP64 Protocol’. 2009. [Online]. Available: `http://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/moldudp64.pdf`

[115] G. W. Morris, D. B. Thomas, and W. Luk, ‘FPGA Accelerated Low-Latency Market Data Feed Processing’, in 2009 17th IEEE Symposium on High Performance Interconnects, Aug. 2009, pp. 83-89. doi: 10.1109/HOTI.2009.17.

[116] Solarflare, ‘Solarflare AOE Line Arbitration Brief’. 2013. [Online]. Available: `http://www.solarflare.com/Content/UserFiles/Documents/Solarflare_AOE_Line_Arbitration_Brief.pdf`

[117] Cisco, ‘White Paper: The Next Generation Trading Infrastructure’. 2012. Accessed: Feb. 01, 2013. [Online]. Available: `http://www.cisco.com/c/dam/en/us/products/collateral/switches/nexus-3000-series-switches/white_paper_c11-720080.pdf`

[118] C. Leber, B. Geib, and H. Litz, ‘High Frequency Trading Acceleration Using FPGAs’, in 2011 21st International Conference on Field Programmable Logic and Applications, Sep. 2011, pp. 317-322. doi: 10.1109/FPL.2011.64.

[119] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch, and V. Natoli, ‘Low-Latency FPGA Based Financial Data Feed Handler’, in 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, May 2011, pp. 93-96. doi: 10.1109/FCCM.2011.50.

[120] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers, ‘A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT)’, in 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, Aug. 2012, pp. 9-16. doi: 10.1109/HOTI.2012.15.

[121] J. Schmidhuber, ‘Deep learning in neural networks: An overview’, Neural Networks, vol. 61, pp. 85-117, Jan. 2015, doi: 10.1016/j.neunet.2014.09.003.

[122] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning. MIT Press, 2016. [Online]. Available: `http://www.deeplearningbook.org`

[123] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, ‘Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling’, Dec. 2014. doi: 10.48550/arXiv.1412.3555.

[124] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, ‘The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0’. May 06, 2014. Accessed: Jun. 24, 2022. [Online]. Available: `https://apps.dtic.mil/sti/pdfs/ADA605735.pdf`

[125] NVIDIA, ‘Parallel Thread Execution ISA’. May 11, 2022. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/ptx_isa_7.7.pdf`

[126] Intel, ‘Intel Stratix 10 Variable Precision DSP Blocks User Guide’. [Online]. Available: `https://cdrdv2.intel.com/v1/dl/getContent/666579?fileName=ug-s10-dsp-683832-666579.pdf`

[127] Szandała, Tomasz. 'Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks'. ArXiv:2010.09458 [Cs] 903 (2021). https://doi.org/10.1007/978-981-15-5495-7.

[128] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, 'Activation Functions: Comparison of trends in Practice and Research for Deep Learning', arXiv:1811.03378 [cs], Nov. 2018, Accessed: Mar. 01, 2021. [Online]. Available: http://arxiv.org/abs/1811.03378

[129] Bengio, Y., P. Simard, and P. Frasconi. 'Learning Long-Term Dependencies with Gradient Descent Is Difficult'. IEEE Transactions on Neural Networks 5, no. 2 (March 1994): 157-66. https://doi.org/10.1109/72.279181.

[130] Hochreiter, Sepp, and Jürgen Schmidhuber. 'Long Short-Term Memory'. Neural Computation 9 (1 December 1997): 1735-80. https://doi.org/10.1162/neco.1997.9.8.1735.

[131] Cho, Kyunghyun, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 'Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation'. ArXiv:1406.1078 [Cs, Stat], 2 September 2014. http://arxiv.org/abs/1406.1078.

[132] Ramachandran, Prajit, Barret Zoph, and Quoc V. Le. 'Searching for Activation Functions'. ArXiv:1710.05941 [Cs], 27 October 2017. http://arxiv.org/abs/1710.05941.

[133] Nair, Vinod, and Geoffrey E. Hinton. 'Rectified Linear Units Improve Restricted Boltzmann Machines'. In Proceedings of the 27th International Conference on International Conference on Machine Learning, 807-14. ICML'10. Madison, WI, USA: Omnipress, 2010.

[134] L. Lu, 'Dying ReLU and initialization: Theory and numerical examples', Communications in Computational Physics, vol. 28, no. 5, pp. 1671-1706, Jun. 2020, doi: 10.4208/cicp.oa-2020-0165.

[135] Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. 'Rectifier Nonlinearities Improve Neural Network Acoustic Models'. In In ICML Workshop on Deep Learning for Audio, Speech and Language Processing, 2013.

[136] Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter. 'Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)'. ArXiv:1511.07289 [Cs], 22 February 2016. `http://arxiv.org/abs/1511.07289`.

[137] Dugas, Charles, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. 'Incorporating Second-Order Functional Knowledge for Better Option Pricing'. In Proceedings of the 13th International Conference on Neural Information Processing Systems, 451-57. NIPS'00. Cambridge, MA, USA: MIT Press, 2000.

[138] Hendrycks, Dan, and Kevin Gimpel. 'Gaussian Error Linear Units (GELUs)'. ArXiv:1606.08415 [Cs], 8 July 2020. `http://arxiv.org/abs/1606.08415`.

[139] M. I. Jordan and T. M. Mitchell, 'Machine learning: Trends, perspectives, and prospects', Science, vol. 349, no. 6245, pp. 255-260, Jul. 2015, doi: 10.1126/science.aaa8415.

[140] S. Z. Gilani and A. Mian, 'Learning from Millions of 3D Scans for Large-scale 3D Face Recognition', in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Jun. 2018, pp. 1896-1905. doi: 10.1109/CVPR.2018.00203.

[141] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, 'Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC', in 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Aug. 2016, pp. 1-4. doi: 10.1109/FPL.2016.7577314.

[142] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, 'Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC', in 2016 International Conference on Field-Programmable Technology (FPT), Dec. 2016, pp. 77-84. doi: 10.1109/FPT.2016.7929192.

[143] T. Gale, E. Elsen, and S. Hooker, 'The State of Sparsity in Deep Neural Networks'. arXiv, Feb. 25, 2019. doi: 10.48550/arXiv.1902.09574.

[144] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun,

A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, ‘Machine Learning at Facebook: Understanding Inference at the Edge’, in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Feb. 2019, pp. 331-344. doi: 10.1109/HPCA.2019.00048.

[145] D. Pasetto, K. Lynch, R. Tucker, B. Maguire, F. Petrini, and H. Franke, ‘Ultra low latency market data feed on IBM PowerENTM’, Comput. Sci., vol. 26, no. 3-4, pp. 307-315, Jun. 2011, doi: 10.1007/s00450-011-0166-0.

[146] Xilinx, ‘CORDIC v6.0 LogiCORE IP Product Guide’, 2021. `https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf`

[147] Xilinx, ‘Floating-Point Operator v7.1 LogiCORE IP Product Guide’, 2020. `https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf` (accessed Mar. 01, 2021).

[148] M. Rawski, ‘Modified Distributed Arithmetic Concept for Implementations Targeted at Heterogeneous FPGAs’, International Journal of Electronics and Telecommunications, vol. 56, pp. 345-350, 2010, doi: 10.2478/v10177-010-0045-9.

[149] S. Xu, S. A. Fahmy, and I. V. McLoughlin, ‘Square-rich fixed point polynomial evaluation on FPGAs’, in Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays, New York, NY, USA, Feb. 2014, pp. 99-108, doi: 10.1145/2554688.2554779.

[150] Xilinx, ‘Performance and Resource Utilization for Floating-point v7.1’. Accessed: Feb. 22, 2022. [Online]. Available: `https://www.xilinx.com/html_docs/ip_docs/pru_files/floating-point.html`

[151] Gupta, Suyog, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. ‘Deep Learning with Limited Numerical Precision’. In Proceedings of the 32nd International Conference on Machine Learning, edited by Francis Bach and David Blei, 37:1737-46. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015. `http://proceedings.mlr.press/v37/gupta15.html`.

[152] F. de Dinechin, M. Istoan, and G. Sergent, 'Fixed-point trigonometric functions on FPGAs', SIGARCH Comput. Archit. News, vol. 41, no. 5, pp. 83-88, Jun. 2014, doi: 10.1145/2641361.2641375.

[153] Z. Wang, H. Huang, J. Zhang, and G. Alonso, 'Shuhai: Benchmarking High Bandwidth Memory On FPGAS', in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2020, pp. 111-119. doi: 10.1109/FCCM48280.2020.00024.

[154] Xilinx, 'AXI High Bandwidth Memory Controller v1.0 LogiCORE IP Product Guide'. 2021. [Online]. Available: `https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf`

[155] Intel, 'FPGA Product Catalog', 2021, Accessed: Mar. 01, 2021. [Online]. Available: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/sg/product-catalog.pdf`

[156] Intel, 'Cyclone 10 Device Overview'. Accessed: Mar. 01, 2021. [Online]. Available: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/cyclone-10-lp-product-table.pdf`

[157] C. Clos, 'A study of non-blocking switching networks', The Bell System Technical Journal, vol. 32, no. 2, pp. 406-424, Mar. 1953, doi: 10.1002/j.1538-7305.1953.tb01433.x.

[158] V. E. Beneš, 'Permutation Groups, Complexes, and Rearrangeable Connecting Networks', Bell System Technical Journal, vol. 43, no. 4, pp. 1619-1640, 1964, doi: 10.1002/j.1538-7305.1964.tb04102.x.

[159] D. H. Lawrie, 'Access and Alignment of Data in an Array Processor', IEEE Transactions on Computers, vol. C-24, no. 12, pp. 1145-1155, Dec. 1975, doi: 10.1109/T-C.1975.224157.

[160] W. J. Dally and B. P. Towles, Principles and Practices of Interconnection Networks. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[161] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, 'Array programming with NumPy', Nature, vol. 585, no. 7825, pp. 357-362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.

[162] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, 'Scikit-learn: Machine learning in Python', Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.

[163] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, 'SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python', Nature Methods, vol. 17, pp. 261-272, 2020, doi: 10.1038/s41592-019-0686-2.

[164] S. Gomar, M. Mirhassani, and M. Ahmadi, 'Precise digital implementations of hyperbolic tanh and sigmoid function', in 2016 50th Asilomar Conference on Signals, Systems and Computers, 2016, pp. 1586-1589, doi: 10.1109/ACSSC.2016.7869646.

[165] A. M. Abdelsalam, J. M. P. Langlois, and F. Cheriet, 'A Configurable FPGA Implementation of the Tanh Function Using DCT Interpolation', in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Apr. 2017, pp. 168-171, doi: 10.1109/FCCM.2017.12.

[166] 'IEEE Standard for Floating-Point Arithmetic', IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1-84, Jul. 2019, doi: 10.1109/IEEESTD.2019.8766229.

[167] M. Langhammer, B. Pasca, G. Baeckler, and S. Gribok, 'Extracting INT8 Multipliers from INT18 Multipliers', in 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Sep. 2019, pp. 114-120. doi: 10.1109/FPL.2019.00027.

[168] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang, and J. Yan, 'Towards Unified INT8 Training for Convolutional Neural Network', in 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2020, pp. 1966-1976. doi: 10.1109/CVPR42600.2020.00204.

[169] T. Chu, Q. Luo, J. Yang, and X. Huang, 'Mixed-precision quantized neural networks with progressively decreasing bitwidth', Pattern Recognition, vol. 111, 2021, doi: `https://doi.org/10.1016/j.patcog.2020.107647`.

[170] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, 'Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools', International Journal of Reconfigurable Computing, vol. 2013, pp. 1-10, Dec. 2013, doi: 10.1155/2013/428078.

[171] Xilinx, 'Performance and Resource Utilization for CORDIC v6.0'. `https://www.xilinx.com/html_docs/ip_docs/pru_files/cordic.html` (accessed May. 28, 2021).

[172] A. P. Renardy, N. Ahmadi, A. A. Fadila, N. Shidqi, and T. Adiono, 'FPGA implementation of CORDIC algorithms for sine and cosine generator', in 2015 International Conference on Electrical Engineering and Informatics (ICEEI), Aug. 2015, pp. 1-6. doi: 10.1109/ICEEI.2015.7352460.

[173] F. D. Dinechin and M. Istoan, 'Hardware Implementations of Fixed-Point Atan2', in 2015 IEEE 22nd Symposium on Computer Arithmetic, Jun. 2015, pp. 34-41, doi: 10.1109/ARITH.2015.23.

[174] F. de Dinechin, 'Reflections on 10 Years of FloPoCo', in 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), Jun. 2019, pp. 187-189. doi: 10.1109/ARITH.2019.00042.

[175] T. Wiersema, A. Bockhorn, and M. Platzner, 'Embedding FPGA overlays into configurable Systems-on-Chip: ReconOS meets ZUMA', in 2014 International Conference on Re-

ConFigurable Computing and FPGAs (ReConFig14), Dec. 2014, pp. 1-6. doi: 10.1109/ReConFig.2014.7032514.

[176] Ahmad, S., S. Subramanian, V. Boppana, S. Lakka, F. Ho, T. Knopp, J. Noguera, G. Singh, and R. Wittig. 'Xilinx First 7nm Device: Versal AI Core (VC1902)'. In 2019 IEEE Hot Chips 31 Symposium (HCS), 1-28, 2019. `https://doi.org/10.1109/HOTCHIPS.2019.8875639`.

[177] J. Rose, R. J. Francis, D. Lewis, and P. Chow, 'Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency', IEEE Journal of Solid-State Circuits, vol. 25, no. 5, pp. 1217-1225, Oct. 1990, doi: 10.1109/4.62145.

[178] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, 'Packet Switched vs. Time Multiplexed FPGA Overlay Networks', in 2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Apr. 2006, pp. 205-216. doi: 10.1109/FCCM.2006.55.

[179] C. H. van Berkel and T. van Roermund, 'Scalable Multi-Input-Multi-Output Queues With Application to Variation-Tolerant Architectures', IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17, no. 7, pp. 920-923, Jul. 2009, doi: 10.1109/TVLSI.2009.2012929.

[180] A. K. Jain, S. A. Fahmy, and D. L. Maskell, 'Efficient Overlay Architecture Based on DSP Blocks', in 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, May 2015, pp. 25-28. doi: 10.1109/FCCM.2015.15.

[181] Xilinx, 'Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics'. 2020. [Online]. Available: `https://www.xilinx.com/support/documentation/data_sheets/ds892-kintex-ultrascale-data-sheet.pdf`

[182] Organization for the Advancement of Structured Information Standards (OASIS), 'MQTT 5 Specification'. Mar. 2019. Accessed: Jun. 13, 2022. [Online]. Available: `https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf`

[183] J. Postel, ‘User Datagram Protocol’, Internet Engineering Task Force, Request for Comments RFC 768, Aug. 1980. doi: 10.17487/RFC0768.

[184] Xilinx. ‘MicroBlaze Processor Reference Guide’, 2021. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf`.

[185] Intel. ‘Nios V Processor Reference Manual’, 2021. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug20343.pdf`.

[186] Xilinx, ‘Alveo Product Selection Guide’, 2021. `https://www.xilinx.com/support/documentation/selection-guides/alveo-product-selection-guide.pdf` (accessed Aug. 31, 2021).

[187] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, ‘CoMeFa: Compute-in-Memory Blocks for FPGAs’. arXiv, Mar. 23, 2022. doi: 10.48550/arXiv.2203.12521.

[188] G. Elber, I.-K. Lee, and M.-S. Kim, ‘Comparing offset curve approximation methods’, IEEE Computer Graphics and Applications, vol. 17, no. 3, pp. 62-71, May 1997, doi: 10.1109/38.586019.

[189] ‘Central Limit Theorem’, in The Concise Encyclopedia of Statistics, New York, NY: Springer New York, 2008, pp. 66-68. doi: 10.1007/978-0-387-32833-1_50.

[190] A. Podobas, K. Sano, and S. Matsuoka, ‘A Template-based Framework for Exploring Coarse-Grained Reconfigurable Architectures’, in 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP), Jul. 2020, pp. 1-8. doi: 10.1109/ASAP49362.2020.00010.

[191] D. Capalija and T. S. Abdelrahman, ‘A high-performance overlay architecture for pipelined execution of data flow graphs’, in 2013 23rd International Conference on Field programmable Logic and Applications, Sep. 2013, pp. 1-8. doi: 10.1109/FPL.2013.6645515.

[192] R. Rashid, J. G. Steffan, and V. Betz, 'Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS', in 2014 International Conference on Field-Programmable Technology (FPT), Dec. 2014, pp. 20-27. doi: 10.1109/FPT.2014.7082748.

[193] Xilinx, 'Versal ACAP AI Core Series Product Selection Guide'. 2021. [Online]. Available: `https://www.xilinx.com/support/documentation/selection-guides/versal-ai-core-product-selection-guide.pdf`

[194] A. Podobas, K. Sano, and S. Matsuoka, 'A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective', IEEE Access, vol. 8, pp. 146719-146743, 2020, doi: 10.1109/ACCESS.2020.3012084.

[195] G. D. Knott, Interpolating Cubic Splines. Boston, MA: Birkhäuser, 2000. Accessed: Aug. 31, 2021. [Online]. Available: `https://doi.org/10.1007/978-1-4612-1320-8`

[196] T. Banchoff and J. Wermer, 'Least Squares Approximation', in Linear Algebra Through Geometry, T. Banchoff and J. Wermer, Eds. New York, NY: Springer US, 1992, pp. 291-295. doi: 10.1007/978-1-4612-4390-8_34.

[197] G. H. Golub and L. B. Smith, 'Algorithm 414: Chebyshev Approximation of Continuous Functions by a Chebyshev System of Functions', Commun. ACM, vol. 14, no. 11, pp. 737-746, Nov. 1971, doi: 10.1145/362854.362890.

[198] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, 'Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions', IEEE Access, vol. 6, pp. 47980-48009, 2018, doi: 10.1109/ACCESS.2018.2866491.

[199] P. D. Francesco, I. Malavolta, and P. Lago, 'Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption', in 2017 IEEE International Conference on Software Architecture (ICSA), Apr. 2017, pp. 21-30. doi: 10.1109/ICSA.2017.24.

[200] T. Nguyen and A. Zakhor, 'Distributed video streaming with forward error correction', in Packet video workshop, 2002, vol. 2002.

[201] C. Wang, Q. Wang, K. Ren, and W. Lou, 'Ensuring data storage security in Cloud Computing', in 2009 17th International Workshop on Quality of Service, Jul. 2009, pp. 1-9. doi: 10.1109/IWQoS.2009.5201385.

[202] Xilinx, 'Alveo U280 Data Center Accelerator Card Data Sheet'. 2020. Accessed: Mar. 01, 2021. [Online]. Available: `https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf`

[203] C. Lameter, 'NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors.', Queue, vol. 11, no. 7, pp. 40-51, Jul. 2013, doi: 10.1145/2508834.2513149.

[204] F. Winterstein, K. Fleming, H.-J. Yang, J. Wickerson, and G. Constantinides, 'Custom-sized caches in application-specific memory hierarchies', in 2015 International Conference on Field Programmable Technology (FPT), Dec. 2015, pp. 144-151. doi: 10.1109/FPT.2015.7393141.

[205] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, 'A Survey of Performance Modeling and Simulation Techniques for Accelerator-Based Computing', IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 1, pp. 272-281, Jan. 2015, doi: 10.1109/TPDS.2014.2308216.

[206] F.-X. Standaert, 'Introduction to Side-Channel Attacks', in Secure Integrated Circuits and Systems, I. M. R. Verbauwhede, Ed. Boston, MA: Springer US, 2010, pp. 27-42. doi: 10.1007/978-0-387-71829-3_2.