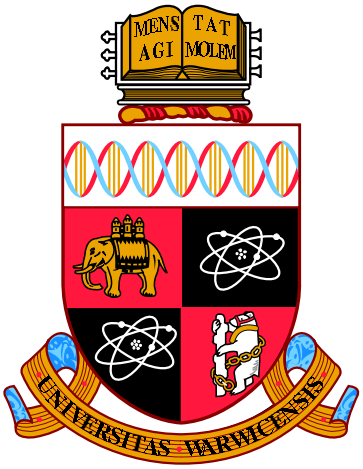


Modelling and Characterisation of Distributed Hardware Acceleration

by

Ryan A. Cooke

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

School of Engineering

July 2020

Contents

List of Tables vi

List of Figures vii

Acknowledgments ix

Declarations x

Abstract xi

Abbreviations xii

Chapter 1 Introduction 1

1.1 Motivations . . . 3
1.2 Objectives . . . 4
1.3 Contributions . . . 5
1.4 Thesis Roadmap . . . 5
1.5 Publications . . . 6

Chapter 2 Background and Literature Review 8

2.1 Computing Platforms . . . 9
2.1.1 Central Processing Units . . . 9
2.1.2 Field Programmable Gate Arrays . . . 11
2.1.3 FPGA Accelerator Design . . . 16
2.1.4 Graphics Processing Units . . . 21
2.1.5 Application Specific Integrated Circuits (ASICs) . . . 23
2.1.6 Summary . . . 24
2.2 Accelerator Integration . . . 24
2.2.1 PCIe . . . 24
2.2.2 Network interface . . . 26

2.2.3 Tightly coupled SoC . . . 27
2.2.4 Interface overheads . . . 28
2.3 Summary . . . 28
2.4 Networked computing systems . . . 29
2.4.1 Edge Networks . . . 29
2.4.2 Datacentre Networks . . . 30
2.4.3 In-network computing . . . 32
2.4.4 Network elements . . . 33
2.4.5 FPGAs for network applications . . . 34
2.4.6 Cloud Computing . . . 37
2.4.7 FPGAs in the cloud . . . 38
2.5 Mathematical Modelling . . . 42
2.5.1 Sensor network modelling . . . 42
2.5.2 Distributed Stream Processing Models . . . 42
2.5.3 Summary . . . 43

Chapter 3 Modelling distributed computing with heterogeneous hardware 45

3.1 Introduction . . . 45
3.2 Contributions . . . 48
3.3 Related Work . . . 48
3.3.1 Edge/Fog Computing . . . 49
3.3.2 Hardware acceleration . . . 49
3.4 Scenario and Metrics . . . 50
3.4.1 Latency . . . 50
3.4.2 Bandwidth . . . 50
3.4.3 Energy . . . 51
3.4.4 Financial Cost . . . 51
3.5 Proposed Model . . . 51
3.5.1 Tasks . . . 53
3.5.2 Implementations . . . 54
3.5.3 Platforms . . . 54
3.5.4 Network . . . 55
3.5.5 Sources and Data . . . 55
3.5.6 Allocation Variables . . . 56
3.5.7 Constraints . . . 56
3.6 Performance Metrics . . . 57

3.6.1 End-to-End Latency . . . 58
3.6.2 Throughput . . . 61
3.6.3 Data-rate . . . 62
3.6.4 Energy Consumption . . . 63
3.6.5 Financial Cost . . . 64
3.6.6 Combined Evaluation Metrics . . . 64
3.7 Case Study . . . 65
3.7.1 Network . . . 65
3.7.2 Tasks . . . 66
3.7.3 Platforms . . . 66
3.7.4 Centralised Software . . . 68
3.7.5 In-network software . . . 69
3.7.6 Centralised Hardware . . . 69
3.7.7 In-network hardware . . . 69
3.7.8 Optimal Placement . . . 70
3.7.9 Summary . . . 71
3.7.10 Event Driven Simulation . . . 71
3.8 Further Analysis . . . 74
3.8.1 Relative Computing Capability . . . 74
3.8.2 Task Data Reduction . . . 77
3.8.3 Network Structure . . . 78
3.8.4 Hardware Acceleration . . . 80
3.9 Generating In-Network Task and Hardware Placement with Heterogeneous Hardware . . . 80
3.9.1 Objective function formulation . . . 80
3.9.2 Case Study . . . 81
3.9.3 Evaluation with Synthetic Networks . . . 84
3.9.4 Summary . . . 91
3.10 Summary . . . 91

Chapter 4 Quantifying the Latency Overheads of FPGA Accelerators 92
4.1 Introduction . . . 92
4.2 Contributions . . . 94
4.3 Related Work . . . 94
4.4 Experiments . . . 95
4.4.1 Latency . . . 95
4.4.2 Throughput . . . 96

4.4.3 Platforms . . . 96
4.5 Results . . . 99
4.5.1 Median Latency . . . 99
4.5.2 FPGA Latency Breakdown . . . 100
4.5.3 Latency Distributions . . . 101
4.5.4 Tail Latencies . . . 101
4.5.5 Packet Size . . . 102
4.5.6 Throughput . . . 103
4.6 Discussion . . . 104
4.6.1 PCIe Accelerators . . . 104
4.6.2 Network-attached Accelerators . . . 105
4.7 Summary . . . 106

Chapter 5 Near-Edge FPGA Acceleration for the Internet of Things 107
5.1 Introduction . . . 107
5.2 Contributions . . . 109
5.3 Related Work . . . 109
5.4 Design and Experiments . . . 110
5.4.1 Application . . . 110
5.4.2 Measurements . . . 111
5.4.3 Platforms . . . 112
5.5 Results . . . 118
5.5.1 Isolated Edge Node Measurements . . . 118
5.5.2 Impact of Multiple Edge Devices . . . 121
5.5.3 Discussion . . . 123
5.6 Accelerator Location . . . 124
5.6.1 Results . . . 126
5.6.2 Discussion . . . 129
5.7 Summary . . . 131

Chapter 6 Conclusions and Future Work 132
6.1 Summary of Contributions . . . 132
6.1.1 Mathematical representation of in-network and near edge computing . . . 132
6.1.2 Optimising hardware and task placement . . . 133
6.1.3 Quantifying costs associated with FPGA accelerators . . . 133
6.1.4 Demonstration of in-network FPGA acceleration . . . 133
6.2 Future Research . . . 133

6.2.1 Practically validating the model 134
6.2.2 Improving optimisation runtime 134
6.2.3 Developing generalised in-network FPGA infrastructure . . . 134
6.2.4 Combination of model and FPGA infrastructure 135
6.2.5 Defining accelerator communication protocols 135
6.3 Summary 135

List of Tables

3.1 Summary of symbols used in formulation. 57
3.2 Case study task values 67
3.3 Case study platform values 68
3.4 Case study SW results 68
3.5 Case study HW results 70
3.6 Performance metrics for MILP optimisation of model 71
3.7 Different placement policies used in simulations 75
3.8 Summary of task parameter values. 82
3.9 Summary of available platforms. 83

4.1 Latency results 99
4.2 Network-attached FPGA delays 100

5.1 Experimental results 119
5.2 Network traversal times 125

List of Figures

2.1 Representative modern FPGA architecture 11
2.2 Xilinx 7 Series CLB arrangement 12
2.3 Xilinx DSP48E1 architecture 12
2.4 Partial reconfiguration example 16
2.5 Example network infrastructure 29
2.6 Example datacentre network 31

3.1 Example networked system 46
3.2 Network node abstraction 52
3.3 Difference between software and hardware nodes 60
3.4 Case study network structure 66
3.5 Model vs simulation results 72
3.6 Compute capability results 76
3.7 Reduction factor results 77
3.8 Network fanout results 79
3.9 Naive vs. model placement 84
3.10 Configurations generated by optimization 85
3.11 Optimal cost with latency constraint 86
3.12 Sythnthetic networks optimised for cost with latency constraint . . . 86
3.13 Synthetic networks optimised for energy and latency with cost constraint 87
3.14 Optimal vs pushed for varying task centralisation 89
3.15 Optimal latency and energy vs pushed down 90
3.16 Bandwidth consumtion for varying centralisation of tasks 90

4.1 Outline of the experimental setup 95
4.2 Accelerator configurations 97
4.3 Tail latency results 100
4.4 CDF of latencies 101

4.5 Packet size differences 102
4.6 Throughput results 104
4.7 Packet rate results 105

5.1 Experimental testbed 110
5.2 Zynq accelerated edge node 113
5.3 FPGA network switch accelerator 116
5.4 Experimental results 119
5.5 Multiple edge nodes offloading to cloudlet 122
5.6 Multiple edge nodes offloading to networked FPGA 122
5.7 Analysed network structure 124
5.8 Estimation results 127
5.9 Network-attacehd latency redcution 128
5.10 Effects of 200ms base computation time 129
5.11 Effect of 100ms base computation time 130

Acknowledgments

I would like to thank my supervisor Suhaib Fahmy for giving me this opportunity. His guidance has been invaluable, and I am extremely grateful for all of the time he spent helping me.

I would also like to thank my friends Lenos and Alex, also part of the Connected Systems group, who at this point are still completing their PhDs. Their advice and company is much appreciated, and helped get through the day to day in the lab.

Lastly, I thank my partner Eleanor, whom I met for the first time shortly after beginning the PhD. Her patience, love, and kindness have kept me motivated and I wouldn't have been able to complete this thesis without her.

Declarations

This thesis is submitted to the University of Warwick for the degree of Doctor of Philosophy. The work contained in this thesis comprises my own work. This thesis has not previously been submitted for a degree at another university.

Parts of this thesis have been published by the author:

1. Ryan A. Cooke, Suhaib A. Fahmy, *In-network online data analytics with FPGAs*, in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2017 [1].

2. Ryan A. Cooke, Suhaib A. Fahmy, *A model for distributed in-network and near-edge computing with heterogeneous hardware*, in Future Generation Computer Systems (FGCS), vol. 105, 2020 [2].

3. Ryan A. Cooke, Suhaib A. Fahmy, *Quantifying the latency benefits of near-edge and in-network FPGA acceleration*, in Proceedings of the International Workshop on Edge Systems, Analytics and Networking (EdgeSys), 2020 [3].

4. Ryan A. Cooke, Suhaib A. Fahmy, *Characterizing latency overheads in the deployment of FPGA accelerators*, in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2020 [4].

5. Ryan A. Cooke, Suhaib A. Fahmy, *Exploring Hardware Accelerator Offload for the Internet of Things*, submitted to: it - Information Technology

Abstract

Hardware acceleration has become more commonly utilised in networked computing systems. The growing complexity of applications mean that traditional CPU architectures can no longer meet stringent latency constraints. Alternative computing architectures such as GPUs and FPGAs are increasingly available, along with simpler, more software-like development flows. The work presented in this thesis characterises the overheads associated with these accelerator architectures. A holistic view encompassing both computation and communication latency must be considered. Experimental results obtained through this work show that network-attached accelerators scale better than server-hosted deployments, and that host ingestion overheads are comparable to network traversal times in some cases. Along with the choice of processing platforms, it is becoming more important to consider how workloads are partitioned and where in the network tasks are being performed. Manual allocation and evaluation of tasks to network nodes does not scale with network and workload complexity. A mathematical formulation of this problem is presented within this thesis that takes into account all relevant performance metrics. Unlike other works, this model takes into account growing hardware heterogeneity and workload complexity, and is generalisable to a range of scenarios. This model can be used in an optimisation that generates lower cost results with latency performance close to theoretical maximums compared to naive placement approaches. With the mathematical formulation and experimental results that characterise hardware accelerator overheads, the work presented in this thesis can be used to make informed design decisions about both where to allocate tasks and deploy accelerators in the network, and the associated costs.

Abbreviations

API Application Programming Interface

AR Augmented Reality

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

BRAM Block Random Access Memory

CLB Configurable Logic Block

CNN Convolutional Neural Network

CPU Central Processing Unit

DMA Direct Memory

DRAM Dynamic Random Access Memory

DSP Digital Signal Processing

DNN Deep Neural Network

FIFO First In First Out

FFT Fast Fourier Transform

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High Level Synthesis

IC Integrated Circuit

IoT Internet of Things

ISP Internet Service Provider

IP Internet Protocol

KNN K-Nearest Neighbour

LAN Local Area Network

LTE Long Term Evolution

LUT Look Up Table

MAC Media Access Control

MILP Mixed Integer Linear Programming

NFV Network Function Virtualisation

NIC Network Interface Card

NN Neural Network

NoC Network on a Chip

NPU Network Processing Unit

OFDM Orthogonal Frequency Division Multiplexing

PCIe Peripheral Component Interconnect Express

PL Programmable Logic

PLL Phase Locked Loop

PR Partial Reconfiguration

PS Processing System

RAM Random Access Memory

SDN Software Defined Networking

SoC System on a Chip

SRAM Static Random Access Memory

SVM Support Vector Machine

TCP Transmission Control Protocol

ToR Top of Rack

TPU Tensor Processing Unit

UDP User Datagram Protocol

VM Virtual Machine

WAN Wide Area Network

Chapter 1

Introduction

Computational offloading is a term used to describe a scenario where one computing system transmits data to another, where computation is carried out, and the result transmitted back. The total computation latency of a task, the power consumed by the hardware, and the processing throughput can all be improved using this approach, when the target platform is more capable.

It can be used to describe a range of scenarios. Early implementations of this technique involved central processing units (CPUs) offloading to a co-processor such as separate floating point units, which were optimised for floating point arithmetic [5]. Computation latency could be drastically reduced, and in some cases the main processor would be free to carry out other tasks while waiting for the result. As processing tasks became more complex with the growth of applications such as graphics rendering, co-processors could begin to take the form of separate chips on the same board, or as chips hosted on a separate expansion board, connected through external interfaces such as peripheral component interconnect (PCI) [6]. Modern system-on-chip (SoC) architectures comprise multiple different hardware cores to accelerate particular tasks such as encryption, digital signal processing, or graphics rendering [7].

The growth of the Internet and networking technologies presented opportunities for computational offload to distinct machines over a network [8; 9; 10; 11; 12; 13]. As network bandwidth improved, it became more viable to transmit significant amounts of data to a remote computing resource. The platform carrying out the computation no longer had to be restricted to the same location as the data source. The arrival of data centre cloud computing allowed for computing resources to be centralised and shared across many clients [14]. There are numerous benefits to this approach. Computing resources can be scaled with much less friction and

hardware can be modified to enhance processing capability without disturbing the data source. This is particularly valuable in applications where the data source is in a hard-to-reach location.

Transmitting data to another processing platform has an associated communication cost, however [15]. The improvement to performance, whatever the metric may be, offsets this cost. Recently, there has been a surge of interest in low-latency, 'real-time' applications, driven by factors such as the internet of things (IoT) and a growing demand for responsive, complex web applications [15; 16; 17]. Industrial IoT systems rely on networked sensors and data acquisition systems, and have strict latency requirements on closed-loop control. Mobile applications requiring complex image processing such as augmented reality (AR) have to meet latency targets to provide an acceptable user experience [16]. Smart vehicles must process data and communicate with other vehicles in real-time to satisfy safety constraints [18].

This presents a problem – processing may be too complex to perform at the data source in the required time interval, so must be offloaded. While transmitting data to the cloud can reduce the computation latency, the communication penalty may be too great, and thus the total latency would still violate the required constraints. This has led to the paradigm of 'edge computing', where computation is offloaded to resources closer to the data source. This could mean improved hardware at the data source itself - but this results in difficult management of hardware and increased cost. Alternatively, smaller data centres or servers could be placed on local networks with the data source, removing the need for data to be transmitted over the Internet. Processing could even be moved into the networking elements that facilitate the transfer of data between machines, resulting in even less communication time. This is known as 'in-network' computing [19]. The resultant scenario is one where there are a range of processing elements in different locations, with different capabilities and constraints, that all offer a target for computational offload. There is now an opportunity for improved processing times, with lower communication costs than when using computation entirely at the data source, or entirely in the cloud. A new set of key challenges must now be addressed to best make use of these systems.

One of these challenges is where in the network to offload tasks. An application may comprise many tasks, each with different requirements and dependencies on other tasks, and must be allocated across a set of heterogeneous offload platforms with varying computational properties. This is a complex decision, where the profile of the application and the available resources must be accounted for. As these networked systems increase in scale, this becomes even more challenging. Systems

can become large enough that manual evaluation of task placement can become impossible. Design decisions must be made to balance opposing performance metrics, and to make the best use of finite resources.

Another challenge is ensuring that the in-network or edge computing resources can provide adequate computational performance. Despite being closer to the data sources, this hardware must still reduce computation latency enough to justify offloading to them. Hardware must be able to be shared across multiple client devices, and be able to perform complex processing quickly. In the case of in-network computing, networking elements are being extended to perform additional processing, so hardware that causes minimal disruption of the usual networking functions must therefore be used. The integration between the network flow and the computations datapaths can also influence latency significantly.

Hardware acceleration is necessary in order to fulfil these requirements. Devices called field programmable gate arrays (FPGAs) are particularly well suited to this. They comprise of an array of simple logic blocks and other computational resources, linked through a configurable interconnect, which can be used to implement accelerator architectures optimised to perform a particular task, providing a computational benefit compared to standard CPU architectures. Being reconfigurable means they can support a range of applications and be dynamically configurable at runtime. They are particularly useful for in-network computing as some network elements such as switches, base stations, and routers already utilise them to carry out networking tasks, and are good at packet processing. FPGAs offer a greater range of deployment and integration capabilities compared to other hardware accelerators such as GPUs [20], and are a key enabler in meeting performance requirements at the network edge as applications scale. In Chapter 2, the differences between accelerator platforms are discussed in more detail.

1.1 Motivations

Computing is becoming increasingly connected. Networked computing systems are becoming more complex, larger in scale, and operate under greater performance constraints. Various factors, such as the growth of the IoT, commercial cloud computing and developments in hardware have been significant drivers of this.

A key challenge in effective use of networked systems for computational offload of tasks is the partitioning and placement of sub-tasks amongst computational nodes. Where computation is carried out is becoming more important. Ad-hoc placement of tasks may have been viable for smaller systems, but as the number

of variables grows, finding solutions that meet required constraints is not viable. Not only are there systems with more connected devices, the devices within the system are becoming more capable. Network elements that were historically passive and simply passed data between nodes, can now be tightly coupled with compute, increasing the number of possible placement solutions. Additionally, edge nodes such as sensors and other data sources which typically just produced data, are also becoming more capable due to the greater availability of cheap microcontrollers and single board computers, such as the Raspberry Pi family of devices [21]. These platforms can handle data acquisition, networking, and computation, and have a small enough form factor that they can be deployed in a range of scenarios. Increased hardware heterogeneity, and the growing market for hardware accelerators such as FPGAs add even more variables to networked system deployment. Taking this into consideration, it becomes clear that a holistic view accounting for modern trends and advancements, must be used when making decisions regarding hardware deployment and task placement. There are such an abundance of options and implications that any ad-hoc or naive decision making is likely to lead to sub-optimal performance.

As more heterogeneous computing platforms become available for lower costs, and existing platforms improve their capabilities, the decision on what combinations of hardware to use becomes more important. FPGA acceleration provides an opportunity to increase processing capability at selected nodes, and unlike other accelerator platforms, allow for a greater flexibility in how they are integrated into the network. Unlike alternative accelerator platforms such as GPUs, FPGAs need no CPU host, and can be connected directly to the network [22].

A greater understanding of the implications of using this approach is needed. The costs and benefits of utilising these accelerators must be quantified, and generalisations for types of applications made. Understanding of the trade-offs between computation and communication is vital to making informed decisions regarding accelerator deployment.

1.2 Objectives

The objectives of this research are as follows:

1. To develop a methodology for exploring and evaluating heterogeneous networked systems used to offload complex applications, and to quantify key performance metrics.

2. To quantify the costs associated with the deployment of reconfigurable accelerators in alternative offload configurations.

3. To characterize the performance of emerging application types with different compute placement strategies, taking into account the variation in computing capability and connectivity of heterogeneous platforms.

1.3 Contributions

The main contributions of this work are a set of tools, measurements, and investigations that can be used to evaluate the placement of offloaded tasks across a network of heterogeneous computing and networking elements. These can be summarised as follows:

1. A mathematical formulation of this scenario, which allows for the detailed description of a heterogeneous network, tasks, and accelerator hardware. The model, unlike other related works, is generalised to account for heterogeneous hardware accelerators, complex task structures, and is easily extensible to model a range of scenarios, capturing all the important metrics of interest.

2. Derived from this model, generalised insights into how computation should be offloaded across networked devices depending on various application and network characteristics.

3. This model can be used within a mixed integer linear programming optimisation to generate hardware and task allocations to meet various performance objectives typically used when considering computational offload.

4. Experiments used to determine inherent latency limits in the use of FPGA hardware accelerators in various deployment scenarios in a networked context.

5. The use of an FPGA network switch extended to perform additional computation in an edge computing case study, and show the resulting performance benefits against other competing strategies.

1.4 Thesis Roadmap

Chapter 2 is a comprehensive background and literature review. It contains background and comparisons of different computing architectures, how they are integrated into networked systems, and where they can be deployed. How this deployment has been modelled in other works is also discussed.

Chapter 3 details a mathematical model that can be used to describe and evaluate the deployment of hardware acceleration and the offload of computing tasks within a network of connected devices. It also demonstrates the usage of this model to generate optimal task and hardware placement for a given scenario and set of constraints. The optimisation is evaluated against naive placement strategies using a representative case study as well as synthetically derived scenarios.

Chapter 4 focuses on experiments designed and carried out to determine latency characteristics in the deployment of FPGA accelerators. Both hosted PCIe and server-less in-network deployments are characterised.

Chapter 5 is a case study comparing in-network FPGA acceleration with other offload strategies for a complex image processing application. The in-network approach uses an augmented FPGA network switch, and is compared to edge node and cloudlet offloaded computation. This chapter details the design of the experimental testbed, results, and general insights that can be derived from the experiments. Results show that the host ingestion latency is comparable to the network traversal time.

Chapter 6 discusses further work and conclusions drawn from the work presented in this thesis.

1.5 Publications

Work presented in this thesis has featured in the following publications.

1. Ryan A. Cooke, Suhaib A. Fahmy, *In-network online data analytics with FPGAs*, in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2017 [1].

2. Ryan A. Cooke, Suhaib A. Fahmy, *A model for distributed in-network and near-edge computing with heterogeneous hardware*, in Future Generation Computer Systems (FGCS), vol. 105, 2020 [2].

3. Ryan A. Cooke, Suhaib A. Fahmy, *Quantifying the latency benefits of near-edge and in-network FPGA acceleration*, in Proceedings of the International Workshop on Edge Systems, Analytics and Networking (EdgeSys), 2020 [3].

4. Ryan A. Cooke, Suhaib A. Fahmy, *Characterizing latency overheads in the deployment of FPGA accelerators*, in Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), 2020 [4].

5. Ryan A. Cooke, Suhaib A. Fahmy, *Exploring Hardware Accelerator Offload for the Internet of Things*, submitted to: it - Information Technology.

Chapter 2

Background and Literature Review

When considering modern networked computing systems, there are two primary considerations: the computing resource used to complete the processing, and where they are located in the network. As discussed in Chapter 1, these systems are increasingly heterogeneous, utilizing a variety of hardware architectures to perform the required computation. Each of these platforms have varying capabilities and trade-offs. Additionally, there are a growing number of network nodes capable of hosting these hardware platforms, increasing the deployment possibilities.

This means that there are a greater variety of deployment options when considering distributed hardware acceleration. It becomes more difficult to evaluate task placement, and to manually design these systems. Existing mathematical models don't take these recent trends into account, or are designed for runtime task allocation in an environment with heterogeneous hardware architectures.

How these architectures are integrated to distributed, networked processing systems is also a significant challenge, with implications on performance. There are now a greater variety of integration techniques, with little experimental evaluation.

This chapter first examines the variety of processing architectures available in modern systems. It then examines the various integration possibilities, and relationship with the supporting network infrastructure, highlighting the need for experimental comparisons between approaches. Finally, it contains a study of other modelling efforts, and how they aren't sufficient for modelling heterogeneous, distributed systems given modern advances.

2.1 Computing Platforms

In this thesis, the term computing platform is used to define a hardware architecture capable of carrying out a computation. Historically, this processing would be carried out on a general-purpose processor, but as workloads have increased in scale and complexity, alternative accelerator platforms have seen a surge of interest.

2.1.1 Central Processing Units

General-purpose central processing units (CPUs) are fundamental components of computing systems. While there is a significant variety of specifications and implementations, the general execution model is the same across most devices. The task to be carried out is expressed through a program, a set of instructions composed from the processor's instruction set – the set of fundamental operations that a CPU can perform. The program is stored in memory, and instructions are executed sequentially.

Processing Model

At the start of each cycle, the next instruction to be executed is fetched from program memory, pointed to by a special purpose register called the program counter (PC). The instruction typically comprises an opcode denoting the operation to be performed, and register and memory location references to the operands. After being fetched, it is decoded into signals that control the execution unit. The decoded instruction triggers a series of actions that execute the operation. These actions will vary depending on the operation required, but usually involve loading operands into registers, fast memory local to the processor. Arithmetic and logical operations are carried out between these registers.

Application Development

CPUs are designed for flexibility, and must be able to implement a wide range of applications. Users can define their programs at different levels of abstraction. At the lowest level, programs can be written in the assembly language for the target processor instruction set. Programs are written directly in terms of CPU instructions by the user. For most applications, this approach is impractical and tedious, and is not realistic to write anything more than simple code at this level. The resultant program is also only able to be executed by an architecture that uses the same instruction set. Most programs are expressed in higher level languages that allow

for architecture independent, complex functions to be expressed more succinctly than at the instruction level. For some languages, such as C or C++, the program is compiled from this higher level expression down to machine instructions that are directly executed by the processor. Other languages are compiled down to hardware independent bytecode, which runs on an intermediate virtual machine, which has hardware-specific implementations. This way the same executable can be executed on any hardware, given that it has a virtual machine implementation that can translate the bytecode to native processor instructions.

Types of CPU

Due to the general-purpose nature of CPU-based computation, there are many variants of the basic architecture that are used for difference application domains.

Server class processors such as the Intel Xeon family are designed to handle larger, more complex workloads, and many tasks at the same time. In comparison to other processors, they usually have many cores on the same silicon die. Each core is capable of independent concurrent operation, and is often also multi-threaded, which allows for the core to be rapidly switched between contexts. This effectively allows multiple software processes to be run concurrently on the same, time-shared physical core. These devices are expensive, and compared to processors used in other contexts, have high power consumption and heat dissipation.

The next class of processors are for use in general-purpose, desktop machines, such as the Intel Core series. Compared to server-class processors, these devices have fewer cores and are cheaper.

Processors not designed for use in a desktop or server environment can be classed as embedded CPUs. They are used in application specific deployments, often controlling or interfacing with external actuators or sensors. In comparison to the other classes, they are lower-power and cheaper. Single-board computers such as the Raspberry Pi are often used to control embedded applications, and utilise 32-bit multi-core embedded processors capable of running Linux. Micro-controllers integrate a processor, memory, and other peripherals onto the same die and are generally even more low power and low capability. Small 8-bit or 16-bit processors may be used in devices with severe power constraints such as remote sensor nodes.

In all cases, when offloading to or from processors, data must traverse a network interface and standard bus into processor memory space.

2.1.2 Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are silicon integrated circuits comprising an array of configurable logic resources. There are several primary vendors of these devices, the most notable being Xilinx and Intel. While these vendors hold a majority of the market, other vendors exist, such as Lattice, who specialise in small low-power solutions, and Microsemi, who offer specialized products such as radiation-hardened devices.

Architecture

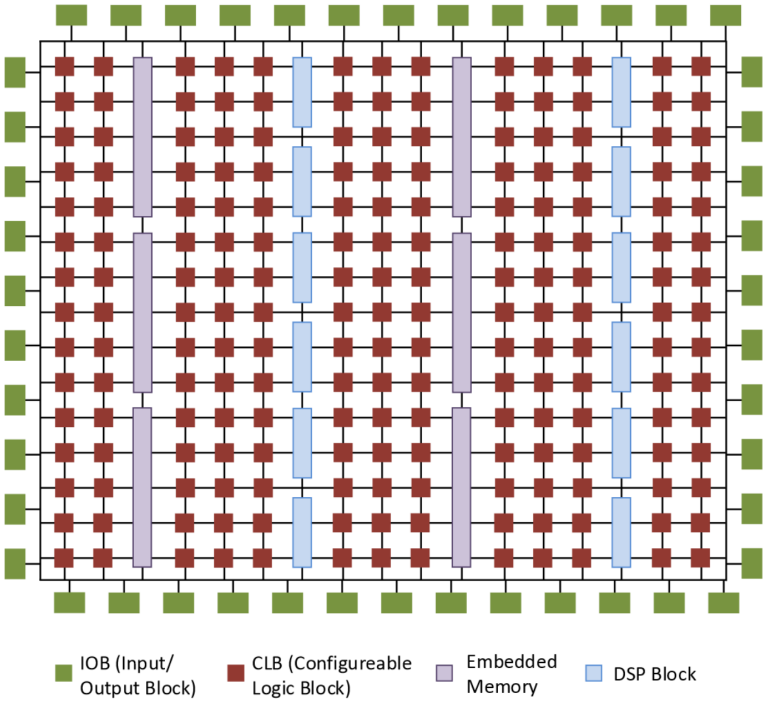


Figure 2.1: Representative modern FPGA architecture [23]

Modern FPGA architectures, regardless of vendor, are based around a generic design. Figure 2.1 shows a high-level overview of a Xilinx FPGA architecture. Users can express a digital circuit in a high-level abstraction, which goes through a flow of vendor tools to convert the circuit to be implemented using these logic resources on the FPGA fabric.

User logic is implemented via an array of configurable logic blocks (CLBs) [24], which are connected to other CLBs through a switching matrix. Each CLB comprises several slices, where each slice typically contains a look-up table (LUT) and

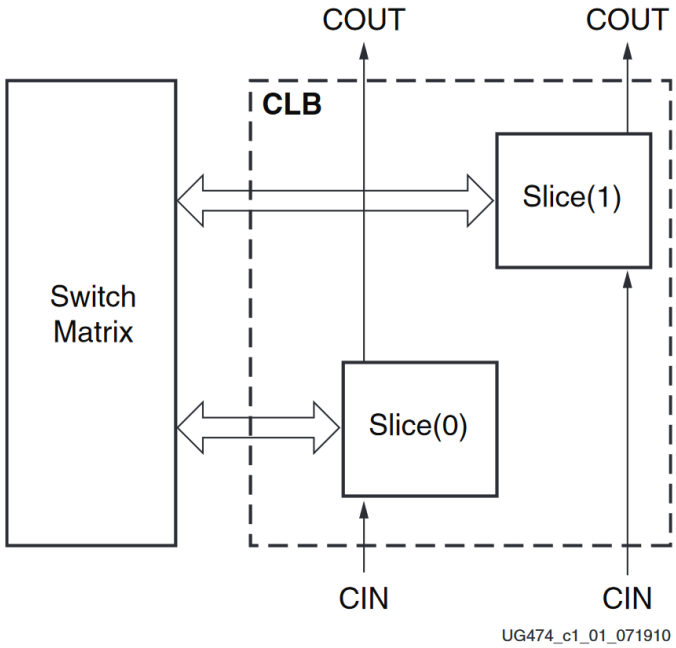


Figure 2.2: Xilinx 7 Series CLB arrangement [24]

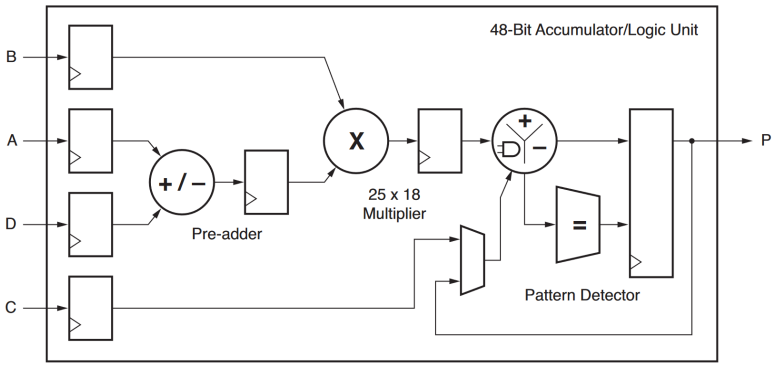


Figure 2.3: Xilinx DSP48E1 architecture [25]

storage elements like flip-flops or small amounts of RAM. For example Xilinx 7 series FPGAs use CLBs with 2 slices, as in Figure 2.2, with each slice having four 6-input LUTs, 16 flip flops and 256-bits of distributed RAM. The CLB LUTs can be configured to implement a large variety of logic functions, and chained together to implement even more complex operations. In addition to CLBs, most modern devices also have hardened blocks of logic optimised for digital signal processing, such as the DSP48 [25] blocks built into Xilinx FPGAs (Figure 2.3). DSP slices can be used to implement complex arithmetic functions with fewer resources.

Hardened Block RAM (BRAM) is also usually present in modern FPGA architectures. This provides storage within the fabric, reducing the need for costly off-chip memory transfers. IO-blocks (IOB) are configurable blocks that contain the appropriate signal conditioning to allow for signals to be brought into or out of the FPGA. Any signal on any pin must go through an IOB. IOBs are arranged into banks - for example, Xilinx 7 series banks contain 50 IOBs. high-speed transceivers are also commonly implemented, to allow the FPGA to interface with high-speed signals such as those used in PCIe and Ethernet. Implementing this functionality using CLBs would consume significant resources, or not meet required performance standards.

All resources are connected through a routing network that routes signals throughout the FPGA. Dedicated interconnect is used to route clock signals. Clocks can be introduced to the FPGA through IO pins. Global clock lines can be used to route the clock to multiple elements throughout the device. These lines ensure minimal skew in comparison to the general-purpose routing resources. Many FPGAs include built-in PLLs or DLLs to synthesise signals of a configurable frequency.

Application Development

The user can express their design at different levels of abstraction. As designs and FPGA devices are complex, it would not be viable to specify the design at the individual gate and wire level. The lowest level of abstraction typically used by designers is the register transfer level (RTL). This is a paradigm where digital circuits are modelled as combinational transformations on data as they move between registers. Designers write this RTL description in a hardware description language (HDL), the most popular being 'Verilog' and 'Very high-speed Integrated Circuit Description Language' (VHDL). All mainstream vendor tools accept these languages. Once a design has been described with a HDL at the RTL level, it is converted to a low-level description by vendor tools in a process called synthesis. The synthesised circuit is then mapped to the target FPGA resources by the vendor place-and-route tool, creating a device specific configuration that implements the desired circuit. This configuration is then encoded into a bitstream that can be loaded into the FPGA configuration memory.

Designing at the RTL level requires knowledge of HDL, digital logic design, and some knowledge of the target FPGA architecture. These present one of the main barriers to entry for developing for FPGAs, leading to the development of language and design flows that let users work at an even higher levels of abstraction, an approach commonly referred to as high-level Synthesis (HLS). Here, designers

describe the design at an algorithmic level using languages such as C++. Both Xilinx and Intel offer HLS tools for their platforms.

Vivado HLS is the commercial Xilinx HLS tool, where users express accelerator designs using C or C++ [26]. The user writes functions in these languages that are compiled into HDL descriptions. Features such as loop unrolling are available which allow for the user to control to what extent loop constructs are implemented spatially in parallel in the resulting FPGA design. This can increase the performance of the design with the penalty of it using more resources. Python-based HLS has also been demonstrated with the open source Migen framework [27]. The framework also includes a set of free IP cores written in python that can be instantiated into user designs. Testbenches can also be written in python.

The open computing language (OpenCL) framework is also utilised for FPGA HLS. This framework allows for cross-platform code to be written in general-purpose languages such as C++, which can then be targeted to heterogenous platforms such as GPUs and FPGAs. Intel offer this as part of their FPGA SDK [28], and Xilinx as part of their SDAccell platform[29]. The benefit of this approach is that it allows for side-by-side development of the base software application that will run on the host, tightly coupled, or soft-core processor and the hardware accelerator. These tools hide details regarding the software and hardware interfacing from the user. They represent a growing trend of heterogeneous computing system designs. Various academic works look to extend OpenCL FPGA integration for use with other languages. The work in [30] enables the coupling of Java programs with FPGA accelerators within th OpenCL ecosystem. This is useful for large-scale analytics applications as many of the frameworks used for this, such as Hadoop, utilise Java frontends. Another work [31] presents a similar strategy for accelerating python code with FPGAs using OpenCL.

In addition to using general-purpose languages for HLS, many works have examined using domain specific languages (DSL) to generate FPGA designs. These are languages designed to either simplify or enhance the development of applications for a specific domain. An early work in this area [32] mapped a network processing DSL called CLICK to an FPGA. The DSL was compiled to a HDL module that could then be synthesised with vendor tools. Network processing languages such as P4 have also had HLS integrations [33; 34]. Hipacc [35] is a DSL than can be used to create FPGA accelerators for image processing. It utilises OpenCL, and can also be used to target other platforms such as GPUs. The framework also allows for more image processing kernels to be added to the database of supported operations. Another image processing DSL, Halide, is adapted to generate FPGA accelerators

in [36]. An end-to-end flow is presented, and an evaluation the generated accelerators performed around 4× faster than CPU implementations. The framework presented in [37] allows for SQL queries to be compiled to FPGA accelerator blocks to be loaded into PR regions at runtime. SQL is a DSL used to interact with relational databases. A generic backend named FROST, capable of automatically adapting multiple different DSLs to HLS is demonstrated in [38]. It can extract an abstract representation of the functionality described by the DSL and then generate a C++ equivalent which can then be used with Xilinx HLS and SDAccel tools. The tool also exposes the same optimisations offered by Vivado HLS such as loop unrolling to the user.

One of the key barriers to the utilisation of FPGAs within networked computing systems is the considerable design effort required. This includes both datapath and system-level design. Advances to high-level development flows such as HLS simplifies accelerator designs that use standard interfaces, greatly reducing friction when deploying FPGAs as accelerators in networked systems.

Types of FPGA

FPGAs can vary in terms of the amount of available resources, hardened blocks, and IO. In general, more expensive devices will have a greater number of CLBs, DSP slices, and more available BRAM, as well as higher bandwidth I/O. Devices such as the Xilinx 7 or Ultrascale series are designed for high-speed networking or accelerating datacentre workloads, so have high-speed transceivers and large amounts of on-chip memory available.

FPGAs designed for smaller scale and even embedded applications also exist. The lattice ECP5 is a cheaper, mid-range device that's designed for automotive or embedded applications. The Lattice ICE40 is a very low density FPGA with few resources used for ultra-low-power embedded applications running on limited power supplies.

Many modern FPGAs can utilize partial-reconfiguration (PR). This feature allows for reconfiguration of selected regions of the FPGA at runtime, without having to reconfigure the whole device [39]. Hardware modules must be partitioned into partially reconfigurable regions, and then these regions mapped to the FPGA fabric through floorplanning. The vendor tool generates partial bitstreams that implement these modules with resources in the reconfigurable regions. These partial bitstreams can then be loaded at runtime to reconfigure these regions without affecting the rest of the design (Figure 2.4). PR has several benefits over full reconfiguration: resources can be time-shared, reducing the overall resources required, designs are

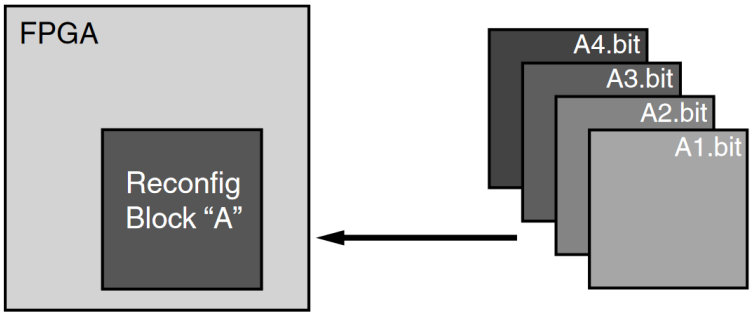


Figure 2.4: Partial reconfiguration allows for selected regions of the FPGA to be reconfigured using partial bitstreams [39]

more flexible and can be modified at runtime, and the same FPGA can be used to independently serve multiple applications in isolation.

2.1.3 FPGA Accelerator Design

FPGAs can provide improvements to application performance through the deployment of computing logic optimised for a given task. This is in contrast to CPUs, which execute software on a general-purpose architecture. The task, algorithm or process is implemented using logical resources on the FPGA. This may be the entire algorithm, or only computationally intensive functions that act as a bottleneck when performed in software. Custom FPGA implementations achieve improvements over software through several avenues that will be discussed in this section.

Static Accelerators

Static accelerators are architectures that are designed to implement a single function. While these application specific architectures may be reconfigured through changing register values, there is no dynamic, device level reconfiguration. To change the core functionality of the accelerator, the entire FPGA must be reconfigured.

One of the key benefits of FPGA accelerators is the ability to implement architectures that exploit temporal parallelism. A sequence of operations can be ‘pipelined’,where each stage of the pipeline can operate concurrently – resulting in an improvement to throughput. Another of the key features of FPGA acceleration architectures is the exploitation of spatial parallelism. A function that may take many iterations of a loop in software can be ‘unrolled’, and multiple iterations of that loop can be implemented with independent FPGA logic, allowing them to execute in parallel. This property is exploited in many data mining and machine learning

algorithms, which often involve performing large quantities of the same operations on a large data set. Utilising a tiled design, of the same operations repeated spatially across the FPGA fabric, is commonplace. The SVM accelerator in [40] demonstrates an example of this approach. This design uses a set of tiles that operate on different parts of the dataset in parallel. Within each of these tiles is another tile structure. The outputs of these tiles are aggregated by additional logic that is in sequence with the tiles. This architecture outperformed a software implementation of the same algorithm by 2–3 orders of magnitude as a result. Decision tree algorithms also commonly use FPGA accelerators, and benefit from spatial parallelism. The architecture in [41] again uses a tiled design, where each comparison in the tree is implemented in parallel. An implementation of a random forest classifier takes this further in [42], implementing multiple trees operating on the same data, in parallel. The design had a greater performance per watt compared to CPUs and GPUs. The paper additionally highlights an issue with spatially parallel designs - the finite resources in the fabric can limit performance gains, and the size of the algorithm that can be implemented. To extend the forest classifier beyond a given size for the available hardware, the authors had to use multiple FPGAs, while still needing only one unit for the GPU implementation.

These decision tree works use static tiles and tree structures. This means that to implement a different tree, the FPGA must be reconfigured. The work in [43] proposes a structure that uses generic, dynamically configurable tiles that use configuration data stored in on-chip RAM. The tree is converted into a rules table and written to RAM to reconfigure the accelerator without having to reconfigure the entire device. Utilisation of an array of flexible tiles is a technique often employed with convolutional neural network (CNN) accelerators. CNN inference requires many 3D convolutions, which include large sets of multiplications. The number of operations required means that unrolling the application in its entirety would be impossible due to resource limitations, therefore a set of processing elements that can be reconfigured dynamically is typically used, operating on chunks of data sequentially. The CNN accelerator in [44] uses an array of processing elements that are configured by a soft-core processor on the FPGA. Weights and data are fetched from off-chip memory. This accelerator improved processing time by 7× compared to a CPU implementation, and had a 24× reduced power consumption. Data transfer from off-chip memory is a bottleneck for these systems, so on-chip BRAM is used for temporary buffering. This reduces the number of costly off-chip memory transfers, and allows for processing to be offset from data transfers temporally. Another CNN accelerator demonstrated in [45] attempts to optimise these on-chip buffers to take

data re-use into account, greatly reducing the number of required off-chip transfers. They also develop a 3 dimensional array of processing elements as opposed to a traditional 2 dimensional one, allowing for greater re-use of processing resources.

In all, FPGAs can be used to accelerate a large variety of intensive tasks, reducing the computation latency in comparison to CPU architectures.

Dynamic Partial reconfiguration

Dynamically reconfigurable accelerators are desirable in a number of cases and can be achieved through partial reconfiguration (PR), a feature available on most modern FPGAs that allows for reconfiguration of only selected regions of the FPGA fabric.

Most PR accelerator designs use an architecture comprising a static shell that contains communication and control logic, and reconfigurable regions that can be loaded with different accelerator logic at runtime using PR. This allows the device to continue running and operating within a larger system while the accelerator is being modified.

One scenario where this technique is useful is when designing an accelerator for adaptive systems. These systems must alter accelerator properties at runtime to respond to external stimulus. In [46], an FPGA accelerator is deployed as part of a wireless sensor network application that can track points of interest. The application is too computationally intensive to use microcontrollers, and a static FPGA design would be unsuitable as full device reconfiguration would take too long, as well as sacrifice communication with the network. Different filters are swapped in to the reconfigurable region at runtime depending on the targets characteristics such as velocity, noise and priority. Deploying all filter datapaths on the device at once would not be possible due to limited resources. A PR-based accelerator is similarly deployed in [37], for database query processing. Acceleration of these queries is essential for large databases to ensure throughput targets are met. Implementing all possible query pipelines on the FPGA at once was not possible as it would have consumed too many resources. Reconfiguring the entire accelerator for each query would be too slow. This design therefore loads the appropriate query processing pipeline as they arrive using PR. Compared to a software solution, the FPGA accelerator achieved between 1.4× and 6.15× speedup depending on the query, with greater relative performance gains for more arithmetically complex queries.

An adaptive K-nearest neighbour (KNN) accelerator is developed in [47], where PR is used to modify the design based on user parameters. This allows for faster reconfiguration times for latency-sensitive workloads, and again allows the

accelerator to be integrated into a communication infrastructure mpre easily. The design provided a 68× speedup over a CPU implementation, and using PR instead of full-device reconfiguration resulted in around 5x faster reconfiguration. The user parameters for KNN applications are frequently changed, with users often experimenting and comparing results for different parameter values, making this faster reconfiguration time beneficial for this application. PR has been used in cognitive radio applications [48] to swap in different functions depending on the conditions such as the signal-to-noise ratio of the transmission channel, or available power. For example, a function performing the same function but with a lower power consumption can be loaded on to the FPGA if a more efficient implementation can be used. An OFDM cognitive radio is implemented in [49] to support different OFDM standards. This design uses PR only for modules that require significant changes to support a different standard, and parametrised modules for those requiring less change, and the additional resource overhead caused by the parametrisation is below a threshold. This was shown to give performance benefits compared to the traditional approach of making using PR modules for the entire pipeline, or a monolithic module containing the entire design.

PR-based accelerators are also used to time-multiplex FPGA resources, overcoming spatial constraints. The entire application may not fit within the FPGA at once, so PR is used to swap in different tasks at runtime. An example of such an accelerator can be found in [50]. The bio-informatics application examined benefits from comparing the outputs of different classifier algorithms. Using partial reconfiguration, classifiers can be loaded sequentially, saving FPGA resources, and can be loaded 8× faster than complete device reconfiguration. Similarly, this time-multiplexing technique can be seen in [51], for a KNN application. The accuracy of the algorithm can be improved by comparing the results of an ensemble of classifier configurations. Resources can be saved by loading classifiers with different parameters sequentially. This method has the benefit of making the design more scalable, and compatible with a greater number of devices with different resources available. For large FPGAs with an abundance of resources, more classifiers can be instantiated in parallel, while for smaller devices, more time-multiplexing can be utilised. Time-multiplexing using PR is exploited for a NN inference application in [52]. The results of each layer are dependent on the results of the layer before, so each layer is loaded sequentially using PR, saving resource consumption. This makes the design more compatible with smaller devices. Similarly, the video broadcasting decoder accelerator is demonstrated in [53], which again takes advantage of time-multiplexing using PR to save resources. Again, this is a highly sequential algorithm where the

results of a given part are highly dependent on the previous parts.

Dynamically reconfigurable accelerator devices are beneficial for applications that must be flexible at runtime. Additionally, dynamic reconfiguration allows for time-sharing of resources and isolation of tasks, which are key enablers of device virtualisation. These properties are particularly advantageous in networked computing systems where resources are shared across multiple tasks, and process constantly changing workloads.

Overlays

Related to this design paradigm is the concept of overlays. This technique involves implementing an intermediate, coarse-grained architecture on an FPGA, quite often an array of generic processing elements, which can then be programmed without reconfiguring the FPGA. Designers express their requirements using a higher level of abstraction, and a custom compiler transforms and maps this design to an architecture pre-loaded onto the FPGA. This gives added flexibility in some scenarios, especially when fast reconfiguration is needed. It also has the added benefits of reducing the difficulty in designing an accelerator, and reduces the time taken to generate a design as no synthesis or place and route is needed.

An example of an overlay architecture can be found in [54]. This design resulted in a 140× speedup and an improved area-time product compared to a soft-core running on an FPGA. One issue with overlays is that they consume more area and can limit throughput and operating frequency compared to fine-grained designs optimised for a specific application. Some overlay architectures have attempted to solve this by designing overlays more closely around the underlying FPGA architecture [55; 56]. These works utilise functional units built around the DSP48E1 primitives found on modern Xilinx devices. As a result, there were large improvements to resource consumption, and improved throughput and reconfiguration time.

Overlay architectures can be targeted to specific domains in order to alleviate their performance penalty compared to hard logic implementations. Optimising the overlay for specific domains sacrifices flexibility for increased performance. The 'DLA' architecture presented in [57] is targeted at neural network inference acceleration. Using this overlay, the authors were able to achieve a throughput of 900fps for GoogleNet. One of the factors enabling this high throughput was the fast reconfiguration times offered by overlay architectures – layers' filters could be quickly loaded in ahead before they were needed. The overlay in [58] was similarly targeted specifically at NN inference, but expanded the overlay and supporting instruction set to be able to support a greater range of NN architectures, and allow for a more

fine-grained control of operator deployment - resulting in more efficient utilisation of overlay resources. Their platform showed significant improvements in performance per watt compared to both GPUs and FPGA implementations demonstrated in other NN inference works. They also demonstrated improvements over a GPU for a license plate detection applications, achieving 3× greater throughput. An overlay for accelerating DSP workloads is demonstrated in [59]. Fast reconfiguration is beneficial for workloads with variable parameters, or for devices that don't have the resources to allow for the entire application to be mapped onto the FPGA at once. It was evaluated using a 255 tap FIR filter implementation, where the accelerator implemented with the overlay resulted in 10× speedup compared to a CPU, compared to 16× speedup for a full custom design. However for larger data sizes, when reconfiguration time is taken into account, the overlay architecture begins to outperform the hard logic implementation.

While overlays can be used to implement dynamically reconfigurable accelerators, performance limitations exist as the architecture cannot be completely optimised for a given application, and the overlay architecture can limit what logic can be implemented, making it unsuitable for some applications. However they can simplify application development, enable fast reconfiguration, and abstracts away system design details.

2.1.4 Graphics Processing Units

Graphics processing units (GPUs) are another reprogrammable hardware accelerator platform, historically used to accelerate graphics rendering. Compared to a CPU, a GPU comprises large numbers of simpler compute units operating in parallel. Complex functions are decomposed and distributed amongst these cores, leading to reductions in processing time. This processing architecture can be used to accelerate non-graphics rendering applications. In [60] a GPU was used to accelerate quantum chemistry calculations, and resulted in a speedup of 3.8× compared to a CPU. GPUs have also been demonstrated to improve performance in neural network applications such as in [61] where a 2-11× speedup was achieved for a recurrent neural network.

Processing model

GPUs comprise of a vast array of processing elements or cores. Each core has limited functionality compared to a CPU core, but is comparatively highly optimised to perform floating point arithmetic operations. Extraneous logic isn't implemented, which means a much higher density of processing cores can be implemented on the

same die. Instead of computation being carried out sequentially, it is decomposed and distributed to a larger number of cores in parallel, increasing throughput and reducing computation time for tasks with large data sets. Cores are also often gathered into processing groups, that all only execute the same instruction. This removes the need for the CPU fetch, decode, execute cycle.

Application Development

Applications are usually written for GPUs using standard software-based programming languages such as Python or C++, with an explicitly parallel GPU framework that abstracts low-level detail away from the user. CUDA is a framework developed by Nvidia for use with its GPU architectures. As previously mentioned, OpenCL is another framework that can be used. OpenCL allows for users to express applications independently of the target architecture.

Types of GPU

GPUs are often connected to a host processor through the PCIe bus. Datacentre class GPUs have high densities of processing elements, and are hosted on boards with large amounts of GDDR5 memory.

Embedded GPUs are lower density and lower power solutions, often integrated onto SoCs used in mobile phones or single board computers.

Comparison to FPGAs

There are several works comparing GPU to FPGA acceleration. Both were tested with a random forest classifier application in [42]. This study determined that for this application, the FPGA achieved a higher performance per watt, however the GPU produced the best performance per dollar. While the FPGAs were more power efficient, it was also noted that the FPGAs ran into resource comsumption issues for larger forest sizes, resulting in the need for additional FPGA boards. The GPU did not, however had a degrade in performance instead. A RNN application was evaluated with a Zynq SoC and Nvidia Tegra GPU development board in [62], platforms marketed towards embedded applications. The Zynq outperformed the GPU significantly in terms of performance per watt. This observation was mirrored in a K-means clustering application [63], where the FPGA accelerator was significantly more power efficient and energy efficient than the GPU.

An extensive review of FPGA and GPU comparison studies was carried out in [20], which looked at the relative performance of GPUs, FPGAs and CPUs for

a variety of applications such as matrix multiplication, FFTs, and encryption/decryption. It was found that for algebraic operations and complex simulations that involved complex mathematics with floating point numbers, GPUs generally provided greater speedup. FPGAs were better at combinational logic applications like encryption, and for fixed-point or integer signal processing. FPGAs had a lower power consumption in most surveyed applications.

As both platforms excel at different tasks, some efforts have been made to produce heterogeneous systems utilising both. This technique has been used for a medical application requiring very high frame rate image processing [64]. An FPGA acting alone did not have the resources available for very large parallel operations and large buffers required for the application, and a lone GPU ran into issues with data flow bottlenecks. A combined solution led to double the frame-rate of a lone GPU and reduced latency. A hybrid GPU-FPGA architecture was used for a SVM image classifier application in [65], where the FPGA performs feature extraction and the GPU the classification. The FPGA provided low latency processing of the image feed as it could interface with the camera easier. The GPU allowed for highly parallel computation that made up the classification phase.

Overall, FPGAs have been demonstrated to have greater power efficiency than GPUs. Another main difference is that FPGAs allow for more integration possibilities. GPUs rely on a PCIe connection to a CPU host, while FPGAs can be connected directly to the network as standalone devices.

2.1.5 Application Specific Integrated Circuits (ASICs)

Application specific integrated circuits are devices designed and fabricated to carry out a specific group of tasks. Unlike FPGAs, the architecture is fixed at manufacture, and unlike CPUs or GPUs, they offer limited programmability. The benefit of this rigid architecture is it can be completely optimised to perform these tasks. While FPGAs require flexible interconnect which limits the achievable clock rate, ASICs do not. There is also no redundant logic or unused resources.

The work in [66] compared a recurrant neural network application implemented on an FPGA and a 14nm ASIC. They estimate that for this purpose, the FPGA was only 7× less efficient than the ASIC. Furthermore, they identified that the number of available DSP blocks was a limiting factor in the FPGAs performance compared to the ASIC, and that this gap will close as new devices offer these resources in greater densities. In another work, the authors make a similar comparison with binarized neural networks [67], and reach the same conclusion, with the FPGA performing around 8× less efficient than the ASIC implementation.

There have been a number of commercial ASICs developed to accelerate machine learning applications. Google have recently released a programmable ASIC for neural network inference called a Tensor processing Unit (TPU), which allows fast execution of tensor models [68]. This architecture was shown to achieve up to 30× speedup and 80× energy efficiency for DNN inference compared to CPU and GPU architectures tested [69]. Intel's Myriad X vision processing unit (VPU) is an ASIC optimised for computer vision applications, and contains a dedicated neural network processor [70]. There are also versions of these devices designed for operation at the network edge. These platforms are relatively new, and there are very few works comparing their performance to FPGAs.

2.1.6 Summary

In recent years there has been a significant increase in the number of available processing platforms and architectures. These architectures have vastly different computational abilities and execution models. When modelling distributed hardware acceleration, the heterogeneity of modern platforms must be considered.

The variety of processing platforms available introduces further variables to consider when deciding how to distribute processing. Unlike other works, the modelling work presented within this thesis takes this into account. There is also little work comparing the performance of these platforms at a systems level, taking into account both the computation on the platform, and communication to it. Both of these overheads must be considered when evaluating distributed systems.

2.2 Accelerator Integration

Data transfer to and from the device is often a system bottleneck, and provides complex challenges, hence requiring significant engineering effort.

2.2.1 PCIe

FPGA accelerators have often acted as slave devices to host processors through PCIe, similarly to other peripherals such as NICs or GPUs. Utilising PCIe means minimal disruption to the rest of the infrastructure, and provided a scalable high throughput interface already commonly used in datacentre and workstation environments. The host CPU would control the flow of data to the accelerator, which had no direct connection to the rest of the network.

Implementing a PCIe connection between the accelerator and host requires

the user to design logic around a PCIe controller IP on the FPGA, as well as write the software drivers that allow the host to control and communicate with the accelerator, requiring knowledge of PCIe and driver development. Commercial integration frameworks such as Xilinx Xillybus [71] and IBMs CAPI [72] are built on top of PCIe, and abstract this detail from the user. However these solutions had several limitations, mainly being restricted to particular hardware. This led to the development of many open source frameworks. One of the first open frameworks that provided a generalised hardware interface based on the Xilinx PCIe endpoint IP and associated software API was Riffa [73]. Building upon this, other frameworks such as EPEE increased device throughput and allowed for additional features such as user defined interrupts [74]. Dyract [75] was a framework based on the open source Riffa framework that additionally provided the capability for partial reconfiguration over PCIe. Revisions to Riffa [76] introduced support for PCIe Gen3 and more FPGA platforms, and Jetstream [77] provided partial reconfiguration over Gen3 PCIe and multi-FPGA support.

The high throughput of PCIe lends itself towards batch processing applications with large volumes of historical data and large transfer sizes. FPGA accelerators using PCIe have been demonstrated to increase performance relative to CPUs in numerous works. A Virtex-5 based PCIe accelerator was used to accelerate neural network image classification for the Baidu search engine [78]. The accelerator provided a speedup of around 13× compared to a software implementation on their test data sets. These results are when using PCIe Gen2×4, so if upgraded to more the higher bandwidth PCIe Gen3, an even larger improvement would likely be seen. A KNN accelerator was demonstrated in [79], using PCIe Gen2×4 to connect to a CPU host. The accelerator provided a 148× speedup compared to a software implementation. A GPU outperformed the FPGA in terms of application speedup, but provided a 3× smaller performance per joule compared to the FPGA. Additionally, the FPGA was only using PCIe Gen2×4, while the GPU was using Gen3×16, which would account for some of the additional speedup of the GPU. PCIe-based FPGA accelerators have also been used to accelerate relational database queries [80]. The FPGA delivered a 10.7× and 6.7× speedup compared to the host CPU, for compressed data. When data was uncompressed, there was a smaller relative speedup due to the reduced effective bandwidth of data to the FPGA.

PCIe-based accelerators have an associated communication cost. The work in this thesis quantifies these overheads in Chapter 4.

2.2.2 Network interface

Alternatively, FPGAs can communicate with servers via a network interface, typically Ethernet. This allows FPGAs to connect directly to the network without the need for a CPU host. The FPGA can receive and process data without the latency and complexity of a software network stack and non-deterministic CPU behaviour. These characteristics lend themselves well to stream processing applications, where data must be processed immediately as it arrives, and latency is of greater importance than throughput.

One method of networked FPGA accelerator deployment is commonly referred to as a 'bump-in-the-wire', where an FPGA is placed between a servers NIC and the network, so that data goes through the FPGA before reaching the server. One example of this approach is demonstrated in [81]. FPGAs can be used as local compute accelerators, used for on the wire processing of packets from the network, or as a set of shared pooled resources between servers. Networking applications such as line-rate encryption/decryption were accelerated, providing performance benefits over software implementations. Acceleration of the Bing search engine was also implemented, which involved multiple remote FPGAs working together across the network.

FPGAs have alternatively been integrated directly into the NIC, with a PCIe connection to the host server and Ethernet connection to the wider network. Again, this allows the accelerator to operate on data from the network independently of the host, and removes the need for sperate NIC and FPGA accelerator boards. Microsoft have adopted this approach to integrate FPGA acceleration into their Azure cloud platform [82]. FPGA accelerator NICs have also been demonstrated to improve performance per watt in applications such as key-value stores, such as in LaKe [83]. Here, memory caches are implemented on the FPGA BRAM and external DRAM connected directly to the FPGA, which doubles as a NIC for all non-application traffic. Upon reception of a query, the packet is only forwarded to the host server if the requested data is in neither cache. This allows for drastic improvements to latency and power consumption compared to software implementations. Outlier filtering for data mining applications was implemented on a 10GbE FPGA-based NIC in [84]. The FPGA NIC filters incoming data and only forwards packets of interest to the host, drastically increasing the effective throughput. The system was capable of filtering the incoming data stream at 95.8% of 10GbE line-rate.

Utilising high-speed network interfaces with FPGA accelerators also allows for multi-FPGA clusters to be implemented. This technique is used to implement a mapreduce accelerator in [85]. Each FPGA is connected to a host server through

PCIe, but connected to other FPGAs in the system through Gigabit Ethernet. The host mainly controls and configures the FPGA, and to retrieve input data from the host memory. Key-value pairs are passed between FPGAs through the network interface. The FPGA-acclerated system was 15× to 20× faster than the software implementation depending on application parameters. The main bottleneck was identified as the communication from host to FPGA through PCIe. Similarly, a multi-FPGA accelerator for deep learning mapreduce applications is presented in [86]. Instead of using a cluster of host-FPGA pairs, it uses the Xilinx Zynq boards, where the FPGA and CPU are tightly coupled, removing the PCIe bottleneck. They reported 8× to 12.6× speedup for a CNN application with this system compared to a cluster of CPU nodes, as well as significant power consumption reduction.

2.2.3 Tightly coupled SoC

More recently, platforms that feature an FPGA tightly coupled to a CPU in the same package have emerged. Examples include the Zynq family from Xilinx, which couples an FPGA with an ARM core, and some of the Xeon family from Intel. The FPGA can communicate with a processor through high-speed on-chip interconnect, instead of external interfaces like PCIe or Ethernet. Typically these devices use the processor for control and configuration, and the programmable logic for compute intensive processing. While a traditional FPGA may utilise a soft-core processor implemented within the FPGA fabric, this is resource inefficient in comparison. The integrated processor has often led to the reprogrammable SoC devices being targeted towards embedded and edge of network applications.

The tightly coupled processor simplifies software controlled partial reconfiguration of the programmable logic part of the device. This allows for reconfiguration of only specified regions of the fabric, as opposed complete device reconfiguration, enabling time multiplexing of the programmable logic, increased flexibility to modify logic at runtime, and sharing resources across multiple applications. There is an area of research focused on accelerator management with tightly coupled SoCs. ZyCap [87] is a framework that utilises a custom reconfiguration controller and software interface developed to allow for more efficient partial reconfiguration on the Zynq platform. When evaluated it had 3× higher throughput than the Xilinx ICAP alternative. The MiCap framework [88], uses a controller that is designed for smaller bitstreams, and allows for the configuration to be read back. It is targeted at dynamic circuit specialisation applications, where circuit parameters are implemented into the design as constants rather than regular inputs, allowing the design to be specialised around those constants. The work on MiCap is extended in [89].

These tightly coupled architectures are suitable for software applications with a few complex tasks. A PR-based cognitive radio is presented in [90], where the control and configuration is in software in the PS, while the more computationally intensive baseband processing is implemented within the PL. Additionally, the tight coupling with a CPU without the need for an external host makes them effective for embedded applications. In [91], a Zynq is used to implement a NN application to monitor gas sensors to determine the composition of the gas being monitored. The accelerator achieved comparable performance to hosted FPGA accelerators for the same application. A Zynq-based accelerator for a road sign detection application is demonstrated in [92], where the PL is used to interface with the sensor and filter data before it is classified in software in the PS.

2.2.4 Interface overheads

Factors such as I/O, network, and CPU stress have been demonstrated to have significant impact on the magnitude and variability of packet delays [93; 94; 95]. The various contributors of packet latencies in datacentre environments were examined in detail in [96]. The PCIe interface has also been shown to be a significant contributor to latency [97]. It was demonstrated in [93] that virtualisation using Linux Vserver typically added a small delay to packet round trip times, while Xen virtualisation added 3 to 4 times greater latency.

There has been little to no work on the interface overheads for FPGA accelerators in particular. As these overheads can be significant, the lack of this information can make it difficult to evaluate the value of accelerator platforms at a system level.

2.3 Summary

FPGAs have can be integrated into distributed computing systems through various interfaces. Despite this, there has been little work investigating the overheads of these interfaces, especially the emerging network-attached method.

While individual works have presented performance studies focussing on the designs they present, there have been no generalised examinations that can be used to aid in modelling or design decisions. Chapters 4 and 5 of this thesis detail experimental studies on these overheads, and demonstrate their effect on an application at the systems level.

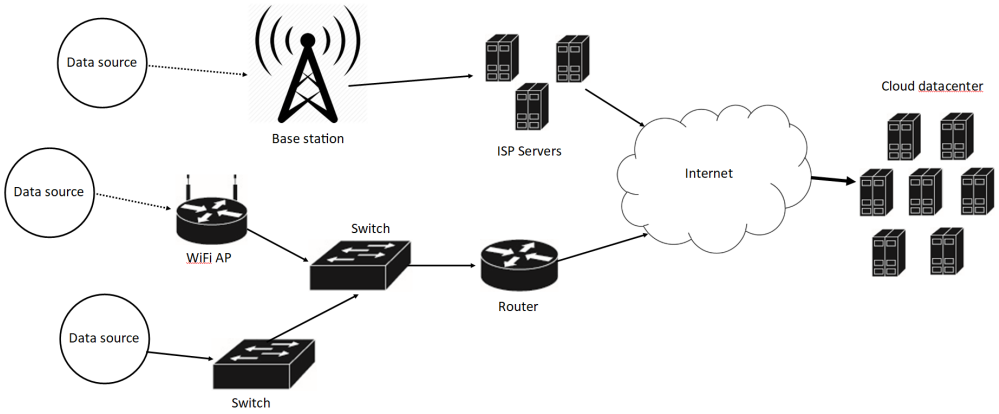


Figure 2.5: Example of network infrastructure supporting computational offload to remote resources.

2.4 Networked computing systems

In networked computing systems, processing is moved from a data source to another more capable network node for processing, to reduce computation time, power consumption, or increase throughput, and the result is sent back. An additional benefit is that the original platform is free to perform other tasks while it waits for the result. The benefits gained must make up for the communication penalty incurred through offloading the data. Depending on the nature of the application, this network structure will vary. An example of such as network structure can be seen in Figure 2.5.

2.4.1 Edge Networks

Endpoint nodes sit at the very edge of the network, and typically act as data sources. Examples of such nodes are IoT sensors, mobile phones, networked cameras, or autonomous vehicles. They offer some limited processing capability, but need to offload data to more capable hardware to meet power or response-time constraints. Typically processing is offloaded to the cloud or a remote datacentre, requiring devices to transmit data across the Internet to reach the target platform. Endpoint nodes could utilise various communication Interfaces. Nodes in remote, hard-to-access locations, or mobile phones, utilise cellular communication such as long-term evolution (LTE) to gain access to a larger scale network. Data is transmitted through the device antenna to the nearest eNodeB or base station [98]. This node facilitates the transmission through the core network to the service provider's datacentre, and then over the Internet. Other endpoint devices may utilise Wifi, in which case data

is transmitted to a local access point, which is then connected to the wired local area network. Data will traverse a series of Ethernet switches and aggregation nodes, and gain Internet access through a router. Applications that require high communication reliability, or are not in difficult to access, static locations may utilise wired Ethernet for the first hop, then follow the same path.

2.4.2 Datacentre Networks

Inside the datacentre itself is another complex network, an example structure can be seen in Figure 2.6. A datacentre is a location with a large collection of large-scale networked servers and other machines [99]. These servers are made up of racks of commodity machines, network resources, storage, power distribution, and cooling. A standard rack is 42U high, which means that 42 1U rack mount servers or other devices can be housed. Blade mounts allow for multiple servers to be hosted within the same 1U enclosure. As a result, server racks may contain a high density of machines, which all need to communicate with other machines in the datacentre. Each rack has a top-of-rack (ToR) switch which facilitates communication between servers in the same rack, and between the rack and the wider datacentre network [100]. External switches allow for racks within the same cluster to communicate with each other, and aggregation switches connect clusters to routers that allow for inter-cluster communication. In the particular topology shown in Figure 2.6, data must go through at least one switch to go from one server to another in the same rack, and at least 3 switches to reach a server in another rack. Quality-of-service policies are implemented on switches and host NICs to prioritise traffic flows under high network load.

While datacentres are typically large centralised environments, smaller scale datacentres are increasingly being deployed to the network edge, closer to the clients they serve. Being closer geographically reduces the latency. These edge datacentres are similar in structure to their larger counterparts, just at a smaller scale. For example, telco central offices traditionally used for telephone switching are being re-purposed as datacentres [101], that are in geographical locations close end users. Micro-datacentres have also been proposed – these are even smaller scale, comprising only 10s of servers in few racks, placed in extreme proximity to premises. This means that platforms requiring offload could potentially have a lower latency option. This approach can be seen in mobile edge computing, where processing data on a mobile device would consume too much power, and doing so in the cloud would lead to high latency [8; 9; 10]. Edge datacentres have been demonstrated in video processing and augmented reality applications [11; 12; 13] where latency is an

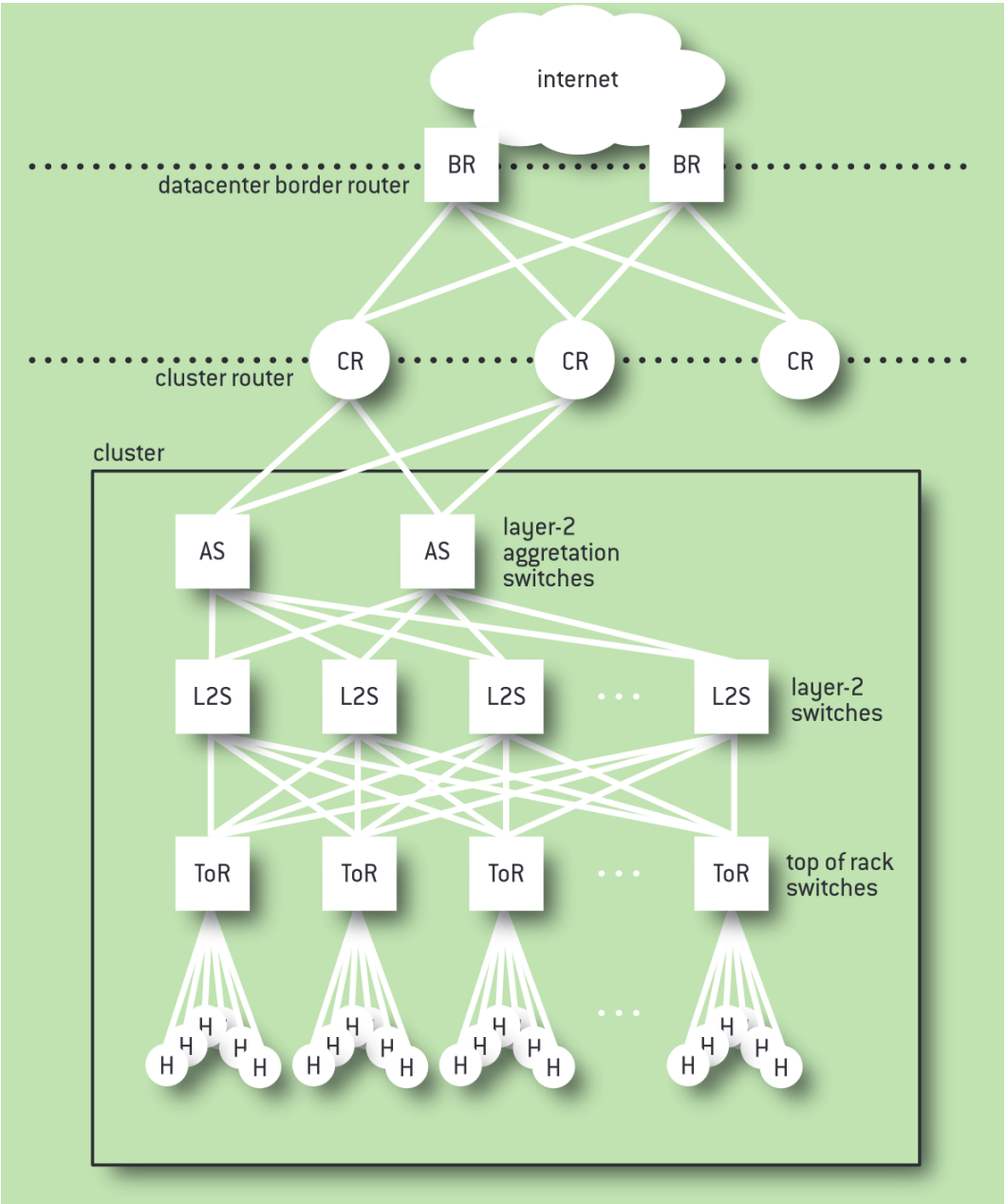


Figure 2.6: Example of datacentre network topology [100]

important consideration. Applications such as face recognition [11; 12] and video surviellance [17] have also been demonstrated on these platforms with significant latency improvements over traditional cloud offloading.

2.4.3 In-network computing

Another emerging method of computing offload is utilising elements traditionally used for moving data, such as switches, gateways, routers, or NICs to perform extra computing on top of their standard networking functions. These elements are discussed in Section 2.4.4. This technique is often termed 'in-network' computing. Cisco utilise this approach in an edge processing context in what they call 'fog computing', where endpoints at the network edge utilise local network elements to assist with computation [102]. In-network computing can also be utilised within datacentre networks, implemented within the dense switching network. There are several benefits to this approach. Firstly, for edge processing applications it provides a computing resource extremely close to the endpoint. The trip from an endpoint to a centralised datacentre over the Internet, or even to a geographically closer edge datacentre or cloudlet is much larger in comparison. For applications with very tight latency constraints, such as closed-loop control of industrial equipment, this is vital. The same holds true for datacentre applications — processing can be carried out within a few hops, and potentially avoid having to be processed on another server, circumventing the latency and jitter of the server's network stack. Keeping processing local using in-network processing can also make applications more resilient to the variable latency caused by dense bursty traffic generated elsewhere in the datacentre. Another benefit of in-network processing is that it can reduce the load on the network, as data is terminated or reduced once processing is complete.

Devices such as network switches and gateways are extended to perform additional data processing as well as their network functions. This technique has been demonstrated to result in a reduction in data and execution latency in map reduce applications [103]. A key-value store implemented on an FPGA based NIC and network switch outperformed a server based implementation [22]. In-network computation using programmable network switches for a consensus protocol was demonstrated in [104].

The result of the computation is typically smaller than the data used in it - for example a neural network may return a class label, or a filtering application removes extraneous data. This can reduce the network load, and the resulting impact on other applications. Additionally, it makes use of elements already utilised within the network, reducing the need for adding extra servers or components, which can reduce space overheads, costs and idle power consumption. In-network computing is an emerging area, but is becoming more readily available due to advancements in programmable switches, smartNICs and the utilisation of FPGAs within networking contexts.

2.4.4 Network elements

Data is moved from machine to machine through a network, passing between other machines and dedicated network elements. Most modern systems primarily utilise the Ethernet networks [105]. In these networks, data is transmitted in bitstreams across either twisted pair or fiber-optic cables between switches. Ethernet networks are switch-based, where machines only communicate directly with the closest switch, and not with other machines. Each device in the Ethernet network has a MAC address, which is used to direct data to the appropriate machine.

Data enters and leaves a device through a network interface, controlled through software running on the device. Embedded devices usually have a network controller integrated circuit (IC) on board to allow for the physical connection to the network through cable or WiFi, and handling of the physical layer protocol. Servers usually use a network interface card (NIC), which are peripheral component interconnect express (PCIe) expansion cards that contain the network controller IC and supporting circuitry and buffers. Upon reception of a packet at the network interface, it is written to a ring buffer in the host via direct memory access (DMA) transfers, and an interrupt is triggered. The operating system then moves the packet into an input queue, and then processed in software by the CPU. This processing involves examining and extracting data from the various headers for each protocol layer in sequence. Once processed the packet is accessible by applications through the sockets interface. A server may host several NICs with varying numbers of ports. Modern NICs may also integrate internet protocol (IP) and transmission control protocol (TCP) stacks on the card before it reaches the host machine. Offloading this processing away from the host and onto the NIC can have significant throughput and latency benefits.

Switches are fundamental network elements that facilitate many-to-many communication between machines. They forward frames from input ports towards the appropriate output port to reach the target machine on the local network. A switch will have input and output arbitration and buffering logic either side of the main switch fabric. Most switch fabrics are implemented using a crossbar design, where each input has a potential connection to each output, but only one input or output can be connected to another at one time [106]. The links through the crossbar matrix are bufferless, and buffers are placed either at the input or output. Unmanaged switches offer mostly simple movement of frames between ports, although can sometimes offer additional features such as basic prioritisation of selected traffic and diagnostic abilities. They offer limited reconfigurability, and are designed to just be deployed out of the box. Managed switches on the other hand

can be reconfigured during deployment, and offer a more comprehensive feature set. These can include additional security, traffic control and quality-of-service capabilities. Smaller, unmanaged switches are more likely to be found closer to the network edge, while more capable managed switches are more likely to be used to aggregate larger groups of links from smaller switches. Routers perform a similar function, but usually work with the layer 3 protocol (for example IP), and are used to connect multiple networks, to each other.

2.4.5 FPGAs for network applications

FPGAs have seen extensive use in networking applications. The growth of software-defined networking, and the need for flexibility in modern networking devices mean that traditional ASIC implementations are no longer viable. On the other hand, software implementations of these functions running on CPU-based hardware cannot meet the strict throughput and latency requirements, especially considering the growth of 40Gb and 100Gb Ethernet in datacentre environments. Specialised network processing units (NPU) are available as a compromise between these options, but still pose a restriction on what functionality can be implemented. FPGAs can provide the both high performance and sufficient flexibility that are needed in modern networking contexts, in a range of applications.

NetFPGA is a platform designed for networking related FPGA research [107]. It is a development board hosting a Xilinx FPGA, memory, PCIe, and 1Gb Ethernet interface. Further revisions of this board host bigger FPGAs, more memory and more Ethernet interfaces [108]. The NetFPGA SUME [109] board is a further revision which has four 10Gb Ethernet interfaces and a Virtex 7 FPGA. The boards come with a library of IP cores to ease development of networking applications.

Packet parsing and classification

A fundamental component of modern networking appliances is packet parsing and classification. This involves identifying and extracting information from fields within packet headers, often nested within headers of other protocols, and classifying packets into flows based on this information. This can help with enforcing security, routing flows to the appropriate network application and meeting quality-of-service constraints. FPGAs are an attractive platform for this function due to the ability to ingest and process packets at high data rates, while retaining the flexibility to be reconfigured to support new protocols or classification policies. Packet parsing at 400Gb/s is demonstrated in [110]. The design implements a pipeline of generic

parsing elements that are configured through microcode by the stages previous to it. A high-level language is also provided to be able to express header formats, which then automatically configure the FPGA parsing logic. The parser presented in [111] aims to reduce the area consumption and latency for high throughput parsers. Embedded applications may have less resources available and require a smaller footprint parser, and applications like high frequency trading require extremely low latency. This architecture allows the user to control the number of pipeline stages to suit their application, and allows for packets to share data words, making better use of the wide bus widths that must be used to achieve high throughput. One of the challenges faced by packet processors implemented on FPGAs is that the clock rate is slow relative to the data rates required, so large bus widths are required, consuming large amounts of resources. The design in [112] utilizes an FPGA with a hardened 'network on a chip' interconnect, that provides a high-speed interface across the FPGA. This allows for lower bus widths and area consumption. Packet classification typically sorts packets into 'flows' based on the contents extracted from the packet headers. Typically this is done through table lookups that match header content to class labels. The memory footprint of the classification ruleset is identified as a key bottleneck in [113], where the authors develop an algorithm that reduces the required memory by breaking down the ruleset into smaller rulesets that can be cross referenced. The properties of the ruleset are exploited to reduce the memory footprint in [114]. An architecture that allows for classification at 40Gb/s data rate is presented in [115]. Memory efficiency is sacrificed in this design in exchange for throughput. This has the added benefit of not relying on the format of the ruleset, making it more applicable to a range of applications.

Network Function Virtualisation

Network function virtualisation involves the replacement of traditionally specialised hardware middleboxes that performed network functions with commodity, general-purpose platforms. Functions can be dynamically allocated to these platforms in order to respond to changing requirements, without having to replace or alter hardware. FPGAs have been highlighted as attractive platforms for NFV as software implementations of these functions have performance limitations, while ASICs do not have the required flexibility. Intrusion detection systems have been implemented on FPGAs in [116] and [117], demonstrating improvements in throughput compared to software implementations. Firewalls have been shown in [118] and [119], using the packet classification techniques discussed above. Deep packet inspection has seen a recent surge in interest, with FPGA implementations being developed in several

works such as [120] and [121]. There has also been work into developing general platforms for FPGA-based NFV, and integration into NFV infrastructure. The architecture in [122] implements a static region consisting of protocol independent switching units, and a partially reconfigurable region that can be loaded with custom network functions. The work in [123] details a framework for the deployment of FPGA networking functions alongside software VNFs within a heterogenous system.

Switching

Recently FPGAs have seen increased interest as platforms for network switching. Devices are being manufactured with more transceivers, memory, and logic resources, and the ability to reconfigure the design to account for new functionality is becoming more favourable. This is decreasing the gap between FPGA and ASIC switching solutions. The feasibility of high-speed network switching using FPGAs is investigated in [124], for up to 50Gb/s per port. A 16×16 156Gb/s aggregate switch is implemented on a Virtex-6 in [125], making use of the hard RAM blocks available on the FPGA. These hardened memory resources can be run a higher clock rate than the general-purpose fabric, and in this design are used to perform a majority of the switching. The authors evaluate the design to have comparable performance to ASIC switches of similar scale. The architecture in [126] utilises a hardened network on a chip to achieve a 16×16 switch with over 900Gb/s aggregate throughput. They use a 64×64 mesh NoC which provides high-speed links across the FPGA, allowing for faster transport than the general-purpose fabric. This technique consumes less area, power and memory resources than the SRAM design. A similar NoC based design is demonstrated in [127]. A 16×16 900Gb/s aggregate throughput is achieved in [128] without the use of a NoC, and is demonstrated to be more resilient to difficult traffic patterns. It does this by removing the crossbar switch structure used in the other FPGA switches discussed, and instead uses a pipelined algorithm to process incoming packets and efficiently organise them at output queues.

Software-defined Networking

While these works focus on the implementation of the switching fabric, other works related to FPGA switches focus on their integration into SDN infrastructure. SDN allows for the control plane of switches to be decoupled from the data plane. The data plane of ASIC devices only allows for configuration between a set of parameters and functions defined at manufacture, while FPGAs allow for a reconfigurable data plane, allowing even more flexibility. An SDN switch compatible with Openflow, an

open SDN standard, is demonstrated in [129].

SDN data planes are often configured using high-level, platform and protocol independent languages. Thus there is a broad set of work focused on allowing these languages to target FPGA platforms. P4 in particular has seen significant interest. P4FPGA [33] extends the standard P4 compiler to generate verilog code that can the be used to generate FPGA logic. To evaluate the compiler, L2/L3 forwarding, Paxos, and a financial trading protocol were implemented. An implementation that reduces the resource overhead on the FPGA is demonstrated in [34]. A full workflow, from the compilation from the P4 program to the bitstream generation, to testing on the hardware is presented in [130]. A P4 to FPGA workflow is used to implement the Paxos consensus protocol in [131]. This is an application typically deployed on standard Servers, but here it is implemented in a programmable data plane within network elements, reducing the number of network transfers and reducing latency significantly.

All of this work demonstrates that FPGAs are well-suited to packet processing, independent of a CPU host, and are present within networking infrastructure. These platforms can be extended to perform additional computation. This provides new opportunities to distribute processing into the network, but increases the complexity of decided where to place tasks.

2.4.6 Cloud Computing

Cloud computing is usually used to describe the flexible, 'on-demand' provisioning of computing and storage resources from a datacentre to clients over the internet. This can be a private cloud, where the datacentre is owned by and serves clients only from the same business or enterprise, or a public cloud, where the datacentre resources are not owned by the same party as the client. Public clouds operate under a service-oriented payment model, where customers pay as they use resources. The most common service model is 'infrastructure-as-a-service' (IaaS). Physical hardware is virtualised, which means that multiple guest operating systems can run on the same host. A hypervisor is a software layer running on the host that enables this. A guest OS is often run as a virtual machine (VM), and all VMs on a host share the same processor, memory and I/O. The Cloud service provider allows users to elastically create VMs in the remote datacentre.

The cloud model provides numerous benefits for users. It gives users access to a range of compute resources without them having to purchase or manage any of the physical hardware. Deployments can be also be scaled up or down dynamically depending on requirements. For applications where data is generated at resource

constrained nodes such as IoT sensors or mobile phones, the cloud provides more capable resources that could be used to offload processing.

There are several primary public cloud providers. Amazon provide a cloud platform called Amazon Web Services (AWS). AWS has a large range of services available for many use cases. The main compute service offered is their elastic compute cloud (EC2), which allows users to generate VMs that can be accessed remotely [132]. Capability can be scaled dynamically, with resources such as CPU and memory being flexible. AWS offers a storage service called S3, which allows for persistent storage in the cloud. Their F1 service provisions VMs with up to 8 attached FPGAs, connected over PCIe [133]. Microsoft have a competing platform called Azure [134], with similar functions to AWS. Google also have a cloud platform called Google Cloud, again providing many of the same basic features [135].

2.4.7 FPGAs in the cloud

In recent years, there has been a surge of interest in deploying FPGAs in the cloud, as a provision-able, shareable resource. The goal is to allow users to remotely access FPGA resources on demand to accelerate compute intensive applications. This growth has been driven by better PCIe-based hosting and the availability of more accessible, software-like development flows.

Commercial cloud FPGA platforms

At present, FPGAs have been integrated into several public commercial cloud platforms. Amazon's AWS platform allows users to directly access an FPGA and implement their own logic, as an extension of their EC2 elastic compute service [133]. EC2 allows users to rent virtualised compute and memory resources. Users can rent an F1 variant of one of these instances, which attaches up to 8 FPGAs. The FPGAs are attached via PCIe to host servers, and are attached to each other in a ring topology using dedicated interconnect, for high-speed inter-FPGA transfer. Developers can design accelerators using the traditional workflow or higher level languages. Amazon provide an interface IP core that user logic must integrate with, and the resulting design is submitted to the AWS servers as an Amazon FPGA image (AFI). This design goes through several stages of checks and can then be deployed to the FPGA instance. The F1 service only allows for the provisioning of entire FPGAs, and these are required to be attached to a CPU instance. There is no virtualisation of the FPGA resources. FPGAs are treated as slave accelerator devices rather than standalone devices, and have no direct connection to the network.

Baidu smart cloud has FPGAs computing instances available in public beta [136]. Each instance has exclusive access to a monolithic FPGA. As part of the service they offer a range of IP cores and software components to simplify development. Similarly to Amazon F1, Baidu provide a static shell and software APIs to handle interfacing between the host and the FPGA, meaning the user only has to develop the computing logic for their application. Huawei cloud has an FPGA cloud server platform [137] which also has up to 8 FPGAs attached to a compute instance over PCIe, with a high-speed network connection between FPGAs.

Microsoft's competing cloud platform Azure also offers FPGA-acclerated instances. In contrast to AWS, which provisions the user an entire FPGA to implement their own designs, Azure offers no direct access to FPGA resources. Based on the work presented in project brainwave [138], Azure gives users the option to deploy pre-trained DNNs to FPGA accelerators, for a specified set of supported model architectures. Users only provide the DNN model parameters rather than any hardware design, and load it onto Microsoft's fabric of FPGA resources within their datacentre using a high-level software API. Another FPGA related cloud service provided by Microsoft is Azure Accelerated Networking. This gives users the option of offloading the network stack to an FPGA smart NIC on the host, allowing the VM direct access to the network interface and bypassing the hypervisor virtual switch implemented in software on the host. This results in greatly reduced latency and jitter [139].

Google cloud offers tensor processing unit (TPU) instances to accelerate DNN inference, but no FPGA platforms. IBM have a cloud platform called Supervessel [140] designed for development and educational use, which utilises FPGA accelerators connected through PCIe to the hosts. Biadu utilise FPGA accelerators to speed up web search [78].

Academic works

The current commercial cloud offerings are ultimately limited, and FPGAs are offered either as a monolithic resource, are not directly available to users, or are restricted to specific sets of applications. Academic research has been focused on overcoming these limitations, and making FPGAs an accessible, virtualised resource in the cloud. Research addresses several key challenges in this area, mainly efficient sharing of resources, and integration of FPGAs into existing cloud management infrastructure.

Resource sharing

Allowing FPGAs to be shared across multiple applications and users in isolation is one of the main objectives of research surrounding FPGAs in the cloud. Offering FPGAs as monolithic resources in the cloud is ultimately not resource efficient, and if a user design takes up less than the total resources available, the extra space is essentially wasted.

Most commonly implemented is an architecture comprising a static 'shell' made up of control and interface logic, with application logic being implemented in reconfigurable regions at runtime. An example can be seen in [141], which partitions the reconfigurable region into virtual FPGAs (vFPGAs) that can host accelerators. The shell implements PCIe connectivity, and manages the arbitration of access to communication and memory resources between the vFPGAs. The hypervisor running on the host manages scheduling and controls the loading of accelerators into the vFPGAs. This design was primarily developed with streaming applications in mind that make limited use of on board memory. Another shell for virtualised PCIe-based accelerators is presented as part of an end to end FPGA cloud platform in [142]. Again, the shared PCIe logic and reconfiguration controller is implemented within the shell, with partially reconfigurable regions used for application logic. This architecture also implements scheduling logic within the shell, which implements policies described in software from the host. Security and error detection logic is also implemented in the shell. A similar architecture can be seen in [143]. This design adds a local soft-core processor to allow low latency management of FPGA resources, and faster access to memory shared with the accelerators. Additionally, this architecture includes on board memory virtualisation, and allows for logic and memory resources to be dynamically allocated to applications depending on the characteristics of the workload. The paper evaluated the design with multiple applications commonly found in the datacentre and demonstrated that sharing the FPGA across multiple applications provided performance benefits compared to monolithically hosting each application in series, due to reduced reconfiguration time.

These approaches treat the FPGA as a coprocessor to a CPU. This presents several issues with cloud deployments. For one, the number of FPGAs available is dependent on the number of hosts and the amount of PCIe interconnect presented. Furthermore, any host failures render all attached FPGAs unusable. Finally, data must travel through the host to reach the accelerator, leading to latency penalties. This has lead to several works arguing in favour of deploying FPGAs as independent resources, connected directly to the the datacentre network. The shells for these accelerators implements a high-speed network interface instead of PCIe. The

network attached accelerator in [144] has resource management logic in the shell that receives commands over the network from centralised management software. This work is extended in [145] in the context of multi-FPGA fabrics. In this case, a master server can co-ordinate the distribution of tasks to multiple network attached accelerators. Feniks [146] again allows for multiple slots of user logic, but allows for management over the network or via a PCIe host connection.

This thesis attempts to quantify the overheads of these approaches.

Integration with cloud management

Along with allowing FPGAs to be shared by multiple users at the FPGA level, the other key challenge is integrating them within existing cloud management infrastructure. This means allowing VM instances to interact with the accelerators, or allowing FPGAs to be managed as traditional software resources are. One example of this is demonstrated in [147], where an FPGA accelerator is integrated with the popular Xen hypervisor. The accelerator is connected to a host with PCIe, and the extra virtualisation layer adds minimal access overhead. The platform presented in [142] is integrated with the Openstack cloud management framework. Openstack is in fact a common target for FPGA cloud integration. Several works have presented standalone, network attached accelerators that can be managed through Openstack. One of the first was [148], where standard Openstack commands can be used to set up or tear down accelerators. Similar approaches are detailed in [144] and [145]. The frameworks present FPGAs as generic cloud resources, and allow for the control of multiple independent devices. Galapagos is a full stack framework for integrating FPGAs in the cloud [149], which automates and simplifies the deployment of multi-FPGA clusters. Under this framework, FPGAs are directly connected ot the datacentre network as standalone devices. The framework manages the physical setup of device and provisioning from a pool of shared resources.

Summary

FPGAs are utilised within the network infrastructure for various networking functions. These platforms could be extended to perform additional computation, in addition to these base tasks. This increases the number of locations within a network that tasks can be allocated to, making manual placement significantly more complex.

FPGAs are also present within the cloud ecosystem. The technologies developed to facilitate this allow for the sharing of FPGA resources and integration

within a networked environment. There has been no work examining the overheads of the virtualisation methods used to achieve this. These overheads are examined in part in chapter 4 of this thesis.

2.5 Mathematical Modelling

In the previous sections it is noted that the increased availability and heterogeneity of hardware accelerators, and the growing number of ways they can be distributed throughout a network make manual evaluation and determination of performant deployments of tasks and hardware impractical. Mathematical representation of this scenario is therefore important.

Fundamentally, this is a graph embedding problem - a set of connected tasks must be allocated to a graph of networked nodes. This sections details existing modelling solutions, and why a new method is required.

2.5.1 Sensor network modelling

Models that deal with in-network processing go back to early research on data aggregation in wireless sensor networks. These networks were often meshes of sensor nodes with very limited computational resources, for example simple microcontroller based systems such as Mica motes [150]. Computations were simple, mostly consisting of basic aggregation functions such as averages and maximums. The models' goals were to allocate these functions to sensor nodes to minimise energy consumption, thus maximising the lifetime of the sensor nodes with limited power supplies and in difficult to access locations. Due to the low power-consumption hardware and simple compute functions, along with energy hungry wireless communication between nodes, computation was generally assumed to be cheaper in terms of energy than communication. This meant that these models for the most part push computation towards the leaves to a substantial degree in order to reduce communication costs as much as possible. These models include TAG [151], directed diffusion [152], EADAT [153], and MERIG [154]. These either assume uniform sensor nodes or do not account for the computation time at each node, rather focusing on reducing network load and minimising energy consumption.

2.5.2 Distributed Stream Processing Models

The allocation of streaming tasks to networked processing nodes has been explored in a variety of existing work. Applications are represented as a graph of tasks

with edges representing dependencies, while networks are represented as a graph of compute nodes with edges representing links.

Earlier models such as Aurora/Medusa [155] focused on load balancing in task placement, primarily for the allocation of tasks to multiple servers in a datacentre environment. However, network costs are not modelled, making them unsuitable for scenarios that consider larger scale networks where communication and network costs are more significant. Work on more network-aware placement [156; 157; 158; 159; 160] was tailored towards networks of machines that are more widely distributed, and include network utilization and latency in their formulation. These models are all focused on placing operators to optimise specific objectives, for example bandwidth utilisation, meaning that they aren't generalisable when wanting to model a range of different performance metrics. Since these online optimisations are run dynamically, the models are significantly simplified to minimise their impact on the application. These models consider homogeneous processor platforms and do not support alternative hardware platforms with different computational models and metrics.

Recently, more generalised placement models have emerged [161; 162; 163]. These focus on creating a general representation of the operator placement problem, developing formulations based on integer linear programming instead of focused heuristics. They are still limited as they assume a fully connected cluster of machines, and their models of computing resources and tasks are coarse grained. The work in this thesis is concerned with a scenario where hardware acceleration may be utilised at certain computing nodes, using a different computational model to that of a processor, which is considered solely in these models.

2.5.3 Summary

Most of the existing work regarding task placement within a network of connected compute resources are focused on dynamic runtime allocation. The time taken to generate a solution is therefore a major constraint. A static allocation model does not have runtime execution constraints, so can be designed with more complexity, including more variables and taking into account more detail. Existing works also assume fixed hardware at each node.

As demonstrated in this chapter, hardware acceleration is more commonplace, and can be deployed at many locations within the network. A different modelling approach is needed that not only allows for the allocation of tasks, but the allocation of additional accelerator hardware at existing network nodes to facilitate this. This model could be used statically, pre-deployment as part of a design phase,

and allow for more detailed modelling of both tasks and hardware.

Chapter 3

Modelling distributed computing with heterogeneous hardware

3.1 Introduction

Distributed data processing applications involve the processing and combination of data from distributed sources to extract value, and are increasing in importance. Emerging applications such as connected autonomous vehicles rely on complex machine learning models being applied to data captured at the edge, while also involving collaboration with other vehicles. Further example applications include factory automation [164], smart grid monitoring [165], and video surveillance and tracking [166]. Such applications present a challenge to existing computational approaches that consider only the cloud and the very edge of the network. Computationally-intensive algorithms must now be applied to streams of data, and latency must be minimised. In these applications, data sources transmit streams of data through a network to be processed remotely, with a focus on continuous processing, and potentially involvement in a feedback loop, as opposed to other applications that involve large-scale storage and delayed processing. Latency, the time taken to extract relevant information from the data streams, and throughput, the rate at which these streams can be processed, are key performance metrics for such applications.

Centralised cloud computing is often utilised in these scenarios, since the data sources do not typically have adequate computing resources to perform complex computations. Applications also rely on the fusion of data from multiple sources,

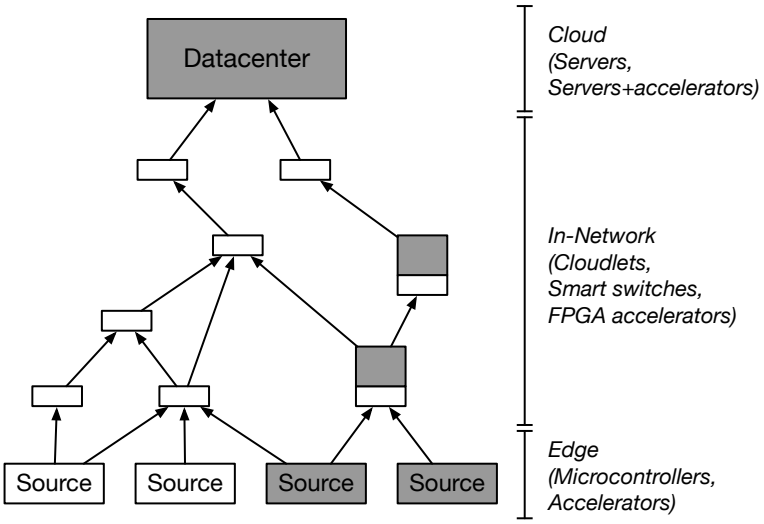


Figure 3.1: An example of the type of networked system that the proposed model targets. Shaded nodes can perform computation.

so centralised processing is useful. The cloud also offers benefits in scalability and cost, and has been shown to provide benefits in applications such as smart grid processing [165; 167] and urban traffic management [168].

However, many emerging streaming applications have strict latency constraints, and moving data to the cloud incurs substantial delay. Furthermore, while the data generated by sources can be small, a high number of sources means that, in aggregate, the volume of data to be transmitted is high. For example, in 2011, the Los Angeles smart grid required 2TB of streamed data from 1.4 million consumers to be processed per day [165]. Some applications, such as those dealing with video data, must also contend with high-bandwidth data requirements.

These limitations have led to an increased interest in 'edge' or 'fog' computing, a loosely-defined paradigm where processing is done either at or close to the data sources. This could mean at the source, such as on a sensor node with additional processing resources [169]. It can also encompass performing processing within the network infrastructure, such as in smart gateways [170], or in network switches or routers. Cisco offer a framework that allows application code to be run on spare computing resources in some network elements, and Ethernet switches from Juniper allow application compute to be closely coupled with the switching fabric.

Edge computing can also include the concept of 'cloudlets', which are dedicated computing server resources placed a few hops away from the data sources. These can vary in scale, from a single box placed on a factory floor to a small scale datacenter comprising multiple networked machines. While the data sources

themselves may not have the required computing capabilities, these resources can support complex applications and are accessible at shorter latencies than a remote cloud [171].

In complex applications, it is likely that some processing, such as filtering and pre-processing can be performed at the edge, greatly reducing the volume of transmitted data, and additional processing and fusion of data can be carried out in the cloud. The benefits of this approach are that latency sensitive parts of the application can be done locally, while more computationally-intensive operations that may require more processing power or additional data can be done centrally. Stream processing applications are well suited to being partitioned and distributed across multiple machines, as is common in stream processing frameworks such as Apache Storm and IBM Infosphere Streams. Additionally, cloud service providers such as Microsoft Azure have edge analytics platforms that allow processing to be split between the cloud and the edge.

Edge and in-network computing is an emerging area. Cloudlets have been utilised for image processing applications [11; 12] and augmented reality [13]. Platforms such as Google's Edge Tensor Processing Unit demonstrate that there is a trend towards moving complex computation closer to the data source. In-network computing has seen application for network functions, machine learning [103], and high data rate processing [22].

In order to explore the implications of distributing application computation across a network of heterogeneous compute platforms, a suitable model is needed. This would allow for the evaluation of different deployment strategies using metrics such as throughput and end-to-end latency. Existing models that deal with placement of processing on distributed nodes do not consider hardware resources, varied connectivity, and application features together.

To this end, this chapter will propose a generalised formulation that can represent applications and target networks with heterogeneous computing resources. It supports reasoning about in-network and near-edge processing scenarios that are emerging including both general processor-based machines and hardware accelerator systems.

Figure 3.1 summarises the application scenario of interest, giving an example of the type of networked system that the proposed model targets. Edge nodes such as sensors and microcontrollers transmit data through a network towards centralised computing resources. In a traditional cloud computing setup, only the central resources perform computation (shaded). In edge computing, the edge nodes are capable of performing some computation (shaded). In-network computing al-

lows some tasks to be performed in the network as data traverses it, using smart switches (shaded).

3.2 Contributions

The key contributions of this chapter are:

- A model for evaluating different in-network computing approaches is developed, encompassing:
  - Multiple levels of network structure, unlike existing models that focus on clusters of machines.
  - Hardware heterogeneity including accelerator platforms, and the resulting differences in computing and networking.
  - Realistic representation of performance metrics, alongside energy and financial cost.
- The model is used to examine a case-study scenario and draw general lessons about in-network computing on different platforms using a set of synthetic applications.

3.3 Related Work

The allocation of streaming tasks to networked processing nodes has been explored in a variety of existing work. Applications are represented as a graph of tasks with edges representing dependencies, while networks are represented as a graph of compute nodes with edges representing links. Those works are discussed in Chapter 2.5.

In contrast to these works, the model proposed in this thesis is not focused on finding an optimal allocation of tasks to a given set of resources at runtime. Instead, it is to be used to investigate the implications of placing computing resources at different locations in a network and to understand the benefits and costs of doing so. Since the model is not concerned with dynamic optimisation of operator placement within a time constraint, the model can include more fine grained detail for tasks and hardware, accounting for hardware acceleration, heterogeneous resources required by tasks, the financial cost of adding additional compute capability to network nodes, and energy consumption. The model also considers the networked system as a whole, from the sensor nodes to the datacenter, instead of focusing on a cluster

of computational servers. The focus of this work is not limited to optimisation, but rather an analysis of different distributed computing paradigms in the context of streaming applications. The model can still be used for an optimisation however.

3.3.1 Edge/Fog Computing

In response to increasing demand for low latency in distributed streaming applications, efforts have been made to move computation closer to the data source, or the 'edge' of the network. Where processing occurs varies, and it is rare that the application is entirely pushed to the edge. Typically operations such pre-processing and filtering take place at the edge, with aggregation and decision making centralised. This approach has been applied to domains such as smart grid, radio access networks, and urban traffic processing [172; 173; 174]. The model developed in this chapter is capable of representing this scenario.

In-network computing is another emerging paradigm in which traditionally centralised computation is distributed throughout the networking infrastructure. As the capability of this hardware improves, this method in which networking elements are used for both moving data as well computing, is becoming more viable. Extending such capabilities to broader applications requires the ability to analyse applications composed of multiple dependent tasks and determining how to allocate these to capable nodes. The proposed model allows this to be explored in a manner not possible using existing distributed computing models.

3.3.2 Hardware acceleration

A primary motivation for this work is the increasing complexity of applications, growing volumes of data, and more widespread availability of alternative hardware such as GPUs and FPGAs that can boost the performance of these applications. As discussed in Chapter 2, recent work has demonstrated the use of hardware acceleration for a variety of algorithms relevant to networked systems. To reflect the trend towards heterogeneity, the model proposed in this thesis encompasses the idea of distinct hardware platforms with different computational characteristics. This further differentiates this work from others that consider only traditional processor based compute architectures.

3.4 Scenario and Metrics

The scenario of interest comprises a set of distributed data sources producing continuous streams of data, connected through a network comprised of intermediate nodes (for example gateways, routers, or cluster heads) to a central data sink, such as a datacenter. These data sources could be cameras, streams of documents, environmental/industrial sensors, or similar. An application consisting of a set of tasks and their dependencies processes these streams to make a decision or extract value. These tasks operate on the different streams of data, and some combine information from multiple (possibly processed) streams. Individual tasks affect the data volume through a reduction factor that determines the ratio of input data to output data, which reflects the properties of many stream processing tasks. An example of such an application is a smart surveillance system that monitors video streams from many cameras to detect specific events. Video streams can come from a mix of fixed cameras and mobile platforms, with different resolutions, frame-rates, and interfaces, requiring different amounts of processing. The application uses processed information to adapt how the cameras are deployed and positioned.

In order to evaluate alternative allocations of resources and tasks, following key metrics of interest are considered, with some explanation of how they are impacted below. The comprehensive formulation of these metrics is provided in Section 3.6.

3.4.1 Latency

Latency is important when data is time-sensitive. Fast detection of an event may have safety or security implications, or in some applications, there could be real-time constraints. In this case-study, transmitting all video streams to the cloud introduces large communication delays and competition for resources in the cloud can add further latency. Performing computation closer to the cameras, whether at the cameras or in network switches can reduce these communication delays, and distributing the tasks to different network nodes reduces the delays from sharing centralised resources. Even with less powerful hardware, latency can improve as a result of this stream processing parallelisation.

3.4.2 Bandwidth

Processing sensor data often reduces the size of data, outputting filtered or aggregated data, or simple class labels. Hence, if this processing is performed nearer to the data source, bandwidth consumption further up the network can be reduced sig-

nificantly. There may also be scenarios where early processing can determine that a particular stream of data is useless, and hence further transmission can be avoided. In this example, some cameras may use low resolutions or frame rates, and hence be less costly in terms of bandwidth, while others might require significantly higher bandwidth, which would be more efficiently processed nearer to the cameras. It is clear once again that this decision depends on the specific application and tasks.

3.4.3 Energy

Energy remains a key concern as cloud computing continues to grow; the power consumption of datacenter servers and the network infrastructure required to support them is significant. One approach vendors have taken to try and address this is to introduce heterogeneous computing resources, such as FPGAs, to help accelerate more complex applications while consuming less energy. However, these resources add some energy cost to the datacenter, in the hope that this will be offset by significantly increased computational capacity. There is similarly an energy cost for adding accelerators in the network infrastructure but this is likely less than the cost of full server nodes, and leads to a reduced load on the datacenters as they then only deal with processed data. However, it is clear that energy consumption is heavily dependent on where such resources are placed. It is also possible that energy constraints at source nodes can impact what can be done there. In this example, battery-powered drones carrying cameras may have constrained power, so performing more computing there may not be viable.

3.4.4 Financial Cost

Adding computing capabilities to all data sources is expensive, especially where the tasks to be performed are computationally expensive, possibly requiring dedicated hardware. In this example, the cameras would have to be smart cameras with dedicated processing resources attached, and this is likely to increase cost significantly. While centralising all computation is likely to be the cheapest solution in terms of hardware, placing some computation in the network can come close to that cost, while offering significant benefits in the other metrics.

3.5 Proposed Model

The proposed model defines a network topology, task/operator graph, and hardware platforms. Tasks and hardware platforms can be allocated to network nodes, and

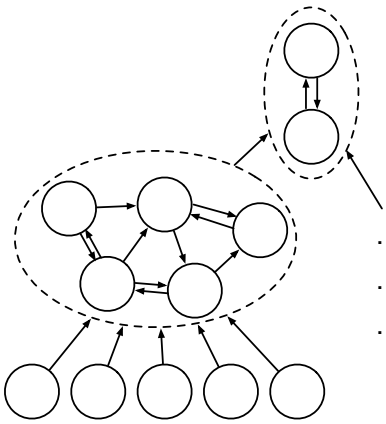


Figure 3.2: Nodes in the network graph can represent a single device or a cluster of networked devices.

values for the previously mentioned performance metrics can be calculated. The network communication topology is assumed to be pre-determined, though not the hardware at the nodes or the task allocation. The model is flexible enough to be used in a range of situations.

The logical topology of the network is represented as a graph, $G_N = (N, E_N)$, where N is the set of network nodes, with bidirectional communication across the set of edges between them, E_N. Application data travels through these nodes and edges towards a central sink. A node can represent either a single machine in the network, such as a gateway, switch or, server, or a 'tier' or 'level' of the network infrastructure. In this case a node represents multiple machines but the connectivity between them is not modelled at the higher level in the graph (see Figure 3.2). Using this representation allows the network topology to be represented in a tree structure as suits the application models considered.

To represent the application, a directed acyclic graph (DAG) is used to define the relationships between tasks, $G_T = (T, E_T)$, where T is the set of tasks and E_T defines the dependencies between them. G_T is a tree structure with a global task at the root, with other nodes representing sub-tasks, such as aggregations and pre-processing. This task model is based on the stream processing model, where data is processed per sample as it arrives. DAGs are commonly used to model stream processing flows in other works, which are acyclic in nature as data is processed as it arrives with minimal storage.

Each task $t \in T$ can be assigned to a node through an *implementation*. Implementations are pieces of software or hardware logic that can perform the required task. This allows for selection between software implementations and hardware ar-

chitectures that may have different benefits and drawbacks. This is in contrast to previous work which typically do not allow for the possibility of alternative implementations of a task. Implementations and tasks are treated as black boxes that take inputs and produce outputs, and have already been benchmarked to determine an estimate of processing time and energy consumption on a reference platform with no other tasks running. A set of platforms that can be assigned to nodes, P, to execute the tasks can have varying computational models and available resources.

3.5.1 Tasks

$T = \{t_1, t_2, t_3 ... t_T\}$ is the set of application tasks to be allocated to nodes in the network. Individual tasks represent functions to be carried out on a data stream. Together tasks represent the operations performed on each data stream, and specify how they are combined and manipulated to extract value. In this model, data is consumed by a task and transformed, with the result passed to the parent task. Task dependency is captured in the DAG, with each task unable to begin until all of its child tasks have been completed on a given instance of data—tasks with multiple children are typically aggregation operations. Each task $t \in T$ is defined by $t = (f_t, M_t, C_t, a_t)$.

- The set $C_t \subset T$ contains the prerequisite tasks for t that must be completed before task t can begin—its child tasks;
- $a_t \in T$ is the parent task of t, which cannot begin until t has finished.
- f_t is the reduction factor, where $0 < f_t \leq 1$. This parameter represents the amount that a task will reduce the volume of data it operates on;
- $M_t \subset M$ is the set of implementations that can implement the functionality of t;
- The data into (operated on by) a task t, denoted δ_t, is the sum of the data out from all sub tasks, $\delta_t = (\sum_{i=0}^{|C_t|} d_i)$;
- The data output from a task, d_t, to be processed by the task's parent task, is given by $d_t = f_t \delta_t$.

This representation of tasks supports different types of operations, for example, a filtering tasks that reduces a data stream, or aggregation tasks that merge multiple streams. Traditionally, aggregation tasks that process several data streams would have to be centralised but in this model they can be placed at intermediate nodes that has access to the requisite streams.

3.5.2 Implementations

$M = \{m_1, m_2, m_3...m_M\}$ is the set of all implementations, which are the pieces of software or hardware that implement the functionality of a task. Implementations can represent different software algorithms or hardware accelerator architectures that give the same functionality but have different computational delays or hardware requirements. Each task $t \in T$ has a set of implementations M_t, and each $m \in M$ is defined by $m = (t_m, \tau_m, R_m, h_m)$

- $t_m \in T$ is the task that is implemented by m;
- the set $R_m = \{r_{m1}, r_{m2}, r_{m3}...r_{R_m}\}$ contains the amount of each resource needed to be able to host the implementation, such as memory, FPGA accelerator slots, etc; These are integer values representing the quantity of each resource available.
- τ_m is the time taken for this implementation to complete the task it implements per unit of data, compared to a reference processor;
- $h_m = \{0, 1\}$ signals whether the implementation is software or hardware. A value of 0 is software, 1 is hardware.

3.5.3 Platforms

Platforms represent the systems in a network node that can carry out tasks. It can be defined that $P = \{p_1, p_2, p_3...p_P\}$ are the set of platforms that could be assigned to node $n \in N$. Each platform $p \in P$ is defined by $p = (e_p, c_p, w_p, R_p, h_p)$, where:

- e_p is the execution speed of the platform relative to a reference processor—this represents different processors having different computing capabilities;
- c_p is the monetary cost of the platform;
- w_p is the power consumption of the platform;
- $R_p = \{r_{p1}, r_{p2}, r_{p3}...r_{pR}\}$ is the set of resources available on the platform, such as memory, FPGA accelerator slots, etc. Resources are required by implementations;
- $h_p = \{0, 1\}$ indicates whether the platform runs software or hardware versions of tasks. A value of 0 means the platform is a processor that executes software, and a value of 1 means the platform is a hardware accelerator that executes application-specific logic. This is used to ensure correct allocation of software and hardware implementations.

Unlike existing work, this model makes the distinction between platforms that execute software code and hardware acceleration platforms such as FPGAs as they have different computational delay models, discussed in Section 3.6.1. Hardware acceleration platforms incur no latency penalty when multiple tasks are present on the same node, whereas software platforms do, as a result of contention for computing resources.

3.5.4 Network

$N = \{n_1, n_2, n_3 ... n_N\}$ is a set of the network nodes, for example sensors, gateways, and routers, or servers. Each $n \in N$ is defined by $n = (a_n, C_n, P_n, b_n)$, where:

- $a_n \in N$ is the parent node of n linking it to towards the central data sink;
- $C_n \in N$ is a set of child nodes of n linking it to towards the source(s);
- $P_n \subset P$ is the set of platforms that can be assigned to node n. For example, a large datacenter class processor that must be housed in a server rack cannot be placed on a drone;
- b_n is the outgoing interface between the node n and its parent node, and represents the data-rate in terms of data per unit time.

3.5.5 Sources and Data

$S = \{s_1, s_2, s_3 ... s_S\}$ is the set of data sources. Data is modelled as continuous streams, as this work is interested in applications that process and merge continuous streams of data. A data source could represent a sensor, database, video, or other source that injects a stream of data into the network. Each $s \in S$ is defined by $s = (n_s, t_s, d_s, e_s)$.

- $n_s \in N$ is the parent node of the source, the node where the data stream enters the network;
- $t_s \in T$ is the task to be performed on data being produced by the source;
- d_s is the amount of data in one instance from this source per period e_s;
- e_s is the period between subsequent units of data of size d_s entering the network.

The model assumes a constant periodic stream of data from the source, such as a periodic sensor reading, frame of a video, or set of captured tweets for example.

There are some systems that do not fit this model – for example where sensors may only send out data if there is some change detected. This case can still be represented in the proposed model, as the sensor is still continually capturing data as a source and the detection component can be modelled as a filtering task that reduces it.

3.5.6 Allocation Variables

Boolean variables represent the allocations of tasks and hardware to network nodes. $x_{nm} = \{0, 1\}$ represents the allocation of an implementation $m \in M$ to node $n \in N$. Similarly, $y_{np} = \{0, 1\}$ represents the allocation of platform $p \in P$ to node $n \in N$. $z_{nmp} = \{0, 1\}$ represents the allocation of platform $p \in P$, and task $m \in M$ to a node $n \in N$, using a set of constraints.

A summary of the symbols used in the model is presented in Table 3.1.

3.5.7 Constraints

Constraints are used to ensure correct allocation of tasks, platforms, and nodes.

Allocate tasks only once

$$\forall t \in T, \sum_{i=0}^{|N|} \sum_{j=0}^{|M_t|} (x_{ij}) == 1 \tag{3.1}$$

One platform per node

$$\forall n \in N, \sum_{i=0}^{|P|} (y_{ni}) == 1 \tag{3.2}$$

Resource availability

Allocations cannot exceed the available resources for the platform assigned to a node:

$$\forall n \in N, \forall e \in R, \sum_{i=0}^{|T|} \sum_{j=0}^{|M_i|} (x_{nj} r_{je}) \leq \sum_{k=0}^{|P|} (y_{nk} r_{ke}) \tag{3.3}$$

Additional constraints

The model allows for additional constraints to be added in order to better model a specific system or set of requirements. Constraints can be added to give certain

Symbol	Meaning
x_{nm}	allocation of implementation m to node n
y_{np}	allocation of platform p to node n
z_{nmp}	allocation of m and p to n
$u_{nm_1m_2p}$	allocation of m_1, m_2, and p to n
τ_{max}	maximum path delay
g	throughput
$K_t \subset T$	set of tasks lower than t in task sub-tree with t at the root
$K_n \subset N$	set of nodes lower than n in network sub-tree with n at the root
$D_s \subset N$	set of nodes on path from s to root node
v_{np}	1 if $p \in P_n$, 0 otherwise
$P_h \subset P$	set of all platforms that run hardware implementations
$P_s \subset P$	set of all platforms that run software implementations
H	set of all paths from leaves to root in task graph
$H_t \subset H$	set of tasks on path from leaf task t to root
O_{H_t}	Set of all other tasks not on path H_t
$I \subset M$	Set of all software implementations
ϕ_{mpt}	Time to complete task implementation on node
q	Data-rate of streams / tasks
$L \subset T$	Set of tasks with no child tasks
S_{K_n}	Set of all sources that lie beneath node n

Table 3.1: Summary of symbols used in formulation.

tasks deadlines, constrain bandwidths, restrict specific nodes to certain platforms, and more.

3.6 Performance Metrics

As previously mentioned, there are five main metrics of interest in this analysis. Latency, throughput, data rates and energy consumption, and financial cost. This section presents the formulation of these metrics, and discuss how the formulation allows each to be evaluated.

3.6.1 End-to-End Latency

The end-to-end latency is the total time between an instance of data entering the network and its root task being completed. For example, this could be the time between a sensor reading or image being taken and a fault or anomaly being detected. This value is of interest in time-sensitive applications such as those concerned with safety or closed-loop control, such as for industrial equipment, or coordinated control. The model incorporates several assumptions and behaviours that are relevant for this metric:

- Sources $s \in S$ produce continuous streams of data of an amount d_s, every period of time e_s. For the purpose of the formulation, the model considers a 'snapshot' of the network at any instance of time, and say that data is entering the system at this instant from all sources, of an amount d_s. The equation formed gives the latency of the data instances entered at the beginning of this 'snapshot';
- Only one software implementation can run at a time on a node. Software runs as soon as possible;
- Hardware implementations of tasks operate independently from one another so can operate in parallel;
- A task cannot begin until all of its child tasks have been completed;
- Tasks start as soon as all of the data required is available, as soon as possible, and once completed send the result to the next task as soon as possible;
- Communication and computation happen independently and can be parallel to each other;
- There is no communication time between tasks on the same node.

As tasks can only begin once their child tasks are complete, the root task of the graph $G(T, E_T)$ can only start once all paths to it are complete. The end-to-end latency is therefore equal to the longest path delay of the task graph, including network and computation delay.

Computation Delay

The time to complete one task on the node it is allocated to can be represented:

$$\phi_{mpt} = \tau_m e_p \delta_t \tag{3.4}$$

For a task t, implemented with m on platform p. The time to complete the task is the product of the time taken to complete the task per unit of data for that implementation, the amount of data, and the processing speed of the hardware platform. To find the end-to-end latency, the values of ϕ_{mpt} for each path in the task tree are summed, and the maximum value determined. This is because the total time taken to complete the tree is dependant on the slowest path.

In the case of software implementations, nodes are assumed to carry out one task at a time. So in the cases of multiple tasks being assigned to the same node, in the worst case scenario, a data instance must wait for all other tasks not in the path to finish before beginning the next task. Note that this applies even if a node supports concurrent software tasks, since it is assumed that multiple software tasks suffer degraded performance in proportion to the parallelism applied. Unlike some other works, which are only concerned with preventing the allocated tasks exceeding a measure of available resources on a platform, running multiple software tasks at once on the same node in the model presented in this thesis affects computational delay. For hardware implementations, no such assumption is made, as they can operate in parallel as separate streams since they are spatially partitioned, and so it is sufficient to only sum the path of interest, though available hardware resources are factored in, as discussed later. This distinction between software and hardware implementations of tasks better represents the growing trend of using alternative computing platforms to accelerate computation, compared with previous work that only accounts for software running on processors. Figure 3.3 shows this difference in scheduling for software and hardware nodes. On software nodes, tasks are performed in series in the worst case, and on hardware nodes, tasks can be performed concurrently. In this example, this means that tasks C and D can be performed in parallel to tasks A and B. Task E is dependent on tasks B, C and D, so must happen once they are completed. The added concurrency of hardware accelerator nodes helps reduce task execution latency when multiple tasks are assigned to a node.

In order to represent this behaviour, a set of new allocation variables is introduced: u. Each one of these $u_{nm_1m_2p} = \{0, 1\}$ represents the allocation of two implementations m_1 and m_2 to node n, assigned platform p.

The set of tasks on the path from a leaf node on the task graph t to the root of the task graph is $H_t \subset T$. Let the set H contain all of the task path sets ($H_t \in H$). The set O_{H_t} is declared, containing all other tasks not on the path H_t. The set $I \subset M$ is defined as the set of all software implementations. The computation time for a path H_t, τ_{H_tc} in the task tree is given by:

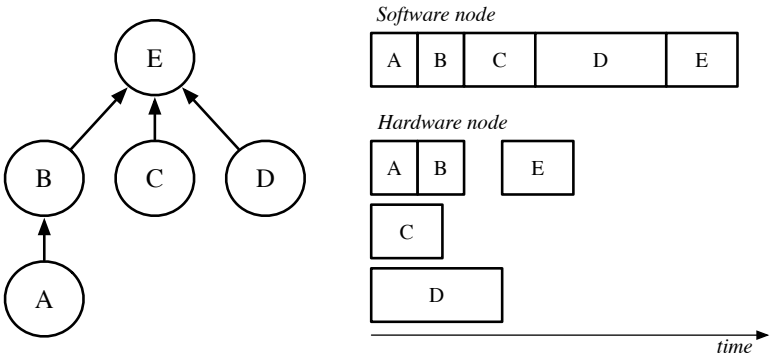


Figure 3.3: The difference in how a set of tasks allocated to a single node are scheduled on software and hardware accelerator nodes.

$$\tau_{H_tc} = \sum_{i=0}^{|N|}\sum_{j=0}^{|H_t|}\sum_{k=0}^{|M_j|}\sum_{l=0}^{|P|}(z_{ikl}\phi_{klj} + \sum_{m=0}^{|O_i|}(\sum_{q=0}^{|I_m|}(u_{ikql}\phi_{qlm}) - \sum_{r=0}^{|I_{a_j}|}(u_{ikrl}\phi_{qlm})) \tag{3.5}$$

The z_{nmp} term in this equation is the sum of delays on all paths of the task tree. The $u_{nm_1m_2p}$ terms represent the extra delays on the path caused by having multiple tasks not on the same path allocated to the same node in software. The computation times of any other tasks allocated to the same node as any task in the path are added. The subtraction is present to ensure that this computation time is only added once for each set of tasks in a path allocated to the same node.

Communication Delay

A simple communication model is used where tasks send data to the parent task as soon as it is ready. There is no communication cost between tasks, only between nodes. Communication and computation can occur simultaneously and independently. If a node receives data for a task not assigned to it, it forwards this data immediately to the next node.

Data is transferred from one node to another when a task's parent task is allocated to another node. Similarly to the computational delay, total communication delay τ_{H_tm} between tasks in each path in the task tree $H_t \in H$ can be expressed:

$$\tau_{H_tm} = \sum_{i=0}^{|H_t|}\sum_{j=0}^{|N|}(\sum_{k=0}^{|K_j|}(\sum_{l=0}^{|M_i|}(\frac{x_{kl}d_i}{b_k}) - \sum_{m=0}^{|M_{a_i}|}(\frac{x_{km}d_i}{b_k})) \tag{3.6}$$

If an implementation is allocated to a node, the communication time to the next task is added. If the parent task is allocated to the same node, this time is then subtracted.

The delay from the data source s on path H_t to the node that performs the first task on it, $\tau_{H_t s}$, is given by:

$$\tau_{H_t s} = \sum_{i=0}^{|D_s|}(\frac{d_s}{b_i}) - \sum_{j=0}^{|K_i|}\sum_{k=0}^{|M_{t_l}|}(\frac{x_{it_l}d_{s_l}}{b_i}) \tag{3.7}$$

Where t_l is the leaf in the task path.

The total communication delay in a path $\tau_{H_t k}$ is thus:

$$\tau_{H_t k} = \tau_{H_t s} + \tau_{H_t m} \tag{3.8}$$

The proposed model can be extended to incorporate different communication delays for software and hardware tasks as would be the case for network-attached hardware accelerators that can process packets with lower latency. The computation and communication latencies are likely to vary in reality. This model considers the worst case latency where a node processes all other tasks first and transmits the results last.

Total Delay

The total latency for a path, τ_{H_t}, is equal to:

$$\tau_{H_t} = \tau_{H_t k} + \tau_{H_t c} \tag{3.9}$$

The largest of these values is the total latency, τ_{max}.

Although this section has discussed a scenario where only a single task graph is present, the model allows the possibility of multiple independent task graphs representing separate applications. Using the same method and equations, a τ_{max} can be formulated for other task graphs.

3.6.2 Throughput

The throughput of the system is the rate at which results are output, and dependent on the node with the longest processing time in the network. A continuous variable g can be introduced to represent the maximum delay processing stage. For software implementations, where only one task can run on a node at any time, this can be

expressed:

$$\forall n \in N, g \geq \sum_{i=0}^{|T|} \sum_{j=0}^{|P_s|} \sum_{k=0}^{|M_i|} (z_{nkj}\phi_{kji}) \tag{3.10}$$

Where P_s is the set of all platforms that run software implementations. This equation assigns the value of the largest total computation times for all tasks on a node to the value of g. For platforms that run hardware implementations, P_h:

$$\forall t \in T, g \geq \sum_{i=0}^{|P_h|} \sum_{j=0}^{|M_t|} (z_{nji}\phi_{kji}) \tag{3.11}$$

Here, only the value of the longest computation time task is assigned, as processing is carried out in parallel.

The throughput, v, can then be expressed:

$$v = 1/max(g) \tag{3.12}$$

3.6.3 Data-rate

Data-rate can be very significant in scenarios involving information sources with dense data and for large networks and applications. Poor utilization can also lead to additional communication delays.

The data-rate of a data stream at a source s, q_s is given by:

$$q_s = \frac{d_s}{e_s} \tag{3.13}$$

The data-rate of a task t, denoted q_t, is given by:

$$q_t = f_t \sum_{i=0}^{|C_t|} (q_i) \tag{3.14}$$

Where the output data-rate is the sum of the output data-rates of all child tasks, reduced by the reduction factor of the task f_t.

For leaf tasks t_l where $|C_t| = 0$, it is given by:

$$q_{t_l} = f_t q_s \tag{3.15}$$

The total data-rate at the output of a network node is the sum of the data-

rates of all streams passing through it.

$$q_{n_c} = \sum_{i=0}^{|K_n|}(\sum_{j=0}^{|T|}(\sum_{k=0}^{|M_j|}(x_{ik}q_j) - \sum_{l=0}^{|C_j|}\sum_{m=0}^{|M_l|}(x_{im}q_j)) \tag{3.16}$$

If a task is allocated to any node beneath that node in the network tree, the data-rate consumption is added, and if the parent task is allocated to any of these nodes, it is then subtracted.

The data not yet processed by any tasks must also be taken into account. If $S_{K_n} \subset K_n$ is the set of all sources that lie beneath n in the network graph, and $L \subset T$ is the set of all tasks where $|C_t| = 0$:

$$q_{n_l} = \sum_{i=0}^{|K_n|}(\sum_{j=0}^{|L|}(\sum_{k=0}^{|S_{K_n}|} q_k - (\sum_{l=0}^{|M_j|}(x_{il}q_{s_j})) \tag{3.17}$$

Where q_{s_t} is the data-rate of the source that leaf task t operates on.

The total data-rate at a node $n \in N$ is given by:

$$q_n = q_{n_c} + q_{n_l} \tag{3.18}$$

This is the sum of the data-rate of all streams passing through a node, and the data-rate of the stream from the data source to that node, if applicable. This gives the data-rate at each link between nodes.

3.6.4 Energy Consumption

The energy consumption of the network can be relevant for a variety of applications. In an application that deploys remote nodes with limited power sources for example, such as a wireless sensor network, energy usage can be a significant constraint. Most related works do not consider computational energy costs. The energy used at a node $n \in N$ depends on the power consumption w_p of the platform $p \in P$ at that node, and the times taken τ_m to complete the implementations $m_t \in M$ of tasks $t \in T$ allocated to the node. Just as when formulating an equation for the end-to-end latency, taking a 'snapshot' of the network, the energy consumed by the network per data instance is given by:

$$\sum_{i=0}^{|N|}\sum_{j=0}^{|T|}\sum_{k=0}^{|M_j|}\sum_{l=0}^{|P|}(z_{ikl}\phi_{klj}w_l) \tag{3.19}$$

This equation adds the product of the computation time and the power consumption of the platform, if a combination of platform of task is allocated to a node.

3.6.5 Financial Cost

The simplest metric is the financial cost of the solution. An equation can be formed that represents the total cost of the system, based on the platforms selected at all of the nodes. The total cost of the solution is given by:

$$c_{max} = \sum_{i=0}^{|N|} \sum_{j=0}^{|P|} (y_{ij} c_j) \tag{3.20}$$

This is a simple equation where the cost of a platform is added to the total cost, if a platform is allocated to a node.

Financial cost is a concern as it is ultimately one of the key drivers in the decision of where to place computing capability, and will always be one of the largest barriers to achieving the best possible placement. The model considers the added cost of the computing platforms required to implement the in-network computation.

3.6.6 Combined Evaluation Metrics

This section has presented formulations for the 5 important performance metrics relevant for evaluating heterogeneous distributed systems. The goal was to keep these distinct as the proposed model is designed to be flexible enough to use for different scenarios and purposes, where the relative importance of these five metrics will vary depending on the application. Users of the model are able to build more complex metrics based on the requirements of their analysis, combining whichever of these five is relevant to their evaluation, and suitably weighting the different components.

This model to be used in the design and evaluation of alternative structures for deploying heterogeneous applications. In such scenarios, a constraint-driven approach is more sensible than a combined metric, and the model supports such evaluations. For example, a required financial budget or latency target can be set and other metrics evaluated for different designs. If used to compare designs, the primary metric of importance can be evaluated, with constraints placed on the other metrics, such as the best latency for a fixed financial cost and energy budget.

The flexibility in this model in determining general lessons around the placement of tasks and hardware resources is demonstrated in the evaluation in Section 3.8.

3.7 Case Study

This section investigates the implications of different placement strategies in a distributed object detection and tracking system. While the formulation presented in Section 3.5 can be used to create an optimal placement of computing resources and tasks for a given application and network, it might be argued that such a bespoke design would not be highly practical, since a more uniform approach to deploying computing resources is generally required, and the variability of applications might make a static allocation less ideal. Hence, this section evaluates strategies for a representative application to learn general lessons about the placement of computing resources in such networks. This study considers a network of cameras, some fixed and some mobile, such as drones, tasked with surveying an area to detect human presence. The images collected by each camera are processed through a sequence of tasks including the histogram of oriented gradients (HOG) and an SVM classifier to detect objects of interest, and a tracking algorithm is applied that relies on the fusion of data from multiple cameras.

3.7.1 Network

A network structure was chosen that is generally representative of that seen in an application such as this. The outermost layer represents the very edge of the network, comprising the cameras themselves (layer A). The next layer represents an access or gateway layer, that connects the cameras to the larger network (layer B). Each gateway and the connected sensors represent different areas that are to be monitored – for example rooms or neighbourhoods. Cameras connect to this layer through interfaces such as 100 Mb Ethernet or 802.11 wireless LAN. A transfer time of 10 ns per bit of data was selected for this layer. The next layer is a routing layer that connects the local network to the wider network, with higher speed and bandwidth interfaces such as 10G Ethernet (layer C). Here a latency of 0.1 ns per bit of data is modelled. Finally, there is the cloud layer, which houses the remote computing resources. To reach this layer data must travel through the internet, for which a communication time of 1 ns per bit is assumed, based on round trip times to AWS EC2 instances measured in [175]. These communication times are estimates and ignore frame/packet overheads, and many other delays, but are there to model variation in transfer time between different layers.

The topology used in this case study is shown in Figure 3.4. It includes a mix of nodes with high and low fanout, and nodes at all of the layers discussed above. Links appear unidirectional as it is assumed data must flow through these

layers in order to reach the cloud/datacenter. It is important to note that the layer B/layer C nodes do not represent individual machines, but rather layers of the network hierarchy, comprising multiple machines. Communication within these nodes is neglected in this case study.

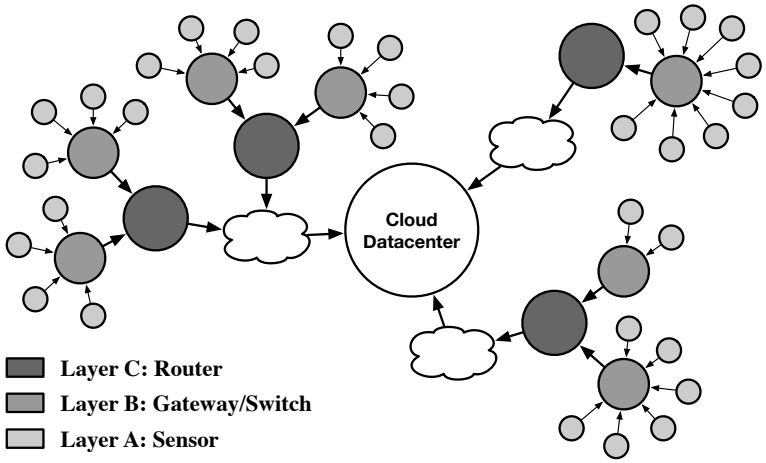


Figure 3.4: Network structure used in this case study.

3.7.2 Tasks

The HOG algorithm used in this case study has been previously implemented on a variety of computing platforms [176; 177; 178]. For the sake of this case study the algorithm is broken down into 3 tasks: gradient computation, normalisation, and classification. While there are more tasks that form this algorithm, these 3 take a majority of the computation time and have a significant effect on data size. Estimates are obtained from [176] for the reduction factor of each task. The tracking algorithm uses these HOG features and a KLT tracker [179], relying on fusion of data from multiple cameras. Therefore this task must be placed elsewhere in the network, at a location that can access all necessary cameras.

In this case study, each camera has a set of tasks, *gradient comp* $\rightarrow$ *histogram* $\rightarrow$ *classification*, associated with them, and then each area of cameras has a tracking task that processes the result from multiple camera chains.

3.7.3 Platforms

Computation time for each task on each platform can be estimated based on previous work. Though these are estimates, and different implementations may have varying optimisations, the relative computation times are the important factor for this case

Platform	Grad,Hist	Normalisaton	Classification	Tracking
Cortex A9	2,000	3,200	1,900	2,000
Intel i7	40	60	35	40
Intel Xeon	2.6	4.0	2.3	2.6
Xilinx Zynq	260	400	240	260
Xilinx Virtex 6	1.3	2.1	1.2	1.3
Reduction factor	0.77	0.004	0.16	0.16

Table 3.2: Computation times in milliseconds for each task on different platforms, from work referenced above.

study. If computation is placed at a camera node, an embedded platform is assumed. An embedded Arm Cortex A9 is used in [176] to implement the HOG algorithm, so this case study uses the computation times presented there.

If computing is placed at the access or routing layers, it can be assumed a more powerful CPU is available. The work in [178] implements the algorithm on an Intel Core i7 processor. Finally, the cloud layer would use server-class processors, such as the Intel Xeon platform used to implement the algorithm in [177]. The work in [176] presents an FPGA design that gives a speed-up of around 7× on a Xilinx Zynq platform that could be embedded at the camera. An FPGA accelerator implemented on a larger Xilinx Virtex-6 FPGA was reported in [177], and the case study assumes this is the FPGA platform available at other layers. The relative performance on these platforms is used to estimate the computation time of the tracker task. Table 3.2 summarises time taken for each task on each platform per frame.

The costs of each platform are also relevant. This case study considers the extra costs associated with adding computing resources to different layers of the network. A Cloud/datacenter node is present regardless of other node placements. Table 3.3 summarises approximate costs and power consumption for each of the platforms in arbitrary currency and energy units based on costs determined from OEM suppliers, and manufacturer power estimation utilities.

The FPGA resource utilization estimates in the previously cited works at the top of Section 3.7.3 suggest that both FPGA platforms can implement 3 full pipelines of the algorithm pipeline each, so 12 tasks. It is assumed CPU-based platforms have no limit to the number of tasks that can be running, though, as discussed in the formulation, there is a latency penalty for sharing resources. The case study focuses on latency, throughput, energy consumption, and financial cost

Platform	Cost	Power Consumption
Arm Cortex A9	10	1
Intel Core i7	300	5
Intel Xeon	2000	100
Xilinx Zynq	250	5
Xilinx Virtex-6	1000	10

Table 3.3: Financial cost and power estimates for each platform.

Placement	Latency	Throughput	Cost	Energy
Centralised	1.95s	3.43 frames/s	2000	30.03
Layer C	1.97s	0.88 frames/s	3200	23.00
Layer B	1.93s	0.94 frames/s	4100	23.00
Layer A	7.16s	0.14 frames/s	2300	241.2

Table 3.4: Performance metrics for different placement strategies using software platforms.

as the metrics of interest.

The model is used to build the above scenario and evaluate different computation placement strategies. Results are presented in Table 3.4 for software platforms and Table 3.5 for hardware platforms. The output is the latency, throughput, financial cost, and total energy consumption of the entire system. Bandwidth results are not shown in this table as they are calculated per node in this model.

3.7.4 Centralised Software

A typical approach to such an application would be to centralise all tasks, performing them in software, transmitting all data to the cloud or datacenter. In this case study, this gives a latency of around 1.95 s, and a throughput of 3.4 frames per second for each camera. Note that this is in the worst case, where all camera streams compete for CPU resources. The large communication latency coupled with the large amount of data being transmitted undermines the extra computing power provided by the cloud. Energy consumption was also joint highest with this approach, as the Centralised hardware has the highest power consumption.

3.7.5 In-network software

An alternative approach is to push processing tasks down into the network. One possibility is placing the gradient computation, normalisation, and classification tasks on the camera nodes (layer A), and placing the tracking tasks at the appropriate layer B nodes as they require information from a set of cameras. This results in a latency of around 7.16 s and a poor throughput of 0.14 frames per second, unsuitable for real time applications. The energy consumption seems high, but this value is the energy consumption of the entire system - the consumption at each individual node is much lower. While there is communication latency, and fewer tasks competing for the same resources, the computing capability of these edge nodes is so low that the latency and throughput are much worse than the centralised placements.

Distributing tasks within the intermediate network infrastructure offers improved latency relative to placing tasks in layer A, but has minimal impact when compared to centralised placement. In this scenario, the reduced communication latency is offset by the increased computation latency. Layer B and layer C approaches introduce additional costs of 2100 and 1200 currency units respectively. The centralised solution also has 3.65× higher throughput than these approaches. This is because of its increased computing capability relative to these other nodes, meaning that there is less computation latency. Energy consumption is less than centralised software, due to the lower power consumption of the hardware. This energy consumption is also spread across a greater number of nodes, meaning each node consumes less energy.

3.7.6 Centralised Hardware

Utilising FPGA acceleration at the server node reduces the latency to 1.68 s, and increases throughput to 133 frames per second, as a result of reductions in computation latency. While the FPGA should in theory provide a greater performance boost than this, the time taken for data to travel to the cloud limits the improvement that can be achieved for the application. The energy cost of running these tasks in hardware is also much lower than in software. The FPGA accelerator has a lower power consumption, as well as lower computation time.

3.7.7 In-network hardware

Adding FPGA accelerators to layer C reduces latency to 0.84 s, and increases throughput to 133 frames per second due to the performance of the FPGA accelerators dramatically reducing computation latency. Placing FPGAs in layer B further im-

Placement	Latency	Throughput	Cost	Energy
Centralised	1.68s	133 frames/s	13000	1.56
Layer C	0.844s	133 frames/s	14000	1.56
Layer B	0.8s	133 frames/s	16000	1.56
Layer A	0.94s	1.10 frames/s	11600	30.6

Table 3.5: Performance metrics for different placement strategies using hardware platforms.

proves latency to 0.83 s. These placements give improvements over the centralised FPGA approach due to the reduction in communication latency. There is little difference in latency between placing tasks predominately in layers B or C, as the fast link between these layers means that there is minimal communication delay. The disadvantage of the in-network FPGA approach is the additional cost, with the layer B and C methods costing 16000 and 14000 currency units respectively. Moving all tasks in hardware to the layer A camera nodes offers improvements over the software equivalent due to the increased computing capability. It also improves over centralised approaches due to the reduced communication latency. However the higher computation latency relative to layers B and C means that there is a higher overall latency, and worse throughput. While the total energy consumption for the layer A approach looks high, it is spread across a greater number of nodes. Each layer A node actually has a power consumption of approximately 0.956. The same processing hardware is implemented on the FPGAs in layers B and C, as well as when centralised. This results in the throughput being equal in all circumstances, despite the higher communication latency.

3.7.8 Optimal Placement

The model outlined in this thesis can be used with a Mixed Integer Linear Program (MILP) solver to generate a specific task and hardware placement strategy to optimise any of the performance metrics detailed in Section 3.6. To do this, the Python PULP front end was used to interface to the CBC solver. In this case, the system is optimised for latency, as in this example, energy and throughput are directly related to latency. First generated is the optimal latency placement, then ran the optimisation again with a latency constraint 5% higher than this value, but optimising for cost. This forces the solver to generate the cheapest placement that achieves a latency within 5% of the optimal value. As a result, the model generated a placement with the metrics shown in Table 3.6. This is presented for completeness;

it may be argued that customising a network for a specific application is unlikely to be a common requirement. Hence, this section has focused primarily on the general lessons learnt in terms of placement strategies for hardware in the network.

Placement	Latency	Throughput	Cost	Energy
Optimised	0.87s	133 frames/s	9000	1.56

Table 3.6: Performance metrics for MILP optimisation of model

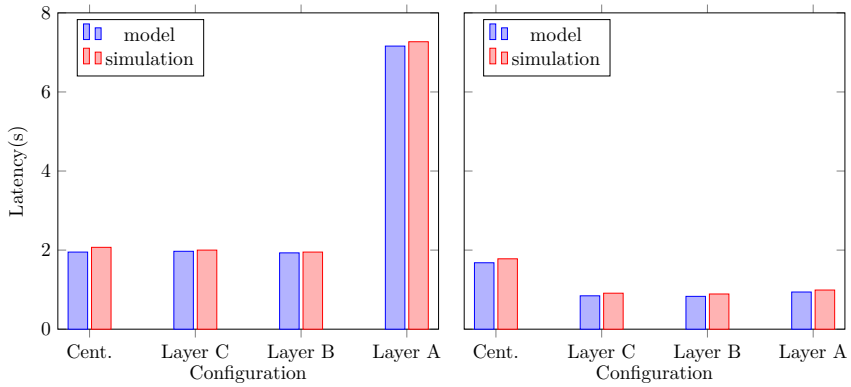
3.7.9 Summary

Improvements can be made to streaming application latency by pushing tasks into the network in either software or hardware. This also offers improvement in energy consumption at each individual node, important when there may be a limited power budget. There is a balance between the communication latency to reach higher capability nodes, and the benefits to computation latency that they provide. Placing tasks at the very edge of the network minimises communication latency but is limited by poor computational capability. The cloud offers the highest computing capability but there is a communication latency bottleneck. The downside of using in-network task placement is the additional financial cost of the extra hardware. However, with the price/performance ratio for embedded devices scaling significantly faster than for server class CPUs, this should improve over time.

3.7.10 Event Driven Simulation

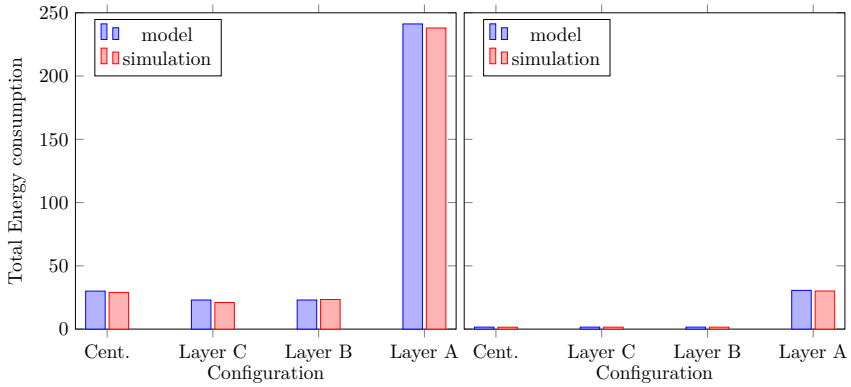
As part of this thesis, a discrete event simulator was written in Python using the SimPy library, to test the validity of results produced by the model. Data sources emit periodic packets of data into the network with the same topology and task structure. The tasks are allocated to the relevant nodes, and are executed at the nodes in a first-in first-out fashion, with priority given to the oldest data packets.

Differences should be expected in the reported latencies from the model and simulator primarily due to the more detailed task and communication scheduling in the simulator. The simulation processes individual packets as opposed to the considering abstract streams in the model. The data sources in the simulator emit packets with fixed periods, sources are unsynchronised, whereas the model implicitly assumes synchronisation. The simulation also takes into account a small switching delay at nodes, representing the transfer of data form received packets to the



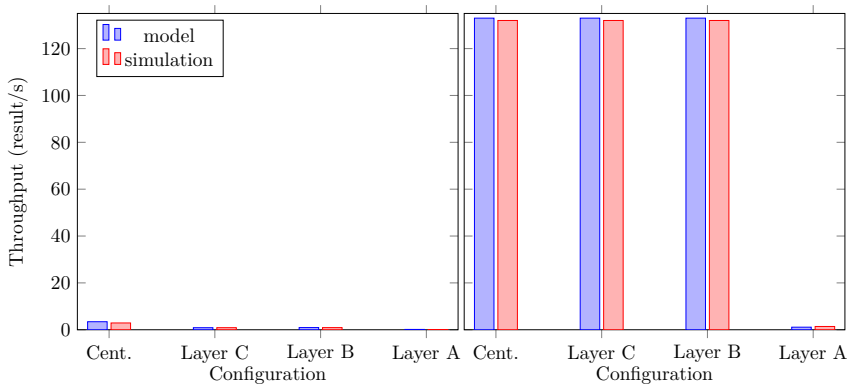
(i) Software platforms (ii) Hardware platforms

(a) Latency



(i) Software platforms (ii) Hardware platforms

(b) Total Energy Consumption



(i) Software platforms (ii) Hardware platforms

(c) Throughput - A higher value indicates superior performance

Figure 3.5: Difference between values calculated through the formulated model and a discrete event simulator for the same configurations and parameter values.

computing platform. Various network related parameters are not included in the simulation, as these are not influenced by the allocation of tasks and platforms.

Simulations of the above scenario were run for 20,000 packets entering the network from each source. The sources were fixed to the same period, but set out-of-sync with each other, to a degree determined from a uniformly distributed random variable. Figure 3.5 shows the deviation between the metrics predicted by the model and those measured in the simulation. Financial cost is not shown, as there will be no difference between the simulation and model, and data-rate is not shown as it is calculated for each individual node, not the system as a whole.

It can be seen that if considering only software platforms, the difference between the model and simulator is close to 6%, and in hardware 7%. These differences stem from the data sources being out of sync, and the switching delays introduced at each node, not represented in the model. The ratio between computation time and network switching delay impacts this error, and hence in the case of hardware, where computation time is reduced, the overhead is more significant. However, these deviations are still well within tolerable levels.

Additional Model Validation

The discrete event simulation detailed above provides only limited validation of the model. While it models the flow of individual packets as opposed to the abstracted streams used in the model, the results cannot be cited as external validation of results generated by the model. This presents a limitation of the model presented in this chapter. The model can be used to evaluate the performance of various placement strategies relative to each other, verified by the simulator. This is still useful within the context of a design space exploration, eliminating the clearly sub-performant solutions.

Further work must be carried out to verify that the model generates performance metrics that match real-world implementations. To carry out this work, a substantial test-bed would be required, as well as the increased maturity of various technologies such as network attached computation. This makes practical validation at scale difficult. In addition, existing modelling tools, as outlined in the literature review, aren't designed to model the same scenario, making validation using other models challenging.

As a result, it is suggested that this validation forms an important piece of further work.

3.8 Further Analysis

While determining a fixed optimal solution for a given application and network topology is possible by using an MILP solver as discussed, this sections considers synthetic scenarios in an attempt to draw general lessons about distributed, accelerated in-network computing. It explores how application and network properties influence the decision on where to place computing resources for this range of scenarios. Since latency is the primary metric of interest, it is the focus of this section.

For this analysis, a Python script is used to pseudo-randomly generate application task graphs. These are in a tree structure with a maximum depth of 4 tasks, reflect a realistic partitioning granularity rather than a very fine-grained structure that would skew further towards in-network computing. The analysis uses a fixed network structure, with a similar layer A/B/C hierarchy and interface specification as used in the case study in Section 3.7, however with 8 layer C nodes, each serving 2 layer B nodes, each of which serves 5 layer A nodes.

Several constraints are placed on the task generation. The tree is built up leaf tasks, with a random variable determining whether each task is connected to a new task or joins one already existing in the tree. Tasks can only join other tasks whose leaf tasks originate from nodes that share the same layer B parent. The script generates 100 random task trees in this manner, and the same 100 trees are used to evaluate each placement strategy, summarised in Table 3.7. For the purposes of this analysis, it is assumed that there are no restrictions on the number of tasks that can be allocated to a node, and all tasks are to be executed in 'software' - meaning that in this model there is a latency penalty dependant on the number of tasks allocated to the same node.

This section is focused on the latency metric, as latency reduction is one of the main motivations behind in-network and edge computing. The case study in Section 3.7 showed that latency and throughput are closely related for these types of streaming applications.

3.8.1 Relative Computing Capability

A key factor that determines where to place tasks is the relative computing capability that can be accessed at different layers of the network. In general, the closer to the centre a node is, the greater the computing capability, since the cost is amortised across more streams. The resources at the edge of the network are more likely to be limited due to space, energy, or cost constraints, while nodes further up in the hierarchy will have access to better hardware. However using better resources further

Strategy	Explanation
Centralised	All tasks allocated to root node
Pushed	All tasks pushed down toward the leaf nodes as much as possible
Intermediate	All tasks pushed down as far as possible, but not to leaf nodes
Edge/Central	Leaf tasks placed at leaf nodes, others placed centrally
Edge/Network	Leaf tasks placed at leaf nodes, others pushed down as far as possible but not to leaf nodes

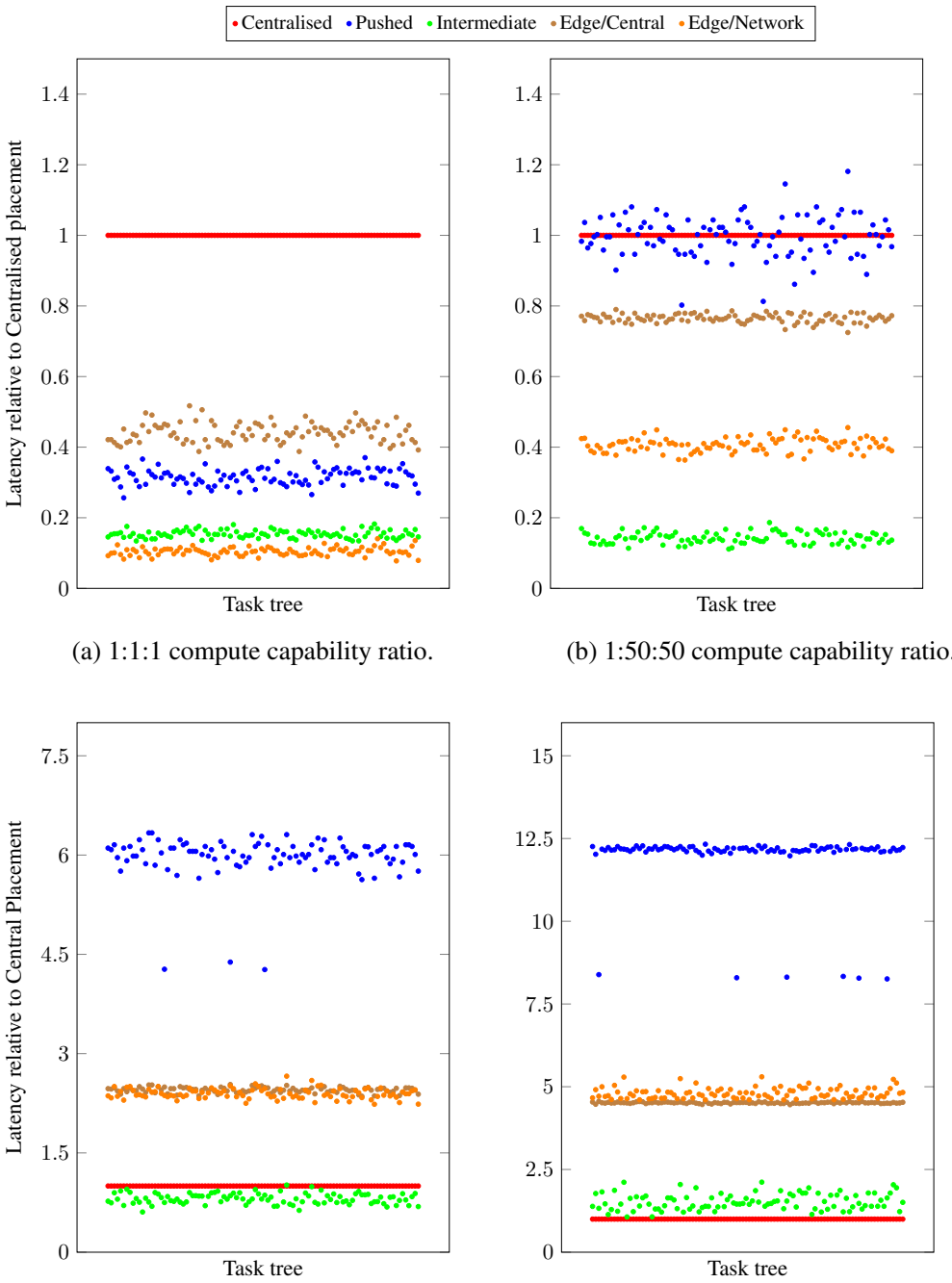
Table 3.7: Different placement policies used in the simulations presented in this section.

up the network entails a communication latency penalty, which must be overcome by improved computation latency. For this comparison, tasks are set to have a reduction factor of 50% and equal latency on the same platform. Figure 3.6 shows how different placement strategies impact latency, for different relative computing capabilities.

In the unlikely case where computing capability is equal across all layers (i.e. a Centralised:B/C:A computing capability ratio of 1:1:1), pushing all tasks as close to the data sources as possible yields the lowest latency as there is minimal communication delay, and no benefit to placing tasks higher up. This may be the case if the network is a commodity cluster of homogenous machines. Computation time is also improved since the tasks are distributed across many nodes resulting in less contention than for a centralised placement.

If the compute capability at the data sources is significantly smaller (50× in this case), while the rest of the network offers equivalent computing capability (a ratio of 1:50:50), pushing tasks down to intermediate nodes offers the best latency. In this case, the slight reduction in communication latency gained through placing tasks at the data sources is outweighed by the computation latency penalty. Placing them any closer to the central node adds further communication latency with no additional benefit, and causes contention due to more tasks being allocated to fewer nodes.

The more likely case is that resources at the central node are more capable than intermediate nodes, which offer greater capability than the edge. In the case of the central node being 5× more capable than the intermediate nodes (a computing capability ratio of 1:50:250), pushing tasks as low as possible into the intermedi-



(a) 1:1:1 compute capability ratio.

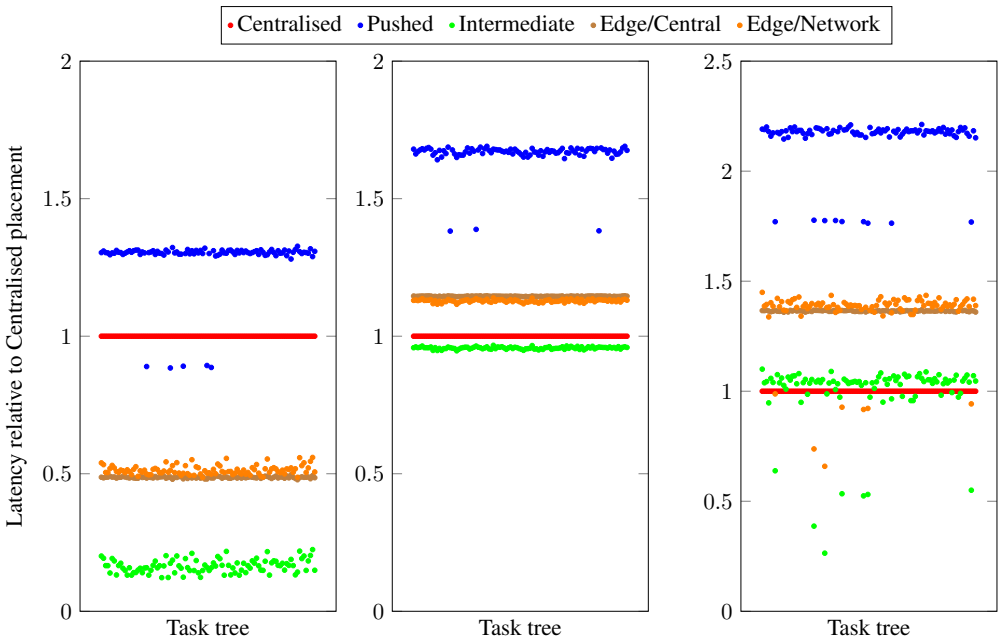
(b) 1:50:50 compute capability ratio.

(c) 1:50:250 compute capability ratio.

(d) 1:50:500 compute capability ratio.

Figure 3.6: Latency comparison for different Layer A:B:C computing capability ratios.

ate nodes still outperforms the centralised solution, as tasks are distributed to a larger number of nodes, reducing computation latency. Increasing the difference in



(a) Significant reduction at edge tasks.

(b) Moderate reduction at edge tasks.

(c) No reduction at edge tasks.

Figure 3.7: Latency comparison for different edge task data reduction factors.

computing power to 10× (1:50:500) causes the central solution to become dominant.

Hence, it can be seen that a key requirement for in-network computing to be feasible is that suitably capable computing resources be employed for executing tasks in the network. The more capable the edge nodes are in comparison to the root node, the greater the benefits of placing tasks further towards the edge.

3.8.2 Task Data Reduction

The time taken to transmit data further up the network is tied to the amount of data being transmitted. Tasks can reduce data by varying degrees, and this impacts the balance between computation and communication latency. For this experiment, reduction factors of tasks are modified to observe the impact on latency. It uses the same network topology as in the previous experiment, and the same method of generating task trees. A 1:50:500 relative computing capability configuration is used, as discussed in Section 3.8.1. This is to model the difference in compute resources at the different levels of the network.

Figure 3.7 shows how different placement strategies impact latency, for different task reduction factors. If data is dramatically reduced by tasks close to the edge of the task tree, placing tasks as close as possible to the data source is more

likely to provide a latency improvement as the communication cost for every other transfer between nodes is reduced. It can be seen that intermediate placement reduces latency by 5× compared to a centralised allocation in such a scenario. Placing all tasks at the edge results in 30% worse latency, despite the reduced communication latency, due to the low computing capability of these nodes. Placing only the leaf task at the edge and the rest either in the network or at the central node also provides a significant reduction in latency in this scenario.

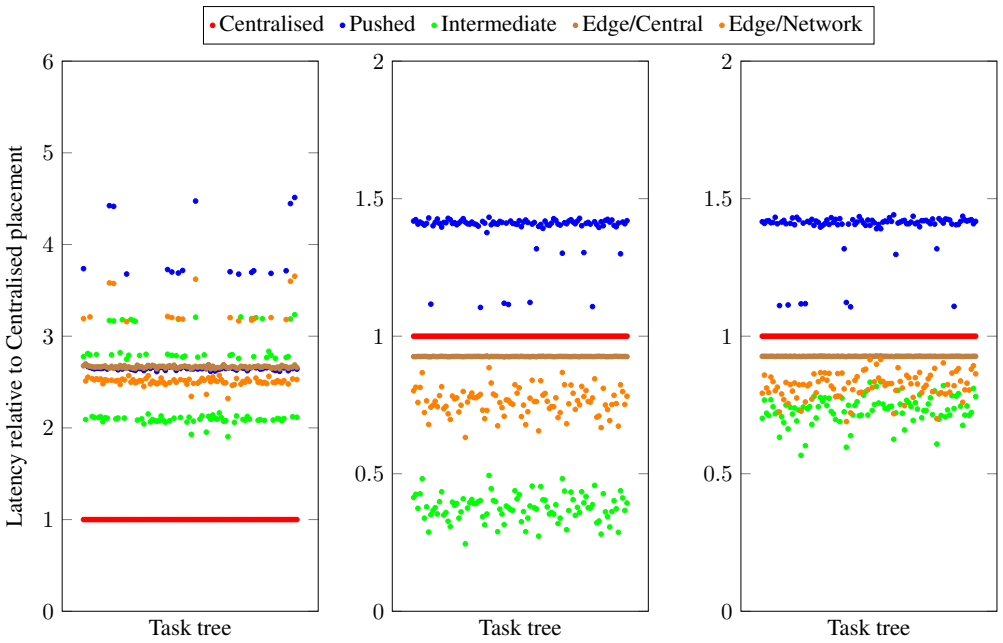
If data is not significantly reduced in the task tree, or only at tasks higher up in the tree, placing tasks towards the central sink is preferred, especially if those resources are more capable. A centralised placement provides the best latency in a majority of cases, although only by a slight margin. For some task trees, the in-network approach is superior. This result is impacted by the relative computing capability of layers. For scenarios where the central node is much more capable than the rest of the network, the instinct is to place tasks there. However, if data is reduced significantly at the leaf tasks then placing tasks in the network can reduce communication latency significantly.

It can be seen that, generally, the closer to the edge tasks that data is reduced, the greater the benefits of placing tasks closer to the edge of the network.

3.8.3 Network Structure

The structure of the network determines to what extent tasks can be distributed and parallelised and how much they must compete for resources. Related to this is the structure of the application task graph; having tasks that require data from multiple sources closer to the root of the tree means that tasks cannot be pushed down into the network to a layer with more computing nodes. To investigate this factor, this section considers different network structures and their impact on latency, as shown in Figure 3.8. The tasks were generated with the same method as before, and network nodes had the same computing capability as in Section 3.8.2. All tasks were set to a fixed reduction factor of 0.5.

Firstly, consider a network with low fanout, where layer B nodes each have 2 layer A nodes attached. While this means that there was more available resources towards the edge of the network, in many cases pushing tasks into the network results in almost 2× the latency of a centralised solution. Tasks that require data from more than one source must be pushed further up the network, adding additional communication latency. Additionally, as there are few layer C nodes, these nodes are over-utilised. Increasing the number of layer C nodes, or the computing capability of these nodes would offer performance benefits in this scenario.



(a) Layer A fanout of 2. (b) Layer A fanout of 5. (c) Layer A fanout of 20.

Figure 3.8: Latency comparison for different network fanout factors.

Raising the fanout of the layer B nodes to 5 instead of 2 increases the benefits of pushing tasks into the network. As more sources share the same paths towards the central node, there is a higher chance that a task that works with data from multiple sources can be placed closer to the edge. Increasing the number of nodes at layer C in this case again slightly decreases latency, as tasks that do have to be placed there have access to more resources.

Further increasing the fanout of the layer B nodes to 20 starts to increase latency again, up to around 0.45× the centralised placement. Increasing it to 40 increases the latency to around 0.7× the central placement.

A larger fanout at layer A (the edge layer), up to a point means that there is a greater benefit of pushing tasks down towards the network edge, as there are more opportunities to place tasks that require data from multiple child tasks closer to the edge. However if the fanout is too great, resource competition starts to reduce the benefits of this approach.

It can be seen that there exists a trade-off between having multiple sources connected to the same path of nodes, and creating too much resource contention by having too many tasks assigned to the same intermediate nodes.

3.8.4 Hardware Acceleration

There are several key points that can be taken away from this investigation. In-network computing is more effective in situations where the edge and intermediate nodes are comparable in capability to the central node. While this is unlikely with traditional software processing platforms, it makes the case for trying to integrate hardware accelerators such as FPGAs into the network, as they can provide processing times closer to the more powerful processors found in a datacenter environment. It can also be seen that in-network computing provides more benefits in applications where data is more greatly reduced in tasks closer to the edge of the task tree. These tasks can often be large filtering or pre-processing tasks, and in order to place them close the the edge of the network, more capable hardware is required. This again makes the case for hardware acceleration. Finally, high-fanout network topologies benefit more from in-network computing as there are more opportunities for data fusion between tasks. The ability of hardware acceleration architectures to process streams of data in parallel is well suited to these scenarios, suffering less of a latency penalty due to resource contention.

3.9 Generating In-Network Task and Hardware Placement with Heterogeneous Hardware

As systems scale, manual placement of tasks and the selection of appropriate hardware becomes more cumbersome and potentially impractical. Additionally, naive placement strategies may result in sub-optimal performance. This means that a model able to determine optimal placement has significant value. An optimisation is also useful during the infrastructure planning phase when decisions need to be made about which systems to deploy.

This section outlines how the model can be used to generate task and hardware placement, for different objectives. It compares the placements generated by the model with various naive placement strategies for an example scenario, and then evaluate the generated placements with synthetically generated network topologies.

3.9.1 Objective function formulation

Section 3.6, outlines equations to calculate the bandwidth, energy consumption, end-to-end latency, financial cost, and throughput for a given network.

Each of these equations could be used as a singular objective function to be used with a mixed integer linear programming (MILP) solver. The solver will

attempt to minimise the value of the function, given the constraints outlined in Section 3.5. The experiments in this section use PULP, a Python front-end for the CBC MILP solver.

There is the possibility of multiple optimal solutions existing for any one objective function. The Python implementation of the model allows for the definition of an order of preference among the metrics. For example, the order *(Latency, Cost)* would provide the cheapest version of the lowest-latency solution. This is achieved by running the solver with the latency objective function, then adding a constraint to the linear program to force the latency to be lower than or equal to the newly found optimal value within 5%, and running the solver again with the cost objective function. This forces the solver to generate the cheapest placement that achieves a latency within 5% of the optimal value. For these experiments the value of 5% is used, but any threshold could be used depending on the requirements of the application. An issue with this approach is that it is inefficient, as the solver must be run multiple times. It also doesn't allow for mixtures of these performance metrics to be targeted. Further work could be done to construct more complex objective functions that combine multiple metrics.

Users could alternatively select one of the five available objective functions, and add constraints on the other metrics. This technique may be enough for a 'design' phase scenario as users will probably have a set budget and performance constraints that have to be met. For example, the cheapest solution that satisfies a given latency and energy constraint can be generated, or the lowest-latency solution for a given energy and financial budget could be found.

3.9.2 Case Study

To demonstrate optimisation of placement strategies with the model, it is applied to an example scenario. The goals are:

1. Demonstrating that computation in the network is feasible and offers advantages over centralised cloud, and fully-distributed edge computation.

2. Quantifying the improvements provided by the optimisation compared to the more naive approaches.

For this case study, complete formulation and optimisation took an average of 5 minutes. While compute time does increase with the scale of the network and task graphs, the initial use of this model is offline when planning the deployment of accelerators, and so this is an acceptable delay in light of the model's flexibility.

In most scenarios, it is likely that a more constrained search is sufficient due to existing hardware placement and other constraints. The model is also useful in scenarios where after planning hardware placement, new tasks are to be allocated to those existing resources. These additional constraints will cause the problem to be further reduced in the pre-solve stage and to a decrease in the total time to solve.

Example Scenario

The scenario examined is based on the smart surveillance application detailed in the Section 3.7, using the same HOG algorithm and task structure. Using the values for task execution time for the various platforms detailed, more generalised estimates can be extracted. Consider a network of cameras, some fixed and some on mobile platforms such as drones, tasked with surveying an area to detect human presence. Each camera source is capable of hosting a computing platform and can send data to a central data sink hosting a more powerful computing platform. This study uses an arbitrary network topology that includes several possible structures—nodes with high fanout, long and short paths, data sources at different depths—as depicted in Figure 3.10.

Parameter values for each task are shown in Table 3.8. It is assumed there is one software implementation of each task, and one hardware implementation. Each hardware implementation of a task consumes one 'slot' on a hardware platform. The experiments also assume infinite tasks can be allocated to software platforms.

Task	τ_{sw}	τ_{hw}	f_t
Gradient Computation	0.16	0.0496	0.770
Normalisation	6.33	1.9500	0.004
Real AdaBoost	650.00	201.5000	0.160

Table 3.8: Summary of task parameter values.

Available platforms are detailed in Table 3.9, with their relative performance to each other. The 'Processor' platform represents a server-class processor and can only be hosted at the root node. The FPGA platforms have no *speed* as the same logic deployed on different FPGAs is likely to have roughly equal compute times, if the logic is the same, and implementation the same. The same holds true for power consumption - given the same logic implemented on the different sizes of FPGAs, the differential power consumption for that task should be similar. The values in the table, derived from [176] are designed to be representative, and can be replaced

with real measured values for specific applications.

Platform	Cost	Speed	Power	Slots
Microcontroller	30	1	0.1	–
Processor	500	8	1	–
Very small FPGA (fpgaVS)	35	–	0.4	1
Small FPGA (fpgaS)	70	–	0.4	3
Medium FPGA (fpgaM)	85	–	0.4	6
Large FPGA (fpgaL)	100	–	0.4	9

Table 3.9: Summary of available platforms.

Performance Compared to Naive Placement

The generality of the proposed model means direct comparison against other published work is difficult. Hence, this section analyses how it performs against the more traditional centralised cloud and distributed edge approaches.

The naive placement options considered are:

- Allocation of all tasks to a central server node, housing the 'Processor' platform, similar to a cloud deployment (Cent);
- All tasks 'pushed down' as far as possible towards the data sources—what is often referred to as the 'edge', performed on microcontrollers (PushM);
- All tasks 'pushed down' as far as possible towards the data sources, performed on FPGAs (PushF);

The solutions generated by the model to minimise latency using only microcontrollers (OptM), solutions generated by the model to minimise latency using any available platforms (OptF), and solutions generated by the model to minimise the time between results using only microcontrollers (OptT), can be compared;

Figure 3.9 shows the results for each of the metrics under these conditions, normalised against the centralised approach. Energy, and latency are improved, as is 'throughput' (throughput is plotted as the time between results, so lower is better).

The model finds solutions that improve upon the fully-distributed 'edge' (PushM and PushF) approach significantly in terms of cost, while also offering improvements in the other metrics, with the ability to optimise for a preferred metric. Figure 3.10 shows two of the generated configurations.

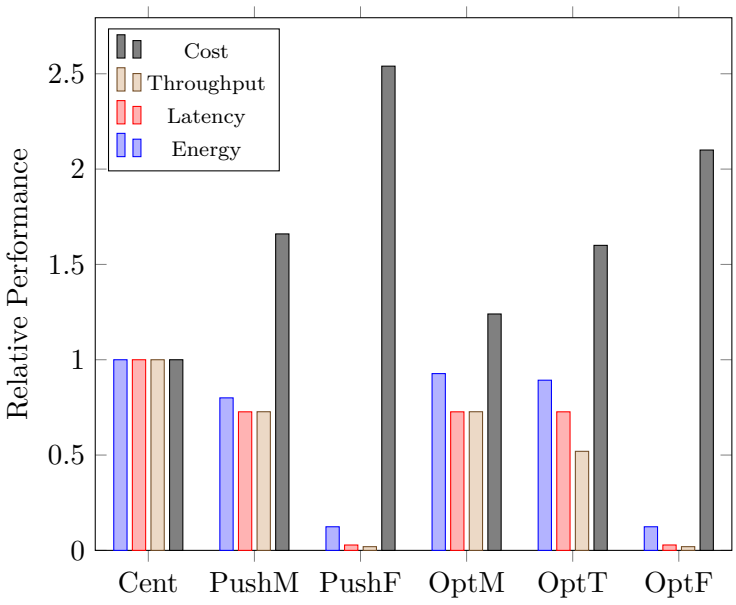
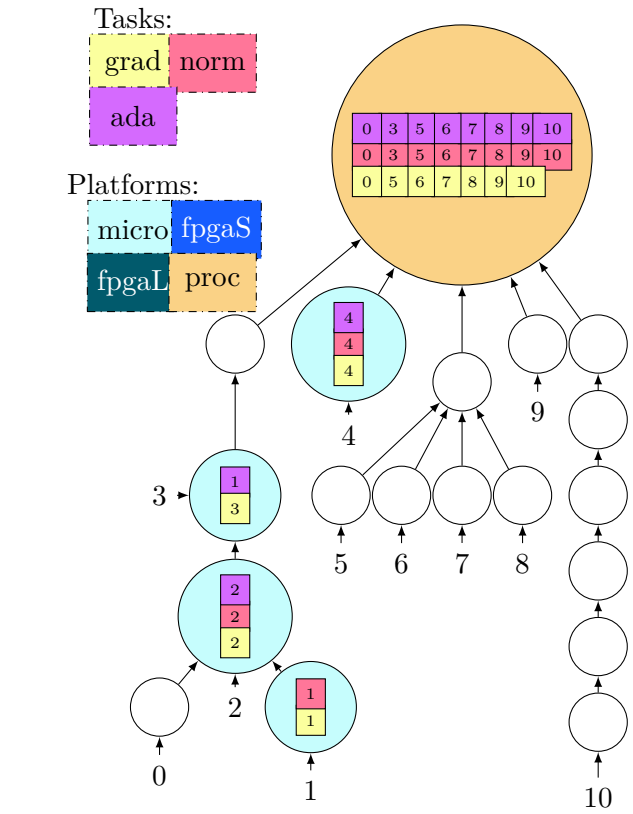


Figure 3.9: Naive vs. model placement. A lower value is better for all metrics.

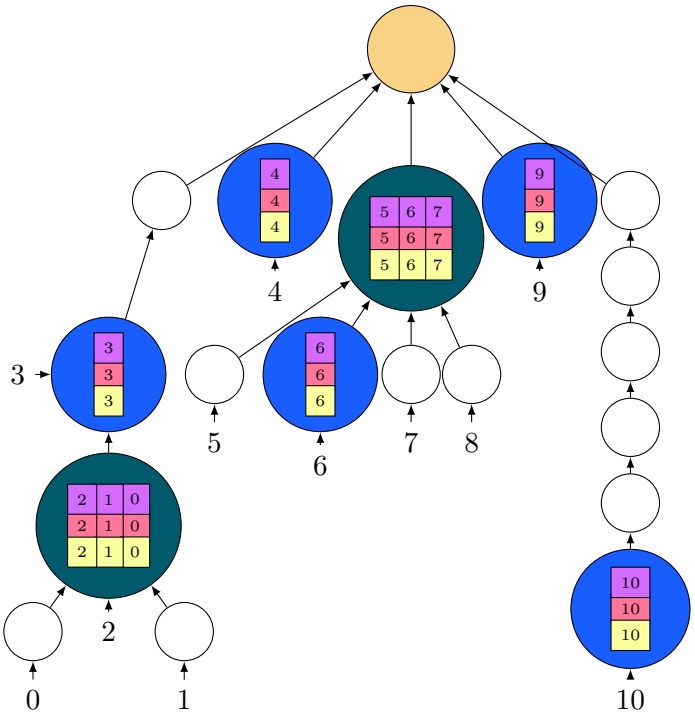
A realistic application of the model could be to find the cheapest resource and task allocation possible that still meets an end-to-end latency constraint. In the case of the surveillance camera system used as a case study, the latency between image capture and a final decision may be a critical factor. Figure 3.11 shows the cost of the solution generated by the optimization to achieve different latency constraints, compared to the cost of pushing all tasks to the leaf nodes. The optimisation approach offers a set of solutions for each latency constraint that improves on either of the distributed approaches in terms of cost, with savings of up to 33.8% in this particular case, depending on the performance required. Centralised computation in this case would not meet the latency constraint for all plotted values. Finally fully-distributed execution on microcontrollers does not meet the tighter latency constraints, demonstrating the importance of this model including hardware heterogeneity.

3.9.3 Evaluation with Synthetic Networks

For a more thorough evaluation, the model can be applied to a number of pseudo-randomly generated network topologies. For this study, the number of network nodes is limited to between 20 and 40, with each node potentially having a data source input as well as connections to other nodes. Generation parameters control the breadth and depth of the network topologies in order to ensure generation of a



(a) Generated configuration using microcontrollers only



(b) Generated configuration including FPGA options

Figure 3.10: Configurations generated by the optimization

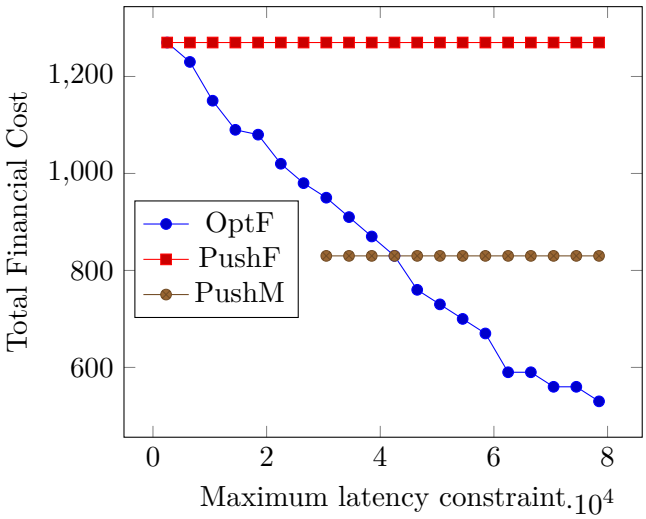


Figure 3.11: Cost of solutions to meet a latency constraint set by the user.

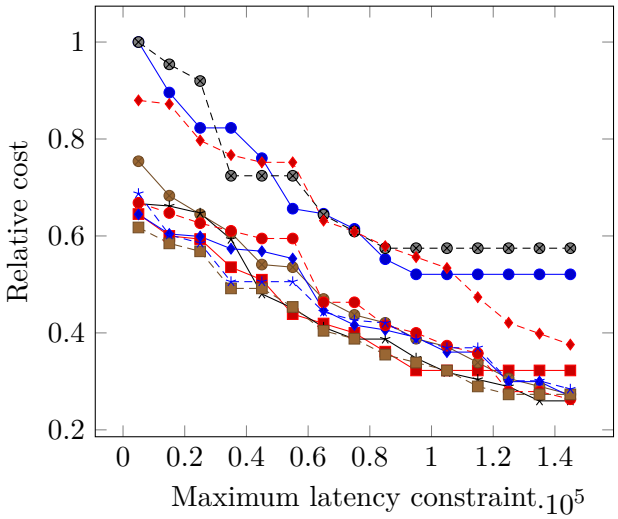
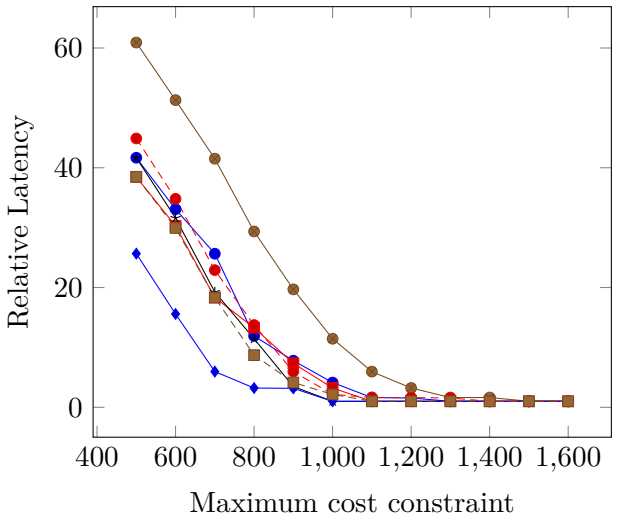


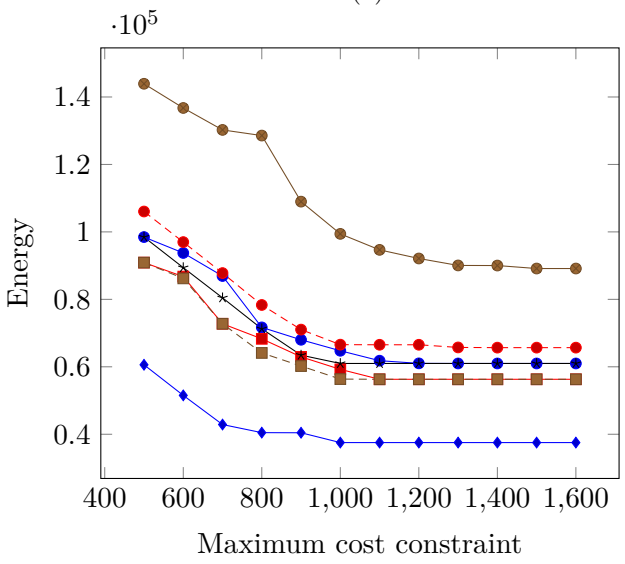
Figure 3.12: Relative cost of solutions to meet a latency constraint set for pseudo randomly generated network topologies. Cost is relative to pushing down all processing for that network topology. Each line represents a different generated network.

variety of structures. Data sources can be connected to any node, meaning data can be entered into the network at any level. The task structure and application is kept the same as the previous experiments, so each data source has an associated set of tasks. The platform options also remain the same. For each topology the model is applied to optimise the cost of the solution while meeting a maximum allowable latency constraint, as in the previous case study.

Results of the generation are presented in Figure 3.12. The minimum possible



(a)



(b)

Figure 3.13: Relative latency and energy consumption of solutions generated under a maximum cost constraint. Latency is relative to pushing down all processing for that network topology. Each line represents a different generated network.

latency is achieved by placing an FPGA at each node that a data source connects to. This is, however, expensive. Fully-centralised processing cannot typically meet tight latency constraints due to large communication latencies and sequential processing. Figure 3.12 shows how as the latency constraint is relaxed, the model finds solutions that offer significant cost reductions, of almost 40% on average. It is also clear, that there are often results that offer the same cost, but differing latency characteristics.

In complex networks, manually finding the cheapest solution to achieve a given end-to-end latency would not be feasible.

Another common situation is one where a given financial budget is available, and the best performing solution is sought. Again, for complex network topologies and applications, manually deciding where to place appropriate hardware and tasks to achieve this would not be possible. Further experiments with these synthetic networks were carried out where financial costs were constrained and the model was directed to find the lowest latency and energy placements.

Figure 3.13 shows how as the cost constraints are relaxed, even slightly, significant improvements in latency and energy can be obtained. This is only possible as this work is concerned with allowing the computation to happen at different places in the network rather than the traditional approach of full centralisation or de-centralisation. The results also show that significant relaxation of cost constraints yields diminishing returns in terms of latency and energy. This makes sense when compared to the earlier case-study where similarly performant configurations could be found at much lower cost, given the ability to place compute in the network.

Figure 3.14 shows the relative energy, latency, and cost results (compared to a fully pushed down on FPGAs allocation) for the different solutions offered by the model for six different network topologies. This is plotted against a centralisation metric that is computed by considering how many tasks are allocated to each 'level' of the network tree, computed as:

$$Decentralisation_i = \frac{\sum_{i=0}^{N} t_i i}{Decentralisation_{pushed}} \tag{3.21}$$

Where N is the number of levels on the network tree and t_i is the number of tasks at level i, where the level is the number of hops from the root node.

The decentralisation value is relative to the value computed when all tasks are allocated as close to the edge as possible. For example, a value of 1 would mean all tasks are allocated as close to the edge as possible, and 0 would mean all tasks are allocated to the root node.

It can be seen that there exists solutions in all cases that achieve similar latency and energy to the fully decentralised approach, while offering significantly improved cost. In all cases, these cannot be determined by inspection.

Figure 3.15 shows the relative energy and latency (compared to fully pushing all tasks to the leaves) with the area of each point being proportional to the solution's financial cost. It can be seen that the lower latency solutions cost more since more hardware is placed near the edges of the network. Energy consumption

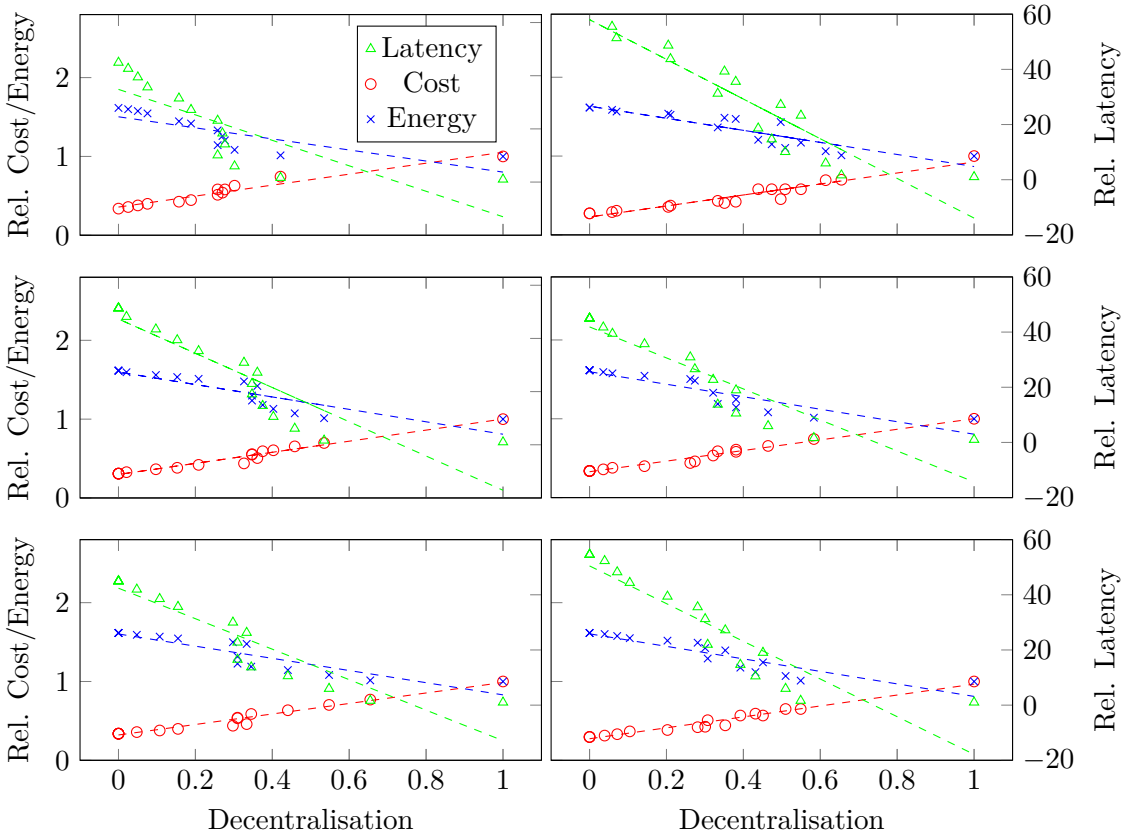


Figure 3.14: Cost, energy and latency relative to a fully-pushed-down-solution for varying Centralisation of tasks. Each graph is for a different generated network.

tracks latency primarily due to its dependence on task computation time. It is also apparent that numerous points with similar energy and latency results vary wildly in terms of cost, further justifying the need for a model that can incorporate all these metrics.

Figure 3.16 shows the bandwidth requirements relative to a centralised solution for each level in the network tree for one of these networks. Level 1 is one hop from the root node and shows the most significant decrease as tasks are moved towards the leaves, there is significant data aggregation at this level. Tasks being carried out at or before these nodes provide a significant bandwidth benefit. The leaf nodes see a less marked reduction in bandwidth requirements as the model places more tasks in intermediate layers of the network, due to the cost requirements. The model also allows bandwidth constraints to be placed on any individual link in the network, allowing modelling of network links with restricted bandwidth.

This case study has demonstrated a small fraction of the capabilities of the

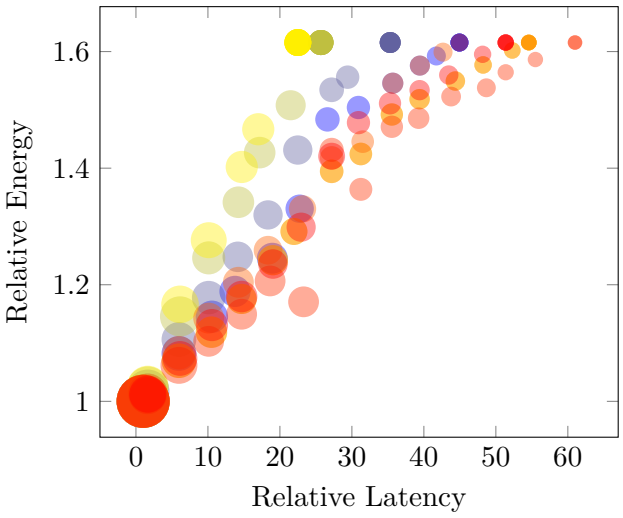


Figure 3.15: Latency and Energy relative to a fully pushed down solution for several network topologies. The area of the points is proportional to the cost of the solution, and each colour represents a different network topology.

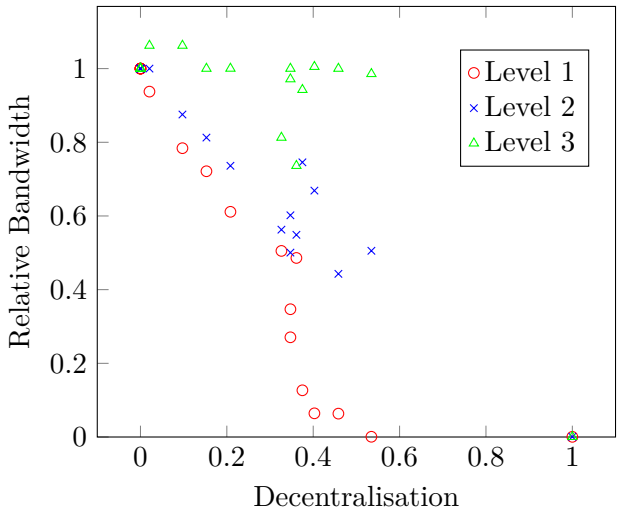


Figure 3.16: Total bandwidth consumed at each level of network graph for varying decentralisation of tasks for one network topology. Bandwidth is relative to a centralised solution.

formulation. The model allows for more fine-grained constraints to be tailored for a given application. For example, energy constraints can be placed on individual nodes, useful if they have a limited power supply such as those found in sensor networks. Latency constraints can be placed on individual tasks, useful in a situation where a task is safety or time-critical. Bandwidth constraints can also be put on any node or groups of nodes. Constraints can be placed that force certain tasks

or platforms to particular nodes, useful when extending an existing solution. The result is a flexible model capable of representing a wide range of applications and requirements.

3.9.4 Summary

This section demonstrates how the mathematical formulation of task and hardware placement presented in this chapter can be used to determine optimal configurations based on a set of constraints. The model allocates tasks and platforms to nodes while minimising latency, cost, energy, throughput or bandwidth. The optimisation generates solutions with equivalent latency performance to theoretical maximums for lower cost. It outperforms naive placement methods where computation is pushed down towards the edge as much as possible. This model is used to demonstrate that computing in the network offers significant advantages over fully centralised and full decentralised approaches.

3.10 Summary

The placement of computing resources and allocation of tasks in distributed streaming applications has a significant impact on application metrics. This chapter presents a model that can be used to reason about such applications. It models data sources that inject data into this network, applications composed of dependent tasks, and hardware platforms that can be allocated to nodes in the network. The model can be used to evaluate alternative strategies for allocating computing resources and task execution, offering information on latency, throughput, bandwidth, energy, and cost. This model has been used to demonstrate that computing in the network offers significant advantages over fully centralised and fully decentralised approaches, using an example case-study of an object detection and tracking system. Synthetically generated applications have also been used to explore the key application factors that impact the effectiveness of the in-network computing approach.

Chapter 4

Quantifying the Latency Overheads of FPGA Accelerators

4.1 Introduction

FPGAs have seen increased deployment within the datacenter to accelerate compute-intensive workloads. The growing complexity and scale of applications and the associated data, alongside the stalled performance scaling of CPU architectures, has resulted in heterogeneity being explored as a way of addressing performance, throughput, and power consumption challenges[180]. FPGA acceleration has been demonstrated to provide considerable benefits in computation latency for a variety of applications [40; 181; 182; 183], and improved performance per Watt compared to GPUs [42]. Virtualisation of FPGA resources is also emerging [184; 185; 141], allowing a single FPGA to be shared by multiple users and applications. These designs comprise a static shell that manages communication and control for multiple partitioned reconfigurable regions. Accelerators are swapped into these regions as needed, hence requiring accelerator allocation middleware to manage data stream sharing at runtime.

Latency reduction is a key challenge within the datacenter, in particular for streaming applications, where data is processed itemwise, and is often time-sensitive. Low latency can be a critical requirement in some applications such as high-frequency trading, or customer-facing web applications. The performance of distributed applications such as Memcached has been shown to be degraded by fundamental datacenter latency factors [186]. While individual packet latencies are

small, they can accumulate in applications where data is spread across multiple packets that travel through multiple switches or servers. Latency variability is also an important consideration. Processing is often distributed across many machines in parallel, and overall completion time is dependent on the slowest response [187]. Some applications rely on large fan-out requests for data across distributed sources, and though average latencies may be low, the effect of small variations can be amplified to cause significant degradation in performance.

While hardware accelerators improve computational latency, they can also introduce additional communication latency. Accelerators are typically attached to a host server through PCI Express (PCIe), which can achieve high throughput communication. Streaming data from the network must then traverse the host NIC and software network stack, subsystems that have been identified to contribute significantly to both average and tail latencies [96]. Inter-accelerator interconnect is also gaining importance for larger applications[188]. More recently, driven by the demand for low latency, there has been a growing interest in network-attached accelerators, where an FPGA is connected directly to the network [189; 190; 191], processing data in-line. While these deployments do reduce communication latency, their impact on overall system latency has not been studied in detail. Finally, the virtualisation required to allow FPGA resources to be shared can contribute further overhead.

Past work has characterised the components that contribute to datacenter latency including the different software components of host machines, the network interconnect, the NIC, PCIe, and switch latencies [96]. An in-depth study of PCIe communication latency and bandwidth as relevant to NICs was presented in [97]. The latency implications of software virtualisation were investigated in [93].

There has thus far been no such study into the communication overheads for FPGA accelerators in the datacenter. This chapter presents experiments that characterise these important overheads, in particular in the context of streaming applications, where latency is a key performance metric. They characterise the latency characteristics for PCIe-hosted and network-attached FPGA accelerators and isolate the additional delays introduced through virtualisation. These are compared to latency measurements for a typical host server, allowing us to isolate the latency contributions of host networking and management and data transfer to the PCIe accelerators.

Some work has compared the performance of PCIe and network-attached FPGA accelerators for specific applications[190] and minimising streaming data latency for embedded accelerators [192]. The work presented within this chapter

characterises the fundamental delays more generally, to gain insights into the costs of accelerator deployments and inform design decisions for a wide range of scenarios.

4.2 Contributions

The contributions of this chapter are as follows:

- It quantifies the communication latency and throughput for different arrangements of accelerator hardware, with a particular focus on FPGA accelerators;
- It explores the implications of these latencies on various aspects of application performance;
- It investigates the effects of these latencies on a consensus application as part of a case study.

4.3 Related Work

FPGA accelerators connected to a host via PCIe have been commonly deployed in the datacenter for various applications, such as machine learning [78; 79] and database processing [80]. PCIe offers a high throughput interface, existing supporting infrastructure, and a way for the host to control and configure the accelerator. The focus of such integration is high throughput, moving larger batches of data to minimise the impact PCIe transfer overheads. Distributed workloads, however, often comprise streams of data arriving over a network, which must be received over the host's network interface, written to memory via DMA transfers, with file descriptors pointing to packets stored in the driver ring buffer. Packets are then processed by the kernel's network stack before being added to the socket receive queue, which can then be accessed by an application running in user space. In order to transfer data to an accelerator, the user space application typically uses API calls to write data to a memory buffer and issues a command to the FPGA to initiate the transfer. The FPGA then reads the data to be transferred from this buffer. These processes all add to overall data latency when receiving data from the network for processing in an FPGA accelerator.

A solution to the latency problem for high data-rate streaming applications is for the FPGA accelerator to interface directly with the network, bypassing a host networking stack. This is possible due to the high I/O performance flexibility afforded in modern FPGA architectures and is a model that cannot be considered for accelerators like GPUs that rely on a host CPU for management. This approach has

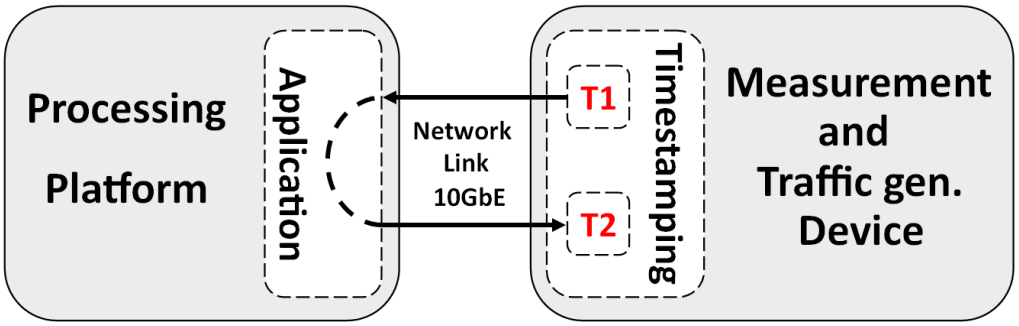


Figure 4.1: Outline of the experimental setup

seen use in a variety of scenarios [82; 83; 84], and has been demonstrated to lead to reductions in latency for specific applications compared to PCIe or purely software solutions. There has not been a characterisation of the detailed latency components introduced by this approach however, and there has not been an investigation into the effects of virtualising these devices. This model of accelerator integration poses additional challenges. Without a CPU-based host, control of the accelerator and virtualisation logic is more complex. Additionally, in some cases it can be more difficult to operate on large datasets, due to limited available storage.

4.4 Experiments

This section details the experimental testbed and the experiments carried out to characterise the communication latencies and throughputs of PCIe and network-attached FPGA accelerators.

4.4.1 Latency

An external device is used to measure round-trip times in order to achieve accurate measurements and ensure fairness between different scenarios. This is a Xilinx KC705 evaluation board programmed with specialised hardware to transmit, receive, and timestamp packets. Packets are sent over 10 Gb/s Ethernet using SFP+ transceivers and an optical cable.

The measurement procedure is shown in Figure 4.1. The measurement device sends packets to the platform under test, recording a timestamp as it leaves. The platform under test receives each packet up to the point where the application accelerator would process the data, then returns it to the measurement device,

where a second timestamp is recorded on reception. The round-trip time is the difference between these two timestamp values. Time values are captured using a free-running 64-bit counter implemented in the FPGA measurement device fabric, driven by the 156.25MHz physical board clock, giving a measurement precision of 6.4 ns. Measurements were taken over 20,000 back-to-back transmissions using a closed-loop model where the latency of one packet is recorded before the next experiment starts. The time taken for the packet to travel out of the measurement FPGA and to the platform under test over an optical fibre from all results (96ns) is subtracted. A 1 m optical cable was used to connect the measurement device to the platform under test in each experiment.

4.4.2 Throughput

To measure throughput, and FPGA measurement device generated traffic over the 10 Gb/s Ethernet link to the platform under test at as high a packet rate as possible, with the minimal 12 frame inter packet gap and fully saturating the AXI4-Stream interface to the Ethernet core IP. All packets received at the platform under test are looped back to the traffic generation board.

The measurement device generates traffic for a 5 second interval, and counts the number of packets it sends and receives back from the platform under test in this interval. Each packet sent in the interval is of the same size, and the tests were repeated for multiple packet sizes. Using the packets received per second, an average throughput is calculated.

4.4.3 Platforms

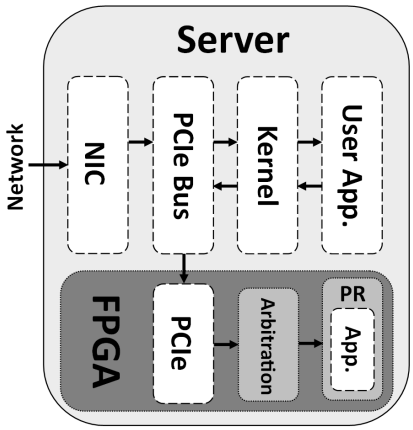
In these experiments 5 deployments were examined.

Host Server

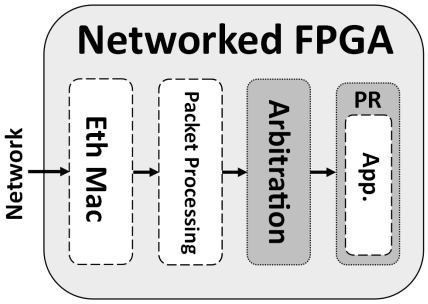
Server platforms typically found in a datacenter are represented in these experiments using a Linux server running CentOS 6.7 on a 12-core 2.20 GHz Intel Xeon E5-2650 v4 CPU with 64GB of RAM. The network card is a 10 Gb/s Mellanox MT26448, using SFP+ transceivers. The latency characteristics for CPU-based server platforms have been studied in detail in other work [96]—the measurements detailed in this chapter are used to provide a baseline for fair comparison only.

Data is sent over the network to this machine to be processed by a C++ application running in user space. The application is pinned to a processor core to improve performance.

For the throughput experiments, to maximise the number of packets receivable per second, the NIC ring buffer was configured to be as large as possible for the device used in these experiments.



(a) Server setup, with and without either a virtualised or non-virtualised FPGA accelerator



(b) Network attached FPGA accelerator, with and without virtualisation.

Figure 4.2: Configurations used to measure server based and network attached accelerators. Virtualised accelerators have arbitration logic and utilise PR regions for application logic.

PCIe Attached FPGA

These experiments used a Xilinx VC709 FPGA evaluation board hosting a Virtex 7 XC7V690T FPGA as the accelerator platform. Tests were carried out tests for both virtualised and non-virtualised FPGA accelerators. For both accelerator configura-

tions used PCIe Gen3×8 was used. For non-virtualised accelerators the design was based on the open-source RIFFA framework [193]. It compiles the PCIe communication interface with a fixed accelerator and provides a simple API that abstracts low-level transfer mechanics. This static accelerator cannot be replaced without significant interruption to the whole system. The architecture, shown in Figure 4.2a, is typical of a fixed-function accelerator found in the datacentre, comprising a generic shell of communication logic that simplifies the deployment of applications.

For this scenario, the C++ application running on the host receives the packets from the tester FPGA, writes them to the FPGA accelerator using the RIFFA API, reads them back, then sends them to the measurement FPGA through the network interface. The write to the FPGA is non-blocking. During the throughput experiments packets were transferred to the accelerator packet-by-packet, and not batched.

To test a virtualised FPGA, a version of the DyRACT [75] framework was used, allowing for the application logic to be modified at runtime without having to reconfigure the entire FPGA. This allows for multiple accelerators to run independently on the same device, with dynamic swapping of accelerators using partial reconfiguration triggered through the same PCIe interface used for data. This framework thus includes extra logic that contributes to additional communication delays as well as additional software components within the driver. The experiments used the same C++ application as with the non-virtualised FPGA but using the DyRACT API.

Network Attached FPGA

Finally, the proposed direct attachment of accelerators to the network is considered, as might be employed in standalone compute units, smart switches/routers, or smart NICs. The Xilinx VC709 evaluation board was used to test this. The board sends and receives 10Gb/s Ethernet through SFP+ transceiver modules attached to the FPGA fabric instead via PCIe. This data travels through the physical interface and Xilinx 10G Ethernet subsystem IP, which includes the PHY, PCS/PMA and MAC layers, interfacing inside the FPGA over an AXI4-Stream interface. A pipeline of 3 state machines is used to remove the Ethernet, IP, and UDP headers, with the UDP payload passed to the accelerator. In these experiments the payload is looped-back out of the application logic, through the network stack and back out of the device, using the same method in reverse. Figure 4.2b shows this architecture.

For these experiments the design was also modified to make it more representative of a virtualised platform, by adding an additional stage that checks the

destination address and directs the data to the correct accelerator among multiple. An additional FIFO buffer is also placed between this stage and the application logic. An arbiter FIFO is implemented before the transmit side of the network stack. This logic uses round-robin arbitration to allow the multiple accelerator slots to share external bandwidth.

4.5 Results

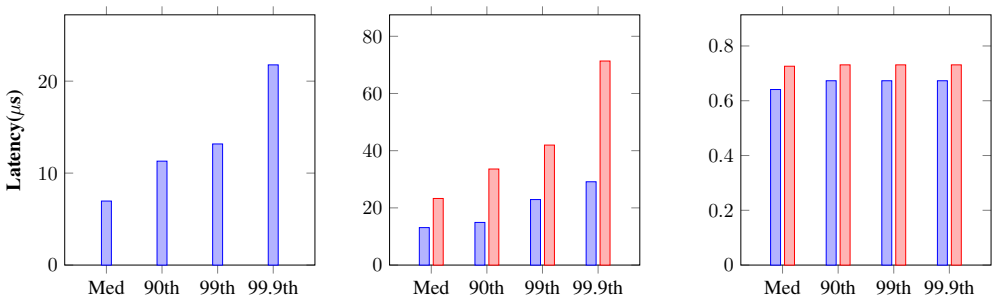
Experiments were conducted using 20,000 back-to-back 80B UDP packet transfers, generating the results shown in Table 4.1, and Figure 4.3. All results are in microseconds. A small packet size, close to the minimum frame size is chosen to give a better indication of the minimum unavoidable latencies in the scenario of interest. Packets were sent using a closed model, where the latency measurement of one packet was taken before the next packet was generated. This was done as uncontrolled transmission from the measurement FPGA to the server based platforms would result in queuing, buffer overflows, and dropped packets.

Scenario	**Median**	**90th perc.**	**99th perc.**	**99.9th perc.**
Server	6.961	11.300	13.170	21.770
Server+FPGA	13.100	14.910	22.910	29.130
Server+VFPGA	23.290	33.590	41.960	71.350
Net FPGA	0.667	0.673	0.673	0.673
Net VFPGA	0.726	0.737	0.737	0.737

Table 4.1: Latency results for 20000 back-to-back UDP packets in microseconds.

4.5.1 Median Latency

It is clear that adding an FPGA accelerator to a server approximately doubles the packet return latency. This is due to the additional movement of the data over PCIe to reach the accelerator and back. Enabling virtualisation on the PCIe-attached FPGA adds a median delay of around 10μs. Part of this is the additional logic added to the FPGA design, but the change in software running on the host server also significantly contributes to this extra delay. Utilising more of the available virtualised slots on the FPGA could also potentially increase latency further if there is significant saturation of the PCIe interface due to contention. Hence an FPGA accelerator must accelerate an application by a sufficient amount to overcome these



(a) Server (1) (b) Server/w FPGA (2/3) (c) Networked FPGA (4/5)

Figure 4.3: Tail latency results of 20000 back-to-back UDP packets for each scenario. Latencies are shown for the Median, 90th, 99th, and 99.9th percentiles.

additional latencies.

The network-attached FPGA has latency an order of magnitude smaller, as packets bypass the PCIe interface of the network card, the server network stack, and the PCIe link to the FPGA. While these delays therefore seem insignificant, network-attached accelerators are often deployed for low latency, or high data rate applications, where even the sub-microsecond delay introduced could be relevant.

4.5.2 FPGA Latency Breakdown

Hardware counters in the FPGA are used to isolate sources of the delays in the device; results are shown in Table 4.2.

Delay Source	**Median Delay(μs)**
Packet-processing logic	0.135
PHY+MAC	0.532
User logic (simple loop-back)	0.013
Virtualisation logic (for V.FPGA)	0.059

Table 4.2: Breakdown of delays measured from round trip times of a packet travelling through a networked FPGA accelerator design, for 80B UDP packets.

The packet-processing layer, which strips the layer 2/3/4 headers and passes data to the accelerator and user logic in these experiments contributes to the total delay less than the PHY and MAC cores. In more complex designs that may implement more complete layer 3 and 4 functionality, packet-processing delay will be higher but a comparable order of magnitude. Additionally, virtualisation logic adds further delay when there are multiple active accelerators present, as network

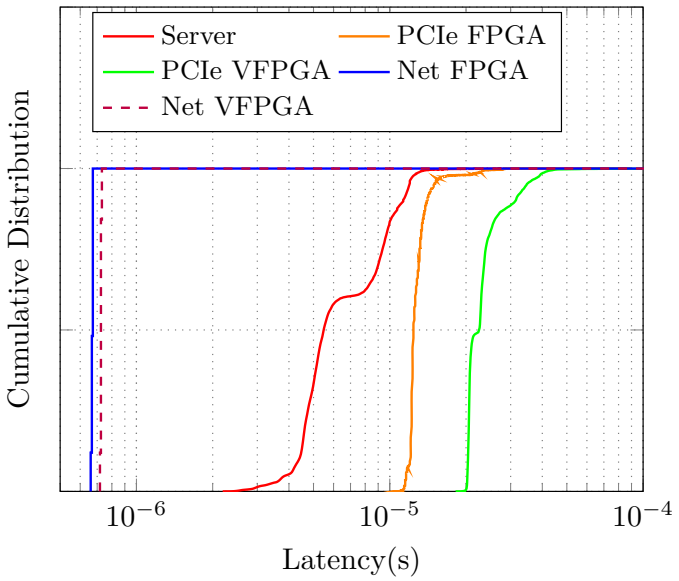


Figure 4.4: Cumulative distribution function (CDF) of latencies for each scenario.

access to different accelerators must be arbitrated. In this design, the accelerator logic loops data back using a FIFO, so adds minimal latency of a few clock cycles. For a real processing task, this latency would depend on accelerator datapath.

4.5.3 Latency Distributions

The CDFs in Figure 4.4 show the latency distributions for each scenario, demonstrating the relative magnitudes and variations for the alternative platforms (note the logarithmic x-axis).

4.5.4 Tail Latencies

The tail of these latency distributions is also important, shown in Figure 4.3. While the median latency is a useful measure, large spikes, even if infrequent, can significantly impact latency-sensitive applications and undermine a system that functions well most of the time [187]. Latencies at the distribution tail have also been acknowledged to have a negative impact on applications where processing is distributed across many machines in parallel, and overall completion time is dependent on the slowest response [187]. To examine these latencies, the 90th, 99th and 99.9th percentile latencies were measured in each scenario, with results shown in Table 4.2.

The server running software has around a 6μs difference between the median and 99th percentile, almost double the delay, while the 99.9th percentile latency is

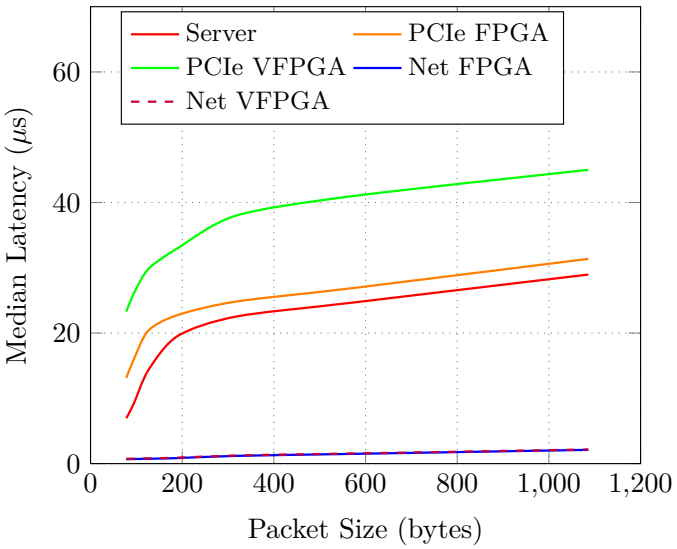


Figure 4.5: Median latency for differing packet sizes.

around 3× the median, meaning that 1 in every 1000 packets may have a latency this high.

The PCIe FPGA accelerators show less of a latency spread relative to the median. There is around a 2μs difference (14%) between the median and 90th percentile for the non-virtualised FPGA, and a 10μs difference (75%) between the median and 99th percentile, suggesting that this variability is primarily due to the host. There is minimal latency variation for the network-attached FPGAs, due to all packet-processing being done in dedicated hardware. As soon as software network stacks are introduced, latencies become significantly less deterministic.

4.5.5 Packet Size

Experiments were repeated with different packet sizes, with results shown in Figure 4.5. Each run of the experiment used 20,000 UDP packets of the same size. All platforms show an increase in median latency as packet size increases. The Server and PCIe platforms show greater sensitivity to packet size, with the virtualisation causing further increases. The PCIe accelerators add a relatively static overhead on top of the host latency, regardless of packet size. Both network-attached FPGAs show smaller increases in latency. The initial increase in latency for packet sizes up to around 200 bytes is significant, but reduces as packets grow further.

4.5.6 Throughput

Results are shown in Figure 4.6, for varying packet sizes. These are values measured with no actual processing and therefore represent the upper limits enforced by the communication infrastructure. These measurements include the reception of the packets at the network interface, and transfer to the accelerator.

Predictably, the network-attached FPGA platforms approach line rate for 10Gb Ethernet, as there is no software involvement or additional interfacing. Adding the extra virtualisation logic has minimal effect on the throughput, as the virtualisation is all hardware-based. What is likely to cause reductions in effective throughput for a particular accelerator is when there are multiple accelerators deployed across the other virtualised slots, and this bandwidth must be shared.

The server-based platforms suffer considerable throughput penalties in comparison. The host running software was limited to a bandwidth of 153MB/s. While the NIC hardware was capable of receiving data at line rate, the application was not able to fetch the packets from the buffer fast enough, causing overflows and thus lost packets. This was despite pinning the application to a CPU core, and maximising the size of the ring buffer allocated to the NIC. Adding the non-virtualised PCIe accelerator into the path resulted in the throughput being reduced to 133MB/s on average, a 13% reduction. The virtualised FPGA accelerator caused an even greater reduction in the total throughput, to around 90MB/s, a 41% decrease. The extra software driver components associated with the virtualisation are likely to be the main cause of this.

The achievable packets that can be received and then transmitted back out for each platform, per-second, are shown in Figure 4.7. This metric can be important for some streaming applications that rely on packet-by-packet processing. The network-attached FPGA platforms again show the ability to process packets at line rate. The decrease in packet-processing rate as packet size increases is due to the maximum number of packets that can be transmitted on a 10Gb Ethernet link decreasing as packet size increases.

The server and non-virtualised PCIe accelerator show a slight decrease in packet rate as packet size increases, likely due to the ring buffer allocated to the NIC and the socket buffer using pointers instead of the packet data itself. This means the buffers can hold the same number of packets regardless of the packet size. The slight decrease in the packet rate can then be attributed to the handling of packet data in the user-space application.

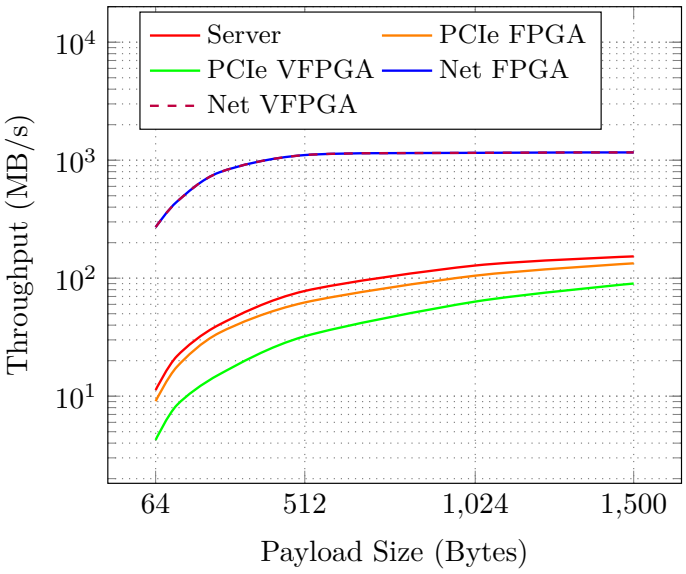


Figure 4.6: Average measured throughput in MB/s for varying frame sizes.

4.6 Discussion

4.6.1 PCIe Accelerators

Utilising PCIe FPGA accelerators results in a significant additional latency for distributed streaming applications. For latency-dependent applications, these measurements can be used to weigh the communication costs against the computational benefits of offloading to the accelerator. For larger accelerators with greater computation times, this communication latency can form a smaller percentage of the total delay associated with the accelerator. Virtualising the PCIe accelerator increases latency, mostly due to the extra software that manages virtualisation. Driver optimisations and improved software control of the virtualised accelerator could significantly improve latency.

While offload to PCIe accelerators has traditionally focused on maximising throughput, especially for large machine learning applications, distributed data across a network introduces a significant throughput bottleneck. In these experiments, the FPGA is hosted on a PCIe Gen3×8 slot which is capable of much higher throughput than measured. The connection to the network via the host prevents this interface being utilised fully. Batching can mininmise PCIe overhead and enhance throughput, but this introduces extra latency, undesirable for streaming applications. Even with large batches, total throughput is limited by the host and its networking stack.

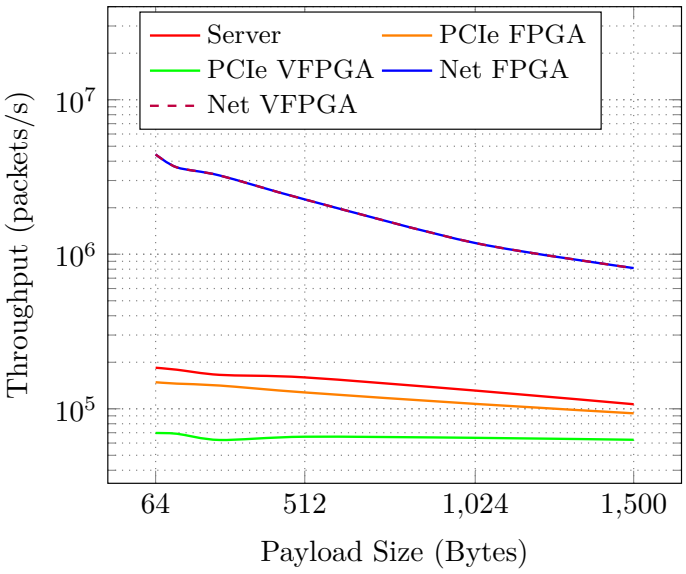


Figure 4.7: Average measured packets-per-second for varying frame sizes.

4.6.2 Network-attached Accelerators

The communication latency associated with the network-attached accelerators is predictably much lower than PCIe hosted accelerators. Communication latencies below a microsecond mean that the application logic is likely to dominate the total latency. Virtualisation logic added minimal additional latency - however when there is competition for the shared communication resources from accelerators in different virtualised slots, this is likely to change. The biggest contributor to latency is the physical and MAC layers, implemented in the Xilinx Ethernet core. Virtualisation of the network-attached FPGA had less of an impact than with the PCIe-attached FPGA, as the virtualisation was entirely hardware based. The network-attached FPGAs had very consistent latency performance, with minimal variation, making them well suited to applications such as consensus, where overall response time is limited by the highest-latency node. Tail latency is a limiting factor in large distributed applications [187], so adding accelerators at high fan-in stages yields the greatest benefit. At these stages, the overall response time is limited by the slowest responding node, and small infrequent spikes can have a greater effect on the overall latency. As such, using platforms with more deterministic latency is beneficial. In terms of throughput the network-attached accelerators can process incoming packets at line rate.

With the datacenter moving towards 40Gb/s and 100Gb/s, the need for direct offload from the network interface to the accelerator is increasing, via smart

NICs or other networking elements that enable FPGA offload. Network-attached FPGA accelerators however pose additional challenges, mainly focused around virtualisation, and managing resources across multiple applications with minimal software involvement.

4.7 Summary

This chapter presents detailed experiments to measure the communication latency characteristics of FPGA accelerators in a datacenter context. It showed the latency overheads inherent in traditional deployments of accelerators hosted on a server through PCIe, and the emerging approach of network attached accelerators. This includes how latency is affected by packet size, and the latency distributions and tail latencies. It additionally detailed the throughput limitations of these deployments in the context of distributed applications where data is received over the network.

These measurements can be used to aid in design decisions in the deployment of FPGA accelerators. For PCIe deployments, latency and throughput measurements can be used to calculate optimal batch and buffer sizes for different applications. For network attached accelerators, the measured latencies were comparatively small, but these devices are often used for applications that require ultra-low latency, where even sub-microsecond communication delays measured are relevant. These small delays will be even more relevant with emerging 40Gb/s and 100Gb/s networking.

Chapter 5

Near-Edge FPGA Acceleration for the Internet of Things

5.1 Introduction

With the Internet of Things driving an explosive growth in the connectivity of resource constrained computing platforms, the latency implications of cloud-based computation offload become an important consideration in deciding how to deploy distributed applications. Edge computing represents the broad paradigm of moving processing away from high performance centralised datacenters towards data sources at the periphery of the network. Computing resources are typically less capable at edge nodes but communication is minimised since data need not be moved up the network to be computed on. Communication latency to the cloud can be significant, but for complex applications where the capabilities of datacenter servers offer a significant improvement in computation latency, overall application latency can be improved. This interplay between communication and computation latencies is heavily influenced not just by the network distance but also the choice of processing platform and the complexity of moving packets from a network interface to the processing hardware. As application latency has become more important and hardware at the network edge has improved, the benefits of offloading to centralised computing resources is now heavily impacted by these inherent communication delays.

In [194], the authors demonstrated the significant impact of cloud offload latency on the performance on neural network applications considering different locations of network "edge" processing. The content within this chapter expands upon that work by considering how the choice of computing platform impacts both communication and computation latency.

There has been an increasing trend of adopting heterogeneous specialised hardware in the datacenter to improve computation performance and efficiency. FPGAs in particular have seen increased use in the datacenter due to their flexibility and increased performance per watt compared to CPUs and GPUs in a variety of applications [181; 141; 195; 182].

Cloudlets, also referred to as edge servers, are small-scale datacenters or servers deployed close to data sources in an attempt to provide cloud-like services a few hops away in the network [12; 13]. Data traverses fewer switches over a LAN instead of the Internet, reducing communication delay and improving predictability. Cloudlets may utilise capable hardware comparable to that found in larger cloud datacenters, including hardware accelerators.

Enhanced computation capability at source nodes is an additional emerging trend, where simple microcontrollers are giving way to more capable single board computers like the Raspberry Pi, allowing more complex computation to take place at the data sources. FPGA acceleration is also possible at these nodes, through the use of FPGA SoC platforms such as the Xilinx Zynq, which tightly-couples an FPGA fabric with an Arm Cortex A9 processor. These have the advantage of running commodity software for programmability, tightly-coupled with custom hardware accelerators for offloaded computation, with the potential for reconfiguration to support dynamic workloads. Application-specific accelerators such as the Google Edge TPU allow for computationally-intensive machine learning applications to run on specialised hardware at the network edge. These more capable embedded devices can also be used as cluster heads for sensor nodes, or gateways between a collection of data sources and the rest of the network.

The emerging paradigm of in-network computing [189] sees the high-performance network switches used to route packets extended with the capability of computing on that data in-flight [196]. Application tasks can be deployed to heterogeneous networks comprising both dedicated computing resources in the network and extended network elements [2]. FPGAs are a key candidate architecture for such a paradigm, as they are well-suited to packet-processing [183], support tight coupling of custom hardware accelerators, resulting in lower latency computation with minimal additional offload latency, and support the dynamic reconfigurability required to support sharing among multiple applications [197].

This chapter uses a case-study of neural network-based image classification application to explore the latency implications of different computation offload strategies, assuming a network of edge nodes interacting with a cloudlet server through an FPGA-based layer 2 switch. Using this, it explores how edge, in-network,

and cloudlet deployments with heterogeneous hardware impact overall performance for a complex application that would traditionally be offloaded to the cloud. The experiments consider how both the communication and computation latencies of different deployment approaches impact overall application performance, including for shared resources. While network round-trip delay contributes significantly to communication latency, this work also considers the packet ingestion latency required to process streaming network data, which becomes a more significant component as processing is moved nearer to the data sources, and is heavily-impacted by platform choice.

5.2 Contributions

The contributions in this chapter are as follows:

- Demonstrating the offloading of computation from edge nodes to more capable networked resources, including hardware accelerators, in an image classification application;
- Detailed latency breakdowns for a range of different types of compute resources used in near edge computing, differentiating between computation and communication delay;
- Demonstrating a method upon which compute can be integrated into an FPGA based network switch;
- Demonstrating how the latency of the application scales with the number of devices;
- Using these results, discussion of how FPGAs integrated into the network infrastructure are well suited to near-edge computing, and scale well with more data streams.

5.3 Related Work

For data to move to a large cloud platform such as Amazon EC2 it must travel through a WAN such as the internet, resulting in large and non-deterministic delays, in the range of 10-100s of milliseconds depending on datacenter location [175]. In Chapter 2, a range of work is discussed that examines the contribution of PCIe and network interface cards to system latency. Edge and cloudlet computing is also

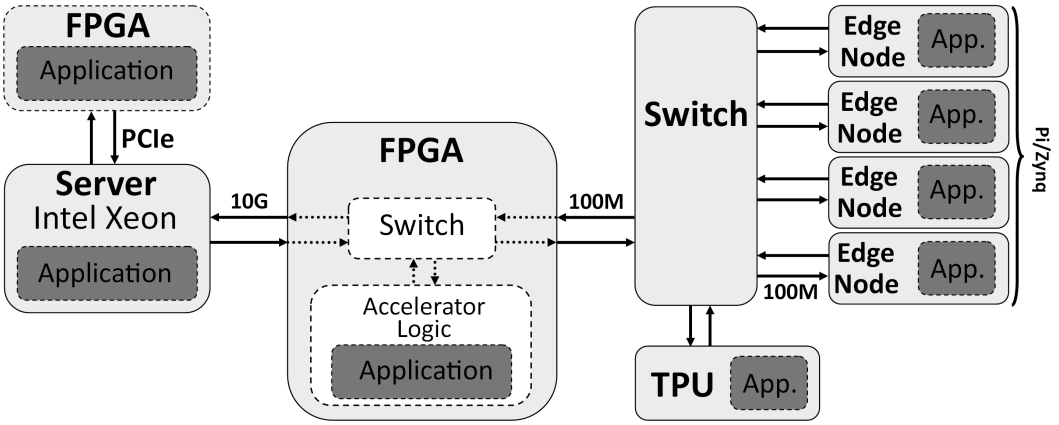


Figure 5.1: Outline of the experimental set up - images are sent from the edge node through the network switch to the Server. In these experiments, latency is measured as the total latency to complete the computation on each platform, returning the result to the source node, including communication time.

discussed, and various works cited demonstrate the benefits in moving processing closer to the edge of the network.

5.4 Design and Experiments

The experiments in this chapter consider a distributed application where data is captured at edge nodes, transmitted through network switches via Ethernet, to a server that acts as a cloudlet platform, a common architecture for IoT applications. An image classification application applied to images generated at the edge of the network is used. The network architecture comprises 3 layers, an edge layer, a switching layer and a cloudlet layer. Edge nodes produce images, and can transmit them via 100Mb Ethernet through layer 2 network switches, which can forward data to a cloudlet server. All layers in the topology can potentially perform the image classification tasks. The edge node has lower compute capability, but has no communication cost, the server has high compute capability but data must travel there and is subject to communication delay. The general structure of the system can be seen in Figure 5.1.

5.4.1 Application

The experiments use a neural network based image classification application that processes images streamed from the edge of the network. The evaluation is not focused on the application itself, but rather the relationship between communica-

tion and computation latency for such distributed deployments. This example has been selected as it involves the transfer of considerable amounts of data, as well as high enough computational complexity, while also demonstrating a good range of computational scaling on these different architectures. This investigation focuses on latency – the time taken to receive a result – rather than throughput. This is important in a variety of safety-critical applications or where data is time sensitive.

The image classification application uses a Deep Neural Network (DNN) called SqueezeNet [198], that has a small memory footprint, designed for resource constrained platforms. In recent years there has been significant research into optimising DNN architectures for inference at the edge on hardware with less capable resources, trading off accuracy, runtime, and energy consumption [199; 200]. These models aim to reduce complexity and memory footprint while maintaining accuracy. In [194], the authors demonstrated how the performance of such compact DNNs scales on traditional CPU-based edge nodes, while also evaluating network delay based on the location of the edge device. SqueezeNet was selected as it is a compact model with a lower number of operations, leading to comparatively lower inference time, feasible for deployment on constrained platforms, compared to other models [201]. This better allows us to focus on the communication latency considerations, rather than applications where computing latency significantly dominates. It also means the lessons learnt are not tied as tightly to the capabilities of the specific architectures used. SqueezeNet is also representative of the type of DNN designed for use on edge devices such as the Raspberry Pi.

Input images are 224×224 pixels with 32-bit combined channel depth. The neural network has 10 layers, and uses 32-bit floating point weights, totalling 4.7MB. It can be compressed to 0.5MB while retaining the same accuracy [198]. The model is pre-trained and weights pre-loaded on all platforms. Images stream from the edge nodes, mimicking connected camera sources. To avoid saturating the network images are sent one at a time, and the time taken to receive a result measured before the next image is sent. Images are processed to determine the probability that each image belongs to one of the 1000 classes classified by the model.

5.4.2 Measurements

The total time taken to carry out the classification task on each platform is measured, focusing on latency, rather than throughput. This includes the computation latency, as well as the communication time to transfer images from the edge node to the target computing platform, and to receive the result.

All measurements are taken utilising specialised timing hardware imple-

mented in the FPGA network switch, allowing for consistent measurements across the different platforms and independence from other tasks running on the various platforms. The edge nodes start transmitting an image upon receiving a start command from the FPGA which records the start time for the experiment as this command is sent. The finish time is recorded when the edge node receives the result. A free running counter running at 200MHz gives a resolution of 6.4ns.

5.4.3 Platforms

Edge Node (1)

Edge nodes may act as a gateway or cluster head, or the data source itself. An example of such a node is implemented in these experiments using a Raspberry Pi Model 3B running Raspbian OS. Up to 4 edge nodes can be connected to the switch in these experiments. It can carry out the full computation using a Keras implementation of SqueezeNet, or transmit the image through the Linux sockets API over 100Mb Ethernet through the network switch to another platform. Image data is sent using raw Ethernet frames, with no layer 3 or 4 headers. 100Mb Ethernet represents a realistic channel for a lightweight edge node, where LPWAN would be too slow or cellular too costly for video streaming. While network bandwidth continues to scale, the ingestion latency required to process received data at a node does not scale proportionally.

To measure the computation latency on a Raspberry Pi, the Python application opens a raw Ethernet socket and waits for the start message from the FPGA network switch. Upon receipt of this frame, the Python script triggers the Squeezenet application. Once computation is complete, it sends an Ethernet frame containing an 'end' message back to the FPGA, triggering the capturing of another timestamp.

The difference between these two timestamps is taken to be the computation time on the Raspberry Pi. In order to take into account the time to send and receive the start/end frames, the same process was performed, with no computation on the Pi, just sending the start and end of test frames, and subtracted this time from the original result.

FPGA-Accelerated Edge Node (2)

Edge nodes may be enhanced with hardware accelerators or co-processors. For these experiments, an FPGA-accelerated edge node was built using the Xilinx Zynq based Arty Z7 small form factor development board suited for edge applications. The Zynq

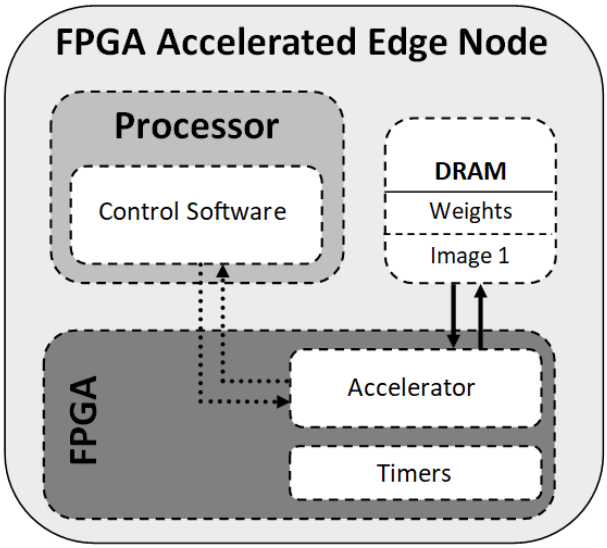


Figure 5.2: High level diagram of the Zynq FPGA accelerated edge node. Software running on the processor subsystem controls the accelerator on the tightly coupled FPGA.

consists of a processor subsystem (PS) and programmable logic (PL) and functions similar to the Raspberry Pi but with compute-intensive functions offloaded to a hardware accelerator in the PL.

For these experiments, the SqueezeNet accelerator logic was generated using Vivado HLS, and is implemented in the PL using a heavily-modified version of the implementation in [202]. HLS allows for the expression of accelerator logic in annotated C rather than a hardware description language like Verilog. Accelerator parameters such as the memory offsets and sizes of layers are configured through an AXI-Lite register interface, through software running on the PS. For each layer of the DNN, the software sets the appropriate parameters in the accelerator before starting it. Weights and image data are stored in on-board DRAM and data can also be temporarily stored in PL memory while being used. In order to measure the latency of the computation, the same method used with the Pi was utilised. The approach is shown in Figure 5.2.

TPU-Accelerated Edge Node (3)

An alternative approach is to attach an application specific accelerator to an edge node such as the Google Coral board hosting an ARM processor and tightly coupled Edge Tensor Processing Unit (TPU) coprocessor. The execution of Tensorflow mod-

els can be offloaded from the host processor to the TPU for significant improvements to latency and throughput.

In order to run the Squeezenet model on the TPU, some optimisation of the neural network parameters to abide by the limitations of the architecture are required, and these can be done using the Tensorflow Lite software. The model was simplified using quantization aware training, and then converted to Tensorflow Lite and compiled using the Tensorflow Edge TPU compiler.

Platforms such as the Edge TPU may be used as shared computing resources available over a network. This scenario is modelled by connecting a Raspberry Pi edge node to the TPU board through the network switch over 100Mb Ethernet. The FPGA network switch generates a start message, triggering the Raspberry Pi to transmit its data through both network switches to the TPU, where the software running on it receives the data through a raw socket. Upon reception, the model is executed on the TPU hardware accelerator, and upon completion, a message is sent back to the FPGA where the time taken for this process is recorded.

Cloudlet Server (4)

The server in these experiments represents a cloudlet, a small form-factor server that can provide compute resources close to the data source. Data would only have to travel through a LAN to reach the cloudlet, as opposed to a WAN to reach a cloud or datacenter. In these experiments, a Linux server running Ubuntu 18.04 on a 12-core 2.2GHz Intel Xeon E5-2650 v4 CPU, with 64GB of RAM is used. This machine has a 10Gb/s Mellanox Technologies MT26448 network card with SFP+ transceivers. The application runs via Python, with frames sent and received through the sockets API and the SqueezeNet model executed using Keras.

To measure the application latency, the server sends a start command to all edge devices, through the switches. Once the FPGA switch has forwarded that frame, it captures the first timestamp. The edge devices then send the images through the switches to the Server. On the server, a Python application runs that listens to a raw network socket for frames received from the edge nodes. Buffers are instantiated to hold image data from each edge node being used in the experiment. The source address field of the received frames are used to direct the data into the appropriate buffer. When a buffer has the required image data, the same Squeezenet model is run using the Keras front-end. Upon completion of the computation, a frame is sent to the corresponding edge node through the FPGA network switch, where the second timestamp is recorded. The two timestamps are then used to calculate the total latency. The Python application uses the multiprocessing library

to generate a separate process for the computation, the sending, and the receiving of data. This technique allows for the opening of multiple concurrent processes, and data can still be sent and received while the model is executing.

In order to separate the computation and communication latency, a variant of the experiment was run without computation - the Python script on the server simply sends a response to the Pi immediately after receiving the full image.

FPGA-Accelerated Cloudlet Server (5)

Computation can also take place on an FPGA accelerator attached to the Linux server via PCIe. In these experiments this is implemented using a Xilinx VC709 evaluation board that has a PCIe Gen 3×8 interface. On the FPGA fabric the DyRACT framework [75] is used to manage communication between accelerator and host. The software application on the host opens a network socket and waits to receive an image from the edge node, triggering the the accelerator when the image is received. Image data is sent to the device using a C API provided by the framework, and logic on the FPGA tracks the amount of image data received. Once the full image is received, it triggers the start of the accelerator, and once computation is complete initiates the transfer of the result back to the host server.

To measure the latency of this platform, the same method is used as for the regular Server. Additionally, to isolate the computation and communication time, additional timers on the PCIe attached FPGA to measure the time taken for the accelerator to complete its computation once the full image had been received.

In-Network Processing (6)

Systems with many networked devices will deploy network switches to facilitate the routing of traffic from the data sources and edge nodes to a cloudlet server or wider network. The testbed uses an unmanaged layer 2 switch and a Xilinx KC705 evaluation board, which hosts a Kintex-7 FPGA to connect edge devices. The standalone layer 2 switch allows 4 edge devices to be connected to the single 100Mb port on the FPGA board. The FPGA switch bridges the RJ45 100M Ethernet interface of the edge nodes and the 10Gb Ethernet SFP+ interface of the cloudlet server, and can also host a hardware accelerator. When a packet is received through the 100Mb interface from an edge node, the destination field in the frame header determines where it is sent as part of the switch logic.

If the field contains a specified address, the payload is transferred to the board DRAM, to be used with the accelerator, otherwise it is forwarded to the 10Gb

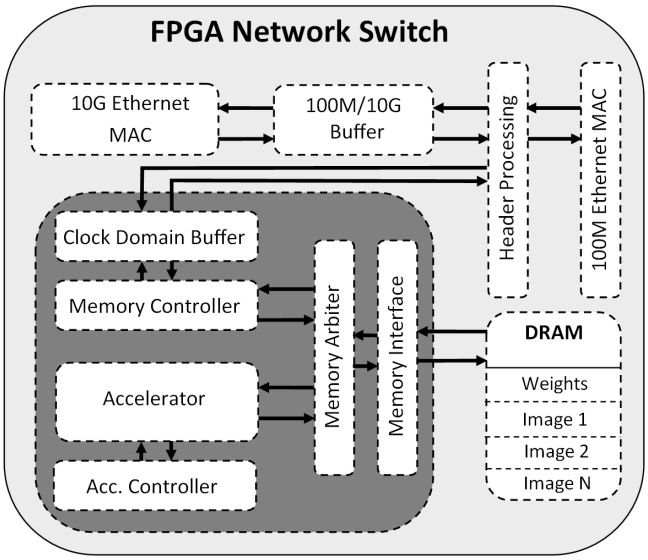


Figure 5.3: Outline of the FPGA based network switch structure. Packets are typically routed from input to output interface, but if they have a specific destination address, are sent to memory for use in computation.

output port. Sending the frames through to the 10G port involves buffering the data in dual clock FIFOs and moving it across clock domains, and then moving it to the 10G Ethernet MAC. As up to 4 edge devices may be transmitting to the FPGA, the source address field of the inbound frames is used to differentiate between data sent by each device. Image data from each device is written to a different memory address offset in DRAM based on the source address. Once determined to be image data bound for the accelerator logic, the payload from the received frame must be buffered and moved to the memory interface clock domain using dual clock FIFOs. Data is then transferred to the DRAM memory interface using AXI4. A custom AXI-4 controller was designed to facilitate the transfers for these experiments. The payload of each frame is 1024 bytes, and a transfer width of 64 bytes with a burst length of 16 are used to minimise the AXI overhead. Counters are implemented to keep track of how much image data has been received from each connected edge device, and are incremented for every frame written to memory. Upon reception of a full image from any edge device, a request is generated to start the accelerator for that device. This approach is summarised in Figure 5.3.

The SqueezeNet accelerator logic was generated using Vivado HLS as for the accelerated edge node, but the expanded resources of the Kintex FPGA on the KC705 allow for greater unrolling of loops and increased parallelisation of the de-

sign. It interfaces to DRAM using an AXI-4 interface, and parameters are configured through a separate AXI-Lite register interface. Custom AXI-Lite logic to monitor and control the accelerator. Before a full image has been received, the controller awaits a request to start the accelerator. Once it receives a request,the controller writes the appropriate parameters for the first layer of the DNN and then starts the accelerator. It provides different memory offsets based on what edge device the computation is for. The accelerator fetches the required weights and image data from DRAM, via an AXI-4 interface. Data is temporarily stored in BRAM within the accelerator logic while in use, to reduce the number of costly DRAM transfers required. This process is repeated for each of the 10 layers of the DNN. Upon completion of the computation, the accelerator writes the set of results to DRAM. The controller then facilitates the transmission of these results through the 100M Ethernet interface, back to the Raspberry Pi edge node. The DRAM interface for Xilinx 7 series FPGAs only allows for a single port, therefore an arbitration mechanism is required to allow the accelerator and network logic to share a single interface. This arbiter has 2 AXI-4 slave ports connected to the network and accelerator and a master port connected to the memory interface. Priority is given to the network interface to avoid buffer overflows and the loss of data. The board DRAM also contains the full set of weights for the accelerator, and weights are loaded prior to the start of the experiment, so there is less accelerator management overhead.

To measure the latency when using the FPGA network switch for computing, as mentioned previously, the FPGA sends a start command to all connected edge devices, and records a timestamp using the free running counter and 200MHz system clock marking the start of the test. The edge nodes run C code that opens a network socket and listens for the start command. Once received, edge nodes send the image data to the FPGA switch via raw Ethernet frames. Once the accelerator has finished its classification for a given edge device, a second timestamp is recorded. The recording of timestamps is independent for each edge device. The difference between the first and second timestamp recorded on the FPGA is taken as the total time.

Additional, secondary timers were implemented on the device, to isolate the time taken for the accelerator to complete its computation only. The timer starts once the full image is received, and stops once the accelerator raises the complete flag.

5.5 Results

As part of these experiments, the total application latency for a distributed image classification application where data is captured at an edge node, and can be transmitted through a network switch to a server with access to an FPGA accelerator, was measured. Measurements were taken when computation was carried out on each of the platforms described in Section 5.4.

1. Computation is carried out on the edge node, a Raspberry Pi 3, in software;
2. Computation is carried out on a more capable edge node, a Xilinx ArtyZ7 FPGA SoC board, with hardware acceleration;
3. Computation is carried out on a specialised Google Coral Edge TPU board, connected via the network;
4. Computation is carried out on a server in software, connected via the network;
5. Computation is carried out on an FPGA accelerator integrated into the server via PCIe, connected via the network;
6. Computation is carried out on the network switch, on the same FPGA fabric that contains the switch fabric;

The experiments measured total application latency, including the time to send the image from the edge node to the platform carrying out the computation, ingestion of the data and computation, and returning of the result back to the edge node. The communication and computation time for each scenario were isolated.

5.5.1 Isolated Edge Node Measurements

Firstly the measurements for a single active edge device were considered. The computation latencies measured differ from raw execution benchmarks, such as in [203], primarily because the latency measurements include the time to ingest input data packets, rather than raw inference time for data already in memory. This distinction is important in the context of offloading computation where data must be received over the network.

It should be clarified that the experiments do not consider the network communication latency to the location of the edge device beyond the single hop, but rather the network ingestion and computation latencies, as these are platform dependent. The experiments in [194] offer an insight into how locating these platforms

may further impact overall latency, but the in-network approach has fundamentally lower latency.

The impact of multiple edge devices competing for shared network and computing resources is discussed in Section 5.5.2.

Computing Location	**Comp. Latency**	**Comm. Latency**	**Total Latency**
1) Edge (Pi)	2380	0	2380
2) Edge (Zynq)	1660	0	1660
3) TPU	210	80	290
4) Server	340	50	390
5) Server + FPGA	60	60	120
6) Network Switch FPGA	60	1	61

Table 5.1: Computation and communication latency for offload from a single edge node in milliseconds.

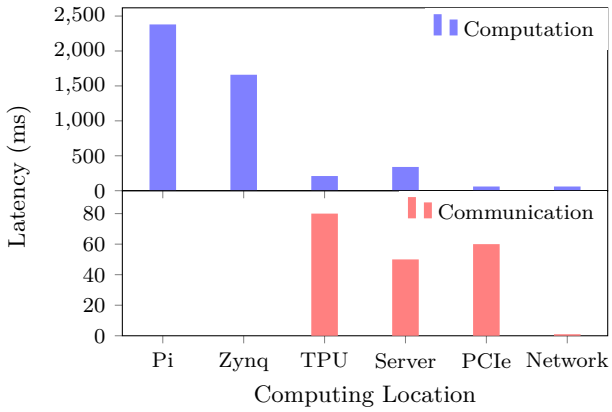


Figure 5.4: Breakdown of latencies per image for different offload scenarios.

It can be seen in Figure 5.4 that performing all computation on the edge node with no acceleration has high computation latency, and despite having no communication latency, this scenario performs worst, justifying computation offload. Less computationally-intensive applications would reduce this disparity between platforms since the computation latency would not be so dominant, but a simpler neural network was selected for this study, and others are likely to show even more disparity.

Replacing the edge node with a more capable Xilinx Zynq SoC platform improves latency by 30% due to the hardware acceleration provided by the FPGA. The embedded FPGA SoC, however, is limited in capacity and cannot fully parallelise

execution of the neural network, and so numerous iterations of hardware execution are managed by software, adding to the latency. Further optimisation is possible, for example, 8 bit fixed -point quantization, which would improve area efficiency, though require further design effort.

The Edge TPU was deployed as a network-connected accelerator on account of its cost compared to other options meaning it is more likely to be a shared resource. It demonstrates a significant reduction in computation latency on account of the optimised parallel hardware and native compilation of the model for this architecture that supports 8-bit arithmetic. As an offload engine attached to the network, there is some communication latency due to the software network stack running on the ARM core on the Edge TPU board. The Zynq FPGA SoC platform can also perform this role with lower ingestion latency by diverting packets directly into the PL for processing by the accelerator without the involvement of the PS processor, as has been demonstrated in [204].

Offloading to the server outperformed both the Raspberry Pi and FPGA SoC platforms due to the more capable server processor. Indeed, it achieves close to the computation latency of the Edge TPU, albeit at much higher power consumption. Improved network stack performance on a server-class processor also results in lower communication latency compared to the Edge TPU.

Adding a more capable hardware accelerator using a server-class FPGA offers even lower computation latency compared to the other architectures discussed so far, as loops can be fully parallelised and there are fewer movements of data to and from memory. This FPGA accelerator is integrated into a server over a PCIe link which adds a modest communication latency as data must first traverse the network stack, then be moved to the accelerator over PCIe. The software stack that manages this accelerator also adds some management overhead that contributes to the communication latency. However, the significant acceleration of the computation means total latency is significantly reduced.

Attaching the FPGA accelerator directly to the switch drastically reduces communication latency as network packets can be directly ingested by the accelerator, and all data movement is managed in hardware. This coupled with the low computation latency of the FPGA accelerator means this deployment has significantly lower latency than those discussed earlier. The communication latency measured here does not include multiple hops over a network, as characterised in [194], which would further increase the magnitude and variability of communication latency, rather the isolated the platform-specific components in these experiments.

While the Edge TPU is capable of very low inference latency, measured at

just 6ms in isolation, packet ingestion and data transfer from the host processor to the Edge TPU silicon increases latency. This highlights, once again, that the data movement is of paramount importance in determining the end-to-end performance of such a connected application.

This chapter is concerned with relationship between computation latency and communication (ingestion) latency, considering deep neural network inference as a case study. Further tweaking of neural network model parameters to optimise for latency against accuracy can improve performance on constrained edge platforms [205]. Alternative neural networks also exhibit different scaling of computation latency on different devices. The effect of varying these parameters on the different accelerators is an avenue for further work, as is the effect of other factors such as network, processor, and I/O stress.

5.5.2 Impact of Multiple Edge Devices

The results presented so far consider one edge device with exclusive access to the network and computing resources. In reality, multiple connected devices will stream data, leading to contention for resources and larger, more unpredictable delays. Computing on edge platforms, such as the Raspberry Pi and FPGA SoC, while having a higher latency individually, is not subject to these resource contention issues, as computation is done locally, with no communication cost. So for large numbers of devices, this approach may scale favourably. Cloudlet servers or networked FPGA accelerators will typically share computing resources across multiple streams.

To examine the effects of resource contention, the previous experiments were adapted to support different numbers of streaming edge devices. The experimental setup allows us to connect up to 4 edge nodes to the network switch and FPGA and take detailed measurements. The same experimental procedure was used: the FPGA switch generates a broadcast 'start' message to all connected devices, upon which all devices transmit their image data through the switch to the appropriate offload resource. Experiments were completed with a closed-loop traffic model, where one measurement was taken before the next experiment was started to avoid flooding the network with data and causing packet loss. While multiple streams were active in the system, the latency of a single stream is measured.

When using the cloudlet server, all edge devices share the output port of the layer 2 switch, the output port of the FPGA switch, as well as the network pipeline on the FPGA switch, in order to reach the server. At the server, the streams share the network and processor resources. Once a full image for the measured stream is

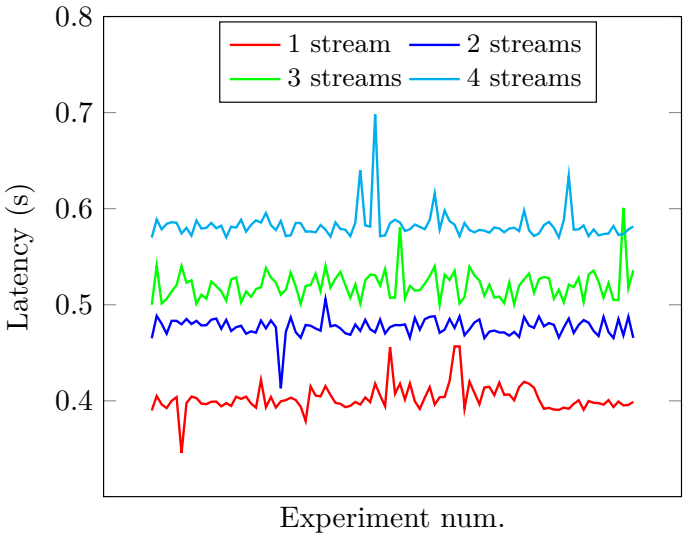


Figure 5.5: Latency for cloudlet server servicing multiple edge devices for 100 experiments.

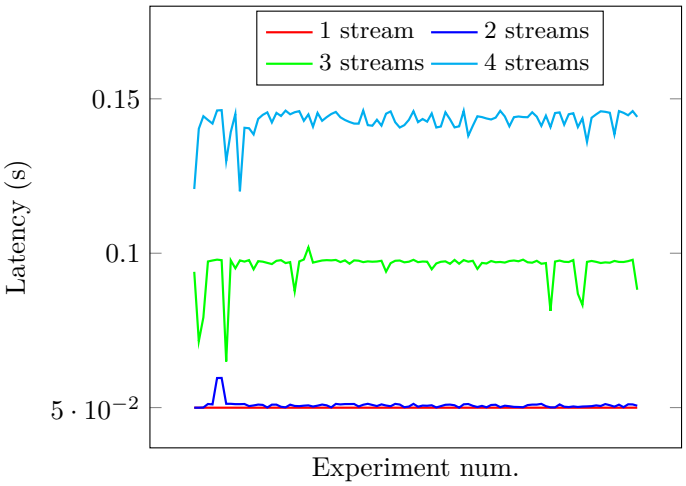


Figure 5.6: Latency for network-attached FPGA servicing multiple edge devices for 100 experiments.

captured, it is processed with the SqueezeNet model in a separate software process. Latency results are shown in Figure 5.5. As the number of data sources scales, overall latency increases, up to 600ms with 4 devices connected.

It can be inferred that increasing the number of devices would further increase latency. This could cause the cloudlet to be slower than accelerated edge platforms such as the Zynq, where computation is performed locally and resources are not shared. With a large number of edge devices serviced by a single cloudlet server,

standard edge nodes without acceleration might offer lower overall latency due to this network/compute contention. The server-hosted FPGA accelerator would suffer similar sharing costs since its computation is managed in software and the communication latency is similar to the cloudlet server scenario.

For the in-network switch-hosted FPGA, the sharing is more fine-grained, so it can be expected to better scale with the number of edge nodes. The results shown in Figure 5.6 demonstrate that an additional edge node has minimal effect on total latency due to the buffering and pipelining of the FPGA design resulting in reduced contention. Adding a third edge node almost doubles total latency, and increases jitter. A fourth edge node adds around 3× the latency compared to a single node, and adds even greater jitter. There are multiple factors that contribute to these increases. The accelerator shares the memory interface with the network pipeline, and to avoid the loss of data, the arbiter gives priority to network data. As the accelerator cannot retrieve image and weight data from memory while the memory interface is busy, latency is higher. Increasing the number of edge node streams means that the memory interface is busy more often. Further increases in latency come from the sharing of a single network interface. More streams means that the data for any given stream is more likely to be in a queue, which also explains the increased variation.

Despite these increases in latency with several connected devices, the network-attached FPGA still outperforms the equivalent number of edge nodes performing computation locally by a large margin. The lower computation latency of the FPGA relative to the server software means that it is more likely to scale well with a greater number of edge node data streams.

5.5.3 Discussion

The interplay between computation latency and communication latency has a significant impact on overall application latency when offloading computation. Network round-trip time is only one aspect of communication latency, and packet ingestion latency also has a noticeable impact. For complex applications, where computation latency dominates, these factors may not be as important. However, with the wider use of hardware acceleration, computation latency is reduced and communication latency, including ingestion latency becomes more important.

Integrating the accelerator into the network switch made communication latency negligible. The TPU hardware, while capable of the fastest inference, had its overall effectiveness reduced by preprocessing and communication time. Going forward, it can be seen that a complete view of computation and communication

latency must be considered when evaluating offload platforms.

While for a single device, offloading from the edge node to any other platform led to large reductions in end-to-end latency, this benefit diminished as the number of edge nodes increased. As systems scale, considerations must be made as to how offload from a large number of devices is handled. In these situations, local accelerators such as the Zynq, while seeming to under-perform in these experiments, will gain value as they are not shared across devices.

5.6 Accelerator Location

The previous section compared computation latency and ingestion latency for a deep neural network (DNN) image classification application using a variety of accelerated computing platforms. That study demonstrated that ingestion latency can have a significant impact on the overall latency reduction achievable for different platforms. Images generated at a Raspberry Pi edge node were transmitted to different offload platforms over a network. Performing the DNN computation on the edge node itself resulted in a computing latency of 2.3s due to the constrained capabilities of the embedded processor on that node. Offload targets investigated included server-based platforms where data enters the system through a PCIe network card and is moved to an accelerator PCIe card via a controlling application running in Linux userspace. This is representative of a typical host deployment in a cloudlet or datacenter. The same experiments were also explored with network-attached FPGA accelerators integrated into a network switch. In this deployment, packets received at the network interface of the switch were forwarded to the accelerator implemented on the same FPGA fabric depending on pre-specified packet headers. From these experiments, an average ingestion latency of 60ms was measured for the

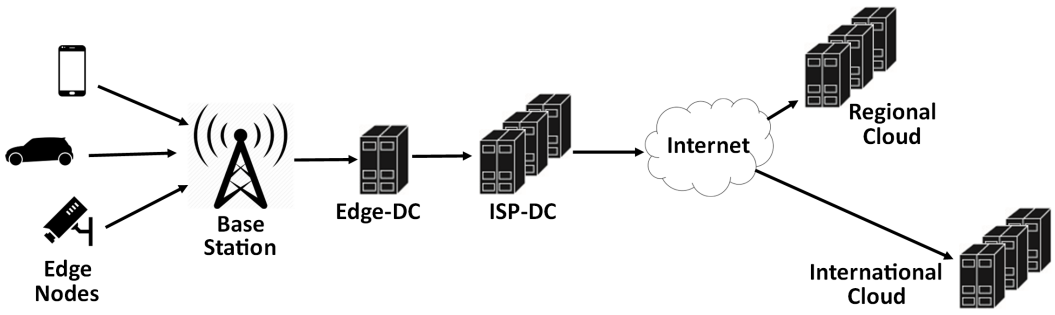


Figure 5.7: Network structure used for the discussion in this scenario.

entire image for a server-based platform, and 1ms for the network-attached FPGA

platform. While both platforms resulted in the same reduction in computing latency, down to 60ms, the low ingestion latency of the network-attached platform gave it superior overall latency.

In [194], the authors carried out a series of experiments measuring the network traversal time of packets from a mobile phone source to various locations that could be used to offload processing. This includes the eNodeB base station, a telco central office re-architected as a datacenter, the ISP datacenter, and various Amazon Web Services (AWS) virtual machines (VMs) in different geographic locations. These different offload locations have varying network distances to the data sources and hence result in a range of different network latencies, as reproduced in Table 5.2.

Location	**Latency (ms)**
BS – Base station	28
EdDC – Edge datacenter	41
ISPDC – ISP datacenter	62
RgCld – Regional cloud	77
IntCld – International cloud	151

Table 5.2: Network traversal time to various offload locations, measured in [194].

The overall network scenario is typical of an Internet of Things deployment, and can be seen in Figure 5.7: data is generated at the edge node and transmitted to the nearest access point, in this case, a base station a single hop away. From there, it travels through multiple network hops to a local edge datacenter, then to the ISP datacenter, and finally to a cloud datacenter, which can be located anywhere across the Internet. With each successive hop, additional network latency is introduced. Once the data reaches the target destination, it must be ingested by the computing platform and processed, with the result sent back to the edge node source. The time taken to complete this process is the total application latency.

Each of these networked locations could potentially host computing platforms to perform this computation. More capable accelerator hardware can also be deployed to be shared across multiple edge node clients and perform the computation with lower latency. Typically, with each successive hop, there is an increase in available computing resources and therefore opportunities to reduce computation latency further. Each location can host a range of computing platforms, which can reduce computation latency by varying amounts.

This case-study aims to examine the relationship and trade-offs present between the amount the offload platform – such as a server or hardware accelerator –

reduces the computation time, and the communication latency due to moving data to the that platform. Using the results previously discussed, total offload latency for similar streaming applications offloaded onto different platforms deployed at these different network locations is estimated, while considering varying acceleration factors achievable for different platforms. The degree to which the computation latency is reduced by the offload platform using an acceleration factor, can be represented such that:

$$\text{acceleration factor} = \frac{\text{latency}_{\text{comp}_{\text{base}}}}{\text{latency}_{\text{comp}_{\text{offload}}}} \tag{5.1}$$

where the base computation latency is the latency when performing processing at the IoT edge node. The acceleration factor is dependent on the platform being used to carry out the computation, such as a server class processor, or hardware accelerator like an FPGA, not on the location where the platform is hosted.

The total application latency hence depends on that computation time, the latency for data to be sent to the offload platform, and the ingestion latency at that platform, estimated as:

$$\begin{aligned}\text{latency}_{\text{total}} = \text{latency}_{\text{comp}} + \text{latency}_{\text{ingestion}} \\ + \text{latency}_{\text{network}}\end{aligned} \tag{5.2}$$

While this model does not capture all the details of a real implementation, such as network congestion, it is detailed enough to allow us to reason about the mix of computation offload platforms and where to locate them for improved application latency.

5.6.1 Results

The results for total latency estimation for each location and platform can be seen in Figure 5.8. Intuitively, as the acceleration factor increases, the computation time decreases, reducing total latency. However this provides diminishing returns, as the communication latency begins to dominate as computation latency reduces. This means that communication latency limits achievable performance when the computational complexity results in comparable computation latencies. Beyond a certain point, further increasing the processing capability of the offload platform results in minimal overall latency improvement improvement.

Utilising network-attached accelerator platforms, shown with the dashed lines, reduces total latency further for a given location, due to reduction of ingestion latency. The communication latency required to reach platforms further

from the edge means less computationally-capable platforms closer to the edge can sometimes provide better performance overall than more capable platforms further away. Similarly, utilising network-attached computing platforms further away can result in lower latency than a more local platform using server-based compute, since ingestion latency is significantly reduced. It can be seen that a strategy of increasing computing capability has limits in terms of achievable latency and that reduction of communication latency is ultimately required to improve overall latency further.

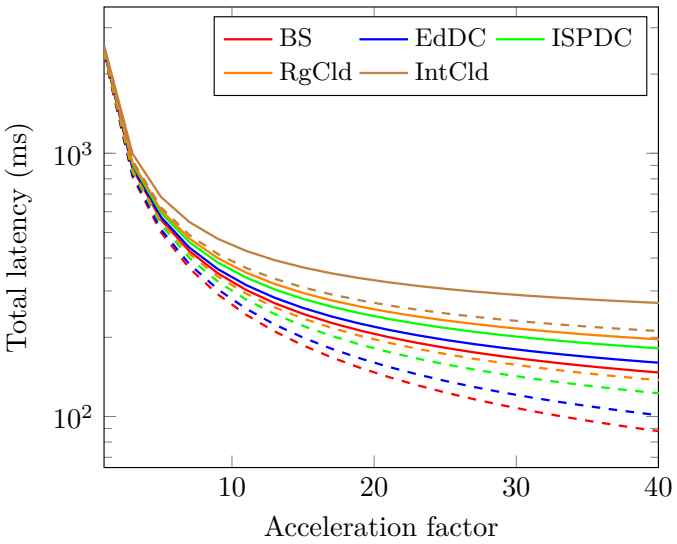


Figure 5.8: Estimated total latencies when computing is offloaded to the network locations in Table 5.2, for varying acceleration factors. Solid lines represent server-based acceleration platforms, and dashed lines represent network-attached acceleration platforms.

The closer processing is moved to the edge, the greater the relative improvement achieved through network-attached processing. The same can be seen as acceleration factor increases – the relative difference between the server and server-less deployments increases in turn. Figure 5.9 demonstrates this further, showing the relative reduction in total latency achieved when using network-attached over server-based acceleration at each location, and similarly shows that network-attached platforms yield greater benefits when applied to platforms closer to the edge, and for higher acceleration factors. This is because as the computation and network traversal times reduce, through improved processing capabilities and moving the compute closer to the source, the ingestion latency becomes a relatively more significant contributor to overall latency. For example, for an acceleration factor of 40×, for a base station offload (BS), using server-based computing, total latency was around 140ms. Ingestion latency contributed 60ms to this total, over 40%, greater than

each of the network traversal and computation latency. By comparison, using a network-attached accelerator resulted in a total latency of 85ms, and ingestion latency contributed only 1%.

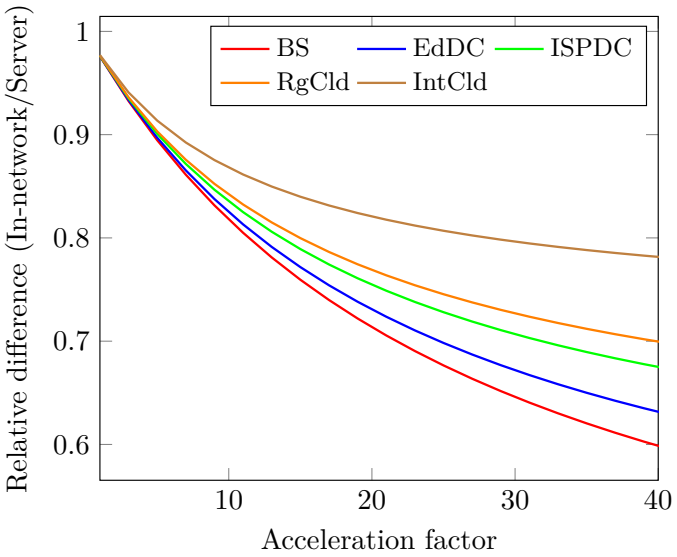


Figure 5.9: Relative reduction of total latency provided by network-attached acceleration compared to server-based, when computing is offloaded to the different networked locations in Table 5.2.

These results are based on the characteristics of the DNN application used in 5.4, which had a base computation latency of 2.3s on the Raspberry Pi edge node. In this situation, the computation latency by far outweighs the communication latency, so benefits can be achieved without requiring a significant acceleration factor on the offload platform. Also of interest is how this analysis changes when the balance of computation latency and communication latency changes. Figure 5.10 shows the results when the base computation latency is 200ms as opposed to 2.3s, and hence closer to the magnitude of the network latency. In this scenario, it can be seen that depending on the location and the ingestion latency of the platform, a greater acceleration factor is required to justify offloading. The base station (BS) must be able to perform processing at least 3× as fast as the edge node if using a server-based platform, but only around 1.4× faster when using network-attached acceleration. In this situation the furthest AWS instance as a server-based accelerator can never achieve an improvement over the edge node, even for extremely large acceleration factors, though network-attached acceleration would still be feasible.

When the base computation time is reduced further, to 100ms, an even greater acceleration factor is needed to justify offload, as shown in Figure 5.11.

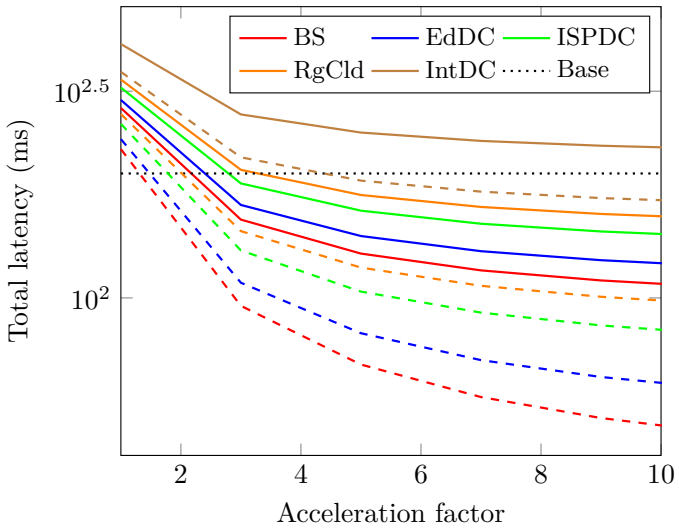


Figure 5.10: Estimated total latencies when a 200ms base computation time is offloaded to the different networked locations in Table 5.2, for varying acceleration factors. Solid lines represent server-based acceleration, and dashed lines network-attached. The black dotted line shows the base computing latency.

Even at the base station, the resource closest to the edge node, achieving increased latency performance would be a challenge unless using network-attached processing. Any of the other platforms wouldn't achieve improvements without utilising this, regardless of acceleration factor.

This study shows that deploying network-attached acceleration offers new opportunities to offload smaller IoT tasks that may have traditionally not have benefited from offloading. Additionally, more lightweight accelerators can be deployed that require less computing power, while still achieving reductions in total latency. These network-attached accelerator platforms also scale better to servicing multiple IoT edge nodes.

5.6.2 Discussion

Computational offload is an attractive method of reducing processing time for latency sensitive applications. While computation time is reduced, there is an associated communication latency caused by the traversal to the offload location, and the time taken to move the data to the processing platform at that location. Computational resources can be increase through the use of more powerful hardware, or using specialised accelerators. This chapter presented a case study for an image processing application where an edge node offloaded processing to one of several locations, ranging from a base station a single hop away, to a cloud datacenter in

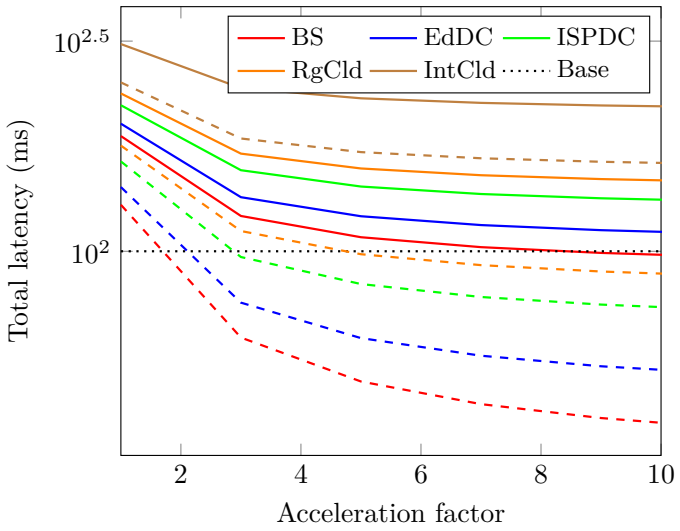


Figure 5.11: Estimated total latencies when a 100ms base computation time is offloaded to the different networked locations in Table 5.2, for varying acceleration factors. Solid lines represent server-based acceleration, and dashed lines network-attached. The black dotted line shows the base computing latency.

another continent. While efforts can be made to improve hardware and bring down processing time, this offers diminishing returns.

Reducing the communication time through moving the processing closer to the data source can reduce the total latency. However this only effects the time taken for data to traverse the network to the offload location. This case study showed that the ingestion latency, the time taken for data to traverse the network interfaces and software stacks of the compute platforms, is a considerable contributor to the overall latency. Utilising network-attached processing platforms that bypass these interfaces reduces ingestion latency, and thus allows greater potential to reduce the total latency through both moving compute to the edge, and increasing hardware capability. As computation improves, and network traversal is reduced, the next step to increasing performance is tackling how data is managed consumed by the compute.

This opens up opportunities to offload smaller tasks that may have traditionally not have benefited from computational offload. Likewise, it could enable the use of more lightweight processing that requires less hardware resources, performing more, with less – as well as open up hardware for sharing across multiple applications or client devices.

5.7 Summary

This chapter explored alternative approaches to accelerated computation near the edge, showing how the benefits of hardware accelerators can be exploited for lightweight Internet of Things nodes. A case study neural network image classification application was implemented on a variety of platforms typically used to move computation closer to the edge of a network. It showed that for an application of significant computational complexity, offloading processing from the edge node to more capable hardware through a network results in overall lower latency, despite the communication delay. Adding an accelerator to the cloudlet server improved performance further despite the additional PCI Express latency, due to significant acceleration achieved from hardware acceleration. Much improved performance was possible using an in-network accelerator approach where the hardware accelerator is not in a server but part of the network switching infrastructure instead. More edge devices sharing these near-edge computing resources leads to increases in total latency due to the sharing of hardware resources and increased network traffic, and hence the benefits of offloading are diminished. However the in-network FPGA accelerator approach is less impacted due to the low latency management of shared network flows and movement of data into the accelerator, meaning it can scale to more requests.

Offloading computation from resource-constrained nodes to more capable networked resources can lead to improvements in latency despite the communication costs, but in order to maximise these benefits, the way data is handled by these systems must be considered. FPGAs allow for large reductions in computation latency without a significant software overhead or additional interconnect.

Chapter 6

Conclusions and Future Work

This thesis examines in-network computation in the context of computational offload, in particular with FPGAs. It is demonstrated that network-attached FPGAs provide significant improvements to latency and throughput compared to server-based systems, which is vital to achieve the increasingly more stringent latency characteristics driven by the IoT and large-scale distributed applications.

6.1 Summary of Contributions

This work approaches several key challenges regarding computational offload. The mathematical model can be used to aid in the decision of where in a network different parts of an application should be offloaded to. It provides thorough analysis on the performance costs associated with utilising FPGA accelerators, and have demonstrated a method to greatly reduce communication latency through extending network elements to perform additional computation.

6.1.1 Mathematical representation of in-network and near edge computing

Chapter 3 outlines a mathematical formulation that could be used to represent a network of heterogeneous compute nodes and complex compositions of application tasks allocated to them. This model can be used to aid decision-making when designing the network, and can be used to evaluate task allocation strategies. The application and network structure can vary greatly, and will have an impact on the suitability of a particular deployment. Thus unlike other works, the model is designed to be generalisable to a greater range of domains, and the MILP structure used makes it simple to model additional parameters and constraints. Determining

where to offload computation is a key challenge as the number of available platforms grows.

6.1.2 Optimising hardware and task placement

As the complexity of networks and task structures increases, manual placement of tasks and allocation of hardware resources becomes more and more cumbersome to the point of being impractical. Additionally, non-intuitive deployments may exist that better satisfy a given performance requirement. The chapter also presents a method to utilise the mathematical formulation to determine task and hardware allocations within a set of networked heterogeneous resources. The optimisation generated equal or improved latency performance at lower costs than naive placement strategies.

6.1.3 Quantifying costs associated with FPGA accelerators

Hardware acceleration is key to achieving decreased computation time, which is one of the main goals of computational offload. FPGA accelerators have seen significant use in this area as highly parallel architectures can be utilised to reduce processing time. However FPGAs are typically deployed in a host-slave configuration, where an FPGA is managed and controlled by a host server. In Chapter 4 it is demonstrated that the communication latency costs associated with this approach. Results showed that the host is a significant contributor to latency, and using serverless, network-attached FPGAs can reduce this.

6.1.4 Demonstration of in-network FPGA acceleration

In Chapter 4, it was shown that network-attached FPGA deployments had significantly lower communication latencies than PCIe based hosted FPGAs. In Chapter 5, a method to take advantage of this is demonstrated. By extending an FPGA network switch to perform additional computation as well as its packet forwarding functions, the total latency of a DNN application was reduced compared to using an ASIC or server-based FPGA. Despite these alternative platforms having the same or lower computation time, integrating the FPGA accelerator directly in to the network resulted in the best performance.

6.2 Future Research

There are several promising areas of future research.

6.2.1 Practically validating the model

The model and optimisation outlined in Chapter 3 would benefit greatly from validating the accuracy of the results it can generate. Currently, it is limited to comparing the relative performance of different task and hardware placement strategies. Confidence could be given to the actual values it generates through practical experimentation, where cases could be physically implemented and performance measured, and compared against results from the model of an equivalent deployment. Performing these experiments at scale would require the resources to construct a substantial test-bed, and additionally would be aided by more readily available off-the-shelf components. As it stands, the lack of practical validation is a limitation of the model.

6.2.2 Improving optimisation runtime

The model and optimisation outlined in Chapter 3 is focused on evaluating and determining hardware placement strategies off-line, while designing the network. With further research, this optimisation could be extended to run post-deployment, to dynamically allocate tasks to nodes in the network. This can be useful to take into account events such as high network traffic or utilisation, nodes becoming unavailable or new nodes added, or the application being modified. The main challenge associated with this would be coming up with ways to reduce the runtime of the optimisation. As the network topology and task structure would mostly be set already in this scenario, certain variables can be fixed which would reduce the time taken to generate a solution. More research could also be carried out into heuristics that could be used to reduce runtime even further.

6.2.3 Developing generalised in-network FPGA infrastructure

Chapter 4 and Chapter 5 demonstrate that integration of FPGAs directly into network elements such as switches had the chance to greatly reduce total latency and increase throughput. This approach has several open challenges regarding management and deployment of accelerators within these devices. A direction of future research would be to generalise the FPGA network switch architecture shown in Chapter 5 to support loading accelerators into the switch at runtime utilising partial reconfiguration. Control information received over the network should be able to trigger this reconfiguration.

It would be extremely valuable to be able to also integrate high level datacenter and cloud management frameworks with such an infrastructure. This would

allow provisioning and management of in-network FPGA accelerator resources with the same tools as any other resource in the datacenter.

6.2.4 Combination of model and FPGA infrastructure

If such an infrastructure was developed, it could be integrated with the placement model. Given changing requirements or conditions, and knowledge of the network and available FPGA accelerator slots, the model could be used to dynamically allocate tasks to these augmented FPGA network elements on-the-fly.

6.2.5 Defining accelerator communication protocols

In order to manage flows of accelerator data, it would be beneficial to develop standard packet formats and protocols. These could allow for more intelligent management of data flows to and from accelerators. Load balancing, security, and QoS policies could be implemented using these protocols.

6.3 Summary

This thesis has contributed to the design and deployment of computational offloading to network elements traditionally only used for packet processing. FPGAs are well suited to this task, and this work demonstrates that using this approach results in greatly reduced latency and improved throughput. The work also addresses the related challenge of determining where in the network tasks should be offloaded to, by developing a generalised mathematical model that can be used to generate placements to meet combinations of objectives. Experiments detailed in this thesis have also been used to quantify the overhead associated with different approaches to accelerator deployment.

Bibliography

[1] R. A. Cooke and S. A. Fahmy, "In-network online data analytics," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, 2017.

[2] ——, "A model for distributed in-network and near-edge computing with heterogeneous hardware," *Future Generation Computer Systems*, vol. 105, pp. 395–409, 2020.

[3] ——, "Quantifying the latency benefits of near-edge and in-network FPGA acceleration," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, 2020, pp. 7–12.

[4] ——, "Characterizing latency overheads in the deployment of fpga accelerators," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2020.

[5] J. Yiu, *The Definitive Guide to ARM® CORTEX®-M3 and CORTEX®-M4 Processors*, 3rd ed. Newnes, 2014.

[6] M. Jackson and R. Budruk, *PCI Express Technology.* Mindshare press, 2017.

[7] G. Yeap, "Smart mobile SoCs driving the semiconductor industry: Technology trend, challenges and opportunities," in *International Electron Devices Meeting*, 2013.

[8] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: at the Leading Edge of Mobile-Cloud Convergence," in *International Conference on Mobile Computing, Applications and Services*, 2014, pp. 1–9.

[9] Y. Jararweh, L. Tawalbeh, F. Ababneh, and F. Dosari, "Resource effiicient mobile computing using cloudlet infrastructure," in *Conference on Mobile Adhoc and Sensor Networks*, 2013.

[10] K. Gai, M. Qiu, H. Zhao, L. Tao, and Z. Zong, "Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing," *Network and Computer Applications*, vol. 59, pp. 46–54, 2016.

[11] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, and W. Heinzelman, "Cloud-Vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture," in *Symposium on Computers and Communications*, 2012, pp. 59–66.

[12] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2016, pp. 73–78.

[13] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *International Conference on Mobile Systems, Applications, and Services*, 2014, pp. 68–81.

[14] Y. Chen, E. Blasch, B. Liu, Y. Chen, E. Blasch, K. Pham, D. Shen, and G. Chen, "A holistic cloud-enabled robotics system for real-time video tracking application," *Lecture Notes in Electrical Engineering*, vol. 309, 2014.

[15] Z. Chen *et al.*, "An emperical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proc. SEC*, 2017.

[16] W. Zhang, B. Han, and P. Hui, "On the networking challenges of mobile augmented reality," in *Proc. VR/AR Network*, 2017.

[17] M. Satyanarayanan, P. B. Gibbons, L. Mummert, P. Pillai, P. Simoens, and R. Sukthankar, "Cloudlet-based just-in-time indexing of IoT video," in *GIoTS*, 2017.

[18] S. Shreejith and S. A. Fahmy, "Smart network interfaces for advanced automotive applications," *IEEE Micro*, vol. 38, no. 2, pp. 72–80, 2018.

[19] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *Symposium on Principles of database systems*, 2005, p. 250.

[20] M. Vestias and H. Neto, " Trends of CPU, GPU and FPGA for high-performance computing," in *FPL*, 2014.

[21] R. P. Foundation, "Raspberry Pi 4," 2020. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/?resellerType=home

[22] Y. Tokusashi, H. Matsutani, and N. Zilberman, "LaKe: the power of in-network computing," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2018.

[23] R. Bajaj and S. A. Fahmy, "Mapping for maximum performance on FPGA DSP blocks," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1–1, 2015.

[24] Xilinx, "7 series FPGAs configurable logic block user guide," 2016.

[25] ——, "7 series DSP48E1 slice," 2018.

[26] ——, "Vivado high-level synthesis," 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[27] F. Kermarrec, S. Bourdeauducq, J.-C. L. Lann, and H. Badier, "LiteX: an open-source SoC builder and library based on Migen Python DSL," 2020.

[28] Intel, "Intel fpga sdk for OpenCL software technology," 2020. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html

[29] Xilinx, "SDAccel development environment," 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html

[30] M. Papadimitriou, J. Fumero, A. Stratikopoulos, and C. Kotselidis, "Towards prototyping and acceleration of java programs onto Intel FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 310–310.

[31] Y. Uguen and E. Petit, "Pyga: A python to fpga compiler prototype," in *International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, 2018, p. 11–15.

[32] C. Kulkarni, G. Brebner, and G. Schelle, "Mapping a domain specific language to a platform FPGA," in *Design Automation Conference (DATE)*, 2004, pp. 924–927.

[33] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A rapid prototyping framework for P4," in *Symposium on SDN Research*, 2017, p. 122–135.

[34] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, "P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, p. 147–152.

[35] O. Reiche, M. A. Ozkan, R. Membarth, J. Teich, and F. Hannig, "Generating FPGA-based image processing accelerators with hipacc," in *International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 1026–1033.

[36] J. Li, Y. Chi, and J. Cong, "Heterohalide: From image processing DSL to efficient FPGA acceleration," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 51–57.

[37] C. Dennl, D. Ziener, and J. Teich, "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 45–52.

[38] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A common backend for hardware acceleration on FPGA," in *International Conference on Computer Design (ICCD)*, 2017, pp. 427–430.

[39] Xilinx, "Vivado design suite user guide: Partial reconfiguration," 2018.

[40] M. Papadonikolakis and C. S. Bouganis, "A novel FPGA-based SVM classifier," in *International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 2–5.

[41] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An FPGA Implementation of decision tree classification," in *Design, Automation and Testing in Europe conference (DATE)*, 2007.

[42] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: multi-core, GP-GPU, or FPGA?" in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012.

[43] Y. R. Qu, "Scalable and dynamically updatable lookup engine for decision-trees on FPGA," in *High Performance Extreme Computing Conference (HPEC)*, 2014.

[44] C. Zhang, G. S. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for convolutional nerual networks," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 161–170.

[45] A. Rahman, J. Lee, and K. Choi, "Efficient FPGA acceleration of convolutional neural networks using logical-3d compute array," in *Design, Automation and Testing in Europe conference (DATE)*, 2016, pp. 1393–1398.

[46] R. García, A. Gordon-Ross, and A. George, "Exploiting partially reconfigurable FPGAs for situation-based reconfiguration in wireless sensor networks," in *Symposium on Field Programmable Custom Computing Machines, (FCCM)*, 2009, pp. 243–246.

[47] H. M. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynamically reconfigurable K-nearest neighbour classifier on FPGA," in *Conference on Adaptive Hardware and Systems*, 2012, pp. 205–212.

[48] J. Delahaye, J. Palicot, C. Moy, and P. Leray, "Partial reconfiguration of FPGAs for dynamical reconfiguration of a software radio platform," in *Mobile and Wireless Communications Summit*, 2007, pp. 1–5.

[49] T. H. Pham, S. A. Fahmy, and I. V. McLoughlin, "An end-to-end multi-standard OFDM transceiver architecture using FPGA partial reconfiguration," *IEEE Access*, vol. 5, pp. 21 002–21 015, 2017.

[50] H. M. Hussain, K. Benkrid, and H. Seker, "Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application," in *International Conference of the IEEE Engineering in Medicine and Biology Society*, 2015, pp. 7667–7670.

[51] H. Hussain, K. Benkrid, C. Hong, and H. Seker, "An adaptive FPGA implementation of multi-core K-nearest neighbour ensemble classifier using dynamic partial reconfiguration ," in *International conference on Field Programmable Logic and Applications (FPL)*, 2012.

[52] N. Chalhoub, F. Muller, and M. Auguin, "FPGA-based generic neural network architecture," in *International symposium on Industrial Embedded Systems*, 2006.

[53] M. Feilen, M. Ihmig, C. Schwarzbauer, and W. Stechele, "Efficient DVB-T2 decoding accelerator design by time-multiplexing FPGA resources," in *Inter-*

national conference on Field Programmable Logic and Applications (FPL), 2012.

[54] N. W. Bergmann, S. K. Shukla, and J. Becker, "QUKU: A dual-layer reconfigurable architecture," *Transactions on Embedded Computing Systems*, vol. 12, 2013.

[55] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput Oriented FPGA Overlays Using DSP Blocks," in *Design, Automation and Testing in Europe conference (DATE)*, 2016.

[56] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, "Adapting the DySER Architecture with DSP Blocks as an Overlay for the Xilinx Zynq," *SIGARCH computer architecture news*, vol. 43, 2016.

[57] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu, "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 411–4117.

[58] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-based overlay processor for convolutional neural networks," *Transactions on Very Large Scale Integration Systems*, vol. 28, no. 1, pp. 35–47, 2020.

[59] S. Mcgettrick, K. Patel, and C. Bleakley, "High performance programmable FPGA overlay for digital signal processing," in *International conference on Reconfigurable computing: architectures, tools and applications*, 2011, pp. 375–384.

[60] J. D. C. Maia, G. A. U. Carvalho, C. P. Mangueria, S. R. Santana, L. A. F. Cabral, and G. B. Rocha, "GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations," *Journal of Chemical Theory and Computation*, vol. 8, pp. 3072–3081, 2012.

[61] B. Li, E. Zhou, B. Huang, J. Duan, Y. Wang, N. Xu, J. Zhang, and H. Yang, "Large scale recurrent neural network on gpu," in *International Joint Conference on Neural Networks (IJCNN)*, 2014, pp. 4062–4069.

[62] A. X. M. Chang and E. Culurciello, " Hardware accelerators for recurrent neural networks on FPGA," in *ISCAS*, 2017.

[63] H. M. Hussein, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized K-means clustering on FPGAs: comparative results with GPPs and GPUs," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011.

[64] P. Meng, M. Jacobsen, and R. Kastner, "FPGA-GPU-CPU heterogenous architecture for real-time cardiac physiological optical mapping," in *FPT*, 2012.

[65] S. Bauer, S. Kohler, K. Doll, and U. Brunsmann, "FPGA-GPU architecture for kernel SVM pedestrian detection," in *Computer Vision and Pattern Recognition*, 2010.

[66] E. Nurvitadhi, Jaewoong 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 *International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–4.

[67] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 77–84.

[68] G. Cloud, "Cloud TPU," 2020. [Online]. Available: https://cloud.google.com/tpu

[69] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[70] Intel, "Intel Movideus Myriad X Vision Processing Unit," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/movidius-vpu/movidius-myriad-x.html

[71] Xilinx, "Xilybus product brief v1.12," 2019.

[72] IBM, "Coherent accelerator processor interface (CAPI)," 2019.

[73] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A reusable integration framework for FPGA accelerators," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 216–219.

[74] J. Gong, J. Chen, H. Wu, F. Ye, S. Lu, J. Cong, and T. Wang, "EPEE: an efficient PCIe communication library with easy-host-integration property

for FPGA accelerators," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014.

[75] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2014.

[76] M. Jacobsen and R. Kastner, "RIFFA 2 . 0 : A reusable integration framework for FPGA accelerators," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2016.

[77] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy, "JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-host and FPGA-to-FPGA communication," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2016.

[78] J. Yan, N. Y. Xu, X. F. Cai, R. Gao, Y. Wang, R. Luo, and F. H. Hsu, "FPGA-based acceleration of neural network for ranking in web search engine with a streaming architecture," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 662–665.

[79] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient KNN algorithm implemented on FPGA based heterogeneous computing system using OpenCL," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 167–170.

[80] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database analytics acceleration using FPGAs," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[81] A. M. Caulfield, E. S. Chung, P. Kaur, J.-y. K. Daniel, L. Todd, and M. Kalin, "A cloud-scale acceleration architecture," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[82] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 51–66.

[83] Y. Tokusashi, F. Pedone, and R. Soulé, "The Case For In-Network Computing On Demand," in *EuroSys Conference*, 2019, pp. 1–16.

[84] A. Hayashi, Y. Tokusashi, and H. Matsutani, "A line rate outlier filtering FPGA NIC using 10GbE Interface," *SIGARCH Computer Architecture News*, vol. 43, 2015.

[85] Y. M. Choi and H. K. H. So, "Map-reduce processing of k-means algorithm with FPGA-accelerated computer cluster," in *International Conference on Application-Specific Systems, Architectures and Processors*, 2014, pp. 9–16.

[86] A. Alhamali, N. Salha, R. Morcel, M. Ezzeddine, O. Hamdan, H. Akkary, and H. Hajj, "FPGA-accelerated hadoop cluster for deep learning computations," in *International Conference on Data Mining Workshop (ICDMW)*, 2015, pp. 565–574.

[87] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, 2014.

[88] A. Kulkarni, V. Kizheppatt, and D. Stroobandt, "MiCAP: a custom reconfiguration controller for dynamic circuit specialization," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015, pp. 1–6.

[89] A. Kulkarni and D. Stroobandt, "MiCAP-Pro: a high speed custom reconfiguration controller for dynamic circuit specialization," *Design Automation for Embedded Systems*, vol. 20, no. 4, pp. 341–359, 2016.

[90] S. Shreejith, B. Banarjee, K. Vipin, and S. A. Fahmy, "Dynamic cognitive radios on the Xilinx Zynq hybrid FPGA," in *International Conference on Cognitive Radio Oriented Wireless Networks*, 2015, pp. 427–437.

[91] X. Zhai, A. A. S. Ali, A. Amira, and F. Bensaali, "MLP neural network based gas classification system on Zynq SoC," *IEEE Access*, vol. 4, pp. 8138–8146, 2016.

[92] M. Russell and S. Fischaber, "OpenCV based road sign recognition on Zynq," in *International Conference on Industrial Informatics (INDIN)*, 2013, pp. 596–601.

[93] J. Whiteaker, F. Schneider, and R. Teixeira, "Explaining packet delays under virtualization," *SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 39–44, 2011.

[94] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *International Conference on Computer Communications (INFOCOM)*, 2014, pp. 1285–1293.

[95] L. Chen, S. Patel, H. Shen, and Z. Zhou, "Profiling and understanding virtualization overhead in cloud," in *International Conference on Parallel Processing*, vol. 2015-Decem, 2015, pp. 31–40.

[96] N. Zilberman, M. Grosvenor, N. Manihatty-bojan, D. A. Popescu, G. Antichi, S. Galea, A. Moore, R. Watson, and M. Wojcik, "Where has my time gone?" in *International Conference on Passive and Active network measurement*, 2017.

[97] R. Neugebauer, G. Antichi, J. F. Zazo, S. López-buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *SIGCOMM Computer Communication Review*, 2018, pp. 327–341.

[98] Z. Xincheng, *LTE Optimization Engineering Handbook*. John Wiley Sons, Ltd, 2017.

[99] B. C. Lee, *Datacenter Design and Management: A Computer Architect's Perspective*. Morgan Claypool, 2016.

[100] D. Abts and B. Felderman, "A guided tour of data-center networking," *Communications of The ACM*, vol. 55, pp. 44–51, 2012.

[101] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow, "Central office re-architected as a data center," *IEEE Communications Magazine*, vol. 54, pp. 96–101, 2016.

[102] Cisco, "The Cisco edge analytics fabric system," 2016.

[103] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computing is a dumb idea who's time has come," in *Workshop on Hot Topics in Networks (HotNets)*, 2017.

[104] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, 2016.

[105] T. G. Robertazzi, *Introduction to Computer Networking*. Springer International Publishing, 2017.

[106] I. Elhanany and M. Hamdi, *High performance packet switching architectures*. Springer International Publishing, 2007.

[107] G. Watson, N. McKeown, and M. Casado, "NetFPGA: A tool for network research and education," in *2nd workshop on Architectural Research using FPGA Platforms (WARFP)*, vol. 3, 2006.

[108] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *International Conference on Microelectronic Systems Education*, 2007, pp. 160–161.

[109] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE micro*, vol. 34, no. 5, pp. 32–41, 2014.

[110] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *International Symposium on Architectures for Networking and Communications Systems*, 2011, pp. 12–23.

[111] V. Puš, L. Kekely, and J. Kořenek, "Design methodology of configurable high performance packet parser for FPGA," in *International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2014, pp. 189–194.

[112] A. Bitar, M. S. Abdelfattah, and V. Betz, "Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA," in *International Conference on Field Programmable Technology (FPT)*, 2015, pp. 24–31.

[113] W. Jiang and V. K. Prasanna, "A FPGA-based parallel architecture for scalable high-speed packet classification," in *International Conference on Application-specific Systems, Architectures and Processors*, 2009, pp. 24–31.

[114] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *Symposium on Parallelism in Algorithms and Architectures*, 2009, p. 188–196.

[115] T. Ganegedara and V. K. Prasanna, "StrideBV: Single chip 400G+ packet classification," in *International Conference on High Performance Switching and Routing*, 2012, pp. 1–6.

[116] C. R. Clark, C. D. Ulmer, and D. E. Schimmel, "An FPGA-based network intrusion detection system with on-chip network interfaces," *International journal of electronics*, vol. 93, pp. 403–420, 2006.

[117] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary, "An FPGA-based network intrusion detection architecture," *Transactions on Information Forensics and Security*, vol. 3, no. 1, pp. 118–132, 2008.

[118] G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A scalable high throughput firewall in FPGA," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008, pp. 43–52.

[119] S. M. Keni and S. Mande, "Design and implementation of hardware firewall using FPGA," in *International Conference for Convergence in Technology*, 2018, pp. 1–4.

[120] J. Cullen, A. Gerbeth, and M. Dorojevets, "FPGA-based satisfiability filters for deep packet inspection," in *Long Island Systems, Applications and Technology Conference (LISAT)*, 2018, pp. 1–4.

[121] M. Ceška, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semric, and T. Vojnar, "Deep packet inspection in FPGAs via approximate nondeterministic automata," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 109–117.

[122] Q. Chen, V. Mishra, and G. Zervas, "Reconfigurable computing for network function virtualization: A protocol independent switch," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2016, pp. 1–6.

[123] N. Tarafdar, T. Lin, N. Eskandari, D. Lion, A. Leon-Garcia, and P. Chow, "Heterogeneous virtualized network function framework for the data center," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[124] J. Meng, N. Gebara, H. Ng, P. Costa, and W. Luk, "Investigating the feasibility of FPGA-based network switches," in *International Conference on Application-specific Systems, Architectures and Processors*, vol. 2160-052X, 2019, pp. 218–226.

[125] Z. Dai and J. Zhu, "Saturating the transceiver bandwidth: Switch fabric design on FPGAs," in *International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, p. 67–76.

[126] A. Bitar, J. Cassidy, N. Enright Jerger, and V. Betz, "Efficient and programmable ethernet switching with a NoC-enhanced FPGA," in *Symposium*

on Architectures for Networking and Communications Systems, 2014, p. 89–100.

[127] M. S. Abdelfattah, A. Bitar, and V. Betz, "Take the highway: Design for embedded NoCs on FPGAs," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015, p. 98–107.

[128] P. Papaphilippou, J. Meng, and W. Luk, "High-performance FPGA network switch architecture," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020, p. 76–85.

[129] S. Wijeratne, A. Ekanayake, S. Jayaweera, D. Ravishan, and A. Pasqual, "Scalable high performance sdn switch architecture on FPGA for core networks," 2019.

[130] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4->NetFPGA workflow for line-rate packet processing," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, p. 1–9.

[131] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, N. Zilberman, F. Pedone, and R. Soule, "P4xos: Consensus as a network service," Research Report 2018-01. USI., Tech. Rep., 2018.

[132] Amazon, "Amazon EC2," 2020. [Online]. Available: https://aws.amazon.com/ec2/?nc2=h_ql_prod_fs_ec2

[133] ——, "Amazon F1," 2020. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[134] Microsoft, "What is Azure?" 2020. [Online]. Available: https://azure.microsoft.com/en-gb/overview/what-is-azure/

[135] Google, "Google Cloud," 2020. [Online]. Available: https://cloud.google.com/

[136] Baidu, "FPGA cloud server," 2020. [Online]. Available: https://cloud.baidu.com/product/fpga.html

[137] Huawei, "FPGA accelerated cloud server," 2020. [Online]. Available: https://www.huaweicloud.com/en-us/product/fcs.html

[138] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin,

M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger, "Serving DNNs in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[139] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *Symposium on Networked Systems Design and Implementation*, 2018, pp. 51–66.

[140] Y. Lin and L. Shao, "Supervessel: The open cloud service for openpower," *White paper, IBM corporation*, 2015.

[141] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *International Conference on Cloud Computing Technology and Science (CloudCom))*, 2015, pp. 430–435.

[142] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *Conference on Computing Frontiers*, 2014, pp. 1–10.

[143] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized execution runtime for FPGA accelerators in the cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017.

[144] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *International Conference on Ubiquitous Intelligence and Computing*, 2015, pp. 1078–1086.

[145] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached FPGAs for data center applications," in *International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 36–43.

[146] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda, "The Feniks FPGA operating system for cloud computing," in *Asia-Pacific Workshop on Systems*, 2017, pp. 1–7.

[147] W. Wang, M. Bolic, and J. Parri, "pvFPGA: accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2013, pp. 1–9.

[148] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with openstack," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014, pp. 109–116.

[149] N. Tarafdar, N. Eskandari, V. Sharma, C. Lo, and P. Chow, "Galapagos: A full stack approach to FPGA integration in the cloud," *IEEE Micro*, vol. 38, no. 6, pp. 18–24, 2018.

[150] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *International workshop on Wireless sensor networks and applications*, 2002, p. 88.

[151] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A tiny aggregation service for ad-hoc sensor networks," in *Symposium on Operating systems design and implementation*, vol. 36, 2002, pp. 131–146.

[152] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion," in *International conference on Mobile computing and networking*, 2000, pp. 56–67.

[153] M. Ding, X. Cheng, and G. Xue, "Aggregation tree construction in sensor networks," in *Vehicular Technology Conference*, vol. 4, 2003, pp. 2168–2172.

[154] H. Luo, H. Tao, H. Ma, and S. K. Das, "Data fusion with desired reliability in wireless sensor networks," *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 23, no. 3, pp. 501–513, 2012.

[155] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *International Journal on Very Large Data Bases (VLDB)*, vol. 12, no. 2, pp. 120–139, 2003.

[156] Y. Ahmad and U. Cetintemel, "Network-aware query processing for stream-based applications," in *International conference on Very Large Databases (VLDB)*, 2004, pp. 456–467.

[157] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The design of the Borealis stream processing engine." in *Cidr*, 2005, pp. 277–289.

[158] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *PInternational Conference on Data Engineering*, 2006, p. 49.

[159] L. Ying, Z. Liu, D. Towsley, and C. H. Xia, "Distributed operator placement and data caching in large-scale sensor networks," in *International Conference on Computer Communications (INFOCOM)*, 2008, pp. 1651–1659.

[160] S. Rizou, F. Dürr, and K. Rothermel, "Solving the multi-operator placement problem in large-scale operator networks," in *International Conference on Computer Communications and Networks*, 2010.

[161] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert, "Resource allocation for multiple concurrent in-network stream-processing applications," *Parallel Computing*, vol. 37, no. 8, pp. 331–348, 2011.

[162] ——, "Resource allocation strategies for constructive in-network stream processing," *International Journal of Foundations of Computer Science*, vol. 22, no. 03, pp. 621–638, 2011.

[163] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *International Conference on Distributed and Event-Based Systems*, 2016, pp. 69–80.

[164] N. Tapoglou, J. Mehnen, A. Vlachou, M. Doukas, N. Milas, and D. Mourtzis, "Cloud-based platform for optimal machining parameter selection based on function blocks and real-time monitoring," *Journal of Manufacturing Science and Engineering*, vol. 137, no. 1, 2015.

[165] B. Lohrmann and O. Kao, "Processing smart meter data streams in the cloud," in *PES Innovative Smart Grid Technologies Conference Europe*, 2011.

[166] W. Zhang, P. Duan, Q. Lu, and X. Liu, "A Realtime Framework for Video Object Detection with Storm," in *International Conference on Ubiquitous Intelligence and Computing*, no. January 2017, 2014, pp. 732–737.

[167] Y. Simmhan, B. Cao, and M. Giakkoupis, "Adaptive rate stream processing for smart grid applications on clouds," in *International Workshop on Scientific Cloud Computing*, 2011, pp. 33–37.

[168] Z. Li, C. Chen, and K. Wang, "Cloud computing for agent-based urban transportation systems," *IEEE Intelligent Systems*, vol. 26, no. 1, pp. 73–79, 2011.

[169] E. Jean, R. T. Collins, A. R. Hurson, S. Sedigh, and Y. Jiao, "Pushing sensor network computation to the edge," in *International Conference on Wireless Communications, Networking and Mobile Computing*, 2009.

[170] L. Hong, C. Cheng, and S. Yan, "Advanced sensor gateway based on FPGA for wireless multimedia sensor networks," in *International Conference on Electric Information and Control Engineering*, 2011, pp. 1141–1146.

[171] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for VM-based cloudlets in mobile computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[172] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Workshop on Mobile Cloud Computing (MCC)*, no. August 2012, 2012, p. 13.

[173] S. H. Park, O. Simeone, and S. S. Shitz, "Joint Optimization of Cloud and Edge Processing for Fog Radio Access Networks," in *Transactions on Wireless Communications*, vol. 15, no. 11, 2016, pp. 7621–7632.

[174] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of Cloud computing and Internet of Things: A survey," *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016.

[175] O. Tomanek, P. Mulinka, and L. Kencl, "Multidimensional cloud latency monitoring and evaluation," *Computer Networks*, vol. 107, no. 1, pp. 104–120, 2016.

[176] M. Hatto, T. Miyajima, and H. Amano, "Data reduction and parallelization for human detection system," in *Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2015, pp. 134–139.

[177] C. Blair, N. M. Robertson, and D. Hume, "Characterising a Heterogeneous System for Person Detection in Video using Histograms of Oriented Gradients: Power vs. Speed vs. Accuracy," *Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 2, pp. 236–247, 2013.

[178] M. Kachouane, S. Sahki, M. Lakrouf, and N. Ouadah, "HOG based fast human detection," in *International Conference on Microelectronics (ICM)*, no. 24, 2012.

[179] B. Benfold and I. Reid, "Stable multi-target tracking in real-time surveillance video," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2011, pp. 3457–3464.

[180] R. A. Cooke and S. A. Fahmy, "A model for distributed in-network and near-edge computing with heterogeneous hardware," *Future Generation Computer Systems*, vol. 105, pp. 395–409, 2020.

[181] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized k-means clustering on FPGAs: Comparative results with GPPs and GPUs," in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, no. 1, 2011, pp. 475–480.

[182] Y. R. Qu, H. H. Zhang, S. Zhou, and V. K. Prasanna, "Optimizing many-field packet classification on FPGA, multi-core general purpose processor, and GPU," in *Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2015.

[183] A. Fiessler, S. Hager, B. Scheuermann, and A. W. Moore, "HyPaFilter: a versatile hybrid FPGA packet filter," in *Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2016.

[184] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized Execution Runtime for FPGA Accelerators in the Cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017.

[185] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on FPGA virtualisation," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[186] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: predictable low latency for data center applications," in *Symposium on Cloud computing*, 2012.

[187] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, 2013.

[188] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 31, no. 1, pp. 94–110, 2020.

[189] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computing is a dumb idea whose time has come," in *Workshop on Hot Topics in Networks (HotNets)*, 2017.

[190] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached FPGAs for data center applications," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[191] R. A. Cooke and S. A. Fahmy, "Quantifying the latency benefits of near-edge and in-network FPGA acceleration," in *International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, 2020, pp. 7–12.

[192] S. Shreejith, R. A. Cooke, and S. A. Fahmy, "A smart network interface approach for distributed applications on Xilinx Zynq SoCs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 186–1864.

[193] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.

[194] A. Cartas *et al.*, "A reality check on inference at mobile networks edge," in *International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*, 2019, pp. 54–59.

[195] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, pp. 10–22, 2015.

[196] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, 2016.

[197] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys*, vol. 51, no. 4, pp. 72:1–72:39, Jul. 2018.

[198] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," 2016.

[199] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017.

[200] H. Kwon, T. Krishna, L. Lai, and V. Chandra, "HERALD: Optimizing heterogeneous DNN accelerators for edge devices," 2019.

[201] D. Velasco-Montero, J. Fernández-Berni, R. Carmona-Galan, and A. Rodríguez-Vázquez, "Performance analysis of real-time dnn inference on raspberry pi," in *Real-Time Image and Video Processing*, 2018.

[202] S. Lanka, "Squeezenet hls implementation," 2017. [Online]. Available: https://github.com/lankas/SqueezeNet

[203] Coral, "Edge TPU performance benchmarks," 2019. [Online]. Available: https://coral.ai/docs/edgetpu/benchmarks/

[204] S. Shreejith, R. A. Cooke, and S. A. Fahmy, "A smart network interface approach for distributed applications on Xilinx Zynq SoCs," in *Proc. FPL*, 2018, pp. 186–190.

[205] N. D. Lane and P. Warden, "The deep (learning) transformation of mobile and embedded computing," *Computer*, vol. 51, no. 5, pp. 12–16, 2018.