University of Stirling

Faculty of Natural Sciences

Division of Computing Science and Mathematics

Domain-Specific Optimisations for Image Processing Algorithms on Heterogeneous Architectures

Teymoor Rasheed Ali

Thesis submitted in fulfilment of the requirements for PhD in Computing Science

29/12/2023

UNIVERSITY *of* STIRLING

Abstract

As real-time embedded vision systems become more ubiquitous, the demand for better energy efficiency, runtime, and accuracy have become vital metrics in evaluating overall performance. These requirements have led to innovative computing architectures, leveraging heterogeneity that combine various accelerators into a single processing fabric. These new architectures lead to new challenges in understanding the most efficient way to partition and optimise algorithms on the most suitable accelerator.

In this thesis, domain-specific optimisation techniques are applied to enhance performance and resource efficiency for image processing algorithms on heterogeneous hardware. Domain-specific optimisations are preferred for being hardware agnostic and their ability to cater to a wider range of image processing pipelines within the domain. First, a literature analysis is conducted on image processing implementations on heterogeneous hardware, high-level synthesis tools, optimisation strategies, and frameworks. The first objective is to develop macro-micro benchmarks for image processing algorithms to determine the suitability of these algorithms on hardware accelerators. The profiling led to the development of a comprehensive benchmarking framework, Heterogeneous Architecture Benchmarking on Unified Resources (HArBoUR). The framework decomposes each algorithm into its fundamental properties that would affect overall performance. A collection of representative image processing algorithms from various operation domains (*e.g.*, Filters, Morphological, Geometric, Arithmetic, CNNs, Feature Extraction) and full pipelines (*e.g.*, edge detection, feature extraction, convolutional neural network) are used as examples to understand the compute efficiency of on three hardware platforms (CPU, GPU, FPGA).

The results show that parallelism and memory access patterns influence hardware performance. GPUs excel for algorithms with large data-size parallel operations and regular memory access patterns. FPGAs better suit lower parallel factor and data-sized operations. In addition, optimising for irregular memory access patterns and complex computations remains challenging on both FPGA and GPU architectures. However, FPGAs offer high performance relative to their resource and clock speed, but their specialised architecture requires careful implementation for optimal results. In the case of feature extraction algorithms, GPU acceleration is preferable for high matrix operation-intensive stages due to faster execution times. At the same time, FPGAs are more suitable for lower arithmetic stages due to comparable performance and energy consumption profiles. Edge detection and CNN pipelines demonstrate GPUs faster performance but at a significantly higher energy consumption than FPGAs. FPGAs exhibit lower latency than GPUs, considering initialisation and memory transfer times. CPUs perform comparably to both hardware in low-complexity and data-dependant algorithms. In CNN pipelines, FPGAs compute particular layers faster but generally have slower total inference times than GPUs. Nonetheless, FPGAs offer flexibility with bit-widths and operation-fused custom kernels.

Domain-specific optimisations are applied to algorithms such as SIFT feature extraction, filter operations, and CNN pipelines to understand the runtime, energy, and accuracy. Techniques such as downsampling, datatype conversion, and convolution kernel size reduction are investigated to enhance performance. These optimisations notably improve computation time across different processing architectures, with the SIFT algorithm implementation surpassing state-of-the-art FPGA implementations and achieving comparable runtime to GPUs at low power. However, these optimisations led to a 5-20% image accuracy loss across all algorithms.

Finally, the research outcomes described above are applied to two constructed heterogeneous architectures aimed at two domains, low-power (LP) and high-power (HP) systems. Partitioning strategies are explored for mapping CNN layers and operation stages of feature extraction algorithms onto heterogeneous architectures. The results demonstrate that layer-based partitioning methods outperform their fasted homogeneous accelerator counterparts regarding energy efficiency and execution time, suggesting a promising approach for efficient deployment on heterogeneous architectures.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me during my PhD except for the following:

- Chapter 4: The results and text are taken from my own 'A Benchmarking Framework for Embedded Imaging' paper.
- Chapter 5: The results and text are from my own 'Domain-Specific Optimisations for Real-time Image Processing on FPGAs' Journal paper.
- Chapter 6: The results and text are from my own 'Energy Aware CNN Deployment on Heterogeneous Architectures' paper.

Signature: Teymoor Rasheed Ali **29/12/2023**

Acknowledgements

I would like to thank my advisers, Dr. Deepayan Bhowmik and Dr. Robert Nicol, who have been instrumental in shaping my research path.

This work was supported by STMicroelectronics, Imaging Division and the University of Stirling, Faculty of Natural Sciences. I express my gratitude to all the colleagues and members within these institutions who have provided me with valuable feedback and discussions on my work. In addition, thanks for allowing me to contribute on many projects that are used by millions of people across the world.

I would also like to thank my fellow university lab colleagues, Amir M. and Alexander C., for the countless discussions and late night chess games. Furthermore, thanks to Calum T., and Howard C. within the ST office for the technical discussions. Thanks to my friends, I have met along the way during my research: Karen F., Dr. Hannuy C., Dr. Ray W. and many more.

Finally, I would like to thank my parents who have enabled and supported me to pursue a PhD.

Contents

Abstract **i**

Attestation **iii**

Acknowledgements **iv**

List of Symbols and Acronyms **xiii**

Statement of Originality **xiv**

1 Introduction **1**

1.1 Motivation 2

1.2 Research Objectives 6

1.3 Thesis Outline 7

1.4 Publications 8

2 Background **9**

2.1 Image Processing Pipeline 9

2.2 Imaging Sensor 11

2.2.1 Image Sensor Characterisation 14

2.3 Interface Technologies 16

2.3.1 Camera Link 16

2.3.2 Peripheral Component Interface Express (PCIe) 17

2.3.3 Ethernet 19

2.3.4 Universal Serial Bus (USB) 21

2.3.5 Mobile Industry Processor Interface (MIPI) 23

2.3.6 FPGA Mezzanine Card (FMC) 24
2.3.7 Summary 25
2.4 Hardware Architectures 26
2.4.1 Multi-Core Central Processing Unit (CPU) 26
2.4.2 Graphics Processing Unit (GPU) 28
2.4.3 Field-Programmable Gate Array (FPGA) 30
2.4.4 Application-Specific Integrated Circuits (ASICs) 32
2.4.5 Heterogeneous Architectures 36
2.4.6 Summary 38
2.4.7 Software Ecosystem 38
2.5 Conclusion 41

3 State-of-the-Art 42

3.1 Hardware Targeting Image Processing 42
3.1.1 Multi-Core CPU Architectures 43
3.1.2 CPU-GPU Architectures 43
3.1.3 CPU-FPGA Architectures 44
3.1.4 CPU-GPU-FPGA Architectures 46
3.2 ASIC Architecture 47
3.3 Image Processing Optimisations 48
3.4 High-Level Synthesis 55
3.5 Benchmarking 58
3.6 Evaluated Image Processing Algorithms 60
3.7 Conclusion 65

4 HArBoUR: Heterogeneous Architecture Benchmarking on Unified Resources 66

4.1 Introduction 66
4.2 Benchmarking Framework for Image Processing on Hardware . 68
4.2.1 Processing Pipeline & Operation Types: 69
4.2.2 Operator Group 70
4.2.3 Heterogeneous Benchmarking Development Flow 76
4.3 Benchmarking Methodology 78
4.3.1 Micro Benchmarking Algorithms 79
4.3.2 Macro Benchmarking Algorithms 82
4.3.3 Performance Metrics 84
4.3.4 Measurement Environments 86
4.3.5 Measurement Approach 87
4.4 Experiments, Results & Discussion 88
4.4.1 Individual ISP Algorithms 88
4.4.2 Combined ISP Pipelines 92
4.4.3 Energy Consumption & Throughput Results 95
4.4.4 Discussions 97
4.5 Conclusions 98

5 Domain-Specific Optimisations 102
5.1 Domain-Specific Optimisations 103
5.1.1 Optimisation I: Down Sampling 104
5.1.2 Optimisation II: Datatype 105
5.1.3 Optimisation III & IV: Convolution 106
5.2 Case Study Algorithms 107
5.2.1 SIFT 107
5.2.2 Digital Filters 114

5.2.3 Convolutional Neural Network 115
5.3 Experimental Results and Discussion 116
5.3.1 Performance Metrics 117
5.3.2 Results and Discussions 120
5.4 Conclusion & Future Direction 126

6 Image Processing Algorithms on Heterogeneous Platforms 130
6.1 Heterogeneous Architecture 132
6.1.1 CNN Development Flow: 132
6.2 Scale-Invariant Feature Transform 133
6.2.1 SIFT Algorithm Analysis 133
6.2.2 SIFT Profiling & Partitioning Strategy 135
6.3 Convolutional Neural Network 137
6.3.1 Convolution Neural Network Architecture: 138
6.3.2 CNN Profiling & Partitioning Strategy: 141
6.4 Experimental Setup 145
6.4.1 CPU-GPU-FPGA Data Communication 148
6.4.2 Execution time 151
6.4.3 Power Consumption 151
6.5 Experimental Results 152
6.5.1 Heterogeneous SIFT Results 152
6.6 Heterogeneous CNN Results 153
6.6.1 Inference 154
6.6.2 Total Execution Time 156
6.6.3 Energy Consumption 158
6.7 Conclusion 159

7 Discussion, Conclusions and Future Work 160

7.1 Discussion 160

7.2 Conclusions 161

7.3 Limitations & Future work 162

7.3.1 Heterogeneous Benchmark Framework 162

7.3.2 Domain-Specific Optimisations 163

7.3.3 Heterogeneous Implementations 163

7.3.4 Domain-Specific Language 164

List of Figures

1.1 Semiconductor Timeline 3

1.2 Wafer Fabrication 5

2.1 Image-Signal Pipeline 10

2.2 Image Sensor Architecture 11

2.3 Image Sensor Circuit Design 12

2.4 3D Stacked Image Sensor and Memory 14

2.5 Image Sensor Noises 15

2.6 Camera Link Architecture 16

2.7 PCIe Architecture 18

2.8 PCIe Link 18

2.9 GigE Vision Stream Packet 20

2.10 MIPI Architecture 23

2.11 FMC Architecture 24

2.12 Multi-Core CPU Architecture 26

2.13 GPU Architecture 29

2.14 FPGA Architecture 31

2.15 Vision Processing Unit Architecture 33

2.16 Heterogeneous Architecture 36

3.1 Generic Soft Processor Architecture 44

3.2 Hardware Pipelining Optimisation 49

4.1 Benchmarking Framework 69

4.2 Framework Pipeline 76
4.3 Exemplar Image Pipelines 79
4.4 Exemplar Image Pipelines 81
4.5 Individual Algorithm Runtime 88
4.6 Individual Power Consumption 89
4.7 Individual Algorithm Throughput 91
4.8 Individual Algorithm EPO 92
4.9 Combined Algorithm Runtime 93
4.10 Combined Algorithm Power Consumption 94
4.11 Combined Algorithms Throughput 95
4.12 Combined Algorithm EPO 96

5.1 SIFT Algorithmic Block Diagram. 107
5.2 SIFT Scale-Space Block Diagram & Scale Neighbourhood 108
5.3 Magnitude & Orientation Assignment and Keypoint Descriptor Generation 109
5.4 High-level block diagram of the SIFT algorithm on FPGA 110
5.5 Gaussian Convolution Module Block Diagram. 111
5.6 Extrema Detection Module Block Diagram. 112
5.7 CORDIC Module Block Diagram. [1] 112
5.8 Descriptor Normalisation Block Diagram 113
5.9 Example approximated 3×3 image filter kernels. 114
5.10 Typical layers implemented within CNN Architectures. 115
5.11 Filter Algorithms Applied onto Input Image 116
5.12 SIFT: FPS and Accuracy 121
5.13 Filter: Runtime comparison for optimisations 122
5.14 Filter: Energy consumption comparison 123

5.15 CNN: Comparison Runtime & Accuracy 125
5.16 CNN: Comparison Energy Consumption 126

6.1 CNN Development Flow 132
6.2 Operation Stage Run-time Profiling: SIFT. 136
6.3 SIFT Algorithm & Partitioning Strategy. 137
6.4 ResNet-18 Architecture & Partitioning Strategy. 138
6.5 MobileNetV2 Architecture & Partitioning Strategy. 140
6.6 ResNet18 Layer Profiling 142
6.7 MobileNetV2, Layer Profiling 144
6.8 Heterogeneous Platforms 146
6.9 Scheduler Architecture 147
6.10 Data-Transfer on Heterogeneous Architecture 149
6.11 Clamp Power Measurement 151
6.12 SIFT Power Consumption 154
6.13 Frames per Second (FPS) for Inference 155
6.14 ResNet18 Energy Consumption 156
6.15 MobileNetV2 Energy Consumption 157

List of Symbols and Acronyms

Acronyms

Acronym	Description
ASIC	Application Specific Integrated Circuit
APU	Application Processing Unit
CNN	Convolution Neural Network
CPU	Central Processing Unit
TPU	Tensor Processing Unit
NPU	Neural Processing Unit
DNN	Deep Neural Network
DSL	Domain-Specific Language
FFT	Fast Fourier Transform
FLOP	Floating Point Operation
FPS	Frames Per Second
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Descriptor Language
HLS	High-Level Synthesis
IP	Intellectual Property
MSE	Mean Square Error
NPU	Neural Processing Unit
PCIe	Peripheral Component Interconnect Express
ReLU	Rectified Linear Unit
ResNet	Residual Network
RMSE	Root Mean Square Error
RTL	Register-Transfer Level
SIFT	Scale-Invariant Feature Transform
SSD	Solid State Drive
SSIM	Structural Similarity Measure
VHDL	VHSIC Hardware Description Language
VPU	Vision Processing Unit

Statement of Originality

The research conducted within the scope of this thesis produced the following novel and unique contributions towards domain-specific optimisation techniques for image processing algorithms on heterogeneous architectures:

Chapter 3

- State of the art analysis of literature found within the heterogeneous computing and domain-specific optimisation research domain.

Chapter 4

- A framework that studies features of image processing algorithms to identify characteristics. These features help partition complex algorithms in determining optimal target accelerators within heterogeneous architectures.
- The approach adopts a systemic and multi-layer strategy that offers trade-offs between accuracy within the imaging sub-domains *e.g.*, *CNNs* and *feature extraction.* Specifically, *HArBoUR* enables support in constructing end to end vision systems while providing expected results and guidance.
- Domain knowledge-guided hardware evaluation of computational tasks allows imaging algorithms to be mapped onto hardware platforms more efficiently than a heuristic based approach.
- Benchmark of representative image processing algorithms and pipelines on various hardware platforms and measure their *energy consumption* and *execution time* performance. The results are evaluated to gain insight into why certain processing accelerators perform better or worse based on the characteristics of the imaging algorithm.

Chapter 5

- Proposition of four domain-specific optimisation strategies for image processing and analysing their impact on performance, power and accuracy;
- Validation of the proposed optimisations on widely used representative image processing algorithms and CNN architectures (MobilenetV2 & ResNet50)

through profiling various components in identifying the common features and properties that have the potential for optimisations.

Chapter 6

- Proposal of an efficient deployment of a CNN that is computationally faster and consumes less energy.
- Novel partitioning methods on a heterogeneous architecture by studying the features of CNNs to identify characteristics found in each layer which are used to determine a suitable accelerator.
- Two heterogeneous platforms which consist of two configurations are developed, one high-performance and the other, power-optimised embedded system.
- Benchmarking and evaluating runtime, energy, and inference of popular convolution neural networks on a wide range of processing architectures and heterogeneous systems.

1 Introduction

The emergence of heterogeneous processor technology has enabled real-time embedded vision systems to become ubiquitous in many applications, such as robotics [2], autonomous vehicles [3], and satellites [4]. Real-time image processing is inherently resource-intensive due to the complex algorithms that demand significant computational power and memory bandwidth. As such, optimising the performance of image processing systems requires a delicate balance between hardware capabilities, software efficiency, and algorithmic innovation to ensure timely and responsive processing. Traditionally, imaging tasks implemented on homogeneous architectures were limited in their adaptability in handling diverse sets of operations. On the contrary, the advent of heterogeneous architectures offers a flexible computing environment that combines multiple accelerators such as CPUs, GPUs, and FPGAs, offering a choice for executing tasks according to their computational requirements.

Integrating such accelerators together poses significant challenges within design and implementation. These challenges are evident in the complexities of scheduling tasks on different hardware units, managing synchronisation, memory coherence, and addressing interconnect requirements. Additionally, the absence of standardised models for heterogeneous systems impacts the programming environment, making it challenging for developers to create cohesive applications. Lastly, performance evaluation becomes a multifaceted task, requiring a comprehensive understanding of the interactions between processing units and their contributions to overall system performance.

However, the primary challenge lies in determining the most effective approach for algorithm partitioning on heterogeneous architectures. Given that each processing architecture executes specific algorithms more efficiently than the other [5, 6]. In addition, navigating an environment with various tool-sets and libraries further compounds the challenge, requiring developers to carefully select and integrate the appropriate tools that align with each processor's properties. Consequently, partitioned algorithms require further hardware and algorithmic optimisations to extract maximum performance. Typically, domain-specific optimisation techniques are often overlooked limiting the full realisation of performance potential and efficiency gains.

Within the scope of the thesis, the aim is to demonstrate that leveraging heterogeneous architectures for image processing algorithms will increase performance in terms of both runtime and energy consumption. Consequently, this work introduces domain-specific optimisation techniques to further improve application efficiency.

1.1 Motivation

The history of the microprocessor can be traced back to 1959 when Fair-child Semiconductors made a significant breakthrough by creating the first integrated circuit. This invention revolutionised the field of electronics by laying the foundation for integrating multiple transistors and other components into a single silicon chip. In the early 1970s, Intel Corporation introduced the first commercially available microprocessor. The Intel 4004 [7], released in 1971, was a 4-bit processor capable of performing basic arithmetic and logical operations, with a clock speed of 740 kHz, it represented a significant leap in computing power compared to previous electronic circuits. The 4004 was primarily designed for calculators and other small-scale applications but soon found use in a wide range of devices. Many manufacturers began to contribute and innovate within the microprocessor space. In 1974, Intel released the 8080 [8], an 8-bit microprocessor that became highly influential.

Continuing through the 1970s and 1980s, microprocessors advanced rapidly, with increasing processing power, efficiency and improved architecture capabilities. The introduction of 16-bit processors, such as the Intel 8086 and Motorola 68000, marked another significant milestone, enabling more com-

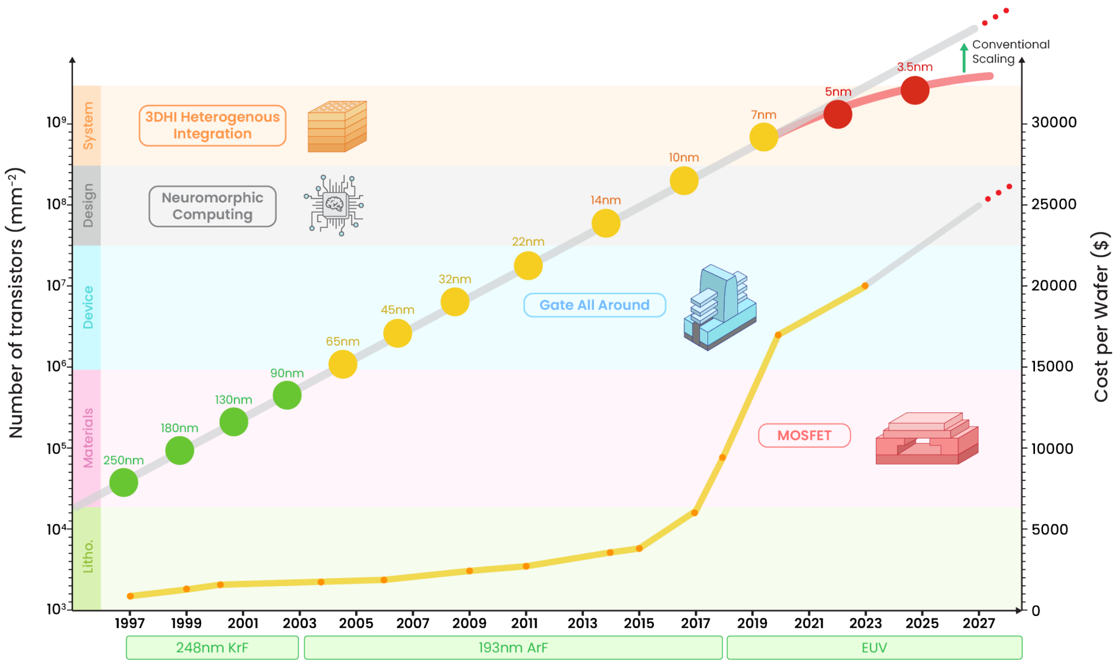


Figure 1.1: Cost, Transistor Count & Gate-length Technology Timeline [10].

plex applications and operating systems. In addition, ARM introduced a new architecture design which used a reduced instruction set paradigm to streamline the execution of instructions. This paved the way for the modern era of computing, with the rise of personal computers and the increasing integration of microprocessors into various devices and industries. In subsequent decades, microprocessors continued to evolve, with advancements in clock speeds, transistor densities, and architectural designs. The transition from 32-bit to 64-bit architectures expanded the memory addressing capabilities and enabled more demanding applications. Multi-core processors emerged in the early 2000s [9], revolutionising computing by enabling parallel processing and significantly improving performance and efficiency.

At around the same time, as CPU processors continued to evolve, two additional specialised architectures emerged to address specific computational needs: GPUs and FPGAs. GPUs were initially designed to handle the complex computations required for rendering high-quality graphics in video games and multimedia applications. However, their parallel processing capabilities and ability to handle large amounts of data made them well-suited for other computationally intensive tasks, such as scientific simulations. On the other hand, FPGAs offer a different approach to computing. Unlike CPUs and GPUs, which are based on fixed instruction sets, FPGAs provide programmable logic that allows users to configure the hardware functionality to suit specific tasks. This flexibility enables FPGAs to be highly optimised for specific applications, such as digital signal processing, data encoding, and real-time processing. FPGAs are particularly valuable in scenarios that require low latency and high throughput, as they can be tailored to perform specific operations with exceptional efficiency.

However, in the past decade, processor architecture designs had begun to coalesce, which resulted in a convergence of approaches and a common set of design principles among different CPU manufacturers. As a result, the X86 and ARM instruction sets are the only remaining architectures used in the majority of the systems available. This shift was driven by the realisation that the exponential performance gains seen in previous years were becoming increasingly difficult to achieve due to physical limitations and power constraints, reflected in Fig. 1.1.

The recent emergence of deep learning has reignited the pursuit of specialised computing units, which has fragmented the ecosystem. Developers have started exploring the potential of domain-specific accelerators such as TPUs or NPUs to meet specific computational needs. As a result, the processor landscape has become increasingly diverse again, with different manufacturers pursuing their unique architectural approaches. The growing set of domain-specific accelerators has driven designers to adopt newer and innovative approaches involving heterogeneity. A chiplet-based approach has emerged as a promising paradigm by disaggregating specialised processing units and integrating them into a cohesive interconnected circuit. Each chiplet serves a specific function, leveraging modularity and specialisation to enhance performance, scalability, and customisation. In addition, new packaging methods are utilised to integrate chiplets together, ranging from 2.5D-IC silicon interposers to 3D stacking. Nevertheless, with the deployment of diverse and heterogeneous architectures, a crucial challenge arises in the form of designing algorithms capable of effectively harnessing the capabilities offered by these novel architectural frameworks. This necessitates the development of algorithmic approaches that can optimise performance, exploit parallelism, and efficiently use the unique features and resources provided by these heterogeneous systems.

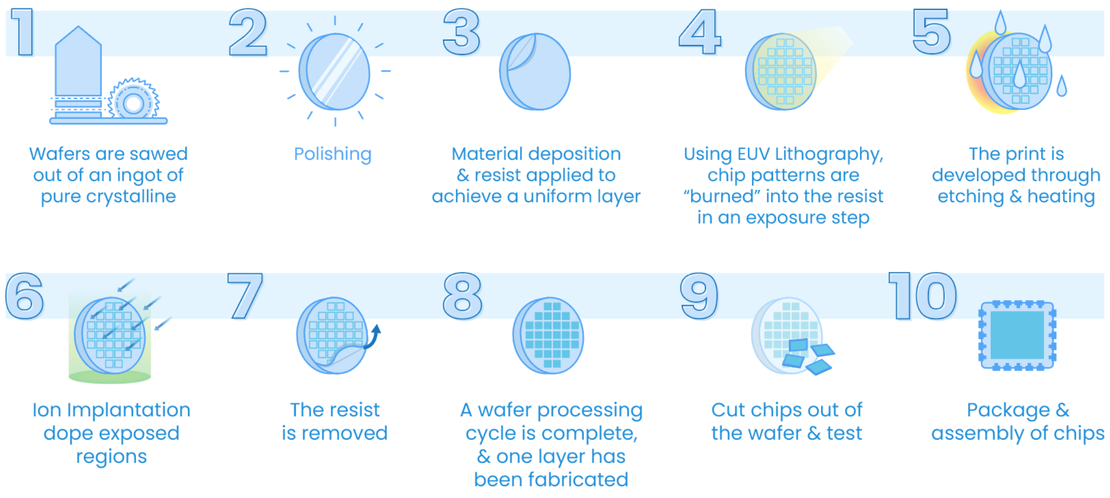

1
Wafers are sawed out of an ingot of pure crystalline
2
Polishing
3
Material deposition & resist applied to achieve a uniform layer
4
Using EUV Lithography, chip patterns are "burned" into the resist in an exposure step
5
The print is developed through etching & heating
6
Ion Implantation dope exposed regions
7
The resist is removed
8
A wafer processing cycle is complete, & one layer has been fabricated
9
Cut chips out of the wafer & test
10
Package & assembly of chips

Figure 1.2: Wafer Fabrication Process Steps to Develop Microprocessors.

Wafer fabrication, involves a series of steps to transform a silicon wafer into an integrated circuit shown in Fig. 1.2, including wafer preparation, photolithography, etching, layer deposition, and testing for functionality and quality. The pursuit of smaller transistor sizes, driven by demands for enhanced memory capacity and processing capabilities, has led to heavy investment in novel lithography technologies. However, the doubling of transistor densities every two years, as predicted by Moore's Law, has started to deviate due to technological limitations and economic costs. Shrinking transistors face challenges from the limitations of lithography wavelengths and the increasing complexity of manufacturing processes, leading to lower yields and higher costs. The production of larger silicon wafers has been debated, with the industry transitioning from small diameters in the 1960s to 300mm wafers as the standard by the early 1990s. While larger wafers offer cost and yield benefits, transitioning requires equipment redesign and cost-effectiveness considerations.

In summary, recent years have brought about major changes in the semiconductor industry, driven by the demand from resource intensive algorithms such as image processing and higher wafer fabrication cost. As a result, heterogeneous architectures serve as a potential to increase system performance further. However, understanding how to efficiently partition algorithms on each accelerator and identifying domain-specific optimisation trade-offs remain key challenges in maximising the potential of these architectures.

1.2 Research Objectives

This thesis aims to conduct research on partitioning and optimising image processing algorithms on heterogeneous architectures to unlock the full energy and runtime performance. This research encompasses a wide range of multidisciplinary domains (*e.g.*, hardware (CPU/GPU/FPGA), compilers, schedulers, optimisations and programming languages). Therefore, the focus is refined to three primary objectives in this thesis, which are listed in detail below:

1. Understanding the properties of image processing algorithms and hardware to determine the suitability in order to map operations to the most efficient hardware to increase performance. In addition, exploring optimised tool-sets and libraries in terms of programmability and performance. The goal of this objective is to develop a comprehensive micro/macro bench-marking framework which distils algorithms into their principle operations and gives heuristics towards mapping the operations to correct architecture. Additionally, providing various metrics to evaluate and compare each accelerator. This work enables the partitioning of algorithms on heterogeneous architectures, as realised in later chapters
2. Investigating domains-specific optimisation techniques which leads to better performance on hardware by exploiting inherent characteristics and structures in the image domain. These optimisations are applied in various combinations to determine the trade-offs in runtime, energy and accuracy metrics. The outcomes of this research enable understanding the efficiency of various hardware-agnostic optimisation methods found within the image processing domain.
3. Development of a comprehensive heterogeneous platform capable of executing image processing operations across all processing units while efficiently scheduling data for optimal performance. This includes designing and developing two complete heterogeneous platforms for high and low-power applications. Furthermore, using novel layer-wise/stage partitioning techniques on convolutional neural networks and feature extraction algorithms to execute on the most suitable accelerator within the heterogeneous platform. The goal of the objective is to uncover the advantages of heterogeneous architectures in image processing and document their performance gains over single-device solutions.

1.3 Thesis Outline

The rest of this thesis is organised as follows:

Chapter 2 presents a technical background on the devices, tools and software deployed in end to end imaging pipelines. This encompasses types of imaging sensors, interfaces, hardware architectures for image processing, high-level synthesis tools and Domain Specific Languages, followed by general discussions of their advantages and drawbacks within the image processing domain.

Chapter 3 critically discusses the state-of-the-art in current literature on optimisations and architectures, which includes HLS/DSL tools, micro/macro benchmarking frameworks and methodologies. Furthermore, an analysis of heterogeneous hardware and their performance in image and domain-specific optimisations.

Chapter 4 presents a novel framework methodology *HArBoUR*, for heterogeneous architectures which deconstructs image processing pipelines into their fundamental operations and evaluates their performance on hardware platforms, including CPUs, GPUs, and FPGAs. The methodology extends its evaluation to include various hardware based performance metrics, enabling a finer-grained analysis of each architecture's capabilities.

Chapter 5 presents the proposition of domain-specific optimisations for various imaging and deep-learning algorithms. Each optimisation strategy is applied individually and in combination, and their effectiveness is validated using runtime, accuracy and energy consumption metrics.

Chapter 6 proposes two algorithm types and their implementations on heterogeneous architectures, two convolution neural networks and one feature extraction algorithm. The accuracy, energy consumption and runtimes are recorded and compared to their discrete counterparts.

Chapter 7 concludes this thesis by summarising the research outcomes, i.e., analysis, proposed benchmarking framework and optimisation strategies on heterogeneous algorithms. Novel contributions are highlighted here along with suggestions on new ideas for future research in this domain.

1.4 Publications

Journals

Ali, T., Bhowmik, D. & Nicol, R. Domain-Specific Optimisations for Image Processing on FPGAs. Journal of Signal Process Systems (2023).
https://doi.org/10.1007/s11265-023-01888-2

Reports

M, Bane, O, Brown, **T, Ali**, D, Bhowmik, J, Quinn, D, Stansby. ENERGETIC (ENergy aware hEteRoGenEous compuTIng at sCale).
https://doi.org/10.23634/MMU.00631226

Under Preparation

Ali, T., Bhowmik, D. & Nicol, R. A Benchmarking Framework for Imaging Algorithms on Heterogeneous Architectures.

Ali, T., Bhowmik, D. & Nicol, R. Energy Aware CNN Deployment on Heterogeneous Architectures.

2 Background

In this chapter, the following sections review central components that make up the image processing pipeline. The components are divided into four categories: 1) Image Sensor Type and Characterisation 2) Interface Technologies 3) Hardware Processing Architectures 4) Software Tool-sets. The first category discusses the most common image sensor designs and various noises sources. The second category observes the data transfer performance of each interfaces between the sensor and processing hardware. The third category explores the components of hardware architectures used to execute algorithms. The final category delves into the tools and libraries employed for the ease of implementation.

2.1 Image Processing Pipeline

Vision applications fundamentally contain a sequence of operations that form a pipeline shown in Fig. 2.1. Firstly, the image sensor captures photons reflected off objects using micro-lens to refract the light into a matrix of wells containing circuits called pixels and the charge produced from the photodiode is converted to a voltage. Once the analogue signal from the image sensor is converted into a digital format, the image data goes through various pixel and frame operations to correct any defects found from the introduction of noise. Furthermore, a full-colour image is reconstructed from the raw frame using a demosaicing algorithm, which may differ depending on the filter pattern *e.g.*, Bayer, X-Trans or EXR. Optionally, the colour image can be compressed into a JPEG format to reduce file size for transmission. The image may contain help-

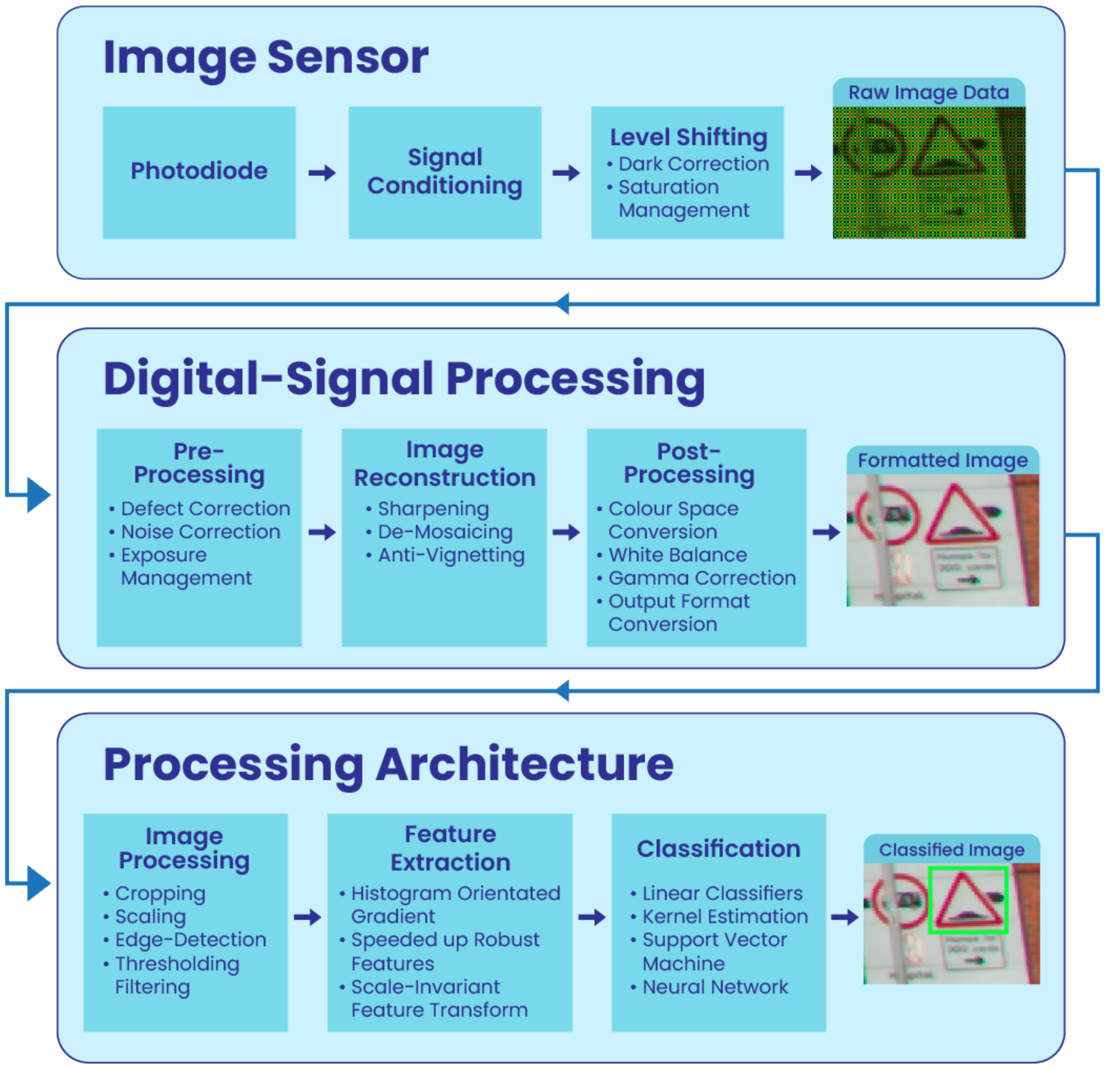
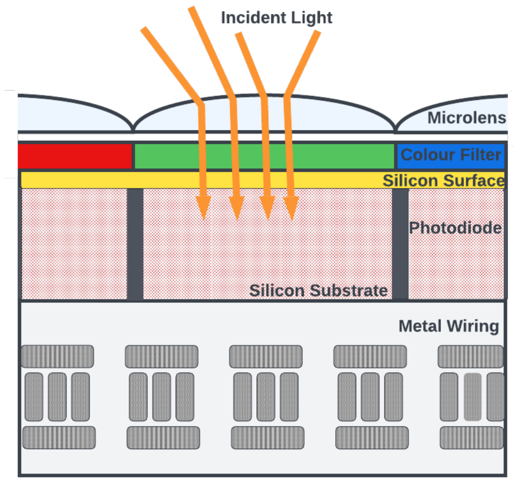


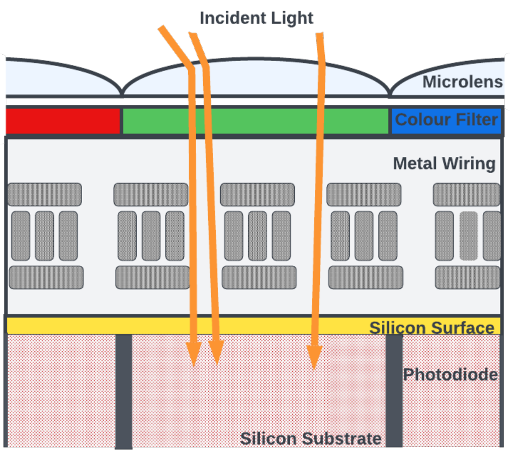
Figure 2.1: Illustrates the key stages in processing an image signal, which includes the image sensor responsible for capturing data, the Digital Signal Processor (DSP) for signal processing, and processing architecture used for further image understanding and labelling.

ful features that define particular objects, such as shape, colour or texture information. Feature extraction algorithms help identify these characteristics and compile the features into a vector. Finally, a feature vector or image is inputted into a classification algorithm such as a convolution neural network to determine a label or 'class'. The sequence of operations within the pipeline can be reordered or removed to fulfil particular design requirements.

The imaging pipeline comprises various hardware and software compo-



(a) Back-Illuminated



(b) Front-Illuminated

Figure 2.2: Cross-Section of CCD and CMOS silicon, revealing internal components fundamental to converting light into a signal.

nents that enable the efficient implementation and execution of image processing algorithms. This chapter presents a complete overview of each component and its limitations. These components include imaging sensors, processing architectures, interface protocols, vision libraries and other tool-sets used to develop a heterogeneous system.

2.2 Imaging Sensor

Image sensors are essential components in modern digital imaging devices, such as digital cameras, smartphones, and surveillance systems. These sensors play a crucial role in capturing and converting light into electrical signals, which are then processed to form digital images. Image sensors work on the principle of detecting and measuring light intensity to create a representation of the scene being captured. The most commonly used image sensing technologies within vision systems are charge-coupled device (CCD) [11], and CMOS image sensor(CIS) [12]. CCD technology was developed first and optimised over time for imaging applications, which allowed it to gain a significant market share compared to the newly developed CIS technology, which suffered in image quality due to higher noise. Therefore, CIS sensors were only used in applications where lower cost was the driving factor over image quality. However, over the years, significant advances in silicon size, power con-

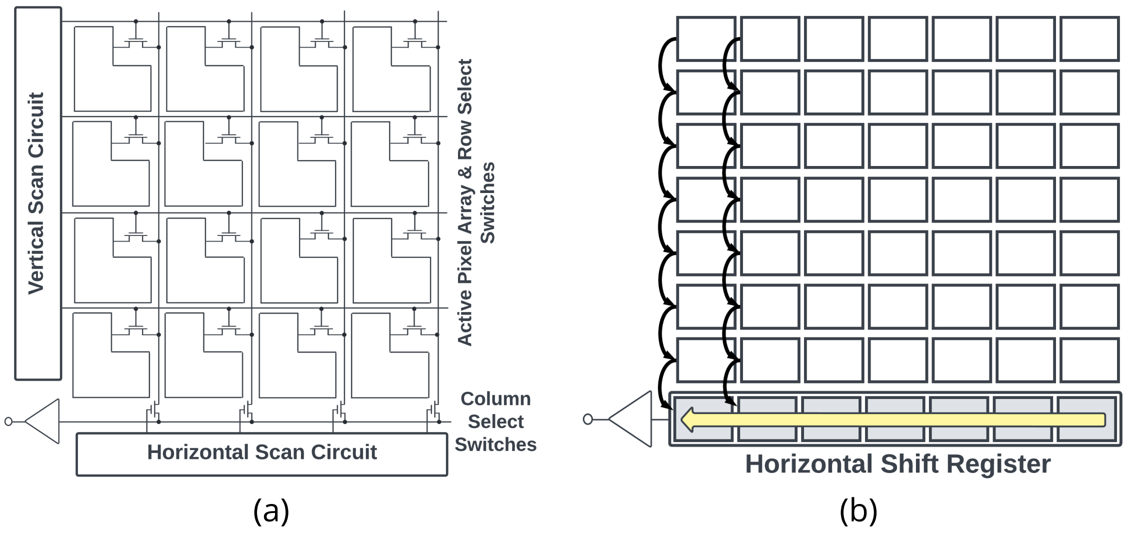


Figure 2.3: a) CMOS Circuit b) CCD Circuit,

sumption, process technology and the reduced fabrication cost of CIS technology resulted in surpassing CCD in market volume. CIS technology can now be found in many applications, from smartphones to medical imaging. Current research on CIS technology focuses on image quality by improving spatial, intensity, spectral and temporal characteristics [13].

Modern image sensors comprise several layers shown in Fig. 2.2 that integrate together to capture and process light. At the topmost layer, microlenses focus incoming light onto the pixel array below, enhancing light sensitivity and overall image quality. Beneath the microlenses lies the pixel array, with each pixel containing a photodiode responsible for converting photons into electric charge. A Bayer pattern colour filter [14], located on top of the pixel array, captures colour information by using red, green, and blue colour filters arranged in a specific pattern. Interpolating algorithms reconstruct the full-colour image from the captured colour data. Wiring and interconnects within the sensor facilitate the efficient transfer of electrical signals from each pixel to the readout circuitry, minimising signal degradation and cross-talk. The silicon substrate forms the foundation for all components, enabling efficient light conversion by the photodiodes and hosting the CMOS circuits for signal processing and readout.

The *CCD* architecture operates on the principle of transferring charge through a sequential shift register. This shift register is a critical component within the CCD chip, responsible for transporting the accumulated charge from each pixel to the output node for further processing. The photons of light strike the pixels of the CCD sensor, which absorbs the incident light, generating an electrical charge proportional to the intensity of the light. The charge in each pixel is horizontally transferred to neighbouring pixels along the shift register. This process, known as "horizontal transfer" in the row direction, uses potential wells to transport charge from one well to the next. After the horizontal transfer, the charge is vertically shifted down the columns. Manipulating voltages in the vertical shift registers moves the charge from one row to the next, guiding it towards the output node. The output node stores the accumulated charge and is connected to an analogue-to-digital converter (ADC) to convert the analogue charge into a digital signal for further processing and storage.

In a *CMOS* image sensor, the conversion of light into voltage involves several technical steps. Each pixel in the sensor consists of a photodiode, which acts as a light-sensitive capacitor. When incident photons strike the photodiode, it generates electron-hole pairs, and these charge carriers are stored as electric charge in the capacitor. The accumulated charge in each pixel's capacitor is then transferred to an associated charge-to-voltage conversion circuit, commonly known as the readout circuit. This circuit typically includes a source follower amplifier or a trans-impedance amplifier. The charge is converted into a corresponding analogue voltage signal, proportional to the intensity of the incident light on the pixel. The output voltage from each pixel is then sent to the image sensor's output circuitry for further signal conditioning and processing. This circuitry may include analogue signal processing components such as analogue filters or amplifiers to enhance the image quality and reduce noise.

In machine vision, the key performance metrics are latency and noise. The differences arise between CMOS and CCD imagers in their signal conversion processes, transitioning from signal charge to an analogue signal and finally to a digital one. In CMOS area and line scan imagers, a highly parallel front-end design enables low bandwidth for each amplifier. Consequently, when the signal encounters the data path bottleneck, typically at the interface between the imager and off-chip circuitry, CMOS data firmly resides in the digital domain. Conversely, high-speed CCDs possess numerous parallel fast output channels, albeit not as massively parallel as high-speed CMOS imagers. As a result, each CCD amplifier requires higher bandwidth, leading to increased

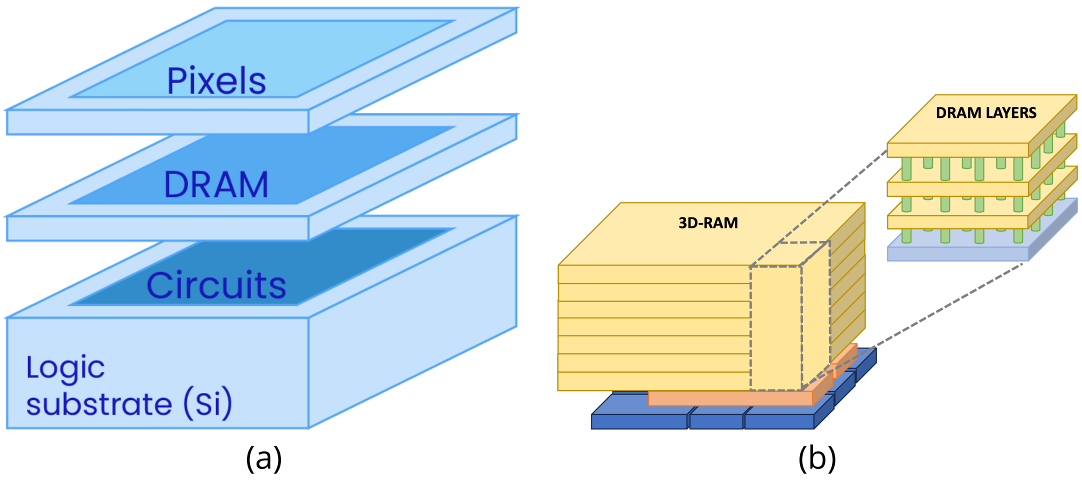


Figure 2.4: a) 3D Stacked Image Sensor b) 3D Stacked Memory; The Configuration Employs Multiple Processing or Memory Layers Vertically Stacked on Top of Each Other, Linked by Through-Silicon Vias (TSVs) and Microbumps.

noise levels. Therefore, high-speed CMOS imagers exhibit the potential for considerably lower noise compared to high-speed CCDs.

In recent years, semiconductor manufacturers have moved onto stacking imaging sensors depicted in Fig. 2.4 to reduce the latency between readout to processing, which was previously developed in the memory domain to increase data storage.

2.2.1 Image Sensor Characterisation

Image sensor characterisation is a process that assesses the performance and capabilities of imaging sensors. The goal is to understand the sensor's behaviour and limitations to ensure optimal image quality and accurate representation of the captured scene. Noise introduced in sensors come from various sources such as thermal noise, read noise, and photon shot noise, which can degrade image quality, as observed in 2.5. Characterisation involves measuring and analysing these noise components to determine their impact on image fidelity. Noise can be separated into two categories:

- **Pattern Noise:**

  This term describes noise patterns that remain constant or fixed over time and across multiple frames or exposures. Fixed pattern noise includes phenomena like Fixed Pattern Noise (FPN), Pixel Non-Uniformity

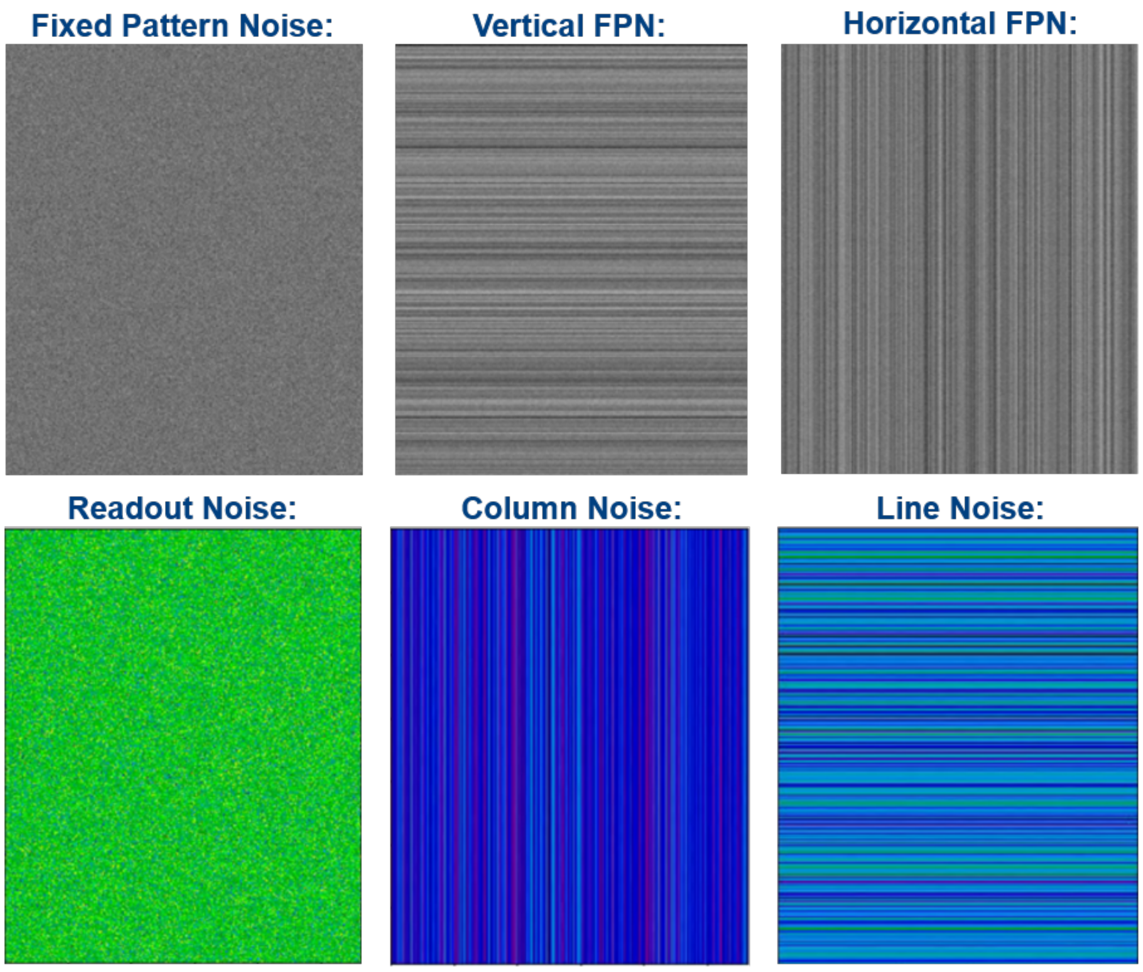


Figure 2.5: The figure illustrates the presence of characteristic fixed noise patterns, often resulting from sensor imperfections or calibration issues, alongside temporal noise, which can manifest as random variations in pixel values over multiple frames.

(PRNU), and other systematic and deterministic noise sources.

- **Random Noise:**

  The random noise relates to noise that varies over time or across different exposures. It includes sources of noise that exhibit randomness and unpredictability from frame to frame, such as Photon Shot Noise, Readout Noise, Amplifier Noise, and Jitter Noise.

Signal-to-noise ratio (SNR) is a standard metric used to quantify the signal quality captured by the sensor relative to the noise in the image. Dynamic range is another parameter that refers to the sensor's ability to capture and distinguish details in a scene's bright and dark regions. A wide dynamic range is essential for preserving details in high contrast scenes without overexposing or underexposing certain areas. Sensitivity and linearity are additional metrics assessed during the characterisation process. Sensitivity determines how well the sensor responds to incoming light, while linearity examines how the sensor's output corresponds to the actual incident light levels.

2.3 Interface Technologies

Vision systems typically rely on input from cameras or other video sources, generating a continuous stream of image frames. Designing algorithms for embedded vision systems requires a detailed understanding of performance and interfacing technologies. The subsequent sections provide an overview of various characteristics related to each technology.

2.3.1 Camera Link

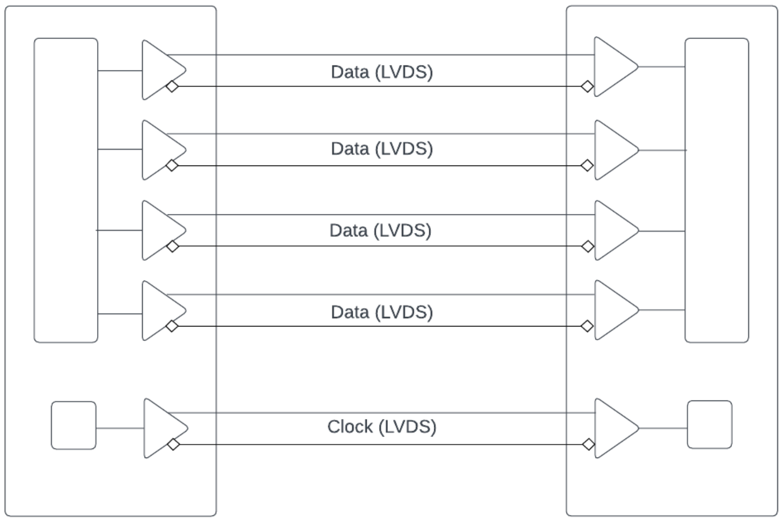


Figure 2.6: Camera Link interface, showing the integration of Low Voltage Differential Signalling (LVDS) technology for noise immunity.

Camera Link [15] is a parallel communication protocol that extends the Channel Link technology and standardises the interface between cameras and frame grabbers. Channel Link provides a one-way transmission of 28 data signals and an associated clock over five LVDS pairs. Among these pairs, one is designated for the clock, while the 28 data signals are multiplexed across the remaining four pairs exhibited in Fig. 2.6, involving a 7:1 serialisation of the input data. A single Camera Link connection allocates 24 bits for pixel data (three 8-bit pixels or two 12-bit pixels) and reserves 4 bits for frame, line, and pixel data valid signals. The pixel clock operates at a maximum rate of 85 MHz. Additionally, four LVDS pairs facilitate general-purpose camera control from the frame grabber to the camera, with the specifics defined by the camera manufacturer. Furthermore, two LVDS pairs are designated for asynchronous serial communication between the camera and frame grabber, supporting a minimum baud rate of 9600 for relatively low-speed serial communication.

For higher bandwidth requirements, the medium configuration includes an additional Channel Link connection, granting an extra 24 bits of pixel data. The full configuration further extends the capacity by incorporating a third Channel Link, resulting in a total of 64 bits of pixel data transmission capability. The versatile nature of Camera Link, with its various configurations, makes it a widely adopted interface standard for high-performance camera systems, particularly in applications demanding real-time image capture and processing.

2.3.2 Peripheral Component Interface Express (PCIe)

The Peripheral Component Interface Express (PCIe) [16] shown in Fig. 2.7, is an open standard serial bus interface protocol designed in the early 1990s to provide a high-speed interconnect between devices such as Ethernet controllers, expansion/capture cards, storage and graphics processing units. The protocol defined a set of registers within each device known as configuration space, allowing software to view memory and IO resources. In addition, the exposure of peripheral data enables software to assign an address to each device without conflict with other systems. Table 2.1 summarises each version of the PCIe specification ratified in the past and future.

The PCIe architecture consists of a root complex that connects the CPU and memory subsystem to the PCI Express switch fabric composed of one or more PCIe/PCI endpoints. The dual-simplex connections between endpoints are bidirectional, as shown in Fig. 2.8, which allows data to be transmitted and received simultaneously. The term for this path between the devices is a Link

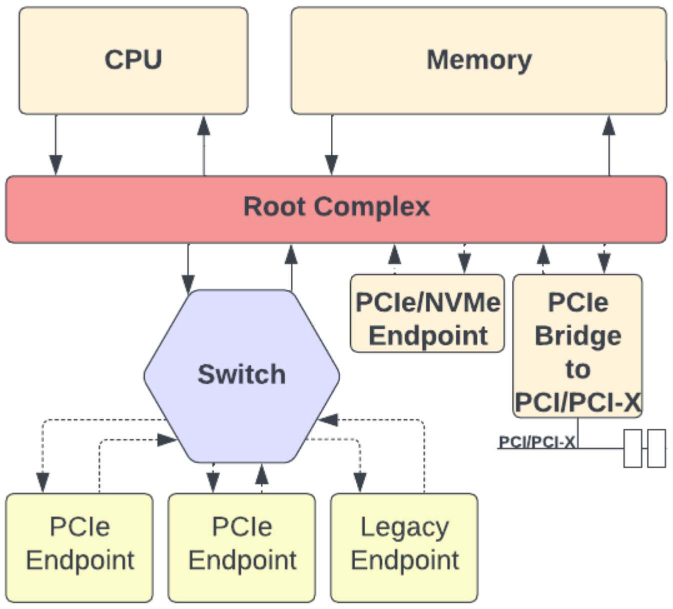


Figure 2.7: The PCIe Architecture consists of Root Complex, PCIe Endpoint Devices, and Memory Subsystem. The Root Complex orchestrates data flow, while PCIe Endpoint Devices serve as endpoints for data transactions.

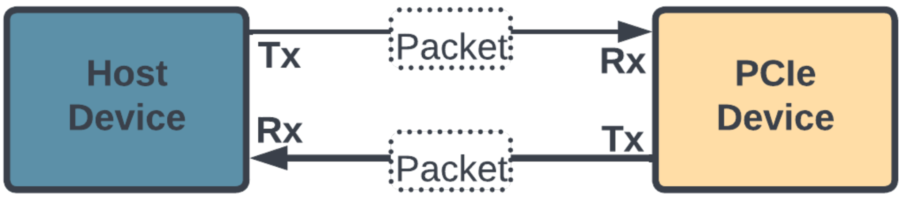


Figure 2.8: The PCIe Communication Data Link between two Devices

and is made up of one or more transmit and receive pairs. One such pair is called a Lane, and the spec allows a Link to be made up of 1, 2, 4, 8, 12, 16, or 32 Lanes. The number of lanes is called the Link Width and is represented as x1, x2, x4, x8, x16, and x32. The trade-off regarding the number of lanes to be used in a given design is that having more lanes increases the Link's bandwidth but at the cost of space requirement and power consumption.

Table 2.1: PCIe Specification Summary.

PCIe Specification	Release Year	Data Rate (GT/s)	Encoding	Total Bandwidth (GB/s)
PCIe 1.0	2003	2.5	8b/10b	8
PCIe 2.0	2007	5	8b/10b	16
PCIe 3.0	2010	8	128b/130b	32
PCIe 4.0	2017	16	128b/130b	64
PCIe 5.0	2019	32	128b/130b	128
PCIe 6.0	2021	64	PAM-4/FLIT	256
PCIe 7.0	2023	128	PAM-16/FLIT	512

2.3.3 Ethernet

Ethernet technology [17] is based on the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method. It operates at the physical layer (Layer 1) and the data link layer (Layer 2) of the OSI model. The physical layer handles the transmission and reception of raw data over the physical medium, while the data link layer is responsible for framing, addressing, and error detection.

One of the main advantages of Ethernet is its flexibility and scalability. It can support various data rates, ranging from 10 Mbps for older versions (e.g., 10BASE-T) to 1 Gbps (Gigabit Ethernet) and beyond for modern implementations. This adaptability allows Ethernet to cater to a wide range of applications, from simple office networks to high-speed data centres and multimedia streaming.

When using Ethernet with FPGAs, designers face the challenge of implementing the higher layers of the OSI model, namely the network layer (Layer 3) and transport layer (Layer 4). These layers are responsible for IP addressing, routing, and end-to-end communication. FPGA designs must include logic to handle IP addressing, packet forwarding, and any higher-level protocols required for data exchange. This complexity can add overhead to the FPGA design and require careful optimisation to ensure efficient data processing.

In FPGA-based systems, the communication between the FPGA and the Ethernet physical layer typically involves a Media Access Control (MAC) core, which is responsible for generating and interpreting Ethernet frames. The MAC core interfaces with the external Ethernet PHY chip, which handles the conversion between the MAC-level signals and the actual physical signals transmitted over the Ethernet cable. In CPU-based systems, handling Ethernet involves the integration of a Network Interface Controller (NIC) or Ethernet controller. The NIC is a hardware component that interfaces the CPU with the Ethernet medium. It manages low-level operations, such as frame reception and transmission, packet encapsulation and decapsulation, and error checking. The NIC communicates with the CPU through driver software that implements higher-level network protocols.

The CPU's involvement in Ethernet communication extends beyond the data link layer. It handles the network layer protocols (Layer 3), such as Internet Protocol (IP), which involves tasks like IP address assignment, routing, and packet forwarding. Additionally, the CPU manages transport layer protocols (Layer 4), such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), responsible for end-to-end communication and data flow control.

GigE Vision

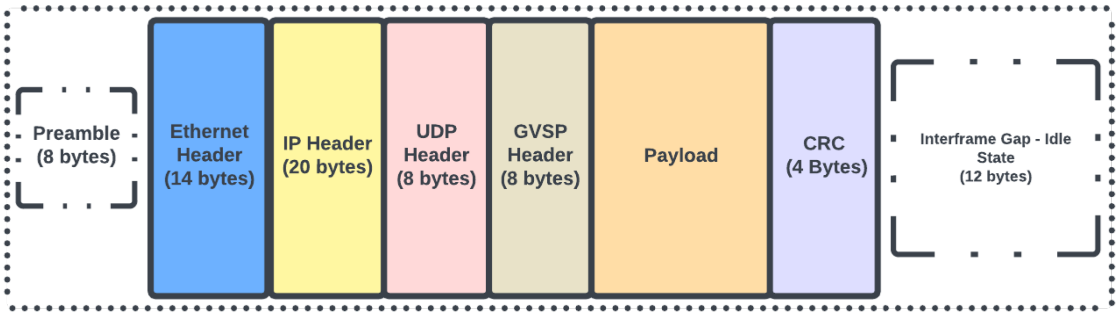


Figure 2.9: A Structure of the Data Frame, Including Payload Data, Header Information, and Ethernet Protocol Stack.

GigE Vision is a standard developed in 2006 by the Automated Imaging Association [18], which extends gigabit Ethernet to transport video data and control information to the camera efficiently. This standard benefits from the widespread availability of low-cost standard cables and connectors, allowing data transfer rates of up to 100 MPixels per second over distances of up to 100 meters.

GigE Vision comprises four essential elements. The control protocol facilitates camera control and configuration communication, while the stream protocol governs the transfer of image data from the camera to the host system, both running over UDP. A device discovery mechanism identifies connected GigE Vision devices and acquires their internet addresses. GigE Vision utilises Gigabit Ethernet (1000BASE-T) for data transmission, enabling a maximum data rate of 1 Gbps for real-time transfer of large image and video data in industrial applications. It employs packet-based communication displayed in Fig. 2.9, dividing images into smaller packets for efficient data transfer, ensuring reliable transmission and minimal data loss. Moreover, many GigE Vision cameras support Power over Ethernet (PoE), receiving power through the Ethernet cable, reducing installation overhead. GigE Vision supports both asynchronous and synchronous triggers for precise image capture control. Asynchronous triggers allow continuous image capture at a specified frame rate, while synchronous triggers enable coordinated capture based on external events for synchronised operation with other devices. Additionally, GigE Vision cameras can be easily configured and accessed using the GigE Vision Control Protocol (GVCP), providing efficient camera parameter adjustments and image data retrieval. Overall, GigE Vision is a versatile and efficient interface for industrial imaging applications, offering seamless data transmission and easy camera control.

In many embedded vision applications, it is more practical to integrate the FPGA system within the camera itself, allowing for image processing before transmitting results to a host system using GigE Vision. To achieve real-time operation, a dedicated device driver is used on the host system. This driver bypasses the standard TCP/IP protocol stack for the video stream and employs Direct Memory Access (DMA) transfers to transfer data to the application directly. By avoiding CPU overhead during video data handling, real-time performance can be achieved effectively. This optimised data transfer scheme ensures smooth and efficient communication between the camera and the host system in GigE Vision applications.

2.3.4 Universal Serial Bus (USB)

USB has emerged as a widely used interface for connecting peripheral devices to personal computers. Over the years, USB technology has evolved signifi-

Table 2.2: USB Versions, Bandwidth, Power Supply, and Specifications.

USB Version	Bandwidth	Power Supply	Additional Information
USB 1.0	1.5 Mb/s	5V (Max 500mA)	Initial USB standard
USB 1.1	12 Mb/s	5V (Max 500mA)	Enhanced data rate
USB 2.0	480 Mb/s	5V (Max 500mA)	High-speed data transfer
USB 3.0	5 Gb/s	5V (Max 900mA)	SuperSpeed USB
USB 3.1	10 Gb/s	5V (Max 900mA)	SuperSpeed+ USB
USB 3.2	20 Gb/s	5V (Max 900mA)	SuperSpeed USB 10, 20 Gb/s
USB 4.0	40 Gb/s	5V (Max 900mA)	Thunderbolt 3 compatible

cantly, supporting increasing link speeds from the initial 1.5 Mb/s and 12 Mb/s to the current 20 Gb/s in the double lane configuration of USB 3.2 Gen 2. As a result, USB has proven to be a viable and versatile interface between FPGAs and SoCs (System-on-Chips) in various applications.

USB operates in a master-slave architecture, where there can be only one host or master controller within a USB network, and the host controller initiates all communication. The communication protocol of USB is structured into four layers: the application and system software interacts with the USB pipe at the topmost layer, while the protocol layer handles packet management. USB packets come in several types, such as Link Management Packets, Transaction Packets, Data Packets, and Isochronous Timestamp Packets. These packets serve to exchange control and status information between the host and the connected devices.

The Data Packet, which carries user data along with a 16-byte header, is essential for transferring information between the host and the device. On the other hand, the other packet types primarily facilitate control and status exchanges. Any data transfer requires initiation by a Transaction Packet before actual data transmission occurs. To enable FPGA access to the USB bus, an external PHY (Physical Layer) chip is necessary. This chip often provides a first-in-first-out (FIFO) interface, which the FPGA can connect to through regular I/O ports. User logic is then required to implement the control logic and interface with the application. While this FIFO interface might limit throughput in certain scenarios, it does not pose any bottleneck for applications like video streaming.

2.3.5 Mobile Industry Processor Interface (MIPI)

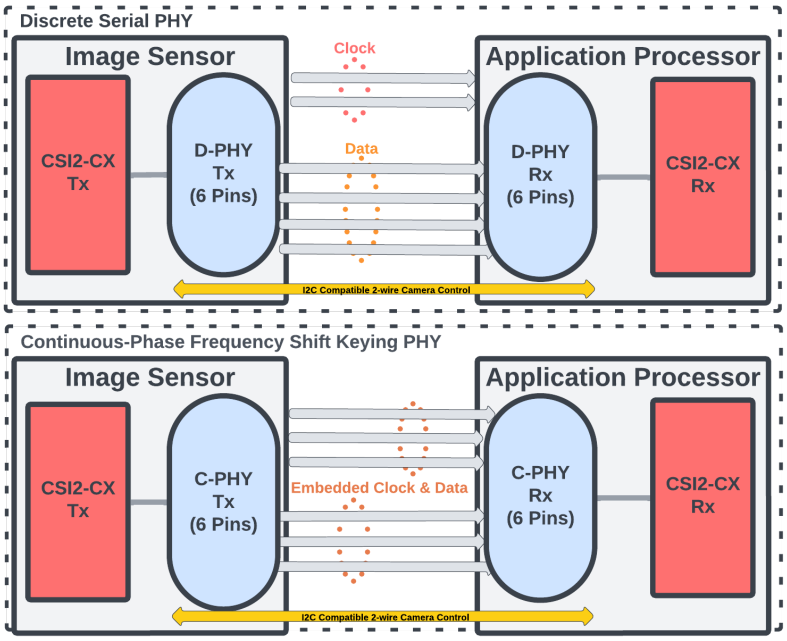


Figure 2.10: C-PHY and D-PHY with two lanes each. C-PHY makes it possible to reach more than double the bandwidth per lane than D-PHY.

MIPI [19] is a serial interface standard developed in 2003 for interconnecting components in mobile devices. MIPI comes in various versions, and one of the widely used versions in mobile camera interfaces is MIPI CSI-2 (Camera Serial Interface 2). The interface can consist of one or more data lanes, each capable of transmitting a stream of image data. A separate clock lane synchronises the data transmission, ensuring accurate data reception. MIPI CSI-2 supports various data types, including RAW image data and metadata, allowing it to accommodate different image sensor formats and data requirements. Furthermore, the concept of virtual channels enables the multiplexing of multiple data types over the same physical data lanes.

MIPI CSI-2 utilises low-voltage differential signalling to transmit image data from the camera sensor to the application processor. C-PHY and D-PHY are two different physical layer specifications, and their usage depends on the specific requirements of the devices shown in Fig. 2.10. D-PHY provides higher data rates and is capable of reaching extremely high speeds, making it suitable for applications that require substantial data throughput. D-PHY supports multiple data lanes (typically 1 to 4 lanes), and it is commonly used in devices with high-resolution imaging requirements, such as high-end smartphones and cameras. On the other hand, C-PHY, or Combo PHY, is a combination of MIPI C-PHY and MIPI D-PHY technologies. It offers a more power-efficient solution than D-PHY, making it ideal for power-sensitive mobile devices. C-PHY leverages both a low-power, single-ended signalling mode and a high-speed, differential signalling mode, providing a balance between data transfer rates and power consumption. It uses fewer wires compared to D-PHY, simplifying the physical design of mobile devices and potentially reducing costs.

MIPI CSI-2 comes with certain drawbacks that may impact its applicability. Firstly, its physical image data transfer (D-PHY) is limited to shorter cable lengths, typically not exceeding 20 cm, which can be restrictive for certain industrial applications. Additionally, the lack of a standardised plug for MIPI CSI-2 means that sensor/camera modules must be individually and proprietary connected. Moreover, the absence of a standardised driver and software stack requires custom adjustments for each sensor or camera module to work with the CSI-2 driver of a specific System-on-Chip (SoC) through a proprietary I²C driver as a Video4Linux sub-device.

2.3.6 FPGA Mezzanine Card (FMC)

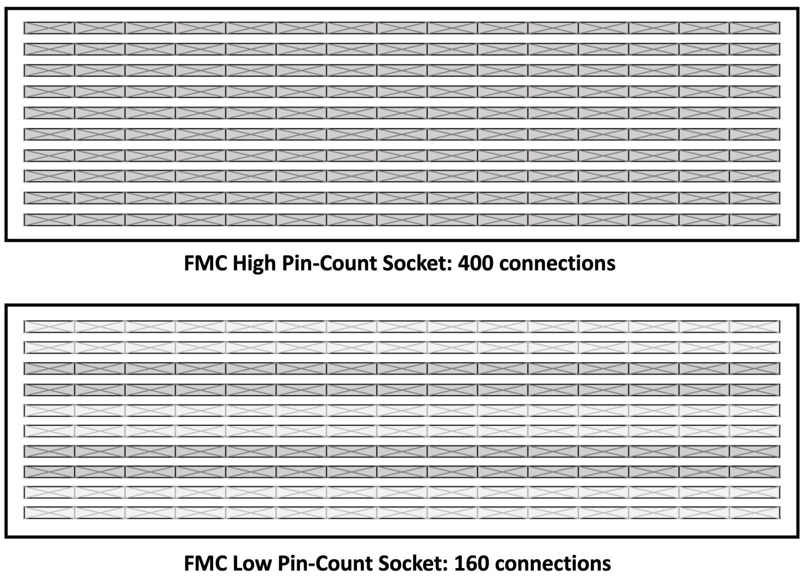

FMC High Pin-Count Socket: 400 connections
FMC Low Pin-Count Socket: 160 connections

Figure 2.11: FMC Low Pin and High Pin Count Sockets.

The FMC interface is a high-speed, versatile standard for connecting external modules to FPGAs (Field-Programmable Gate Arrays). The FMC standard encompasses two form factors: single-width and double-width. Single-width supports one connector, while double-width caters to applications needing more bandwidth, front panel space, or larger PCB areas and supports up to two connectors, offering designers flexibility for optimising space and I/O requirements. Two connector types, Low Pin Count (LPC) with 160 pins and High Pin Count (HPC) with 400 pins, displayed in Fig. 2.11. Both support single-ended and differential signalling up to 2 Gb/s, with signalling to an FPGA's serial connector at up to 10 Gb/s. LPC offers 68 user-defined single-ended signals or 34 differential pairs, along with clocks, a JTAG interface, and optional I2C support for base Intelligent Platform Management Interface (IPMI) commands. HPC provides 160 single-ended signals (or 80 differential pairs), ten serial transceiver pairs, and additional clocks. HPC and LPC connectors use the same mechanical connector, differing only in populated signals, enabling compatibility between them.

2.3.7 Summary

Table 2.3: Image Sensor Interfaces Specifications.

Interface	Interface Type	Bandwidth	Max Cable Length	Max Frequency	Bit Depth	Power (W)
PCIe	Serial	Up to 32 Gbps	0.2 Metres	100 MHz	Varies	75 W
GigE Vision	Serial	Up to 1 Gbps	100 Metres	125 MHz	8/10/12/14/16 bit	12 W
Camera Link	Parallel	Up to 800 Mbps	10 Metres	85 MHz	8/10/12/14 bit	6 W
MIPI CSI-2	Serial	Up to 10 Gbps	0.2 Metres	250 MHz	8/10/12/14/16/24 bit	1.2 W
FireWire	Serial	Up to 800 Mbps	4.5 Metres	400 MHz	8/10 bit	6 W
CoaXPress	Serial	Up to 12.5 Gbps	100 Metres	250 MHz	8/10/12 bit	24 W
USB4	Serial	Up to 40 Gbps	2 Metres	100 MHz	8-32 bit	7.5 W
10 GigE	Serial	Up to 10 Gbps	100 Metres	156.25 MHz	8/10/12/14 bit	15 W
Thunderbolt 4	Serial	Up to 40 Gbps	3 Metres	648.91 MHz	8/10 bit	9.9 W
FMC	Both	Up to 40 Gbps	1 Metre	156.25 MHz	8/10/12/14/16 bit	6.6 W

Table 2.3 provides a summary of common image sensor interfaces utilised in various imaging applications. These interfaces offer diverse specifications in terms of bandwidth, maximum cable length, frame rate, bit depth, and power consumption, catering to specific imaging needs and requirements. USB4, Thunderbolt and USB offers the highest bandwidth at 40 Gbps while CoaXPress and GigE variants supports the longest cable length at 100 meters which is ideal for distant camera setups. USB provides the highest bit depth

options, ensuring better colour precision. MIPI CSI-2 stands out as the most power-efficient interface, consuming only 1.2 W, which is ideal for mobile applications, while CoaXPress requires the most power at 24 W. Although, USB4 can supply up to 240W to cameras or other devices. The only two parallel interfaces are FMC and Cameralink.

2.4 Hardware Architectures

In recent years, the demand for flexible, energy-efficient and higher performance processors has continuously grown. This has pushed designers to develop novel processing architectures to facilitate requirements. This section introduces popular processing hardware used within vision applications.

2.4.1 Multi-Core Central Processing Unit (CPU)

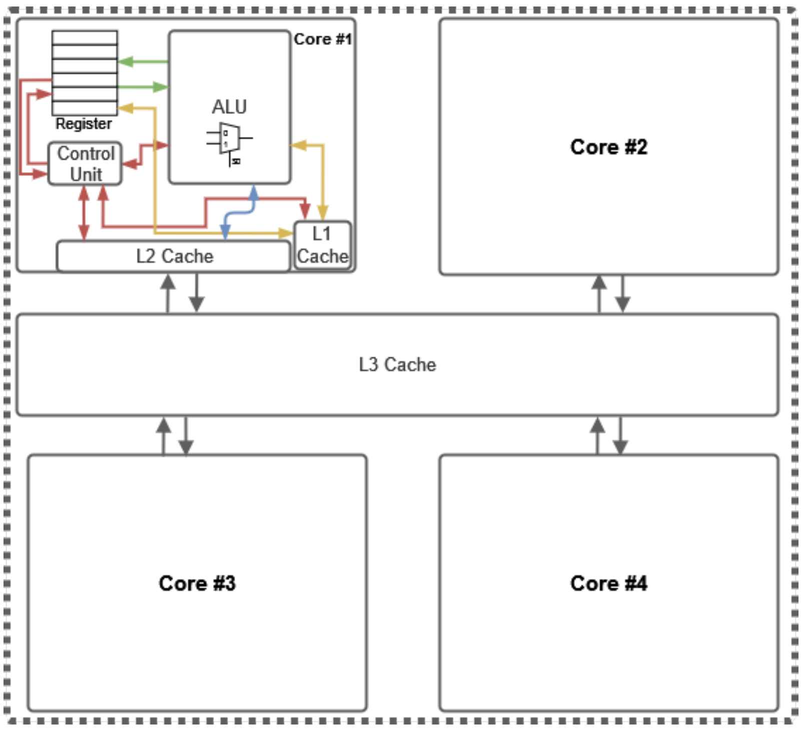


Figure 2.12: The CPU architecture consists of multiple cores, which contain components such as registers, caches, ALU and interconnects.

The CPU observed in Fig. 2.12 is an integrated circuit responsible for executing instructions and performing arithmetic, timing, logic and I/O operations. The CPU architecture involves the design and organisation of various components to optimise performance, power efficiency, and instruction execution. The main components are registers, arithmetic logic units (ALUs), control units, cache memory, and instruction pipelines. Registers are small, high-speed storage units within the CPU used for temporarily holding data and intermediate results during computation. ALUs are responsible for performing arithmetic and logic operations, such as addition, subtraction, NOT and OR. The control unit manages the flow of instructions and data within the CPU, fetching instructions from memory, decoding them, and coordinating their execution. CPU cache memory is used to store frequently accessed data, reducing the time taken to retrieve data from main memory.

Reduced Instruction (RISC) and Complex Instruction Set Computer (CISC) are two CPU microarchitecture approaches. RISC architectures prioritise simplicity and efficiency by employing a smaller set of basic instructions. This streamlined design typically leads to faster and more predictable execution, making RISC processors well-suited for power-constrained devices and applications where speed is critical. In contrast, CISC architectures, exemplified by x86, feature a diverse and extensive set of complex instructions designed to reduce the number of instructions required to perform tasks. While this complexity can provide convenience for programmers, it often results in more intricate hardware, potentially impacting performance and energy efficiency.

Significant research is put into improving the execution speed of instruction pipelines. The pipeline breaks down the execution of instructions into multiple stages, allowing different instructions to be processed simultaneously. Each stage of the pipeline handles a specific task, such as instruction fetch, decode, execute, and write back. This pipelining process increases the CPU's instruction throughput and overall performance. CPU architecture also includes features like branch prediction, speculative execution, and out-of-order execution. Branch prediction predicts the outcome of conditional branches to keep the pipeline filled with useful instructions. Speculative execution allows the CPU to execute instructions before it is confirmed that they are needed, further improving performance. Out-of-order execution enables the CPU to execute instructions in a different order to optimise resource utilisation.

In the past decade, single-core processors have now been outpaced by the shift to multi-core designs. Traditionally, speedup was achieved by increasing the processor's clock speed and decreasing the transistor size to pack more into the silicon area. However, the power density required grew at a faster rate than the frequency which entailed power problems exacerbated by complex designs attempting to extract extra performance from the instruction stream. This led to designs that were complex, unmanageable, and power-hungry. However, chip designers introduced multiple cores onto a single die and leveraged parallel programming to continue pushing for more performance. The primary advantage to multi-core systems is the raw performance increase from extending the number of processing cores rather than clock frequency, which translates into slower growth in power consumption. This can be a significant factor in embedded devices that operate on a power budget, such as mobile devices.

General-purpose multi-cores are becoming necessary in real-time digital signal processing. One general-purpose core would control various signals and watchdog functions for many special-purpose ASICS as part of a system-on-chip. This is primarily due to the variety of applications and functions required. Nevertheless, multi-Core processors give rise to new problems and challenges. As more processing cores are integrated into a single chip, power and temperature are the primary concerns that can increase exponentially with more cores. Memory and cache coherence is another challenge due to the distributed L1 caches and, in some cases, L2 caches which need to be coordinated.

2.4.2 Graphics Processing Unit (GPU)

The GPU is a specialised hardware architecture initially used for graphics rendering. However, GPUs have undergone significant power and cost advancements, which have captured the attention of both industry and academia. Designers have been exploring the potential of GPUs to accelerate large-scale computational workloads.

The architecture of GPUs is designed with a focus on throughput optimisation, allowing for efficient parallel computation of numerous operations.

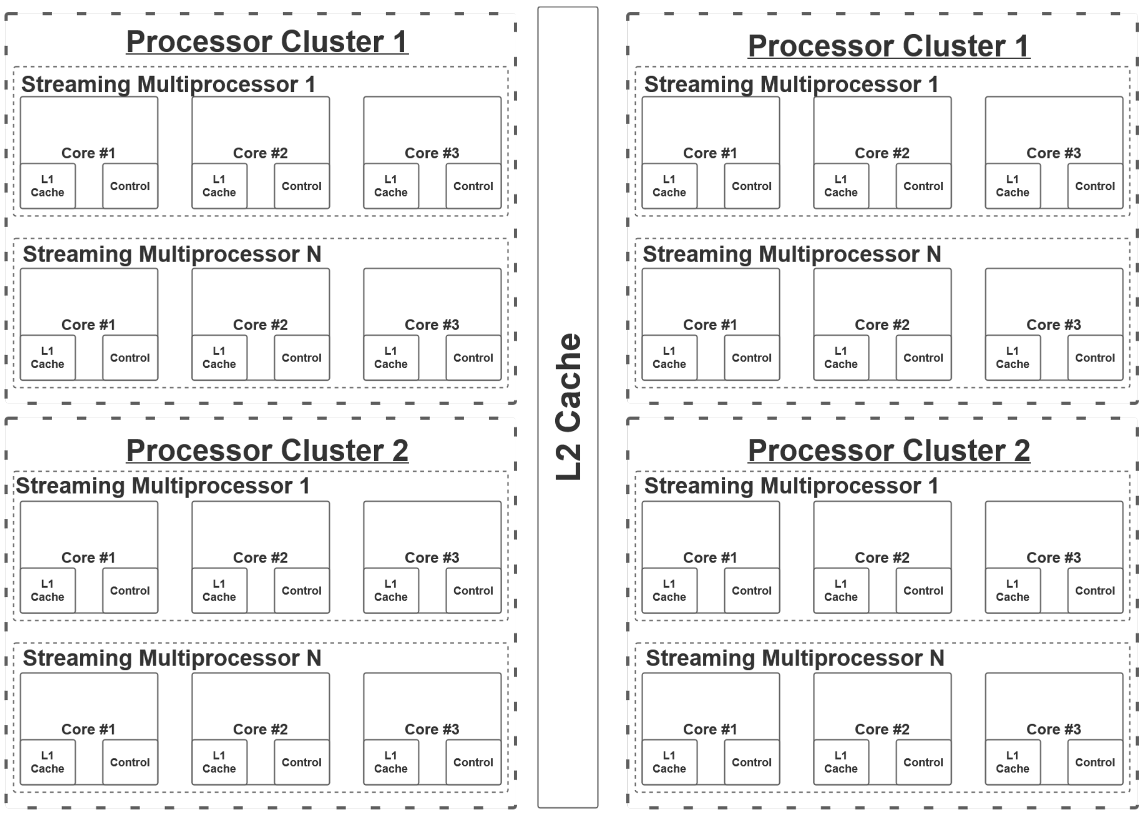


Figure 2.13: The GPU architecture comprises distinct elements, including Streaming Multiprocessors (SMs), Cache Hierarchy, and Compute Cores. SMs serve as the processing engines responsible for executing parallel threads, Cache Hierarchy optimises data access by efficiently managing on-chip memory caches, and Compute Cores perform complex computations and shader operations.

Fig. 2.13 illustrates the high-level GPU architecture. The GPU comprises multiple Streaming Multiprocessors (SMs) that function independently, and these SMs are organised into multiple Processor Clusters (PCs). Each SM incorporates a layer-1 (L1) cache with each core. Typically, each SM possesses its dedicated layer-1 cache, and multiple SMs share a layer-2 cache before accessing data from the global GDDR-5 memory. Newer GPU models integrate tensor cores, which efficiently compute matrices calculations, enhancing their performance in deep learning tasks.

The GPU architecture, initially tailored for 3D graphics rendering, involves a streamlined pipeline with distinct stages. It commences with vertex processing, transforming 3D geometric data, followed by primitive assembly to group vertices into primitives. Rasterisation then translates these into screen pixels or fragments, and fragment processing adds attributes like colours and textures. Finally, the pixel output stage writes processed fragments to the frame buffer, resulting in the rendered image on the screen. The highly parallel nature of the graphics pipeline in GPUs makes them exceptionally well-suited for image processing tasks. Image processing often involves manipulating and analysing large amounts of pixel data concurrently, making it a naturally parallelisable task. Leveraging the parallel processing capabilities, image processing algorithms can be accelerated by providing higher frames per second performance for tasks such as image filtering, edge detection, and object recognition. Additionally, GPU optimised memory hierarchy ensures faster access and storage of larger images, kernels and intermediate data.

There remain drawbacks for GPUs, primarily if they are intended to be used as general-purpose machines. Firstly, the limitations of adaptability and context switching make them less suitable for general-purpose computing tasks. Simple calculations which do not utilise the parallelism are inhibited by lower clock speeds. Communication between the CPU and GPU can introduce bottlenecks and decrease the GPU throughput, especially when waiting for results from the CPU. Memory capacity and bandwidth would also affect GPU performance; for example, an image processing application must wait for the image data to be transferred from the main memory, further delaying the runtime. Lastly, GPUs cannot operate independently without support from a CPU, which contributes to more power consumption from idling.

2.4.3 Field-Programmable Gate Array (FPGA)

Field-Programmable Gate Arrays are versatile integrated circuits which offer direct hardware programmability for diverse applications. They have gained prominence due to their reconfigurability, making them highly advantageous compared to fixed processing architectures such as ASICs. These features enable shorter time-to-market by allowing prototyping and late-stage design modifications. The FPGA architecture, as depicted in Fig. 2.14, comprises a matrix of configurable logic blocks (CLBs) containing a combination of look-up tables (LUTs), shift registers (SRs), and multiplexers (MUXs). These components are interconnected through programmable high-bandwidth pathways and are surrounded by I/O ports.

The fine-grained nature of FPGAs empowers designers to exploit both spa-

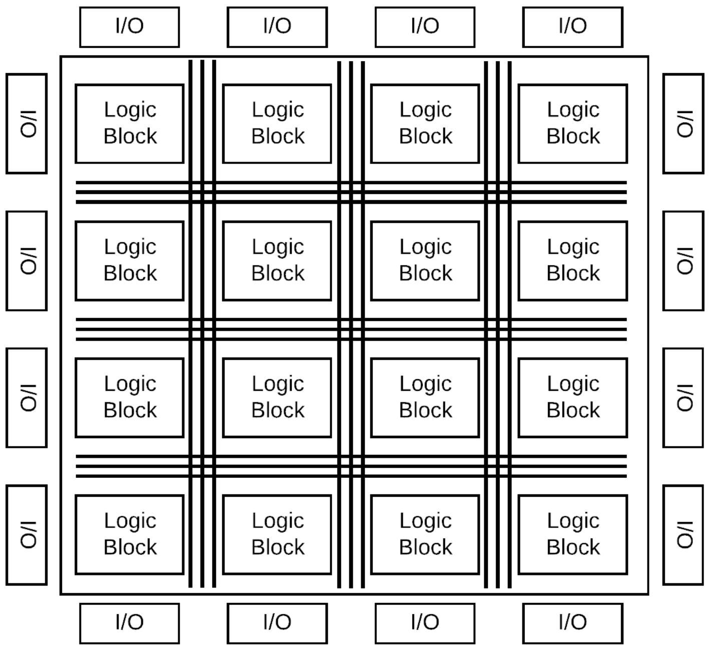


Figure 2.14: The FPGA Architecture contains Configurable Logic Blocks (CLBs), Interconnects, Programmable Routing, and I/O Resources components which define the Versatile and Reconfigurable Nature.

tial and temporal parallelism in their designs, resulting in enhanced performance. In image processing applications, algorithms can be tailored to operate on individual pixels or groups of pixels in parallel. Temporal parallelism can be achieved using techniques like pipelining, where separate processors work on successive stages of data, allowing concurrent processing and better throughput. Spatial parallelism, however, involves partitioning the image frame and processing each segment independently using separate processors.

FPGAs allow seamless integration of I/O, such as image sensors, enabling pixel data to be streamed directly into processing units without latency. Data can be routed efficiently to other embedded processors without external memory access. Block RAMs (BRAMs) within the FPGA enable exploiting data locality in vision kernels by keeping critical data on-chip. However, the main limitation in image processing applications often stems from external memory (E.g. DDR4 RAM) read/write operations, which can impact overall performance.

Advanced extensible interface (AXI) is a standard protocol for efficient communication between IP blocks within an FPGA design. It follows the Advanced Microcontroller Bus Architecture (ARM AMBA) specification, ensuring compatibility with ARM-based processors and systems-on-chip (SoCs). The AXI protocol supports separate read and write channels, enabling simultaneous data transactions in both directions. It also features burst transfers, allowing multiple data transfers within a single transaction to enhance data throughput.

Despite their advantages, FPGA development requires expertise in hardware descriptor languages (HDL), such as VHDL/Verilog. This steep learning curve can be a challenge for new developers accustomed high-level languages and instruction based architectures. In comparison to ASICs, the support functions and additional reconfigurable logic and power consumption overhead, making power efficiency considerations important during the design phase. FPGAs typically have limited on-chip memory compared to GPUs, which can have limitations for applications that require large memory spaces. Overall, FPGAs offer a powerful platform for image processing tasks, but their effective use requires careful consideration of design constraints and optimisation strategies.

2.4.4 Application-Specific Integrated Circuits (ASICs)

ASICs are a specialised type of Very Large Scale Integration (VLSI) technology where integrated circuits are designed specifically for a particular application domain. This involves custom designing at the transistor level to optimise the circuit for performance and silicon area. There are several advantages of opting for an ASIC implementation over other general-purpose accelerators. The custom designed nature of ASIC logic allows designers to create tightly integrated applications, resulting in better performance, reduced power consumption, and minimised silicon usage. ASICs come with intrinsic trade-offs listed below:

- **Fixed Design**: ASICs are designed for specific applications and lack flexibility compared to general-purpose processors. Once fabricated, it is challenging and costly to make modifications or upgrades to their functionality.
- **High Design Cost**: Designing and prototyping involves significant expertise and time, leading to higher initial development costs.

- **Long Development Timeline**: Creating a custom ASIC requires extensive expertise and significant time to design, verify, and manufacture.

Despite these drawbacks, the per-chip manufacturing cost becomes significantly lower during mass production, rendering ASICs more economically viable for high-volume production. The following sections discuss the various types of ASICS targeting specific workloads:

Vision Processing Units (VPUs)

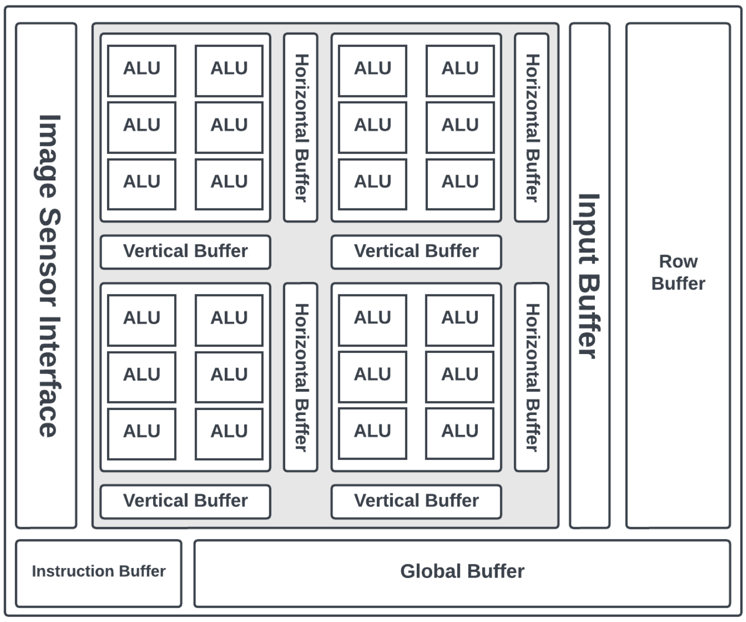


Figure 2.15: VPU Architecture consists of an array of processing elements with horizontal and vertical buffers for efficient image processing.

VPUs are a class of ASIC designed to alleviate the heavy processing load on the central processor by accelerating workload-specific tasks. VPUs shown in Fig. 2.15 have a distinct hardware design that focuses on accelerating specific types of computations, such as deep learning inference, video encoding/decoding, and image processing. They often incorporate dedicated execution units, tensor cores, or specialised instructions to accelerate these tasks efficiently.

VPUs employ hardware architectures and software frameworks tailored to exploit parallelism and optimise performance for these tasks. GPUs, while also capable of accelerating AI workloads, are designed to handle a wide range of general-purpose graphics and compute tasks, making them more versatile but potentially less optimised for specific workloads.

VPUs also prioritise energy efficiency, aiming to deliver better performance per watt over other accelerators. They employ techniques like low-power execution units, reduced precision compute, and power management features to minimise energy consumption. GPUs, on the other hand, focus more on delivering absolute performance, often consuming more power in exchange for higher computational capabilities. In addition, VPUs often have specialised APIs or libraries that target specific applications or frameworks, enabling efficient execution of AI models or video codecs. However, the programming ecosystem for VPUs is limited in comparison to general-purpose architectures.

Neural Processing Units (NPUs)

NPUs initially emerged in embedded devices as efficient AI inference accelerators specifically designed to manage the computational demands of machine learning workloads. The initial NPU architecture integrated high-density MAC arrays such as 2D GEMM or 3D systolic arrays since the majority of the computations are found within convolutional layers, which involve significant matrix multiplications. As CNNs continued to become increasingly complex with higher depth and many layers configurations, NPU has now optimised the MAC array structures to ensure enhanced modularity and scalability. Furthermore, newer features such as:

- Fused operations
- Sparsity acceleration
- Unified High Bandwidth Memory
- Multi-level array partitioning
- Mixed Precision Support

NPUs have expanded their capabilities for other neural network architectures. This includes RNN/LSTM structures, targeting for audio and natural language processing, and transformers.

Neuromorphic Hardware

Neuromorphic architectures are a type of hardware developed to mimic the structure and function of the human brain's neural networks. These architectures aim to replicate the principles of neural function in their operation, seeking inspiration from biological systems. By incorporating concepts such as weighted connections, activation thresholds, short and long-term potentiation, and inhibition, neuromorphic architectures aim to perform distributed computation in a way that resembles how the human brain processes information.

The key objective of neuromorphic architectures is to achieve efficient and parallel processing of data by leveraging the inherent capabilities of neural networks. These architectures often involve the use of spiking neural networks, where information is transmitted through spikes or pulses, similar to how neurons communicate in the brain. This approach allows for event-driven and energy-efficient computation, making neuromorphic architectures suitable for various tasks, including sensory data processing, pattern recognition, and complex decision-making. Despite their promising advantages, they face challenges, including complexity in design and implementation, limited applicability to specific tasks, scalability issues, lack of standardisation, and difficulty in implementing learning and adaptation mechanisms. Balancing energy efficiency and performance is another challenge, and commercial availability remains limited.

ASIC Summary

Table 2.4: ASICs and Their Specifications

Manufacturer	ASIC Type	Clock (MHz)	Power (W)	Process Node	Applications
Intel Movidius Myriad X	VPU	1050	2.5	16nm	Edge devices, AI inference
Google TPU	TPU	1800	10	7nm	Machine learning accelerators
ARM Ethos-U55	NPU	1000	1	7nm	IoT devices, edge computing
Huawei Kirin	NPU	820	1	7nm	Smartphones, AI applications
Graphcore IPU	VPU	500	252	16nm	AI workloads, data centres
Intel Neural Compute Stick	NPU	700	1.5	28nm	Machine translation, NLP, Edge AI

Table 2.4 offers a concise overview of various ASICS. These ASICs serve diverse purposes, from machine learning acceleration to edge computing and AI inference. Notable entries include the Intel Movidius Myriad X, known for its use in edge devices, and the Google TPU, a powerful tensor processing unit designed for machine learning tasks. The ARM Ethos-U55 and Huawei Kirin ASICs are optimised for IoT devices and smartphones, all while operating at low power consumption. Graphcore's IPU, on the other hand, stands out with its high power requirements, tailored for AI workloads in data centres. Lastly, the Intel Neural Compute Stick focuses on applications such as machine translation and natural language processing.

2.4.5 Heterogeneous Architectures

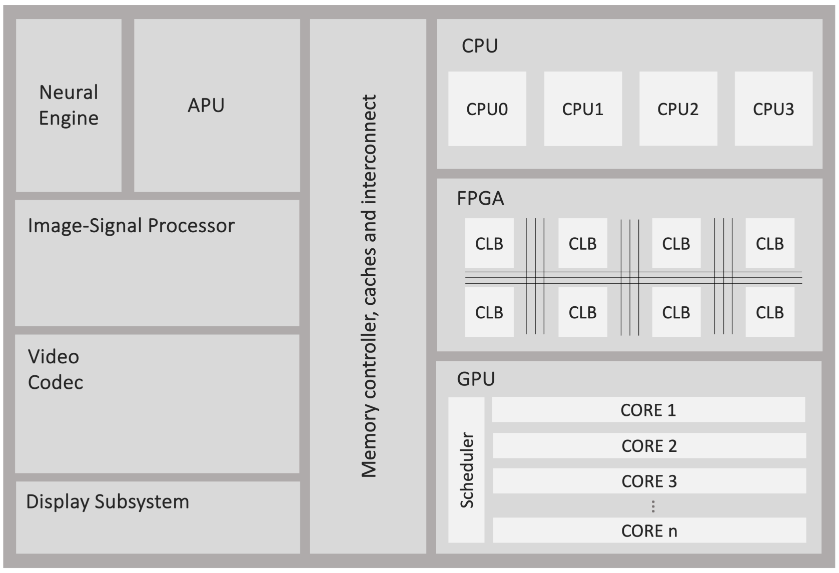


Figure 2.16: Concept Heterogeneous Architecture which integrate multiple specialised processing units onto a interconnected silicon chip.

Heterogeneous architectures have recently gained significant attention and mainstream appeal in various application domains. These architectures integrate different types of accelerators, including CPUs, GPUs, NPUs, and FPGAs, into a single compute fabric, observed in Fig. 2.16. Currently, commercial heterogeneous chips only contain a combination of CPU-GPU-NPU [20]. The primary objective of heterogeneous architectures is to accelerate complex tasks by allocating specific operations to the most suitable specialised cores that can process them efficiently.

One of the key challenges in utilising heterogeneous systems lies in algorithm design. Designing algorithms that can effectively leverage the capabilities of different accelerators is crucial. It requires careful consideration of the characteristics and strengths of each accelerator, as well as the partitioning and mapping of computational tasks to the appropriate cores. Algorithm designers need to analyse the computational requirements, data dependencies, and parallelism inherent in the application to optimise the workload distribution across different cores.

Partitioning and mapping refer to the process of breaking down the computational tasks and mapping them onto the available cores. It involves considering the data dependencies, communication overhead, and resource utilisation to ensure efficient execution. Additionally, scheduling tasks across different cores, managing synchronisation between them, and optimising interconnect requirements are critical aspects of achieving optimal performance in heterogeneous architectures.

The programming environment for heterogeneous architectures can be complex and diverse. Each accelerator may have its own programming model, APIs, and language extensions, making it challenging to develop applications that can fully exploit the capabilities of all accelerators. Furthermore, the availability of libraries and software tools may vary across different compute elements due to differences in instruction set architectures. This can lead to binary incompatibility and limit the portability of applications across different accelerators. Evaluating the performance of heterogeneous architectures requires comprehensive performance evaluation techniques. Benchmarks and performance metrics need to consider the characteristics of the application, workload distribution, and communication patterns to provide an accurate assessment of the system's capabilities.

2.4.6 Summary

Table 2.5: Hardware Architecture Specification Summary.

Architecture	Flexibility	Compute Type		Latency	Execution Paradigm	
		Temporal	Spatial		Dataflow	Instruction
CPU	General	✓	✓	Medium	×	✓
GPU	General	✓	✓	High	×	✓
FPGA	General	×(1)	✓	Low	✓	×
ASIC	Fixed	×	✓	Low	✓	×

(1) FPGA's can support temporal compute, however impractical considering overhead and effectiveness.

The table 2.5 provides a concise overview of various hardware architectures used in compute operations. CPUs and GPUs offer general-purpose flexibility, supporting temporal and spatial computations with medium and high latency, respectively, while following an instruction-based execution paradigm. FPGAs, though generally flexible, are better suited for spatial computations, with limited practicality for temporal tasks due to overhead and effectiveness constraints. They employ a dataflow execution paradigm. In contrast, ASICs are fixed-function hardware designed for specific spatial computations, offering low latency and following a dataflow execution paradigm.

2.4.7 Software Ecosystem

This section explores the software domain employed for targeting hardware architectures and software interfaces. Optimised libraries such as OpenCV, High-Level Synthesis, and Domain-Specific Languages assume a role in bridging the gap between hardware and software application development.

High-Level Synthesis (HLS)

High-level synthesis (HLS) is a tool that enables hardware designers to use a high-level programming language, such as Python, C or C++, to create hardware designs. This is in contrast to traditional hardware design methods, which involve manually writing hardware description languages (HDLs) such as VHDL or Verilog. HLS tools take in the high-level source code and automatically generate the corresponding HDL code. This can greatly simplify the design process, making it more accessible to non-hardware design experts. This means that designers can focus on the functionality of the design and not worry about low-level implementation details. HLS tools also perform optimisation to improve the performance and resource utilisation of the generated hardware. This can result in more efficient designs that use fewer resources and run faster.

Another benefit of HLS is that it allows for faster design iteration. As the design can be expressed in a high-level programming language, it can be easily modified and re-synthesised to see the effects of the changes. This can greatly speed up the design process and allow for faster time-to-market. In addition, FPGAs are often selected for systems where time to market is critical in order to avoid lengthy chip design and manufacturing cycles. The designer may accept the increased performance, power, or cost in order to reduce design time. Modern HLS tools put this trade-off into the hands of the designer; with more effort, the quality of the result is comparable to handwritten RTL (register transfer language). ASICs have high manufacturing costs, so there is a lot of pressure for designers to achieve success on the first attempt. Design iterations can quickly and inexpensively be done without huge manufacturing costs.

However, these tools come with a set of drawbacks. For instance, the initial learning curve can be steep, particularly for those new to hardware design, as they require a solid understanding of both high-level programming and hardware optimisation techniques. While HLS tools automate the allocation of hardware resources based on the provided code, they may not always yield the most efficient designs for complex projects compared to manual, fine-tuned hardware descriptions. One of the key challenges in using HLS tools is accurately predicting the performance of the generated hardware. Factors such as memory access patterns, data dependencies, and the overall architecture can significantly impact performance, making it challenging to estimate how the synthesised hardware will behave. Moreover, debugging HLS-generated designs can be complex. Traditional software debugging methods are often insufficient, as hardware-related issues might not manifest in the same way as in software. This can prolong development cycles and hinder the identification of issues.

Domain-Specific Languages (DSL)

Domain-specific languages (DSLs) are programming languages designed to address specific problem domains rather than being general-purpose languages. DSLs offer higher-level abstractions and syntax tailored to a particular application area, allowing users to express domain-specific concepts more concisely and intuitively. Unlike general-purpose languages, DSLs enable non-experts to work effectively within a specific domain, as they are more focused on the domain's requirements and semantics. DSLs come in two main types: external DSLs, which are standalone languages distinct from the host language (*e.g.*, Cal Actor Language [21]), and internal DSLs, which are embedded within a general-purpose language using its syntax and tools (*e.g.*, Halide [22]). The use of DSLs can lead to improved productivity, reduced error rates, and better code maintainability in specific application areas.

Libraries & Frameworks

Optimised libraries such as OpenCV [23] are essential tools used to develop vision and deep-learning applications. These libraries offer a comprehensive collection of pre-built algorithms and functions for a wide range of image-related tasks. Their significance lies in the substantial time and resource savings they provide, enabling developers to utilise tried-and-tested algorithms, thus reducing development efforts and benefiting from community-driven improvements. Moreover, optimised libraries ensure cross platform compatibility, supporting various programming languages and platforms. They are continually updated to harness advancements in hardware and software, making them key for efficient and adaptable image processing.

Deep learning frameworks such as Pytorch [24] offer abstraction and simplification, allowing developers to focus on high-level tasks. Frameworks encompass a comprehensive suite of support programs, compilers, code libraries, toolsets, and application programming interfaces that provide a cohesive environment that streamlines the development of systems. Therefore, frameworks facilitate rapid prototyping and integration with other tools.

2.5 Conclusion

In conclusion, this section provides an in-depth overview of the imaging pipeline and its fundamental components, establishing the groundwork for the subsequent chapters. It encompasses typical operations present in each pipeline stage, which will used as implementation examples. Furthermore, the section delves into diverse hardware platforms such as CPUs, GPUs, VPUs, and FPGAs, each offering distinct attributes that can accelerate algorithms. To leverage these hardware capabilities, a range of tools and methodologies are introduced, which include high-level synthesis. In the next chapter, the state-of-the-art study on heterogeneous architectures and optimisation strategies related to image processing are discussed.

3 State-of-the-Art

This chapter surveys the literature relevant to the research conducted in this thesis. The work covered in this section spans a wide range of topics, including image processing, CNNs, hardware, algorithmic, and domain-specific optimisation approaches. Additionally, the chapter reviews proposed heterogeneous platforms and partitioning methods. A critical analysis of recent research publications is performed, and potential areas for exploration are discussed throughout the section.

3.1 Hardware Targeting Image Processing

This section introduces imaging algorithms implemented on various architecture configurations found within the literature. Heterogeneous architectures, which integrate diverse computing elements like CPUs, GPUs, FPGAs, and specialised accelerators, have emerged as a pivotal paradigm in modern computing systems, aiming to achieve higher performance and energy efficiency. These architectures cater to the diverse computational needs such as parallelisation or pipelining for tasks involving deep learning to signal processing. In addition to the literature on supporting algorithms that are tailored to exploit the unique capabilities of these heterogeneous components. Furthermore, various optimisation methods are explored for each hardware.

3.1.1 Multi-Core CPU Architectures

While accelerators with numerous cores such as GPUS, have traditionally outperformed CPUs in image processing due to core count, the recent introduction of many-core CPUs boasting thousands of cores has become more competitive in runtime performance. Furthermore, considering the initialisation and memory latency required for GPUs, CPUs may complete kernels within that timeframe [25–27].

Many-core co-processors, relying on simple hardware, place substantial demands on software programmers, while their use of in-order cores struggles to tolerate long memory latencies. In addressing these challenges, work has been done to explore decoupled access/execute (DAE) mechanisms for tensor processing. One software-based method is to use naïve and systolic DAE, complemented by a lightweight hardware access accelerator to enhance area-normalised throughput. This method has shown $2 - 6\times$ performance improvement on a 2000-core CPU heterogeneous system compared to an 18-core out-of-order CPU baseline [28]. Executing fundamental image processing operations, such as Winograd-based convolution, on many-core CPUs (Intel Xeon Phi), has shown comparable performance for 2D ConvNets. Additionally, it has demonstrated $3\times -8\times$ times better runtime performance for 3D ConvNets compared to the best GPU implementations [29].

3.1.2 CPU-GPU Architectures

The CPU-GPU architecture is a widely adopted approach to implementing of complex image processing algorithms. The architecture leverages many simple processing cores, which are efficient in executing parallelised tasks. The CPU is typically responsible for orchestrating the high-level control flow and task management allocation to the GPU. Many works developing image processing algorithms on GPUs [30–32] have exhibited a $10 \sim 20$x speedup in runtime compared to single CPU implementations. In real-time imaging, works such as optical flow [33] and edge-corner detection [34] were evaluated for their algorithmic performance on GPUs and FPGAs. The results observed show that GPUs slightly outperform FPGAs by utilising large amounts of data parallelism and hiding latency. Dynamic thread scheduling on the GPU hides

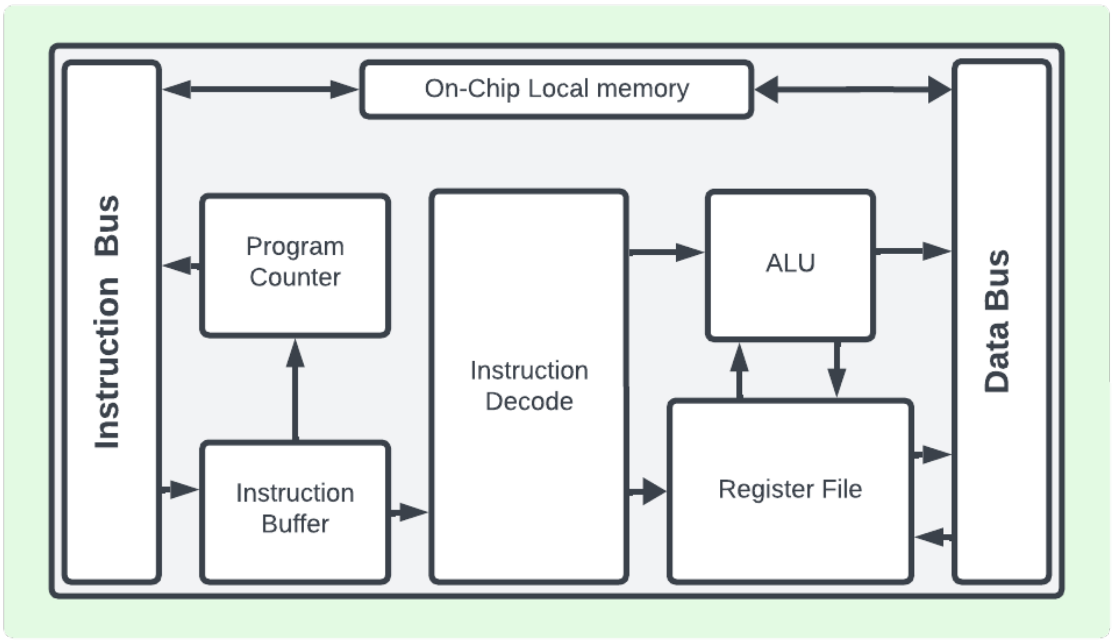


Figure 3.1: Generic Soft Processor Architecture on Programmable Logic.

memory latency by swapping threads and making memory requests with others, as long as there are enough threads to keep the process continuous. In addition, easy programmability of GPUs supports software debug iterations which involve fast edit/compile/execute cycles compared to the much more time consuming FPGA [35].

3.1.3 CPU-FPGA Architectures

FPGA:

FPGAs have been utilised for image processing in order to leverage their unique architectural characteristics, such as parallelism, reconfigurability, and low latency. These features enable FPGAs to excel in tasks that demand real-time analysis of image data and require lower power consumption [36].

One major drawback of using FPGAs for image processing is the need to primarily use fixed-point arithmetic. While FPGAs can handle floating-point arithmetic, it often demands too many resources, especially for parallel processing. Vision research typically relies on floating-point algorithms, and adapting them to fixed-point requires a detailed analysis to determine necessary precision at each stage and to work within the FPGA's resource limits [37]. When compared directly to ASIC devices, disregarding cost and design timelines, FPGA implementations are generally less efficient due to the configuration circuitry overhead, which includes I/O and the required SRAM cells to store the current design. This results in larger device sizes and higher power consumption. ASIC design processes also enable circuitry optimisation for faster clock speeds than those achievable on FPGAs [38].

CPU-FPGA:

Historically, these two components operated independently, each catering to its own application domains. However, in recent years, manufacturers have recognised the complementary strengths of CPUs and FPGAs. This has led to the development of integrated systems, which can be split into two categories, which are soft or hard processors. A soft processor is realised using the programmable logic resources of an FPGA. It's essentially a processor described in a hardware description language, such as VHDL or Verilog, which is then synthesised and mapped onto the FPGA's logic blocks. This design offers flexibility, allowing designers to modify the architecture, add custom instructions, or adjust interfaces as needed, with Xilinx MicroBlaze [39] and Intel Nios V [40] being notable examples. In contrast, a hard processor is a physical processor core embedded directly into the FPGA silicon which is optimised and hardwired for better performance and efficiency. The ARM Cortex cores found in newer Xilinx's Zynq FPGAs [41] are typically connected to the programmable logic elements through an AXI (Advanced eXtensible Interface) protocol for efficient data transfer and communication.

Initially, soft CPUs were utilised for pre-processing, task scheduling, and resource management. However, collaborative execution, where tasks are computed by both accelerators, have emerged as a prominent approach for increasing application performance, as demonstrated in the literature [42–44]. In image processing, soft processors have been shown to be more energy efficient and have comparable runtimes than their counterpart discrete processors for low-high complexity algorithms, which is shown in the works [45–47]. The performance gains extend into the deep learning domain such as CNNs are presented in literature [48–51].

Table 3.1: Energy and Runtime Speedup of CPU-GPU-FPGA heterogeneous architecture implementations compared to single GPU. The Table only includes works where algorithms are partitioned and processed on all accelerators.

Work	Heterogeneous Platform		Partitioning Strategy	Algorithms	Energy Gain	Runtime Speedup
Hyungmin C, etal [52]	GPU+CPU	ARM+P100	Element-wise	Long Short-term Memory	∼0.34x	∼4.2x
	FPGA	Zync Ultrascale				
Hosseinabady M, etal [53]	GPU+CPU	ARM+Jetson TX1	Element-wise	Histogram	∼2.29x	∼1.79x
	FPGA	Virtex-7 Zync Ultrascale		Dense Matrix-Vector Multiplication	∼1.19x	∼1.48x
				Sparse Matrix-Vector Multiplication	∼1.23x	∼1.25x
Yuexuan T, etal [54]	GPU+CPU	ARM+Jetson TX2	Hybrid	LeNet-5 N=16	∼2.11x	∼1.3x
	FPGA	Nexys Artix-7				
Carballo-Hernandez, etal [55]	GPU+CPU	ARM+Jetson TX2	Layer-Wise	SqueezeNet Fire	∼1.34x	∼1.01x
	FPGA	Cyclone-10 GX		MobileNet V2 Bottleneck	∼1.55x	∼1.26x
				Shufflenet V2 Stage	∼1.39x	∼1.35x
Sumeet N, etal [56]	GPU+CPU	ARM+A100	Grouped Layer-Wise	ResNet-18	∼1.14x	-
	FPGA	Xilinx Alveo U280		ResNet-50	∼1.08x	
				VGG16-bn	∼1.12x	

In summary, hard processors typically outperform soft processors in both speed and resource utilisation due to their independence from FPGA fabric speed and separate chip placement, resulting in enhanced clock speeds and efficient data path designs. Soft processors excel in power efficiency and adaptability, catering to scenarios prioritising energy-conscious designs and dynamic modifications during rapid prototyping and development stages. The architectural distinctions align each processor type with specific application requirements within FPGA-based computing landscapes.

3.1.4 CPU-GPU-FPGA Architectures

FPGAs offer the advantage of direct hardware mapping for efficient implementation of CNNs [57], but they are often constrained by limited on-chip resources [58]. The complexity and size of state-of-the-art CNNs often exceed the available logic and memory resources on a single FPGA chip. To mitigate this limitation, a heterogeneous approach can be employed where different layers of the CNN are mapped onto both FPGA and GPU platforms. This leverages the FPGA's efficiency for specific layers while utilising the GPU's computational power for more complex layers, thereby creating a balanced and optimised system.

Recent work into partitioning and executing algorithms onto heterogeneous CPU-GPU-FPGA architectures has been explored in the literature, collected in Table 3.1. The platforms category records the combination of accelerators that compose the heterogeneous platform in which the algorithm has been distributed across. The results for energy gain and runtime speedups are derived from comparing the algorithm executed on a single GPU. Partitioning strategy refers to the level of detail at which operations are divided on a heterogeneous platform. In a coarse-grained implementation, entire algorithms or large functional blocks are distributed across different processors within the system. Conversely, a fine-grained implementation maps individual components or layers of an algorithm to specific processors, allowing for more targeted optimisation and resource utilisation. Across all studies, a range of $1 \sim 4\times$ speedup and $1 \sim 2.3\times$ energy improvement is observed from various partitioning techniques. The key areas identified from all works are that the limiting factors for performance are communication latency, resource availability, coarse partitioning strategies and limited optimisations. In all works, CNN algorithms were implemented partially (*e.g.*, Convolution Layer Only) or did not pass data to other accelerators, therefore not utilising true heterogeneity.

3.2 ASIC Architecture

ASICs are highly efficient because they are purpose-built and don't require additional support hardware. They integrate all necessary components on a single chip, minimising external dependencies and reducing overall system complexity, making them ideal for streamlined and dedicated image processing tasks [59, 60]. VPUs have been shown to achieve similar performance to reference CPUs, GPUs and FPGAs. Additionally, benchmarking of NPUs within mobile platforms has shown to be better in runtime than desktop CPUs and comparable to GPUs while consuming less energy [61].

3.3 Image Processing Optimisations

Optimisations are necessary for improving overall system performance, There are three primary categories. First, Hardware optimisations which include optimising memory architectures, computation engine and integrating additional accelerators. Algorithmic optimisations focus on improving the computational procedures, ensuring efficient solving strategies. Lastly, domain-specific optimisations leverage domain knowledge and characteristics inherent to image processing which aim to improve performance, accuracy, and computational efficiency. This section explores optimisation techniques tailored for image processing and CNN algorithms across various hardware accelerators, primarily focused on FPGAs.

Hardware Optimisations

In image processing, memory usage primarily contributes to overall energy consumption and runtime, especially when algorithms require complete frames to be stored in memory [62]. However, accelerators such as FPGAs have resource limitations, making efficient utilisation critical for meeting performance, size, and power constraints [63]. Hardware-based memory optimisations can be classified into on-chip and off-chip categories:

- On-chip: Involves optimising the use of fast but limited on-chip memory resources like Block RAM in FPGAs or L1/L2 caches in CPUs and GPUs.
- Off-chip: focuses on optimising the use of larger but slower off-chip memory like DDR RAM.

Line buffers are a memory optimisation technique used in convolution-based algorithms by minimising redundant memory access. The first few lines of the image or signal are loaded into the line buffer, marking the only time these lines are fetched from the main memory. As the convolution operation progresses through the image or signal, the lines already in the buffer are reused to calculate multiple output pixels, thereby eliminating the need to fetch the same lines from the main memory again. When moving to the next set of lines, the buffer shifts, discarding the oldest line and fetching a new one to ensure that the lines immediately needed for the convolution

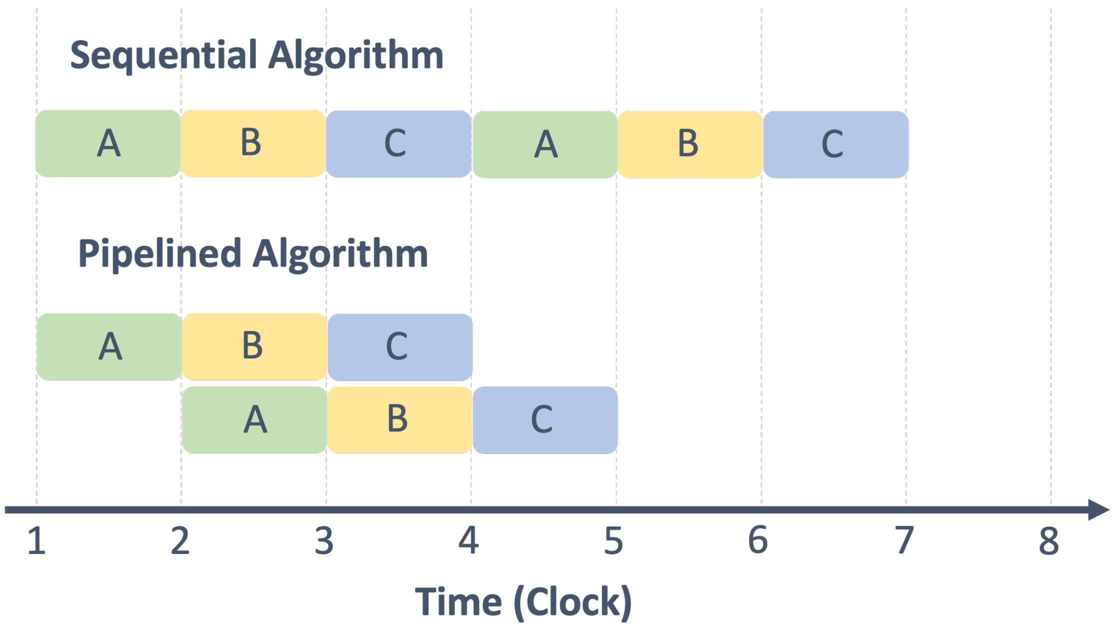


Figure 3.2: Hardware Pipelining: concurrent processing of multiple stages in a computational task, enhancing throughput and reducing latency.

are always available. This data reuse and line shifting minimises the number of times the slower main memory has to be accessed. Line buffers have been explored throughout literature [64–67]. Ping-pong buffers employ a dual-buffering scheme, where two or more buffers alternate roles in a synchronised manner. This approach allows one buffer to be filled with new data while the other is being processed, thereby increasing the throughput by speeding up the read/write process [68–70].

Pipelining is a technique used to increase the throughput by partitioning complex operations into discrete, independent stages implemented within logic elements like LUTs and flip-flops. Each stage performs a specific operation and is clocked separately, allowing for concurrent execution of multiple data elements across stages shown in Fig. 3.2. In the works of, Jiang *et al.* [68] and Bai *et al.* [69], significantly improved throughput in their designs, the data generated by each operation was transferred to the next operation without storage, reducing resource consumption and off-chip latency. Additionally, It is important to ensure stage independence for maximum parallelism and to balance resource utilisation to avoid bottlenecks.

Look-up tables (LUTs) are an effective optimisation technique to increase efficiency [71,72]. LUTs pre-compute and store the results of frequently used operations, allowing for rapid retrieval and eliminating the need for redundant calculations. In addition, for more complex expressions, such as square roots or multiplying and dividing by an arbitrary number, look-up tables (LUT) and raster based incremental methods can offer improved performance.

Memory architecture can significantly impact both performance and energy efficiency, especially when dividing on-chip memory into smaller blocks to allow parallel access and reduce latency. However, the choice of parallelism influences the required memory organisation and, consequently, the total energy consumption, which is explored in literature [73, 74]. Following on previous works, Tessier *et al.* [75] showed on-chip power reduction through converting user-defined memory specifications to on-chip FPGA memory block resources. FPGAs often have fixed-size memory that may not align well with the task at hand, leading to energy overhead. Partitioning techniques are therefore required to efficiently manage the storage and processing needs of image data. In the work, Garcia *et al.* [76], showed that effectively partitioning image frames into BRAMs in order to maximise utilisation (ie, minimise the number of required on-chip memories) can reduce power consumption without affecting the performance. Various off-chip caching systems have been developed to mitigate the latency overheads, such as a three-level memory access architecture proposed by Zhang *et al.* [77]. This architecture includes off-chip memory, on-chip buffers, and local memories. Nonetheless, the system entails significant waiting times for valid signals between Block RAMs (BRAMs) and off-chip memory, introducing delays.

Approximate computing techniques can significantly improve computational throughput and energy efficiency in image processing tasks when implemented on FPGAs [78]. These methods trade off a small degree of accuracy for performance gains. Two primary strategies are generally used: the first leverages approximate arithmetic for reduced-precision calculations, while the second aims to decrease the total number of operations without substantially affecting output quality. These approaches can be integrated into the learning or optimisation stages to balance both accuracy and computational demands effectively. One study has shown using lower-bit precision like INT8 and INT4 significantly speeds up neural network inference on various architectures. For example, INT8 inference led to up to 5.02× speedup on GPUs, and INT4 added another 50-60% speed gain. Mix-precision further improved

ResNet50's speed by 2% without accuracy loss. These benefits extend to non-GPU platforms, achieving up to 2.35× speedup [79].

CNN Hardware Optimisations

CNNs have now become a popular method of feature extraction and classification. Therefore, this section explores hardware-based optimisation techniques that improve performance. Optimising CNNs on hardware accelerators requires careful algorithm-to-hardware mapping and resource management. The convolutional and fully connected layers are typically the most resource-intensive in terms of both computational logic and memory footprint. Specifically, the storage of high-precision weights and biases for these layers can consume substantial portions of on-chip memory, while the multiply accumulate (MAC) operations required for convolutions and activation's demand significant computational resources as discussed by Laith *et al.* [80].

Multiple techniques address these challenges. Firstly, pruning is a compression technique that reduces the model's complexity [81]. These methods identify and remove weights and neurons that contribute minimally to the model's predictive performance, usually based on certain statistical or empirical thresholds. Pruning lessens the memory footprint, reducing the storage required for high-precision weights and biases. At its simplest level, pruning removes the smallest weights, setting them to zero as demonstrated by Song *et al.* [82]. When optimised for energy consumption, pruning techniques that target the least energy-consuming weights achieved a $1.74\times$ gain in efficiency compared to traditional approaches [83]. In both methods, the pruned network is fine-tuned to maintain the classification accuracy. Multiple studies demonstrate that pruning eliminates 53% to 85% of weights in a CNNs convolutional and fully connected layers while losing around $0.5 \sim 1\%$ accuracy [84, 85]. Table 3.2 summarises optimisations techniques found in hardware.

Domain-Specific Optimisations

Domain-specific optimisations within the imaging domain are methods tailored to increase the performance of applications. These optimisations lever-

Table 3.2: Summary of Hardware Optimisation Techniques

Optimisation Technique	Description
Pipelining	Concurrent processing of tasks in stages within a pipeline.
Vectorisation	Performing operations on entire vectors of data in a single instruction.
Cache Optimisation	Enhancing data locality and minimising cache misses for improved memory access.
Line Buffer	Storing and processing a line of data at a time; optimising access patterns and reducing memory bandwidth usage.
Look-Up Table	Using precomputed values stored in a table for quick retrieval; enhancing computational efficiency.
Memory Architecture	Optimising the design and organisation of memory systems for efficient data access.
Approximate Computing	Allowing imprecise calculations without prioritising accuracy.

age the unique characteristics of image processing. Such optimisations often involve exploiting properties like spatial locality, symmetry, and redundancy present in images. In literature, there has been very little research on platform agnostic domain-specific optimisations of imaging algorithms on FPGAs. Domain-specific tools and optimisations, particularly in areas such as compilers [86–88], have been explored but not yet reached maturity.

In the field of image processing, domain-specific optimisations aim to significantly reduce computational load while maintaining consistent accuracy. Examples of such optimisations include down-sampling [89], approximation [90], data-type conversion [91], kernel size adjustments [92], bit-width modification [93], and the complete removal of certain operations. Although hardware acceleration techniques for algorithms on CPUs, GPUs, and FPGAs have been extensively researched [94–96], these studies generally focus only on target algorithms. In contrast, there has been limited work on exploring the performance and accuracy trade-offs of domain-specific optimisations of imaging algorithms specifically for FPGAs.

Downsampling is a popular method used to reduce the amount of data in an image by selectively removing samples. This involves reducing the resolution of an image by eliminating pixels, usually through averaging or taking the value of a representative pixel in a local neighbourhood. The aim is to decrease computational complexity and storage requirements, making it easier to process and analyse the data. However, downsampling comes with the trade-off of losing some level of detail, which may be critical for certain applications. It is essential to choose an appropriate downsampling factor and method to balance computational efficiency with the preservation of important features in the data.

Fast Fourier Transform is an algorithm to compute the Discrete Fourier

Transform (DFT) and its inverse in a more efficient manner. It capitalises on the properties of symmetry and periodicity in the Fourier domain to reduce the number of arithmetic operations. Instead of directly convolving spatial or time-domain signals, FFT first transforms both the input signal and the kernel into the frequency domain. Here, the convolution operation transforms into a simpler element-wise multiplication. After this multiplication, an inverse FFT (IFFT) is applied to bring the data back to the spatial or time domain. The FFT algorithm reduces the computational complexity from $O(n^2)$ for direct convolution to $O(n \log n)$, making it highly efficient, especially for larger kernels. The use of FFTs in image processing is found in many works [97–99]. However, FFT is hardware-intensive due to its high memory bandwidth requirements and arithmetic complexity, which can lead to increased power consumption.

Additional work by Qiao *et al.* [100] proposed a minimum cut technique to search fusible kernels recursively to improve data locality. Rawat *et al.* [101] proposed multiple tiling strategies that improved shared memory and register resources. However, such papers propose constrained domain-specific optimisation strategies that exclusively target CPU and GPU hardware. In related work, Reiche *et al.* [102] proposed domain knowledge to optimise image processing accelerators using high-level abstraction tools such as domain-specific languages (DSL) and reusable IP-cores. Additional optimisation techniques commonly used in general-purpose computing, such as loop unrolling, fission, and fusion, do not map effectively onto FPGA architectures due to the distinct operational paradigms and resource constraints inherent to FPGA design. Consequently, there is a need for the development of accelerator-agnostic and domain-specific optimisation strategies that can be universally applied across diverse computational platforms, including CPUs, GPUs, and FPGAs, for a more cohesive and efficient heterogeneous design.

Algorithmic Optimisations

Algorithmic optimisations refer to techniques employed to exploit mathematical properties or patterns in the data being processed. Strategies may include the use of more efficient data structures, dynamic programming, divide and conquer techniques, and algorithmic transformations. Convolution operations are used in many image processing algorithms which typically account for the majority of computation time. Various algorithmic convolution optimisation strategies are discussed below:

- The *Strassen* [103] algorithm optimises matrix multiplication through recursive partitioning. Given two $n \times n$ matrices, A and B, it divides each into four submatrices and recursively computes seven intermediate products (M_1 to M_7). These products are combined to yield the final matrix using additions and subtractions. The algorithm's time complexity of $O(n^{\log_2 7})$ improves upon the $O(n^3)$ complexity of naive multiplication, particularly advantageous for larger matrices. However, its practicality diminishes for smaller matrices due to increased constant factors and memory requirements associated with additional operations. The algorithm has shown to be effective in reducing the computational complexity without losing accuracy in CNN algorithms [104].
- The *Winograd* [105] filter algorithm utilises minimal filtering algorithms to perform convolutions, particularly advantageous for small kernel sizes $K \leq 3$. It transforms the convolution operation into a set of polynomial multiplications in a transformed domain. The idea is to decompose the convolution into smaller, overlapping tiles and then apply the Winograd transformation to each tile separately. This results in a significant reduction in the number of multiplicative operations, which are computationally more expensive than additive operations. Numerous works in literature implementing and comparing Winograd performance [106, 107]. In comparison to FFT-based methods which also reduce the number of multiplications by transforming the convolution into a point-wise multiplication in the frequency domain. However, these methods introduce the overhead of complex-to-real transformations and are more computationally intensive for small kernel sizes due to the increased number of additions and the need for padding.

Specific image processing algorithms often require sorting pixels (e.g., median filtering). These algorithms can benefit from parallel sorting network algorithm optimisations. Sorting networks consist of a predefined sequence of compare-and-swap operations, often organised in a pipeline or tree-like structure, enabling simultaneous execution. In the case of median filtering, sorting networks such as Batcher's Odd-Even Mergesort [108] can be implemented to sort the values in the input window in parallel, thereby reducing time complexity [109].

Table 3.3: Commercial & Academic High-Level Synthesis Compilers.

Compiler	Owner	Year	License	Input	Output
Vitis [112]	Xilinx	2013	Commercial	C++	VHDL/Verilog
HDL Coder [113]	Mathworks	2019	Commercial	Matlab/Simulink	VHDL/Verilog
Intel HLS [114]	Intel	2017	Commercial	C++	VHDL/Verilog
Hardcaml [115]	Jane Street	2018	Open Source	Ocaml	VHDL/Verilog
Stratus HLS [116]	Candence	2015	Commercial	C/C++/SystemC	RTL
AUGH [117]	TIMA Lab.	2012	Academia	C subset	VHDL
Shang [118]	U. Illinois	2013	Academia	C subset	Verilog

In CNNs, algorithmic optimisations are commonly used to reduce runtime for both convolutional and fully connected (FC) layers. General Matrix Multiply (`GEMM`) is a key method for implementing these layers, as indicated in [110,111]. In the FC layer, `GEMM` proves effective for batch processing feature maps (FMs), organised as a $CHW \times B$ matrix. In which C represents the number of channels, H for height, W for width, and B for the batch size. This approach optimises computational throughput and memory bandwidth by loading weights just once per batch. Given that FC layers house the majority of CNN weights, `GEMM` significantly enhances computational speed, especially with increasing sparsity in the FC weight matrix.

3.4 High-Level Synthesis

High-level synthesis (HLS) is a potential solution to increase the productivity of real-time image processing development on FPGAs. In order to close the performance gap between the manual and HLS-based FPGA designs, various code optimisations that exploit FPGA architecture for potential speedups are made available in today's HLS tools. At present, there are various high-level synthesis compilers that are being developed commercially and in academia, shown in table 3.3.

Efficient code optimisation by HLS compilers is vital in real-time image processing applications, where the goal is to minimise execution time and resource utilisation [119]. Furthermore, the quality of generated Register Transfer Level (RTL) descriptions in High-Level Synthesis (HLS) is found to be influenced by the high-level language used, prompting a need for optimised approaches [120]. Notably, comparative research underscores significant performance gaps between HLS-based designs and manually crafted counterparts for intricate applications [121–123]. Noteworthy disparities of up to 40 times in performance have been documented, particularly evident in demanding tasks like high-definition stereo matching.

Several contemporary avenues in HLS compiler optimisation merit exploration. In the work [124], investigates the impact of compiler optimisations on hardware generated by HLS tools, highlighting the significance of both the optimisation strategies and their sequential application order in enhancing the RTL output quality. In addition, a subsequent study [125], refines the understanding of HLS-based real-time image processing design optimisations. Applying a sequence of optimisation techniques, this approach showcases comparable performance when benchmarked against alternative methodologies and industry-standard HLS tools. The optimisations applied in both works include:

- **Function In-lining:** Incorporating functions directly into the code to eliminate function call overhead and improve overall performance.
- **Loop Manipulation:** Adjusting loop structures to enhance computational efficiency and reduce processing time.
- **Symbolic Expression Manipulation:** Utilising symbolic expressions to manipulate mathematical representations for improved computational speed and precision.
- **Loop Unrolling:** Expanding loops by replicating their bodies to reduce loop-control overhead and maximise parallelism.
- **Array Transformation:** Applying various techniques to optimise arrays, the two commonly applied techniques are listed:
  - **Array Partitioning:** Dividing arrays into smaller, more manageable partitions to enhance data locality and optimise parallel processing.
  - **Array Reshaping:** Modifying the shape of arrays to better align with computation requirements, improving overall efficiency in data processing.

The advantages of HLS tools observed by Wakabayashi *et al.* [126], on elevating abstraction levels highlight how expressing designs in higher-level languages like C and C++ can significantly reduce code complexity, making them better at managing complex designs more effectively.

Domain-Specific Languages (DSL)

Table 3.4: Image Processing Targeted Domain-Specific Languages

Compiler	Owner	Year	License	Frontend	Backend
Halide-Genesis [127]	Akari, Ishikawa Et al.	2019	Academia	Actor/Dataflow	VHDL/Verilog
RIPL [128]	Robert,Stewart Et al.	2018	Academia	Actor/Dataflow	C/Verilog
HiPacc [129]	R., Membarth Et al.	2015	Academia	C++	VHDL/Verilog
CAPH [130]	J. Sérot Er al.	2013	Academia	Actor/Dataflow	VHDL/SystemC
RVC-CAL [131]	EFPL	2010	Academia	Actor/Dataflow	C/Verilog

A domain-specific language (DSL) is a specialised programming language for a particular domain, such as image processing. The following paragraphs look into some recent developments of DSLs within image processing. In contrast to C/C++ with HLS tools, DSLs provide clearer syntax, rigorous semantic checks and possible compiler domain optimisations for improved generated code. A summary of available DSLs is found in Table 3.4.

Richard, Membarth *et al.* [129] proposed a new DSL and source-to-source compiler for image processing called 'Hipacc'. They show that domain knowledge can be captured to generate tailored implementations for C-based HLS from a common high-level DSL description targeting FPGAs and GPUs. The image processing algorithms that were generated in VHDL/Verilog code from the DSL are evaluated by comparing them with hand-written register transfer level (RTL) implementations. The results show that the HLS still has deficiencies in contrast to the RTL but enables rapid design space exploration. The Hipacc framework does not generate the hardware descriptor language but relies on Xilinx's HLS tools for generated HDL optimisations.

In the work by Jocelyn, Serot *et al.* [130] presented CAPH, a DSL suited to implementing stream-processing based applications on FPGA. CAPH relies upon the actor/dataflow model of computation and the tool suite also contains a reference interpreter and a compiler producing both SystemC and VHDL code. CAPH was evaluated by implementing a simple real-time motion detection on an FPGA platform. This was done to validate the overall methodology and to identify key issues. The results established three research directions to improve CAPH. The first is assessing the tools on larger and more complex applications and comparing them with hand-crafted RTL in terms of resource usage and runtime. The second research direction is improving the compiler and optimising the generated VHDL code. Third, applying static analysis techniques to actor behaviours to statically estimate the size of FIFO channels.

Another DSL for FPGAs was proposed by Robert, Stewart *et al.* [128] called RIPL. The aim is to increase throughput by maximising clock frequency and minimising resource usage to fit complex algorithms onto FPGAs. RIPL introduces an algorithmic skeleton to express image processing algorithms which are then exploited to generate deep pipelines of highly parallel and memory-efficient image processing components. The data-flow graph generated is expressed in CAL actor language and is compiled into Verilog. The DSL was used to implement image watermarking and multi-dimensional subband decomposition algorithms.

3.5 Benchmarking

Benchmarking is a relatively established concept in computing, serving as a crucial tool to gauge the performance of various systems. Identifying the most suitable hardware platform becomes imperative, especially when aiming for efficiency and performance. Yet, the challenge lies in determining this without investing significant time and understanding into implementations. Thus, benchmarking measures performance and aids in making informed decisions about hardware compatibility for complex algorithms.

Numerous benchmarking studies have been conducted on a variety of accelerators, including FPGA [132], GPU [133], TPU [134], and NPU [135]. These studies assess performance using specific metrics and facilitate comparisons between different hardware platforms [136]. Additionally, recent interest in the industry has begun to develop suites such as MLPerf [137] and DataPerf [138] to establish a standardised ML benchmarking and evaluation to allow comparison of inference/training chips or models.

Early work in heterogeneous benchmarking was the introduction of the Rodinia suite in 2009 [139]. Rodinia comprises applications and kernels that embody the behaviours of the Berkeley dwarfs. These are a taxonomy of 13 computational patterns widely used in various scientific domains. The suite also addresses communication, synchronisation, and power consumption issues. However, Rodinia does not leverage newer features, such as advanced heterogeneous programming constructs, half/mixed precision, tensor computations, and libraries that enhance communication between architectures. Instead, Rodinia focuses on more abstract algorithms rather than micro-benchmarks that target specific components.

Scalable Heterogeneous Computing (SHOC) [140] developed in 2010 as another benchmark suite targeting heterogeneous systems. The suite primarily focused on scientific computing workloads, including common kernels such as matrix multiply, fast Fourier transform, and Stencil computation. The benchmark is divided into two testing methods: The stress tests use computationally demanding kernels to identify OpenCL devices with bad memory, insufficient cooling, or other device component problems. The other tests measure many aspects of system performance on several synthetic kernels as well as common parallel operations and algorithms. SHOC supports various versions of benchmarking, from serial to testing inter-communication between architectures. Every application in the SHOC suite operates within a cohesive framework that allows users to define specific testing parameters, including the desired number of iterations. The framework can also capture intricate metrics, such as floating-point operations per second (FLOPS). The drawback to SHOC is that it focuses on basic parallel algorithms, thus missing the nuance of real-world applications; just like Rodinia, it has not been updated to test modern algorithms and cannot scale to larger problem sizes.

In recent work, Mirovia [141] builds upon both Rodinia and SHOC, designed to leverage the newer evolving architectures. While also representing a diverse set of application domains. This includes a particular focus on deep neural networks. Mirovia aims to characterise modern heterogeneous systems better. The benchmark suite falls short in the range of hardware it can target, being limited to Compute Unified Device Architecture (CUDA) enabled GPU only. Other work focuses on a single domain area; *QuTiBench* [142] is a multi-tiered framework for neural networks which introduces optimisation strategies such as quantisation, which is essential for specific accelerators such as FPGAs. Only the classification stage of the image-signal processing pipeline is tested within the framework; therefore, determining the full performance scope of a vision system is difficult. Reuther *et al.* [143] proposed a survey and benchmarked machine learning algorithms on commercial low-power ASICs and a CPU. However, such papers propose benchmarks and frameworks for specific algorithms or target singular architectures, focusing only on execution time performance. In addition, power consumption and memory transfer metrics are often missing in such benchmarks, which is a driving factor for embedded systems with limited energy.

3.6 Evaluated Image Processing Algorithms

This section outlines the implementation details on hardware (primarily FPGA) and the rationale behind the selection of various algorithms evaluated in the subsequent chapters. All algorithms were partly chosen due to their popular use in many image processing pipelines and varying complexity. The CPU and GPU image processing implementations of all algorithms are derived from OpenCV [23]. The FPGA implementations have been reviewed in literature to develop widely adopted methods (Algorithmically the same as CPU & GPU for fair comparison), which, at best, are close to the most optimal design.

- **RGB2Grey, Image Addition, Subtraction:** These operations serve as fundamental building blocks in image processing pipelines. Converting images to grayscale simplifies processing tasks, reduces memory usage, and enhances contrast for improved feature detection. Grayscale images directly represent luminance, making them useful for detecting subtle brightness variations. In addition, grey images have widespread compatibility, require less storage space, and can help reduce noise levels, resulting in clearer images for analysis. Image addition can adjust brightness or contrast by adding or subtracting constant values, while subtraction can isolate foreground objects through background subtraction or detect changes in scenes over time.

  The FPGA colour conversion module declares parameters for the dimensions of the image (m columns by n rows) and defines registers to store the input colour image in BMP format and the resulting grayscale image. Upon initialisation, the module reads the input colour image data from a BMP file into memory. Then, it iterates over each pixel in the image, extracting the red, green, and blue colour values and calculating their sum. This sum is divided by 3 to obtain the grayscale value, which is stored in a register. The image addition and subtraction module functions by performing pixel-wise arithmetic operations between corresponding pixels in two input images. The modules primarily use registers and combinational logic to perform pixel-wise addition/subtraction of two images.

- **Resizing:** Resizing operations are essential for various image processing applications, including scaling images for different display resolutions or aspect ratios. Many algorithms and CNN architectures require images to be resized to reduce computation complexity.

  The `imageResize` module facilitates the resizing of input image data by adjusting its width and depth based on specified scaling factors. The resizing operation involves updating counters for tracking column, row, and pixel positions within the image. As the module receives valid input data and a ready signal, it increments these counters accordingly. When the counters reach the desired scaling factors for width and depth, the module resets them to zero, effectively resizing the image. The output image data is synchronized with the clock signal and becomes valid only when both the column and row counters are zero, ensuring proper alignment with the resized image dimensions.
- **Erode, Dilate:** Morphological operations like erosion and dilation are commonly used for tasks such as image segmentation and feature extraction. The module `erodeDilate` operates on grayscale images represented by 8-bit pixel values. A 3x3 buffer array is utilised to store neighbouring pixel values for each input image pixel. Upon initialisation or reset, the buffers and output values are cleared. As new pixel values arrive with each clock cycle, the buffer contents are shifted accordingly.

  For erosion, the module computes the logical AND operation of all pixels in the 3x3 neighbourhood, resulting in the minimum pixel value. Conversely, dilation computes the logical OR operation, resulting in the maximum pixel value. These operations are performed on the buffer contents corresponding to the current pixel position. Finally, the resulting eroded or dilated pixel value is output along with its validity signal.
- **Box Filter, Gaussian Filter:** Filters like the box filter and Gaussian filter are fundamental for image smoothing and noise reduction.

  The Verilog code comprises modules designed for linear filters. The "line Buffer" module functions as a storage mechanism for incoming pixel data, organising it into line buffers to facilitate subsequent convolution operations. It efficiently manages the storage and shifting of pixel data as needed. The "imageControl" module orchestrates the flow of pixel data, determining when to initiate reading and writing operations based on the availability of data and signalling when to begin filtering. It tracks the total number of pixels processed to ensure accurate filtering across the image. The "conv" module is responsible for executing the convolution operation itself. It performs element-wise multiplication of pixel values with corresponding kernel coefficients and aggregates the results to generate the convolved pixel output.

- **Sobel Filter, Median Filter:** The Sobel filter is widely used for edge detection, while the median filter is effective for noise removal. The Sobel operator consists of two 3x3 kernels: one for detecting vertical edges and the other for horizontal edges. The module takes nine input pixel values corresponding to the 3x3 neighbourhood surrounding a central pixel. These input pixels are named according to their relative positions: Within the module, two instances of the `matrix_mul` module compute the gradient components along the x-axis (`gx`) and y-axis (`gy`) using the Sobel kernels. The result of each multiplication operation is squared to obtain the squared gradient magnitudes in the `gx_squared` and `gy_squared` wires. These squared magnitudes are then added together to compute the overall squared gradient magnitude in the `squared_sum` wire. The `sqrt` module calculates the square root of the squared gradient magnitude.

  The `median filter` module takes in pixel values (pixel_in) of image width and operates on a square filtering window of fixed size. Within each processing cycle, the module updates the window with new pixel values, sorts the window values to find the median value, and outputs the median as the filtered pixel value (pixel_out). The algorithm maintains internal registers to store the window and the histogram-based sorted window of pixel values. During reset, the internal registers are initialised, and processing flags are reset.
- **White Balance:** White balance adjustment is essential for achieving accurate colour representation in images captured under different lighting conditions.

  `WhiteBalance` module is designed to perform white balance adjustment on an input image by adjusting the red, green, and blue colour channels.

It takes in pixel values for the red, green, and blue channels of the image.

The module maintains buffers to accumulate the pixel values for each colour channel (`red_sum`, `green_sum`, `blue_sum`) and a register to count the number of pixels processed (`pixel_count`). These registers store the accumulated sums and count, respectively. On each clock cycle, the module accumulates the pixel values and updates the count. Then, it calculates the average pixel values for each colour channel by dividing the accumulated sum by the pixel count. Finally, it applies the white balance coefficients to adjust the colour channels accordingly.

- **Linearization: & Gamma Correction** Linearization and gamma correction are essential in image processing, ensuring consistent pixel representation and adjusting brightness and contrast for good visual images, making them ideal for bench-marking.

  The `Linearization` module is designed to linearize the pixel values of an input image, effectively scaling them to a range between 0 and 1. The module takes an 8-bit pixel value as input and outputs the corresponding linearized value. It employs simple arithmetic logic to divide the input pixel value by 255, the maximum value in an 8-bit system, thereby scaling it to the desired range. The linearization process occurs on each clock cycle, ensuring real-time processing of image data.

  `GammaCorrection` module applies gamma correction to an input pixel value using a real gamma value of 2.2. The module operates synchronously with the clock signal and includes a reset input for initialisation. Inside the module, it takes the input pixel value and applies a power function with an integer approximation of the gamma value. The function then converts the gamma-corrected integer value back to the [0, 255] range to ensure compatibility with pixel representations. The module continuously applies the gamma correction function to the input pixel value on each clock cycle. The output pixel value is updated accordingly. The input and output pixel values are stored in registers.
- **GEMM:** GEMM algorithm serves as a base building block for numerous image processing and deep learning operations due to their algorithmic efficiency in performing matrix multiplications. The algorithm is computed in many areas of ML, such as fully-connected layers, recurrent layers such as recurrent neural networks, Long short-term memory or Gated recurrent unit, and convolutional layers. Benchmarking GEMM

allows for assessing the computational performance and scalability of hardware architectures across a wide range of image processing and deep learning algorithms. The optimised FPGA GEMM implementations used for evaluation are provided by Xilinx [144] library.

- **FFT (DFT):** Fast Fourier Transform (FFT) are found in various operations in imaging algorithms by enabling efficient frequency domain analysis and manipulation of images for tasks such as filtering, convolution, registration, texture analysis, edge detection, compression, and super-resolution imaging. Xilinx provided FFT IP-block [145] is used in the benchmarking. The module implements the Cooley-Tukey FFT algorithm for computing both forward and inverse DFTs of sample sizes that are powers of 2.
- **STREAM:** Memory bandwidth and latency are important metrics because they directly impact the speed and efficiency of data transfer between the processor and memory. Therefore, when benchmarking image processing algorithms, measuring memory bandwidth and latency provides insights into how efficiently the algorithms utilise memory resources, helping assess their overall performance and scalability. The implementation used for FPGAs are from HPCC _FPGA benchmarking suite [146].
- **Demosaicing:** Demosaicing algorithms are essential for reconstructing colour images from Bayer-pattern sensor data which are common filter used in vision systems. The algorithm is a good choice to benchmark since most vision systems use Bayer filter. Xilinx provided de-mosaicing IP-block [147] is used in the benchmarking. The demosaicing algorithm used is a bilinear interpolation, where missing colour values in a Bayer-filtered image are estimated by averaging neighbouring pixel values.
- **SIFT:** Scale-Invariant Feature Transform (SIFT) is a widely used technique for feature extraction in image processing applications. The FPGA implementation is discussed in Section 5.2.1.
- **CNN (Classification):** Convolutional Neural Networks (CNNs) have become significantly popular methods for image classification, object detection/segmentation tasks. Xilinx Deep Learning Processor Unit (DPU) is used to implement both optimised ResNet-18 and MobilnetV2 architectures [148].

3.7 Conclusion

This chapter delves into the discussion and evaluation of diverse heterogeneous architecture implementations, tools, and optimisation strategies. The literature review emphasises that specific processing architectures, such as GPUs, FPGAs, and VPUs, exhibit greater efficiency in executing imaging algorithms than CPUs, owing to their architectural properties (*e.g.*, SIMD, DSP Slice) and depending on algorithmic features like parallelisation and data dependencies. Moreover, these architectures to exploit their advantages leads to further performance gains. Various optimisations have also been shown to enhance the efficiency of image processing algorithms, encompassing techniques such as algorithm replacement, hardware (memory management) and pipelining schemes. However, the knowledge gap is evident in the lack of a systematic approach or set of strategies for partitioning imaging algorithms onto the appropriate hardware accelerators in a heterogeneous platform, nor a comprehensive rationale for such decisions. In addition, there are no end-to-end implementations of CNN and feature extraction algorithms utilising CPU-GPU-FPGA architectures. The absence of image-domain-guided optimisations and understanding of trade-offs further complicates achieving optimal performance and efficiency. Hence, within the scope of this thesis, a robust benchmarking framework that guides algorithm partitioning along with domain-specific optimisation strategies on heterogeneous platforms is proposed.

4 HArBoUR: Heterogeneous Architecture Benchmarking on Unified Resources

This chapter presents a description of *HArBoUR*, a heterogeneous imaging framework used for guidance and implementation of various image processing algorithms presented in later chapters. In addition, the chapter contains benchmarks and evaluations of micro/macro image processing algorithms commonly found in imaging pipelines of vision systems. The algorithm suitability on a specific accelerator is determined from the analysis of the results.

4.1 Introduction

The advances in multi-core processors and accelerators have enabled real-time embedded imaging algorithms to become ubiquitous within many vision application areas such as advanced driver assist systems (ADAS) [149], surveillance [150] and satellites [151]. The growing demand for image processing algorithms on systems with resource and energy constraints requires architectures that perform tasks efficiently. These hardware accelerators come with various architectures ranging from CPUs, GPUs, and FPGAs. Traditionally, embedded imaging designs often involve implementing image processing algorithms on homogeneous architectures, which come with hardware limitations. However, recent developments introduce heterogeneous architectures that combine multiple specialised accelerators on a singular interconnected chip [152]. These novel architectures provide optimum design opportunities for embedded imaging development. However, current developments in targeting heterogeneous platforms are still primitive and require careful consideration of various development languages, tool-sets and performance profiles to fulfil energy and runtime constraints of applications. Furthermore, certain accelerators have pre-written optimised vision libraries, *e.g.*, OpenCV/CUDA (for CPU/ GPU) and xfOpenCV (for FPGAs). However, the design and development of image processing algorithms in a heterogeneous environment is still an arduous task. It requires in-depth knowledge of multiple hardware accelerators, and each differing in performance due to their underlying architectures. Additionally, FPGA designs require knowledge of hardware descriptor languages (HDLs), which have higher learning difficulty despite the existence of recent high-level synthesis tools or domain-specific languages that abstract away from the underlying hardware. This is due to a lack of understanding and the existence of benchmarks for image processing algorithms on different hardware. Therefore, partitioning complex image processing algorithms onto each accelerator remains a difficult task for application designers.

Benchmarking has played an integral part within the computing domain for decades. Since the beginning of computing systems, there has been a persistent need to evaluate and compare the performance of hardware components. Initially, with the early day mainframe computers to the modern era of microprocessors, GPUs, and custom ASICs, benchmarking has provided a standardised measure to gauge the efficiency, speed, and capabilities of hardware devices. Over the years, benchmarking tools and suites have played a pivotal role in driving technological advancements, guiding design decisions, and ensuring that hardware meets the ever-evolving demands of software applications.

To address the emerging demand for high performance vision hardware, there is a need for a suitable benchmarking framework that dissects the imaging/vision algorithms in a disciplined way and benchmarks their performances (both energy and execution time) on all available target hardware (*e.g.*, CPU, GPU and FPGA). To address such a gap, this chapter proposes a new framework providing a systematic way of implementing imaging designs on specialised platforms and perform benchmarks on representative vision algorithms while assessing execution time, memory latency and energy consumption. System designers will use the proposed framework to identify appropriate hardware for the target application and unlock the potential of a true heterogeneous system. The main contributions of this chapter are:

- We propose a framework that studies features of image processing algorithms to identify characteristics. These characteristics help partition complex algorithms into the most optimal target accelerators within heterogeneous architectures.

- The approach adopts a systemic and multi-layer strategy that offers trade-offs between runtime, energy and accuracy within the imaging sub-domains *e.g.*, *CNNs* and *feature extraction.* Specifically, *HArBoUR* enables support in constructing end to end vision systems while providing expected results and guidance.
- Domain knowledge-guided hardware evaluation of computational tasks allows imaging algorithms to be mapped onto hardware platforms more efficiently than a heuristic based approach.
- We benchmark representative image processing algorithms on various hardware platforms and measure their *energy consumption* and *execution time* performance. The results are evaluated to gain insight into why certain processing accelerators perform better or worse based on the characteristics of the imaging algorithm.

4.2 Benchmarking Framework for Image Processing on Hardware

For efficient implementation on heterogeneous platforms, the algorithm design will require partitioning any image processing algorithm according to its suitability for individual components in the pipeline on the target hardware. Therefore, a standard framework is proposed containing a pool of image processing algorithms and their characteristics in accordance with their hardware suitability. We selected a range of low, medium and high-level algorithms from the image processing hierarchical classification domain, providing wider coverage of commonly used operations performed within a vision application pipeline.

Various algorithmic and hardware characteristics that would impact the performance are identified in the heterogeneous benchmark framework, shown in Fig. 4.1. The framework diagram offers a clear overview of the image processing landscape. It maps out the relationship between hardware properties, optimisations, algorithms, metrics and memory access patterns. This concise

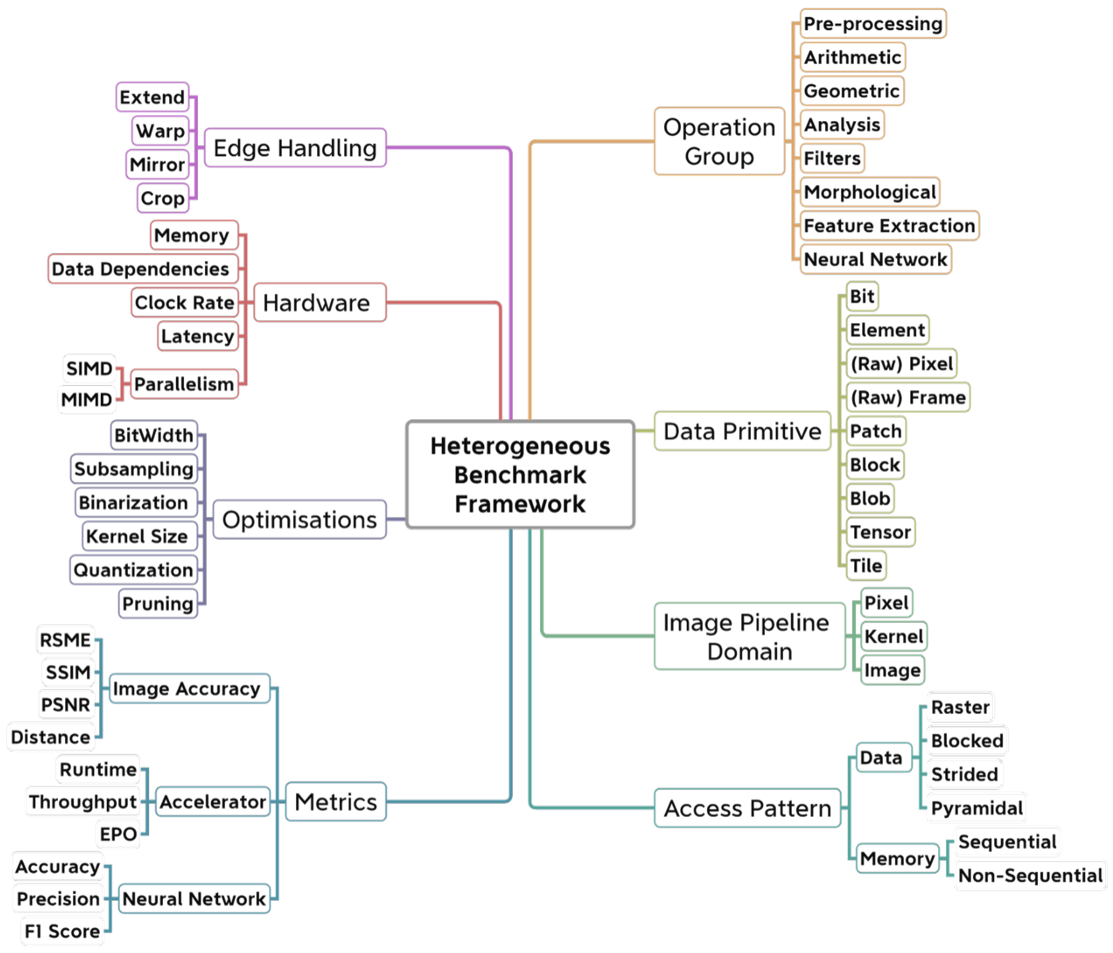


Figure 4.1: Benchmarking Framework for Image Processing Algorithms, Highlighting Key Metrics and Properties

visual guide aids designers by offering insights into potential bottlenecks, suggesting areas for innovation, and guiding the fine-tuning of algorithms on heterogeneous platforms to achieve peak performance in imaging tasks.

4.2.1 Processing Pipeline & Operation Types:

Image processing algorithms are organised into three primary domains: Pixel, Kernel, and Image. The *Pixel domain* focuses on operations that manipulate or query individual pixel values. The *Kernel domain* encompasses algorithms that utilise a small matrix (the kernel) to modify an image. Lastly, the *Image domain* deals with operations that consider the image as a whole, where global features and patterns are essential for labelling.

4.2.2 Operator Group

This label specifies the name of the algorithm. Algorithms may perform a particular operation (*e.g.*, Image Arithmetic), but depending on the stage within the image computation pipeline, the data type of the pixel is defined differently and may contain additional values for calibration, such as a pedestal. It is necessary to be explicit when defining the algorithm within the framework. These image processing algorithms can further be categorised into groups (Operation Group) depending on the type of operation it performs.

Image Arithmetic & Pre-Processing: Image arithmetic and pre-processing are foundational steps in image processing pipelines, preparing images for subsequent analysis. These algorithms predominantly execute primitive operations to transform an input format into a desired one. They typically operate on individual pixels using localised data, which minimises task dependencies. While algorithms like multiplication, accumulation, squaring, magnitude determination, and weighting are arithmetically simple, most architectures can compute them with ease. Given their low initialisation and latency requirements, architectures such as CPUs and FPGAs are particularly well-suited for these tasks.

Geometric Transformation & Image Analysis: Geometric transformations refer to operations that modify the spatial arrangement of pixels in an image. These transformations can be applied for various purposes, such as image registration, scaling, and augmentation. Typically, these algorithms involve convolution and interpolation operations, which can be linear or non-linear. Additionally, the choice of interpolation method, whether nearest-neighbour, bilinear, or bicubic, can significantly impact both the quality of the transformed image and the computational complexity of the operation. Several operations are sequentially bound; forward mapping directly calculates new pixel locations but can leave gaps in the output. Its counterpart, backward mapping, determines source contributors for each output pixel and often requires interpolation, which can be sequential, especially with higher-order methods. Warping with a displacement map, which dictates pixel movement, can also be sequential if complex algorithms determine displacements. Resampling, essential post-mapping, can become sequential with intricate interpolation. Cumulative transformations, where multiple operations are applied in sequence, and error corrections post-transformation, further introduce sequential elements. For operations with inherent sequentiality, such as certain geometric transformations, traditional CPUs are often the most suitable due to their optimised instruction sets for sequential tasks and complex branching. However, for tasks within geometric transformations that can be parallelised, GPUs, FPGAs, and TPUs can offer significant speedups

Image analysis algorithms label and understand various statistical data about a pixel. These algorithms have many irregular memory access patterns (mean, mode, min/max) and branching conditions that negatively impact the performance of processing accelerators.

Image Filters & Morphology: Image filter algorithms modify particular spatial frequencies. The image is filtered either in the frequency or in the spatial domain. Image filters are divided into two categories: linear and non-linear. Linear image filters perform the convolution of an image using a pre-computed kernel for efficiency. The data-independent multiply and accumulate operations coupled with sequential data access of linear filters map well onto GPUs and FPGAs. Contrarily, non-linear filters have varied memory access patterns and have higher arithmetic intensity. Additionally, certain algorithms with branching makes it arduous to implement efficiently on a GPU and FPGA, which can exploit parallelism in these operations. Non-linear filters consist of operations where the output pixel value is determined based on some non-linear function of the input pixel values in its neighbourhood. Examples of algorithms include median, adaptive and bilateral filtering.

In morphological operations, an image is processed with a structuring element, which is a small binary or grayscale mask. The structuring element is moved over the entire image, and at each position, a computation is performed based on the values of the image pixels that overlap with the mask. Architectures with cache hierarchies, prevalent in modern CPUs and GPUs, can exploit this pattern for enhanced cache locality, especially when the structuring element's dimensions are compatible with cache sizes. In row-major order, horizontal traversal optimises memory access, while vertical traversal can introduce inefficiencies due to memory strides. Basic operations like dilation and erosion have regular patterns, but advanced morphological algorithms can cause irregular accesses. These irregularities, often from adaptive structuring or conditional operations, challenge GPU performance through warp divergence. Additionally, these algorithms may require repeated pixel reads, especially with overlapping structuring elements, and in-place operations risk read-write conflicts, which need careful management to prevent race conditions.

Feature Extraction: Feature extraction algorithms, such as SIFT [153], SURF [154], and Oriented FAST and Rotated BRIEF (ORB) [155], are designed to identify and describe local features in an image. These features are often points or small image patches that are distinct and can be reliably and robustly detected in various scales of the same scene. The pixels extracted from an image are not stored adjacently in memory and require expensive computational reads from non-adjacent memory addresses that will impact the performance of all processing architectures. Algorithms such as `ORB` examine a circle of pixels around each candidate pixel. While each keypoint detection is independent and can be parallelised, the algorithm involves conditional checks, which can lead to divergent execution paths.

Regarding hardware suitability, CPUs have a layered hierarchy of caches, encompassing L1, L2, and L3. This architecture is good at offsetting the performance implications of non-sequential memory accesses. Therefore, algorithms with random access patterns, akin to the keypoint detection seen in SIFT or ORB, can leverage this hierarchical structure. Nevertheless, despite their proficiency, CPUs will lag in efficiency when confronted with parallel operations stages within the algorithms. GPUs and FPGAs, on the other hand, are geared towards the parallel processing tasks which are found during the convolution and prefix sum stage. GPUs favour coalesced memory access, where adjacent threads reading consecutive memory locations achieve faster, more efficient data retrieval. Consequently, the random accesses, if frequent, can lead to performance dips due to the resulting uncoalesced memory transactions. However, TPUs are purpose-built for tensor operations, forming the basis of deep learning methodologies. Traditional feature extraction may not have direct advantages from TPUs unless they're integrated into a wider deep learning framework.

Data Primitive:

This characteristic specifies the type of data unit or collection of pixels an algorithm operates on, encompassing attributes like type, size, and representation. Depending on the level of the pipeline, a pixel may refer to an individual or collection of bits of type `word`, `uint8` or `uint12`. The following are the various data primitives found within image processing:

- **Bit:** Refers to individual bits in the binary representation of the image data, encompassing pixel values, metadata, file headers, and more, contingent on the file format.
- **Element:** Denotes a discrete scalar value, formed of bits, quantifying pixel attributes, like colour intensity or luminance.
- **Pixel:** Is a set of elements symbolising a point in an image, with formats such as RGB, YUV, or Bayer encoding.
- **Frame:** Is a structured pixel collection visualised in 2D or 3D, where the resolution indicates the total pixel count.
- **Patch:** Is a small frame subsection targeted by algorithms like blurring; a 3×3, 5×5 Kernel is a matrix sliding over the image, its values multiplied with corresponding patch pixels, and the results summed for a transformed image.
- **Block:** Pertains to a contiguous pixel group, typically of fixed dimensions like 8×8 or 16×16, serving as the atomic unit for algorithms, such as JPEG compression.
- **Blob:** Is an image region defined by properties like brightness, distinct from surrounding areas.
- **Tensor:** Is a multi-dimensional structure encapsulating complex data; a 2D tensor represents greyscale images, while a 3D tensor handles colour images, considering width, height, and colour channels. In deep learning, tensors provide a concise mathematical representation of the problem.
- **Tile:** Is a unique image portion, often rectangular, acting as a processing or representation unit. Unlike overlapping patches, tiles traditionally denote non-overlapping image regions, ensuring each pixel's unique tile membership. Tiling affects memory access patterns and storage; for instance, accessing an image row might require data fetching from multiple tiles if stored tile-by-tile.

Understanding data type precision is essential as it impacts accuracy and hardware suitability. Algorithms requiring higher precision will use floating-point datatype, which is better supported by the FP hard blocks within CPU or GPU architectures, while FPGA solutions suit integer-based calculations. Data primitive choice impacts memory, bandwidth, and computational efficiency. Successful imaging pipeline design hinges on aligning data primitive attributes with processing unit capabilities, thereby optimising image processing workflow in terms of accuracy, efficiency, and hardware compatibility. An optimal pipeline balances various primitives for each operation while maintaining high accuracy.

Access Patterns:

Image pixel locality in memory refers to how the spatial arrangement of pixel data impacts the efficiency of image processing tasks. When pixels are stored in memory, their arrangement influences data access efficiency. Locality in memory means nearby pixels in the image are stored adjacently, aiding faster data access. Various access patterns are discussed below:

- **Sequential Access:** Involves accessing pixels one after the other, typically in row-major or column-major order.
- **Random Access:** Retrieves pixels in a non-sequential manner based on algorithmic requirements.
- **Neighbourhood Access:** Relates to the pixels surrounding a specific pixel, often used in operations with kernels or local filters.
- **Block Access:** Deals with a contiguous region of pixels, common in block-based algorithms or compression methods.
- **Strided Access:** Refers to the method where pixels are accessed at regular intervals or ’strides’.
- **Pyramidal Access:** Is associated with multi-resolution representations, such as image pyramids.
- **Scanline Access:** Involves accessing entire rows or columns of pixels simultaneously, often seen in raster operations.
- **Tile-based Access:** Focuses on square or rectangular pixel tiles, crucial in tiled rendering or processing.

Efficient memory access is crucial in real-time imaging due to computer memory hierarchies. Pixels stored close in memory can be loaded into faster cache levels, reducing time spent waiting for data from slower memory. In image processing, operations like convolution require accessing nearby pixels. Locally stored pixels minimise cache misses and improve computational performance. Techniques like tiling, memory padding, and cache-aware algorithms enhance processing efficiency. By aligning pixel data in memory with spatial arrangement and optimising memory hierarchy, image processing algorithms can leverage hardware effectively for faster performance.

Hardware Characteristics & Edge Handling:

It is necessary to understand the capabilities and limitations of each hardware architecture to obtain optimum accuracy, area and speed of imaging algorithms. These depend on particular hardware properties such as bit-width, clock rate, memory location, data type and data dependency. Image processing algorithms perform poorly on memory systems due to memory hierarchy latency bottlenecks and high cache miss rates. Additionally, architectures containing many processing units require careful division of tasks to avoid load imbalance.

Edge handling involves strategies when operations, such as filtering, convolution and morphological transformations, are applied near the boundaries of an image. Since these algorithms often require neighbouring pixel values, challenges arise at the image edges where full neighbourhoods are unavailable. Common edge handling techniques include zero-padding (extending the image with zeros), replication (duplicating the edge values), reflection (mirroring the adjacent pixels), and circular (considering the image as a continuous loop). Different edge handling techniques can lead to non-uniform or non-sequential memory access patterns. In addition, the choice of method can influence the resultant image, especially in terms of artefacts or discontinuities at the boundaries. Proper edge handling is crucial to ensure consistent and artefact-free processing results.

Optimisations & metrics

The optimisation attribute captures accelerator agnostic optimisations for image processing algorithms. The attribute provides a repository of optimisations that helps designers pick and tune algorithms to find the optimal combination within their design space for specific use cases. These optimisations focus on the inherent properties of the algorithm, such as reducing computational complexity, improving data access patterns, or refining logical structures. The primary reasoning is their broad applicability and ensures a performance baseline. The improvements realised are typically consistent across various hardware architectures, from CPUs and GPUs to FPGAs and ASICs.

The metric attribute standardises performance indicators from benchmarking literature, facilitating an understanding of trade-offs. Metrics such as runtime assess algorithmic execution speed, throughput quantifies data processed over time, and energy per operation provides insights into energy efficiency. For neural networks, accuracy measures how often the model is correct, precision looks at how many of the positive identifications were actually right, and the F1 score balances precision against recall.

4.2.3 Heterogeneous Benchmarking Development Flow

Heterogeneous hardware addresses the growing complexity of modern workloads by combining different processing units, such as CPUs, GPUs, and other accelerators, within a single system. This approach optimises performance and energy efficiency for specific tasks, leveraging the strengths of each component. However, targeting these platforms remains a significant challenge to address.

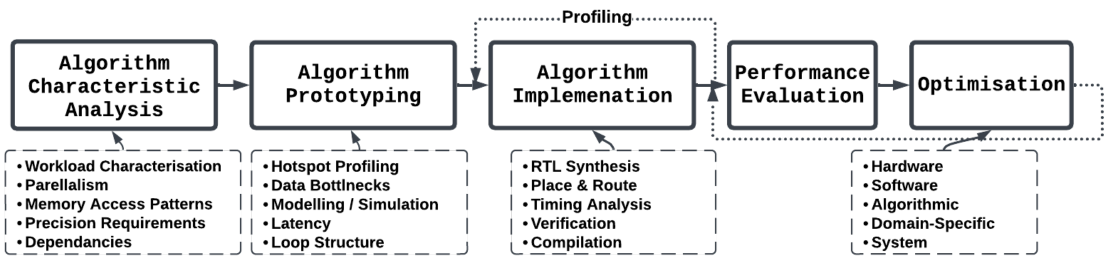


Figure 4.2: Framework Pipeline for Heterogeneous Image Processing

Therefore, To effectively exploit the power of heterogeneous architectures, a robust development flow is crucial. Such a workflow not only aids in pinpointing the optimal architecture but also streamlines the entire development process. As depicted in Fig. 4.2, a comprehensive heterogeneous development pipeline describes a systematic and standardised approach for implementation. This structured flow ensures that developers can seamlessly integrate various computational resources, leverage specialised hardware capabilities, and achieve optimal performance across multiple platforms.

Characteristic Analysis: The first stage involves identifying certain properties of an algorithm discussed in 4.2 and building a model of data, which helps find suitable architectures. Starting with workload characterisation provides a comprehensive overview of the algorithm's computational and data demands, potentially giving insight into areas where algorithms can be partitioned. Specific algorithms require higher precision, which can impact the choice of the target architecture. However, trading off accuracy for speed is a potential opportunity. Memory access patterns significantly affect cache utilisation and memory latency. It's important to recognise an algorithm's parallelism potential while being aware of data dependencies. Moreover, optimising memory access patterns to align with the cache hierarchy can reduce latency and enhance memory throughput. Furthermore, adapting algorithms to harness specific hardware features, such as GPU threads, FPGA pipelines, or TPU matrix multiply units.

Prototyping: This stage involves prototyping designs to identify the initial performance of an architecture to establish a baseline benchmark. Profiling code is done to identify performance bottlenecks, which offers valuable insights into areas suitable for performance enhancement across various architectures. Profiling aims to uncover the 'hotspots,' sections of code where execution time and resource consumption are disproportionately high. In addition, identifying task, data parallelism or pipelining opportunities which benefit certain accelerators. It is essential to pinpoint potential latency issues, both in I/O operations and within memory, which can arise from slow data transfers or inefficient data handling. Identifying loop structures is important for performance; branching within loops in parallel or pipelined processors can negatively affect performance. This is because branching can lead to divergence, where different execution paths are taken simultaneously, causing processing cores to remain idle. Loop unrolling is a common optimisation technique that can mitigate the effects of branching by increasing the number of operations in each loop iteration, reducing the loop's overhead. However, if a loop cannot be unrolled or tiled, it may be more efficient to execute the algorithm on a higher clocked sequential core, which can handle branching.

Implementation: After selecting the appropriate architecture for an algorithm, the next step involves its implementation. Depending on the chosen accelerator, this process may entail simulating and synthesising designs and conducting thorough verification. Finally, timing analysis involves evaluating the propagation delays of signals through the circuit's logic gates and interconnects to ensure they meet the constraints set by the clock cycle time. This helps in detecting and addressing potential timing violations. General-purpose architectures will use compilation methods to high-level code into machine-level instructions tailored for the instruction driven accelerators.

Evaluation & Optimisation: Post-implementation, performance tuning and optimisation techniques are applied to ensure the algorithm runs efficiently, maximising the hardware's potential. Optimisations can be grouped into four categories: hardware, software, domain-specific and algorithmic. Hardware optimisations involve fine-tuning specific hardware configurations or datatypes such as quantisation or bit-width adjustments. Software optimisations are usually applied at the code level either manually or automatically by compilers. These techniques can include inlining, dead code elimination, and vectorization. Domain-specific is tailored for specific application areas such as downsampling or separable filters. Lastly, algorithmic focus on mathematical refinement to reduce complexity and use more efficient data structures. It's also necessary to validate the implementation against reference data-sets to ensure functional correctness. Throughout this process, continuous profiling helps identify bottlenecks and areas for further optimisation.

4.3 Benchmarking Methodology

This section introduces two benchmarking strategies, micro and macro, each offering distinct approaches to evaluate accelerator performance. While micro-benchmarking focuses on assessing individual algorithms or pipelines, macro-

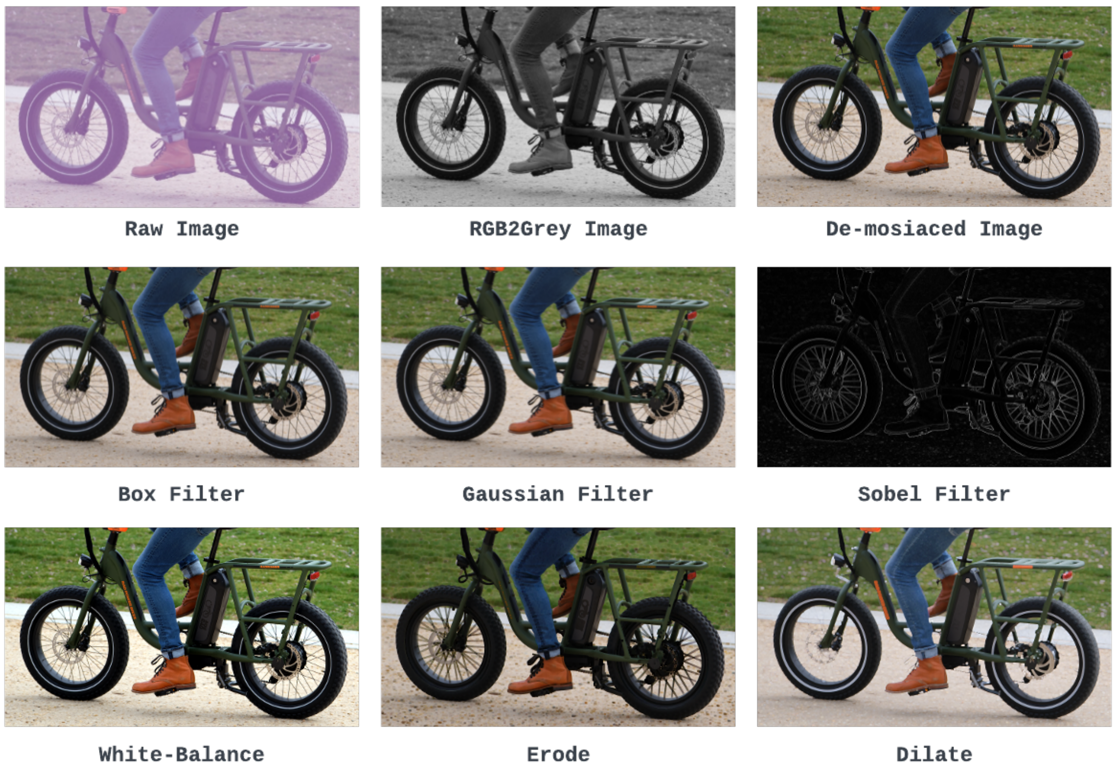


Figure 4.3: Low to High Complexity Image Processing Algorithms in the ISP Pipeline

benchmarking provides an overall view by analysing fundamental operations found in many algorithms within the ISP pipeline.

4.3.1 Micro Benchmarking Algorithms

Image processing pipelines contain many operations varying in complexity. For a comprehensive set of results, many popular individual ISP algorithms are chosen to be benchmarked, listed in Table 4.1 and visually shown in Fig. 4.3. The algorithms, *Addition*, *Subtraction*, *Erode*, *Dilate*, *Box Filter*, *Gamma Correction*, *Linearization*, and *Demosaicing* are chosen because they represent foundational operations in image processing. These algorithms exhibit diverse memory access patterns, computational intensities, and parallelism levels. Their inclusion ensures a comprehensive evaluation of an architecture's capability to handle point-wise and neighbourhood operations.

Table 4.1: List of Implemented Image Processing & Benchmarking Algorithms

Algorithm	Complexity	Operator Group	Configuration	Description
RGB2Grey	Low	Point Operations	R:0.299 G:0.587 B:0.114	Convert RGB image to greyscale
Resizing	Low	Geometric Transformation	2.5x Upsampling	Change image dimensions by interpolation
Image Addition	Low	Point Operations	~	Add two images pixel-wise
Image Subtraction	Low	Point Operations	~	Subtract one image from another pixel-wise
Gamma Correction	Low	Linear Filter	$\gamma = 2.2$	Adjust pixel intensities using gamma values
Linearization	Medium	Point Operations	~	Correct non-linear sensor response
Erode	Medium	Morphological Operations	5x5	Erode image regions
Dilate	Medium	Morphological Operations	5x5	Dilate image regions
Box Filter	Medium	Linear Filter	5x5	Apply simple box averaging
Gaussian Filter	Medium	Linear Filter	5x5	Apply Gaussian blurring
Sobel Filter	Medium	Non-linear Filter	7x7	Detect edges using Sobel operator
Median	Medium	Non-linear Filter	5x5	Replaces each pixel value with the median value
White Balance	Medium	Linear Filter	Gray World	Adjust color balance in images
GEMM	Medium	Fundamental	N=4096	General Matrix Multiply (GEMM) operation
FFT (DFT)	Medium	Fundamental	radix-2 / R2C Zero Padding	Compute Fast Fourier Transform of signals
STREAM	Medium	Fundamental	N=1 $\times 10^7$	Evaluate memory bandwidth and latency
Demosaicing	Medium	Non-linear Filter	Bilinear Interpolation	Convert Bayer-pattern image to RGB
SIFT	High	Feature Extraction	2 Octave 4 Scale	Scale-Invariant Feature Transform for image matching
CNN (Classification)	High	Deep Learning	ResNet18 & MobileNetV2	Feature Extraction and Classification using Neural Networks

Complete Imaging Pipelines

Vision applications usually do not consist of one algorithm but contain many pipelined together to form a complete system. In developing the proposed framework, three exemplars were selected: 1) *Edge Detection*, 2) *Feature Extraction (SIFT)* [153] and 3) Classification, representing low to high-level complexity shown in Fig. 4.4. They are partitioned into nine sub-algorithms, namely, *RGB2Grey*, *Gaussian Filter*, *Sobel Filter*, *Gaussian Pyramid*, *Extrema Detection*, *Orientation Assignment*, *Descriptor Generation*, *Resize*, and *CNN*. The sub-algorithms are common building blocks of many other image processing algorithms with varying complexity and therefore are good candidates for benchmarking. For example, *Gaussian Pyramid* is useful for analysis across different spatial scales and *Extrema Detection* operations are often used in corner detection and image blending algorithms.

The first pipeline is the edge detection pipeline, designed to identify and emphasise edges within images. It starts with the conversion of RGB colour information into greyscale, a method aimed at reducing redundant processing. Following this, Gaussian filtering is used to achieve image smoothing and noise reduction, essential for accurate edge detection. The Sobel edge detec-

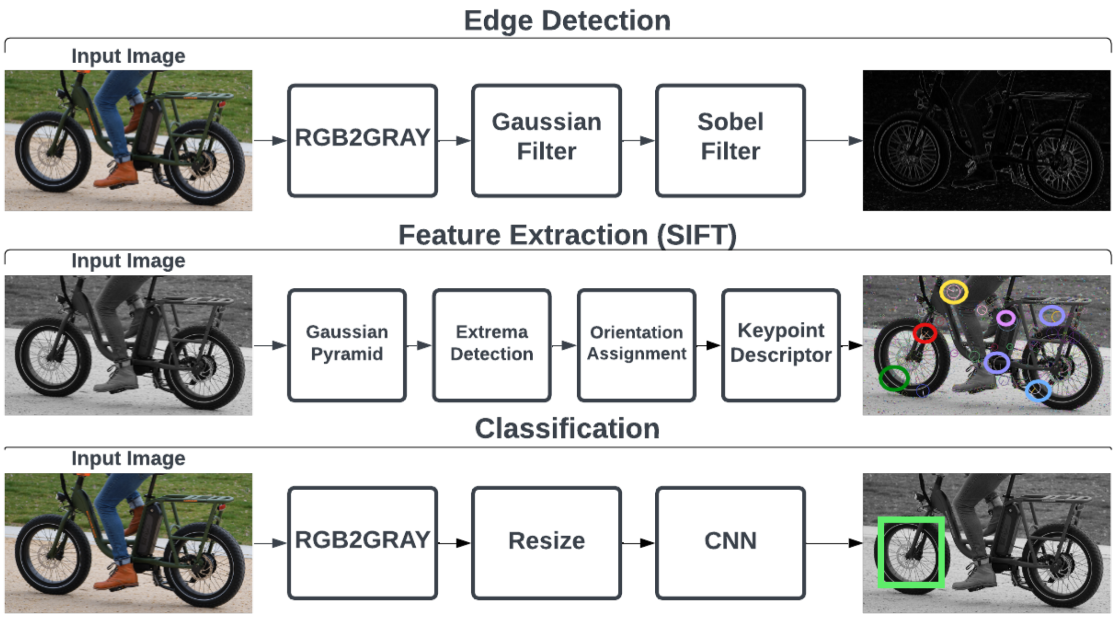


Figure 4.4: Exemplar Image Pipelines Benchmarked on each Architecture.

tion algorithm, serving as the final stage within this pipeline, detects edges by highlighting abrupt changes in pixel intensities.

The feature extraction pipeline captures important features present within images. The first step in the pipeline involves the construction of a Gaussian pyramid, accomplished by generating multiple versions of the input image through Gaussian filtering and downsampling. This pyramid plays a critical role in identifying features across varying scales. Subsequently, extremum detection is executed, facilitating the identification of keypoints or points of interest within the image. These keypoints serve as reference points for subsequent analysis and interpretation. Grayscale images is used as a input for their simplicity in processing and their robustness to changes in illumination.

The classification pipeline can be found in many deep learning based applications, including vision. It begins with the conversion of RGB to greyscale and subsequently, the image resizing operation is performed to standardise dimensions, ensuring compatibility with the CNN architecture. The final stage involves the application of the CNN algorithm, a state-of-the-art deep learning technique known for its proficiency in tasks related to image classification, object detection, and segmentation.

4.3.2 Macro Benchmarking Algorithms

Many image processing algorithms share foundational operations, which can be used to benchmark accelerators to gain better insight. This section reviews a range of algorithms and their properties that make it an ideal case study for evaluation.

Sustainable Memory Bandwidth in High Performance Computers

Stream memory benchmark [156] (STREAM) is a widely used performance evaluation tool for measuring memory bandwidth in computer systems. It assesses the speed at which a system can read and write data to memory. The benchmark primarily focuses on four memory access patterns: Copy, Scale, Add, and Triad. In the Copy pattern, data is read from one memory location and written to another. The Scale pattern involves reading data, scaling it by a constant, and writing it to a different memory location. The Add pattern reads two arrays, adds corresponding elements, and writes the result to a third array. Finally, the Triad pattern combines scaling and addition operations. The benchmark generates a set of memory performance metrics, including the memory bandwidth, calculated as the amount of data transferred per unit time, typically in gigabytes per second (GB/s). The memory bandwidth (B) can be calculated using the following equation:

$$B = \frac{N \times S}{t} \tag{4.1}$$

where N is the number of data elements accessed in the memory, S is the size of each data element in bytes, and t is the time taken to complete the memory operation in seconds. This equation gives us the memory bandwidth in bytes per second. Image processing is very memory-intensive due to the large amounts of data associated with higher resolution images and the need for frequent data access during processing. Each pixel requires multiple bytes of storage, and when processing, these pixels often need to be accessed multiple times, especially in operations like convolution, filtering, or transformations. The STREAM benchmark becomes invaluable in this context, as it provides insights into how efficiently an architecture can handle the memory demands of image processing tasks.

Fast Fourier Transform

Fast Fourier Transform (FFT) is an algorithm that finds extensive applications in signal processing, image analysis, and various fields where frequency domain analysis is essential. In image processing, FFT is particularly valuable for transforming an image from its spatial domain representation to its frequency domain representation. This transformation enables the identification of various frequency components present in an image, offering insights into patterns, textures, and other intricate details that may not be as evident in the spatial domain.

The FFT algorithm computes the Discrete Fourier Transform (DFT) of a signal in an efficient manner, reducing the computational complexity from $O(n^2)$ to $O(n\log(n))$, where n is the number of data points in the signal. In the case of a 2D image, the FFT operation involves applying the DFT algorithm separately to both the rows and columns of the image's pixel values. The 2D FFT of an image $I(x, y)$ is expressed as:

$$F(u,v) = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} I(x,y) e^{-j2\pi\left(\frac{ux}{N}+\frac{vy}{M}\right)} \tag{4.2}$$

Here, $F(u, v)$ represents the frequency components in the transformed image, N is the number of pixels along the x-axis, M is the number of pixels along the y-axis, and (u, v) are the spatial frequency coordinates in the frequency domain.

The resulting 2D FFT representation provides valuable information about the image's frequency content. Low-frequency components (corresponding to slow changes in intensity) are typically located near the centre of the frequency domain representation. High-frequency components (corresponding to rapid changes in intensity, edges, and textures) tend to be found towards the corners. By analysing this transformed image, practitioners can perform tasks like filtering, denoising, compression, and other frequency-based manipulations to enhance or extract specific features from the original image.

Convolution (Matrix Multiply)

The two most common methods for convolution are general matrix multiply (GEMM) and direct convolution. GEMM-based convolution relies on the *im2col* algorithm [157], which results in a large memory footprint and reduced performance. Alternatively, direct convolution has a lower memory footprint but the performance is reduced due to the irregular memory access patterns. GEMM can be can be expressed through Eq. (4.3):

$$C_{ij} = \sum_{k=1}^{K} A_{ik} \cdot B_{kj} \tag{4.3}$$

Here, C_{ij} refers to the element positioned at row i and column j within the resultant matrix C, A_{ik} signifies the element situated at row i and column k in matrix A, and B_{kj} represents the element located at row k and column j in matrix B. The summation spans across the index k from 1 to K, where K corresponds to the number of columns in matrix A (which should equivalently match the number of rows in matrix BB for accurate multiplication). This equation forms the core operation underlying the GEMM benchmark, which orchestrates the calculation of the matrix product between A and B to yield matrix C.

GEMM is found in image processing tasks that require matrix operations. For example, convolutional neural networks (CNNs) extensively use matrix multiplications for convolutional layers, and GEMM's efficiency and parallelism make it highly useful for accelerating these operations. Additionally, transformations, filters, and image manipulations often involve matrix operations, and therefore GEMM's optimised implementations can significantly enhance the performance of image processing algorithms.

4.3.3 Performance Metrics

This section focuses on evaluating implemented image processing algorithms using two key metrics: execution time and power consumption. The analysis provides insights into the strengths and limitations of the image processing algorithms on a heterogeneous architecture.

Execution Time

In evaluating the performance of image processing algorithms, precise time measurements are essential to capture the subtle differences across hardware platforms. For the CPU, the C++ standard library's $high\_resolution\_clock$ is employed, offering a fine-grained temporal resolution suitable for capturing in microseconds. This method involves marking the start and end times surrounding the algorithm's execution and computing the difference to determine the elapsed time. On the GPU side, CUDA events were utilised to measure the time taken for the algorithms to run. CUDA events are specifically designed to capture start and end times in a GPU's environment, ensuring accurate timing measurements that account for the asynchronous nature of GPU operations. Irrelevant processes are stopped within the operating system to prevent interference during the timing measurement. For the FPGA evaluations, the behavioural simulation timing feature of the Vivado software is used. This tool provides a detailed timing analysis, simulating how the algorithm would perform on the FPGA, thereby offering insights into its expected real-world performance. The implementations are written in C/C++/Verilog and uses the time function built into Linux and system performance monitor to measure the runtime of CPU/GPU/FPGA.

Power Consumption

Accurate power estimation is always challenging for software tools. However, systematic steps are taken to minimise assumptions for better accuracy. The approach used to measure the power consumption for the CPU and GPU is obtained by using *HWMonitor* software. The power is initially measured to determine the average base operating power. Then, each algorithm is executed multiple times, and the power is measured during algorithm runtime. The FPGA on-chip power consumption is measured using the reports from the power analyser feature integrated into Vivado Design Suite. The heterogeneous implementation power consumption is calculated using both *HW-Monitor* and Xilinx's System Monitor IP. Static power consumption, refers to the constant energy usage of a device when it's idle, whereas dynamic power consumption varies based on the workload or activity levels, resulting in fluctuations in energy usage

Table 4.2: Summary: Hardware/Software Environment, Measurement Tools.

Architecture	Hardware		Software/ Libraries	Power Measurement	Language	Programmability
	Model	Clock				
CPU	AMD 5900x	4.8 GHz	Pytorch 2.0 [24] / OpenCV [23]	HWMonitor [158]	C++	Easy
GPU	Nvidia GTX 3070	1730 MHz	Pytorch 2.0 / OpenCV	Nvidia-smi [159]	C++	Easy
FPGA (HLS)	Xilinx ZCU102	300Mhz	Vitis 2020.2	MaxPower-tool [160] / Power Analyser	C++	Medium
FPGA	Xilinx ZCU102	300Mhz	Vivado 2022.2	MaxPower-tool / Power Analyser	Verilog	Difficult

4.3.4 Measurement Environments

While FPGA can provide an accurate execution time in the simulation mode, the same is not true for the CPU/GPU accelerators, as there might be other software (including part of the operating system) competing for compute resources. To mitigate this, the execution time for each bench-marked algorithm is measured as the average of 1,000 iterations on CPU/GPU and closing all other non-core application processes before execution. The Nvidia (CUDA) GPU also has an initialisation time often associated with setting up the GPU context and memory allocations, can be significant, especially for smaller tasks. The initialisation time is recorded for both with and without.

Software & Hardware Environments: *OpenCV* and *Pytorch* library is used to implement image processing algorithms and CNNs on CPU and GPU platforms. The FPGA implementation is written in Verilog, using Vivado Design Suite 2019.2. Additionally, for comparison purposes, implementation was done with high-level synthesis (HLS) code using Xilinx Vitis 2019.2. Programmability and flexibility vary across architectures shown in Table 4.2; CPUs and GPUs are general-purpose, which makes them highly programmable for a wide range of tasks, with languages like C++ being commonly used. FPGAs are more inflexible since significant time is needed to change implementation designs and are typically crafted in hardware descriptor languages such as Verilog. The hardware setup consists of a desktop PC running a Linux operating system, with a discrete GPU and FPGA connected via a high throughput PCIe interface to reduce data latency.

4.3.5 Measurement Approach

The algorithms are implemented individually on each hardware and then combined to create the combined pipeline. For a fair comparison, open-sourced (OpenCV) and CNN libraries(Pytorch) were employed, which are highly optimised for their respective architectures. This is with exception of the Verilog implementations that were developed manually. The parameter for each algorithms used float precision and 5×5 kernel size. During the benchmarking, for consistency, an uncompressed 8 bit 1920×1080 greyscale (Colour for *RGB2Gray* algorithm) bitmap image for all experiments and a bayerRAW equivalent for *Demoasiacing* algorithm.

We provide multiple performance indicators to compare between architectures. For each algorithm, the runtime is measured on each hardware to determine which accelerator executed the operation in the least amount of time. The results from the runtimes are used to calculate the estimated throughput using Eq. (4.4). The clock cycles per operation (CPO) in Eq. (4.5) gives insight into the average number of cycles required to execute an instruction. To have a fair comparison across the target hardware, the energy per operation is normalised using Eq. (4.6):

$$\text{Throughput} = \frac{N}{t} \tag{4.4}$$

$$\text{CPO} = \frac{f \times t}{N} \tag{4.5}$$

$$\text{EPO} = \frac{P \times t}{N} \tag{4.6}$$

In these equations, N denotes the number of operations performed and t signifies the runtime of the computational task in seconds. f represents the frequency of the processing unit in hertz, and P is power consumption in watts.

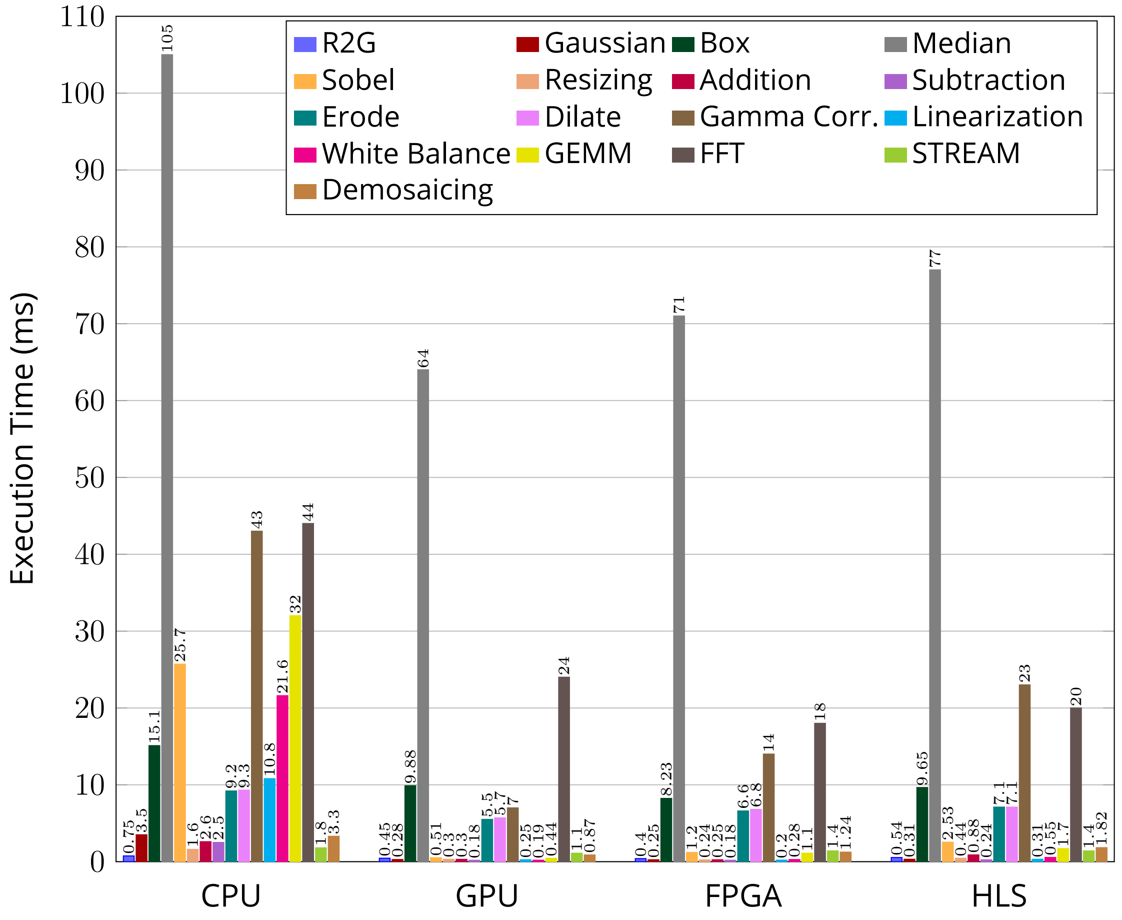


Figure 4.5: Execution times (milliseconds) for *individual algorithms* found within imaging pipelines on each hardware architecture.

4.4 Experiments, Results & Discussion

This section presents the bench-marked results of each algorithm described in 4.3 and an in-depth discussion. The section is divided into two parts, which include the runtime, power consumption, EPO and throughput results for individual ISP algorithms and combined exemplar pipelines.

4.4.1 Individual ISP Algorithms

Fig. 4.5 & Fig. 4.6 plots the execution time (in milliseconds) and power consumption (in Watts) of the selected benchmarking image processing algorithms varying in complexity across different hardware architectures: CPU, GPU, FPGA, and high-level synthesis implementation on FPGA. The results re-

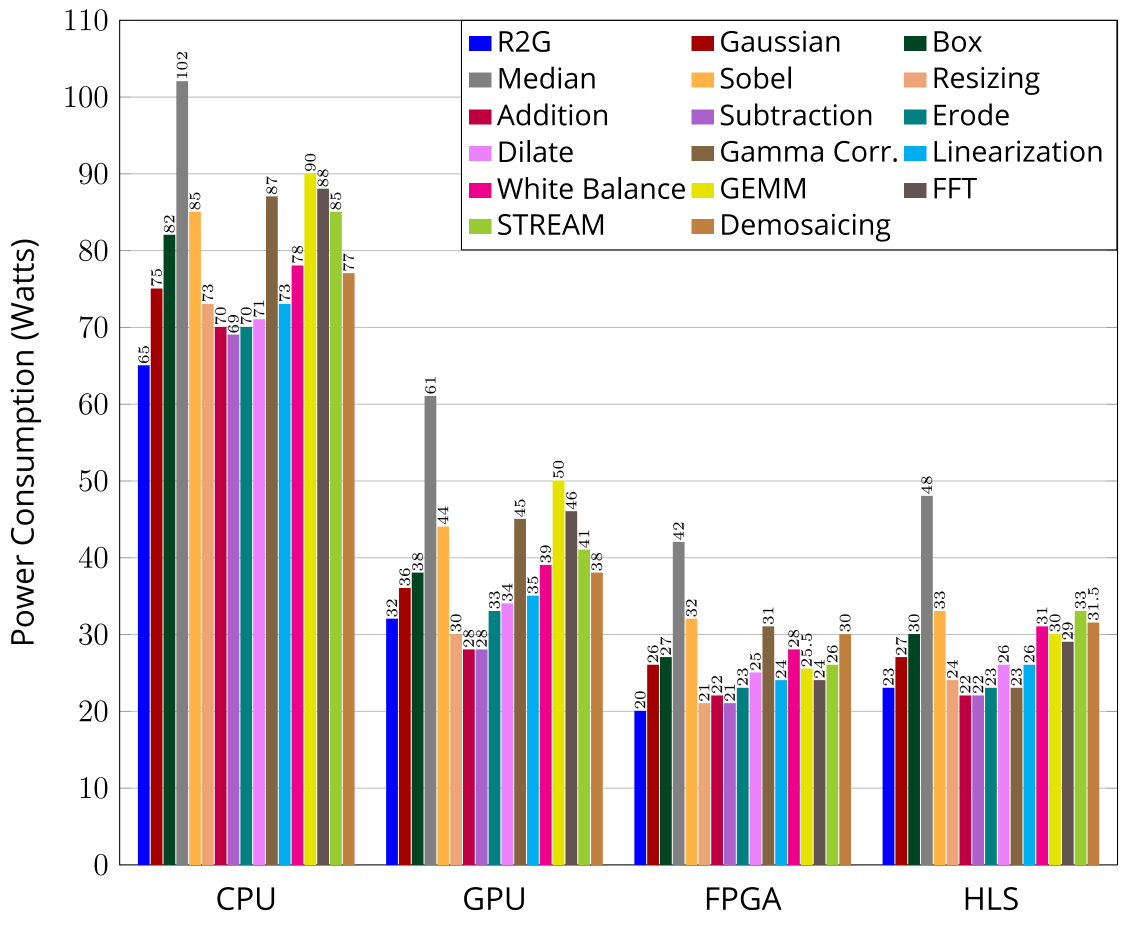


Figure 4.6: Power Consumption (Watts) for *individual algorithms* found within imaging pipelines on each hardware architecture.

veal runtime variations among the architectures for each algorithm. Overall, the CPU performs competitively with the GPU, FPGA and HLS for lower complexity algorithms cases such as *RGB2GRAY* algorithm, *Addition*, and *Subtraction*. It's evident that algorithms involving a small number of operations, such as colour channel conversion, do not require many computational cores to leverage parallelism or memory access requirements. However, the CPU's performance starts to decline as the complexity of the algorithms increases, as seen in cases like *Sobel*, *Gamma Correction*, *GEMM*, and *FFT*, where the large array processing capabilities offered by the GPU and FPGA become more prominent. On average, the CPU is $4.5\times$ and $4.35\times$ slower than the GPU and FPGA, respectively.

Across most algorithms, the GPU consistently demonstrates its capability for calculations that require a great amount of multiply and accumulate operations such as *GEMM* where its vast number of cores are fully occupied. Although FPGAs also contain many processing blocks, their quantity is typically far fewer than GPUs, thus having a small increase in runtime. The results show that GPU implementations outperform FPGAs for larger data/kernel sizes but underperform for smaller sizes where the memory latency and kernel initialisation overhead become significant.

Non-linear algorithms such as *Median*, *Erode* and *Dilate* have a significant impact on all architectures due to having unconventional operations which do not have specialised hard blocks to compute them. In addition, branching conditions involve irregular memory access patterns. The impact can be seen from the GPU and FPGA results, which jump in runtime from linear to non-linear filters where parallelisation is inhibited and the kernel is not separable. The *Gamma Correction* is another non-linear algorithm which operates on a pixel-wise basis, modifying pixel intensities based on their original values and gamma factors. Division operations are generally more complex and relatively slower compared to multiplication and addition but can be pipelined in hardware. Therefore reducing the overall runtime on all architectures and requiring more resources to compute. The HLS implementation proved competitive in runtime with the hand-written FPGA and GPU for many low-medium complexity algorithms in which the compiler can leverage pre-written functions or libraries. On the contrary, algorithms such as *Gamma Corr.* or *Demosiacing*, which are implemented without using pre-optimised libraries, required more compiler effort to generate HDL, resulting in slightly slower execution times.

Related to power consumption, the CPU consumes the most energy in all cases. As the algorithm complexity increases, more power is consumed. The FPGA and HLS are 27% and 7.5% more energy efficient than the GPU in nearly all algorithms, which reveals that direct hardware designs have much less power overhead than the GPU, which needs to support other processes such as host communication. Table 4.4 presents average reduction ratios for each algorithm relative to CPU energy consumption. The GPU exhibits a higher energy reduction ratio in *White balance*, *Sobel*, *Gamma* and *GEMM*, which reveals that the higher clocked GPU can execute the algorithms faster while offsetting the higher operating power consumption.

The algorithm throughput and energy per operation in nanoJoules, are

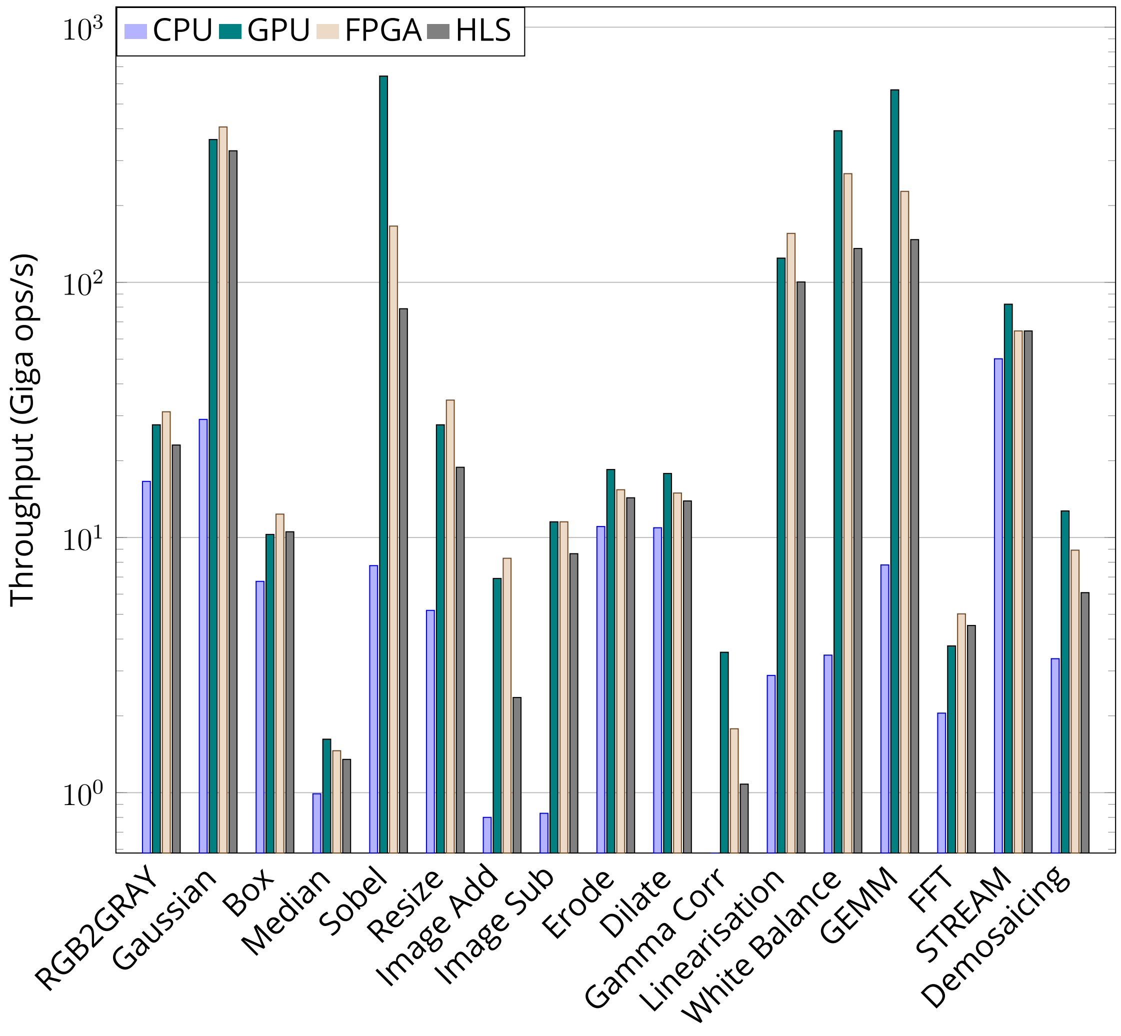


Figure 4.7: Throughput for *individual algorithms* found within imaging pipelines on each hardware architecture (log scale)

shown in Fig. 4.7 and Fig. 4.8. Using nJ, as opposed to watts, offers a more granular and clearer measurement. The trend remains, with CPU lagging in throughput and EPO compared to the other architectures for all algorithms. Although in *RGB2GRAY*, *Median* & *FFT*, the CPU is closer to the other accelerators in both metrics due to lower complexity, input size and non-linearity of the algorithms, which don't map well to highly parallel hardware. The GPU consistently maintains its higher throughput capabilities, as shown by the graphs and lower energy per operation on larger data sized algorithms. The FPGA achieves closer results to the GPU, but being clocked lower and having fewer processing cores results in a slight performance decrease. The HLS tool exhibits comparable performance, closely trailing the GPU and FPGA

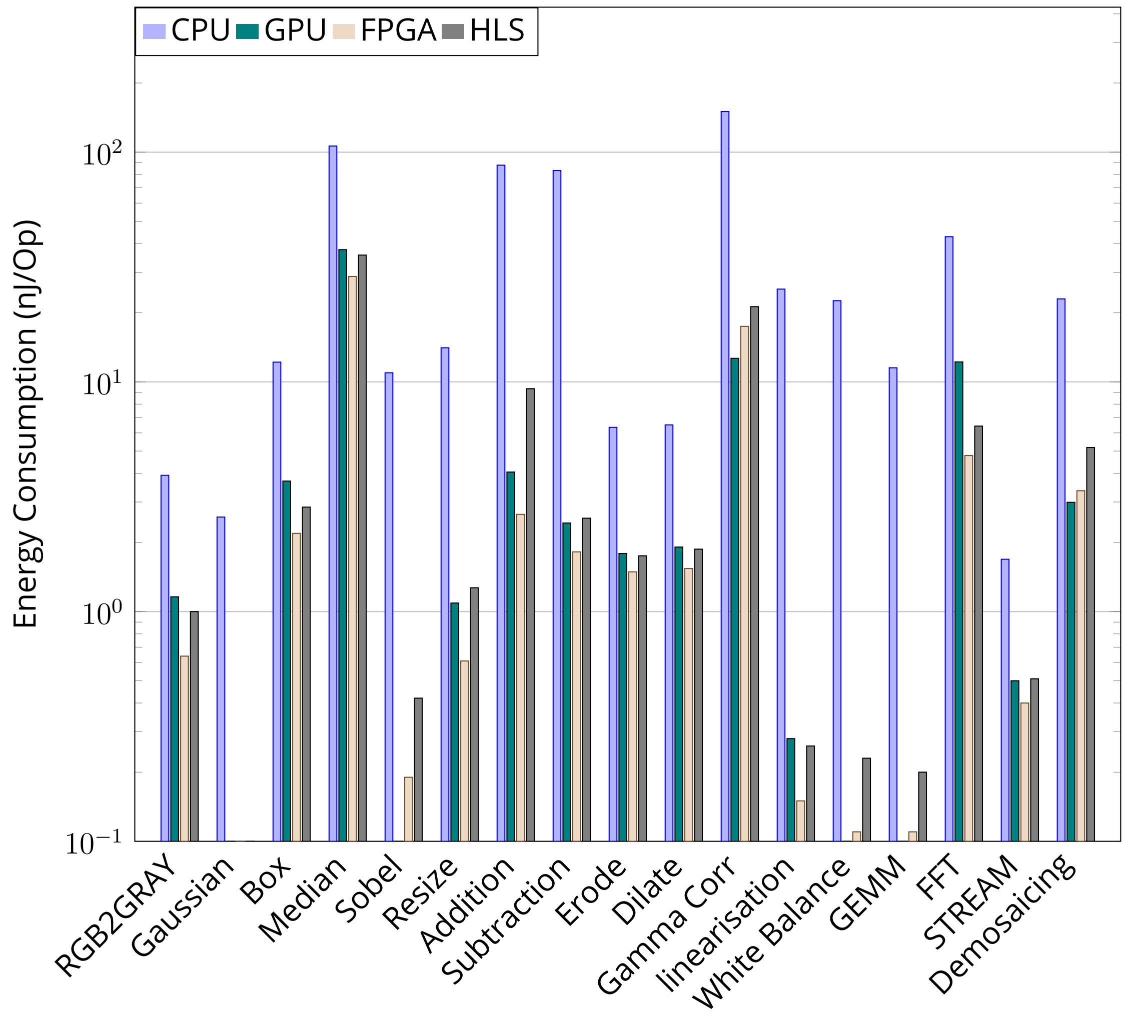


Figure 4.8: Energy per operation (EPO) for *individual algorithms* found within imaging pipelines on each hardware architecture (log scale)

in most algorithms. Although it experiences a slight performance lag which is attributed to challenges in hardware translation, this setback is mitigated by leveraging optimised pre-written functions used in *STREAM*. In such algorithms, the HLS tool demonstrates comparable throughput performance to hand-optimised FPGA implementation.

4.4.2 Combined ISP Pipelines

The combined image processing pipeline algorithm for execution times is reported in Fig. 4.9. The runtime results indicate nearly a $3.46\times$, $2.92\times$, and $1.79\times$ order of magnitude improvement going from CPU to GPU, FPGA, and HLS, re-

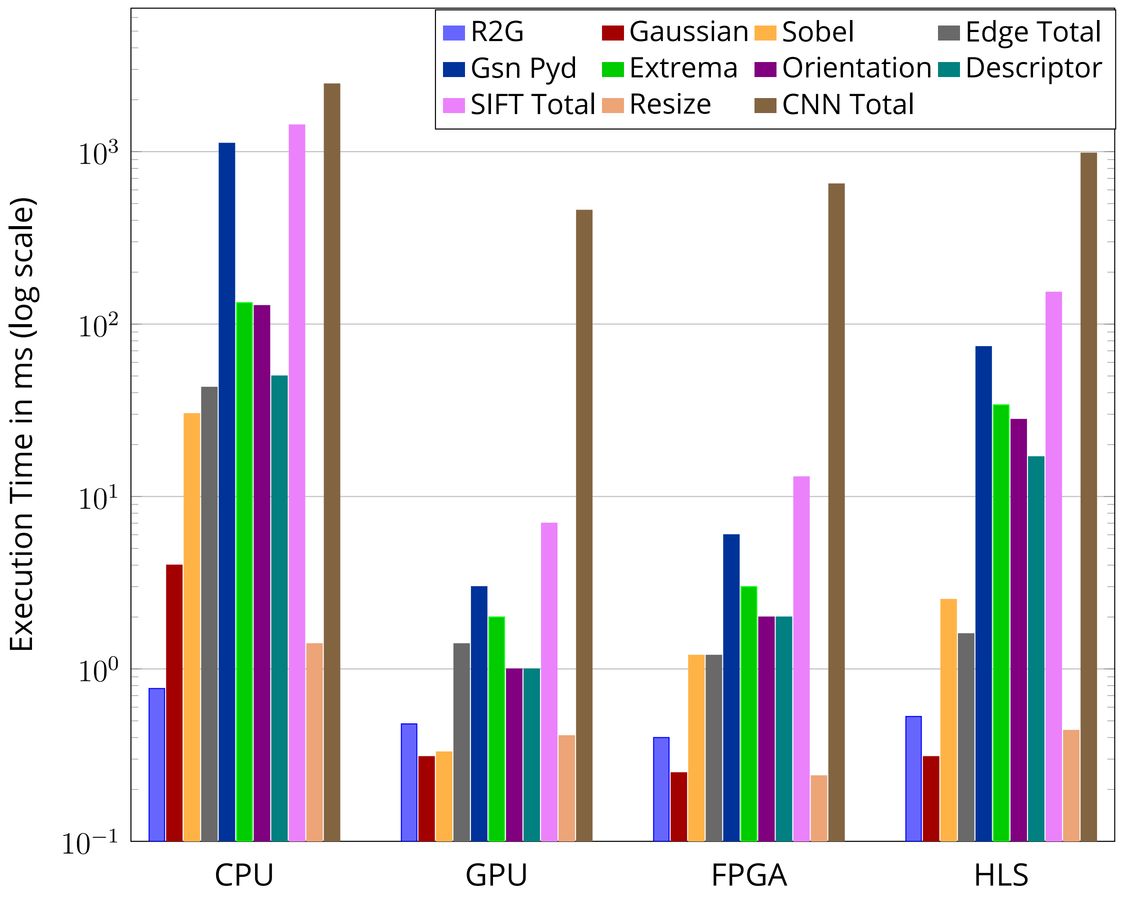


Figure 4.9: Execution times (milliseconds) for combined image processing pipelines on each hardware architecture. Some bars that don't appear are due to values being small

spectively. The GPU also has a slightly marginal $\sim 0.56\times$ runtime improvement over the FPGA implementation and is closer to a double order of magnitude improvement than the HLS. The memory transfer results in the table show that the latency for each algorithm ranges from $20 \sim 70$ms for the CPU and GPU. In contrast, the FPGA can take advantage of stream processing pixels to minimise memory access for most algorithms.

In the execution time of *Edge Total* and *CNN Total* pipelines, the improvement between accelerators is minimal in comparison to the *SIFT Total*, which has substantial differences between each platform. The *Gaussian Pyramid* stage within *SIFT* has the largest contribution to overall *SIFT Total* execution time. The result can be attributed to image filter algorithms with many multiply-and-accumulate operations, which map well to the parallel processing of GPU and FPGAs due to their high number of compute cores. In contrast, the CPU

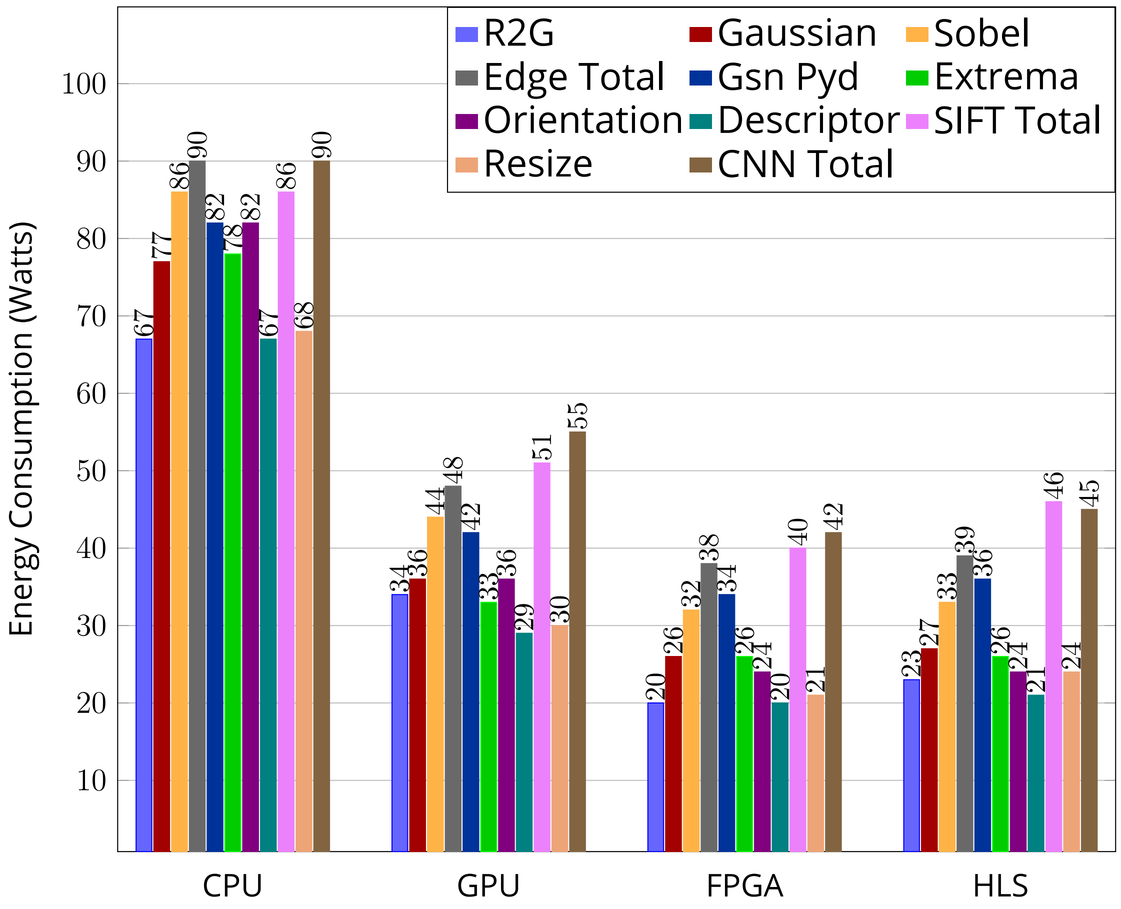


Figure 4.10: Power Consumption (Watts) for combined processing pipelines on each hardware architecture.

suffers due to the lack of processing cores where parallelism is needed, resulting in poorer execution time. In addition, the power consumption results in Fig. 4.10 indicate that the FPGA consumes $1.4 \sim 6$x less power in the *Gaussian Pyramid* stage than the CPU and GPU, with the HLS implementation being slightly less power efficient. The architecture of FPGAs enables tight-knit designs without additional power consumption by other support functions. Furthermore, the efficiency is evidenced in the *Sobel* algorithm; the FPGA makes use of the DSP blocks, which consumes less power, although the algorithm contains more than double the numerical operations than *Box* or *Gaussian*.

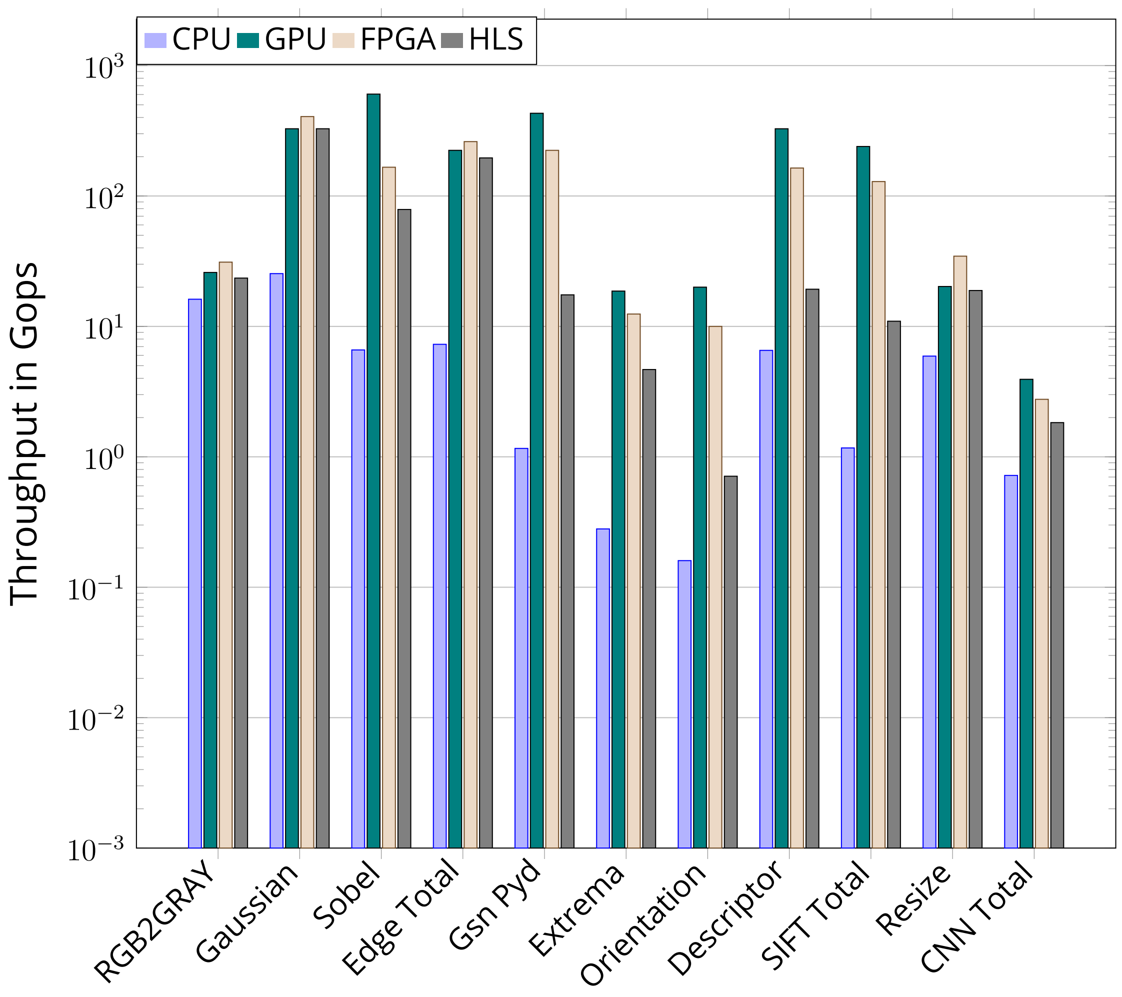


Figure 4.11: Throughput of each algorithm on hardware platforms (log scale) for combined processing pipelines.

4.4.3 Energy Consumption & Throughput Results

The plots in Fig. 4.11 & Fig. 4.12 show the throughput and energy consumed per operation. The throughput difference between each accelerator for low complexity/arithmetic algorithms such as *R2G* was consistent, with all three accelerators are comparable. However, the gap widens between CPU and the other hardware in throughput from *Gaussian* to *CNN Total*. Comparing GPU and FPGA implementations, the FPGA outperforms in throughput for *Edge Total*. The hand-written FPGA is able to stream algorithms more effectively than the GPU without overhead CPU memory management latency. Furthermore, the core initialisation and allocation of the GPU does not offset the time it processes for the *Gaussian* and *Resize* algorithms.

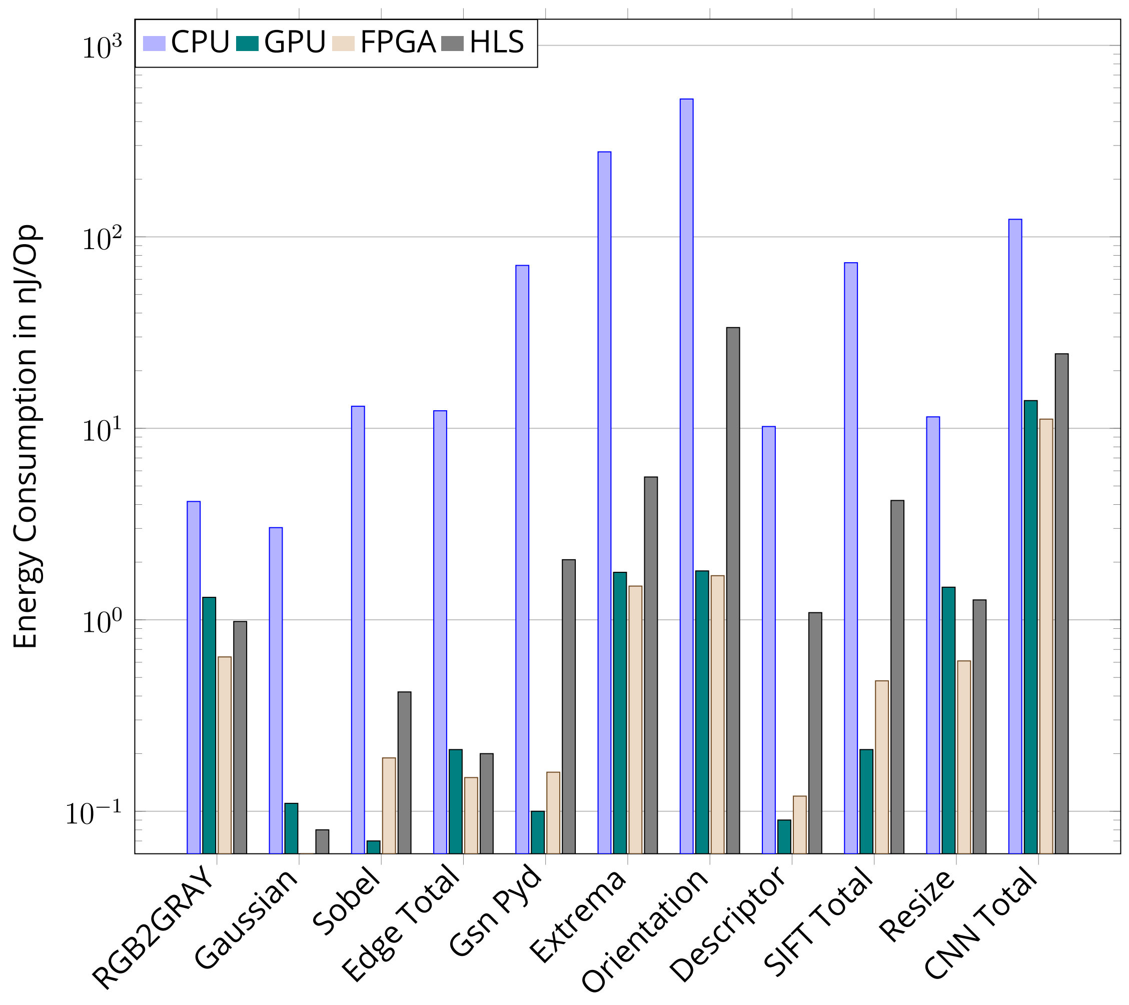


Figure 4.12: Energy per operation (EPO) for each algorithm on hardware platforms (log scale) for combined processing pipelines.

The GPU and both FPGAs consume ~ 2x less energy per operation compared to the CPU for *Gaussian Pyramid, Extrema, Orientation* algorithms. This result is because many cores are at full occupancy which start to compute more data in parallel thus increasing throughput. The HLS implementation of *Orientation* and *Descriptor* found in *SIFT* algorithm has worse energy consumption per operation than the hand-written FPGA and GPU. The throughput loss in the *descriptor* algorithm for GPU and FPGA is due to high data dependency, which leads to sequential processing. In some algorithms, the GPU consumes less energy per operation than the FPGA. The lower power consumption is due to processing the algorithm faster on the GPU than the FPGA to offset the high energy usage from the clock speed and support functions. Table 4.5 summarises the results of each calculation (Throughput, CPO, Energy Consumption per Operation).

4.4.4 Discussions

The results highlight the areas where there are clear performance gaps in the implementations on each accelerator. For example, the HLS may not be the best approach to implement all algorithms due to the compiler not optimally translating custom C++ code to Verilog, resulting in performance degradation compared to hand-written FPGA code. Therefore, our benchmarking framework contains a consistent set of metrics to assess various implementations and understand the trade-offs between each target hardware. The results from the framework indicate two generic conclusions: 1) FPGAs are better suited to meet power budget, whereas GPUs can achieve faster execution time (as anticipated), and 2) most optimised performance can be achieved through heterogeneous computing, especially in real-time imaging.

For example, in implementing SIFT, the sub-algorithms *Gaussian Pyramid* and *Descriptor Generation* are better suited to GPUs due to faster execution time and throughput, whereas *Extrema Detection* and *Orientation* are worthy of targeting FPGAs due to their energy consumption profile. For the first two, GPU consumes $\sim$1 & $\sim$8 nanoJoule more energy per operation, respectively, but in trade for significant speedup. On the contrary, the latter ones are closely comparable using the throughput and execution time metrics on the GPU and FPGA. However, the power consumed per operation for both algorithms is significantly lower for the FPGA, and hence, the FPGA is better suited. Therefore, partitioning SIFT to target a heterogeneous architecture would benefit from both power and speed performance improvements. However, it is worth noting that such partitioning will incur the cost of frequent memory transfer between architectures.

In the case of the edge detection pipeline from *RGB2GRAY* to *Sobel*, the GPU is ~ 2x faster than the FPGA but at the cost of more energy consumption. Therefore, it may be ideal to select the FPGA platform, especially within an embedded system with low power constraints. Furthermore, the low latency of FPGAs would achieve better execution time if the GPU initialisation and memory transfer time are taken into account. Therefore, when considering the deployment of an algorithm on a GPU, it's vital to ensure that the speedup gained from the GPU's parallel processing capabilities outweighs this initial transfer overhead. If the algorithm doesn't run sufficiently fast on the GPU to compensate for this initialisation time, it might not be the optimal choice for real-time or low-latency requirements. In such cases, relying on low-latency architectures such as FPGA offers better overall performance. This highlights the importance of a holistic evaluation of execution times, factoring in initialisation and processing time, before deciding on the accelerator. Additionally, GPUs require a CPU to allocate tasks and manage memory which means additional idle power being consumed.

In the CNN pipeline, the FPGA computes the *RGB2GRAY* and *Resize* faster than the other architectures but is $1.45\times$ slower in CNN inference compared to GPU. However, when considering power consumption and image transfer latency, it is better to pipeline all the operations on an FPGA. The GPU may be better suited to training models or executing larger CNNs that are too big to fit onto FPGA logic. The hand-written FPGA is highly sensitive to implementation and optimisations but allows more granular control over design, while GPUs benefit from a mature ecosystem of compilers and tools. These compilers can automatically optimise and implement core operations, often with high efficiency.

In terms of qualitative visual quality, images processed by CPU, GPU, and FPGA all appear similar since the precision is kept the same for each algorithm. However, for a more comprehensive assessment, deeper analysis can be conducted using visual metrics algorithms (*e.g.*, SSIM, RSME) or through human-based visual experiments.

4.5 Conclusions

This chapter discusses the importance of understanding the properties of image processing algorithms to determine which hardware accelerator is suitable and if it can benefit from heterogeneous architecture, especially for real-time vision applications. To facilitate such insight, a benchmarking framework is proposed to observe the features of image processing algorithms and provide a consistent set of metrics to identify trade-offs in performance and energy efficiency of various hardware accelerators, including CPUs, GPUs and FPGAs. We selected commonly used low, medium and high-level image processing operations as exemplars, dissected them in sub-algorithms, and benchmarked their throughput and energy consumption profiles on each hardware. The results indicate that partitioning algorithms based on their memory latency, energy consumption and throughput profiles have the potential for efficient deployment on heterogeneous hardware in achieving optimised performance at a lower power.

The performance variations are influenced by factors such as parallelism, memory access patterns, and the capability of each architecture to map algorithmic operations efficiently. Algorithms that contain large data size parallel operations and have regular memory access patterns tend to perform well on GPU, while FPGA and HLS architectures may perform better for smaller data sized operations. However, both face challenges optimising for irregular memory access patterns and complex algorithmic computations. The FPGA delivers great performance relative to its size, clock and processing resources, but the specialised nature of its architecture is highly sensitive to how the design is implemented. Translating HLS code to hardware can introduce overheads, potentially affecting performance. While HLS tools expedite design cycles, they might not always match the optimisation of manual hardware designs, leading to possible inefficiencies in resource allocation and data paths.

Table 4.3: Execution times (ms) of combined pipeline (Total) and their corresponding algorithms including **memory transfer.**

Pipeline	Algorithm	CPU	GPU	FPGA	HLS
Edge Detection	RBG2GRAY	20.77	49.48	**0.4**	0.53
	Gaussian	24	51.31	**0.25**	0.31
	Sobel	58.23	45.33	**1.2**	2.53
	Edge Total	63.45	51.4	**1.44**	1.6
Feature Extraction	Gaussian Pyramid	1118	**3**	6	74
	Extrema Detection	133	**2**	3	34
	Orientation Magnitude	128	**1**	2	28
	Descriptor	50	**1**	2	17
	SIFT Total	1434	57	**13**	153
Classification	RBG2GRAY	20.34	48.43	**0.40**	0.53
	Resize	21.4	51.55	**0.24**	0.44
	CNN	2245	280	**234**	265
	CNN Total	2570	310	**254**	285

Table 4.4: Reduction Ratios Relative to CPU Energy Consumption for Individual Algorithms. (Highest reduction ratio in bold.)

Operation Group	Algorithm	GPU	FPGA	HLS
Image Preprocessing	RGB2GRAY	3.39x	**6.09x**	3.93x
	Resize	12.98x	**23.17x**	11.06x
Image Arithmetic	Image Add	21.67x	**33.09x**	9.40x
	Image Sub	34.23x	**45.63x**	32.67x
Image Filters	Gaussian	26.04x	**40.38x**	31.36x
	Box	3.30x	**5.57x**	4.28x
	Sobel	**160.15x**	56.89x	26.16x
	Median	2.82x	**3.70x**	2.98x
Linear Algorithms	White Balance	**227.37x**	214.90 x	98.82x
	Linearisation	90.10x	**164.25x**	97.82x
Non-linear Algorithms	Erode	3.55x	**4.24x**	3.63x
	Dilate	3.41x	**4.22x**	3.48x
	Gamma Corr	**11.88x**	8.62x	7.07x
	Demosaicing	**7.69x**	6.83x	4.43x
Fundamental	GEMM	**130.91x**	102.67x	56.47x
	FFT	3.51x	**8.96x**	6.68x
	STREAM	3.39x	**4.20x**	3.31x

Table 4.5: Algorithm All Metrics Result Summary: Runtime, Throughput, Clock per Operatiom, Energy Consumption per Operation.

Algorithm	Complexity	Runtime (ms)				Throughput (Gops)				CPO (Clock Cycle / Op)				Energy Consumption/ Operation (nJ / Ops)			
		CPU	GPU	FPGA	HLS	CPU	GPU	FPGA	HLS	CPU	GPU	FPGA	HLS	CPU	GPU	FPGA	HLS
RGB2GREY	Low	0.75	0.45	0.40	0.54	16.59	27.65	31.10	23.04	2.89×10^{-1}	6.33×10^{-5}	9.65×10^{-3}	1.30×10^{-2}	3.92	1.16	0.64	1.00
Resizing		1.6	0.30	0.24	0.44	5.18	27.65	34.56	18.85	9.26×10^{-1}	6.33×10^{-5}	8.68×10^{-3}	1.59×10^{-2}	14.08	1.09	0.61	1.27
Addition		2.6	0.30	0.25	0.88	0.80	6.91	8.29	2.36	6.02	2.53×10^{-4}	3.62×10^{-2}	1.27×10^{-1}	87.77	4.05	2.65	9.34
Subtraction		2.5	0.18	0.18	0.24	0.83	11.52	11.52	8.64	5.79	1.52×10^{-4}	2.60×10^{-2}	3.47×10^{-2}	83.19	2.43	1.82	2.55
Erode	Medium	9.2	5.5	6.6	7.1	11.04	18.47	15.39	14.31	4.35×10^{-1}	9.47×10^{-5}	1.95×10^{-2}	2.10×10^{-2}	6.34	1.79	1.49	1.75
Dilate		7.1	5.7	6.8	7.1	10.92	17.82	14.94	13.91	4.39×10^{-1}	9.82×10^{-5}	2.01×10^{-2}	2.16×10^{-2}	6.50	1.91	1.54	1.87
Box		15.1	9.88	8.23	9.65	6.73	10.28	12.35	10.53	7.13×10^{-1}	1.70×10^{-4}	2.43×10^{-2}	2.85×10^{-2}	12.19	3.70	2.19	2.85
Gamma Corr.		43	7	14	23	0.58	3.55	1.78	1.08	8.29	4.92×10^{-4}	1.69×10^{-1}	2.77×10^{-1}	150.34	12.66	17.44	21.26
Gaussian		3.5	0.28	0.25	0.31	29.03	362.88	406.43	327.76	1.65×10^{-1}	4.82×10^{-6}	7.38×10^{-4}	9.15×10^{-4}	2.58	0.10	0.06	0.08
Sobel		0.31	0.41	1.2	2.53	0.99	1.62	1.46	1.35	6.19×10^{-1}	2.72×10^{-6}	1.81×10^{-3}	3.81×10^{-3}	10.96	0.07	0.19	0.42
Median		105	64	71	77	7.76	643.20	166.16	78.81	4.86	1.08×10^{-3}	2.05×10^{-1}	2.23×10^{-1}	106.34	37.65	28.76	35.65
Linearization		10.8	0.25	0.20	0.31	2.88	124.42	155.52	100.34	1.67	1.41×10^{-5}	1.93×10^{-3}	2.99×10^{-3}	25.35	0.28	0.15	0.26
White Balance		21.6	0.19	0.28	0.55	3.46	392.89	266.61	135.73	1.39	4.45×10^{-6}	1.13×10^{-3}	2.21×10^{-3}	22.57	0.10	0.11	0.23
GEMM		32	0.44	1.1	1.7	7.81	568.18	227.27	147.06	6.14×10^{-1}	3.08×10^{-6}	1.32×10^{-3}	2.04×10^{-3}	11.52	0.09	0.11	0.20
FFT		44	24	18	20	2.05	3.76	5.02	4.52	2.34	4.65×10^{-4}	5.98×10^{-2}	6.64×10^{-2}	42.87	12.22	4.78	6.42
STREAM		1.8	1.1	1.40	1.40	50.17	82.10	64.51	64.51	9.57×10^{-2}	2.13×10^{-5}	4.65×10^{-3}	4.65×10^{-3}	1.69	0.50	0.40	0.51
Demosaicing		1.82	0.87	1.24	1.82	3.35	12.71	8.92	6.08	1.43	1.38×10^{-4}	3.36×10^{-2}	4.94×10^{-2}	22.98	2.99	3.36	5.18
SIFT Total	High	1429	7	16	153	1.17	239.63	129.02	10.96	6.58×10^{-1}	7.82×10^{-6}	1.15×10^{-3}	1.53×10^{-3}	73.27	0.21	0.48	4.20
Edge Total		43	1.4	1.2	1.6	7.29	223.89	261.20	195.90	4.09	7.30×10^{-6}	3.58×10^{-3}	2.74×10^{-2}	12.35	0.24	0.15	0.20
CNN Total		2470	457	650	980	0.73	3.93	2.77	1.83	6.59×10^{0}	4.44×10^{-1}	1.08×10^{-1}	1.63×10^{-1}	123.50	13.96	11.17	24.50

5 Domain-Specific Optimisations

In recent years, real-time vision systems on embedded hardware have become ubiquitous due to the increased need for different applications such as autonomous driving, edge computing, remote monitoring, etc. Field Programmable Gate Arrays (FPGA) offer the speed and flexibility to architect tight-knit designs that are power and resource-efficient. It has resulted in FPGAs becoming integrated into many applications [161]. Often, these designs consist of many low to high-level image processing algorithms that form a pipeline. Increasingly, the race for faster processing encourages hardware application developers to optimise the algorithms.

Traditionally, optimisations are domain agnostic and developed for general purpose computing. The majority of these optimisations aim to improve throughput and resource usage by increasing the number of parallel operations [162], memory bandwidth [163] or operations per clock cycle [164]. On the contrary, domain-specific optimisations are more specialised in a particular domain and can potentially achieve larger gains in faster processing and reducing power consumption. This chapter proposes domain-specific optimisation techniques on FPGAs that exploit the inherent knowledge of the image processing pipeline.

In demonstrating the proposition, a thorough analysis is presented of well-known image processing algorithms, emerging CNN architectures (MobileNet [165] & ResNet [166]), and Scale Invariant Feature Transform (SIFT) [167]. The decision to include MobileNet is influenced by its popular use within embedded systems, and ResNet is included for its consistently higher accuracy rates compared to other available architectures. Additionally, SIFT is chosen for being the most popular feature extraction algorithm, owing to its performance and accuracy. Algorithmic properties are exploited with the proposed domain-specific optimisation strategies. The optimised design undergoes evaluation and comparison with other general optimised hardware designs regarding performance, energy consumption, and accuracy. The main contributions of this chapter are:

- Proposition of four domain-specific optimisation strategies for image processing and analysing their impact on performance, power and accuracy; and
- Validation of the proposed optimisations on widely used representative image processing algorithms and CNN architectures (MobilenetV2 & ResNet50) through profiling various components in identifying the common features and properties that have the potential for optimisations.

5.1 Domain-Specific Optimisations

Image processing algorithms typically form a pipeline with a series of processing blocks. Each processing block consists of a combination of low, mid, intermediate and high-level imaging operations, starting from colour conversion, filtering to histogram generation, features extraction, object detection or tracking. Any approximation and alteration to the individual processing block or the pipeline have an impact on the final outcome, such as overall accuracy or runtime. However, depending on the applications, such alterations are expected to be acceptable as long as they are within a certain error range (e.g., $\sim \pm 10\%$).

Many image processing algorithm operations share common functional blocks and features. Such features are useful for forming domain-specific optimisation strategies. Within the scope of this work, image processing algorithms are profiled and analysed to enable potential areas for optimisations. However, such optimisations impact algorithmic accuracy and therefore, it is important to identify the trade-off between performance, power, resource usage, and accuracy.

The hypothesis suggests that understanding of domain knowledge, e.g., processing pipeline, individual processing blocks, or algorithmic performance, can be used for optimisations to gain significant improvements in runtime and lower power consumption, especially in FPGA-based resource-limited environments. Based on the common patterns observed in a variety of image processing applications, this section proposes four domain-specific optimisation (DSO) approaches: *1) downsampling*, *2) datatype*, *3) separable filter* and *4) convolution kernel size*. However, on the flip side, optimisation often leads to lower accuracy in return for gains in speed and lower energy consumption. The effectiveness of these optimisations is compared against benchmark FPGA, GPU and CPU implementations, showing the impact on accuracy. Within the scope of this thesis, four optimisation strategies have been identified and are discussed below:

5.1.1 Optimisation I: Down Sampling

Down/subsampling optimisation reduces the data dimensionality while largely preserving image structure and hence accelerates runtime by lowering the number of computations across the pipeline. Sampling rate conversion operations such as downsampling/subsampling are widely used within many application pipelines (*e.g.*, low bit rate video compression [89] or pooling layers in Convolutional Neural Network (CNN) [168]) to reduce computation, memory and transmission bandwidth. Image downsampling reduces the spatial resolution while retaining as much information as possible. Many image processing algorithms use this technique to decrease the number of operations by removing every other row/column of an image to speed up the execution time. However, the major drawback is the loss of image accuracy due to removing pixels. *down sampling optimisation* used for each selected algorithm is a bilinear interpolation, and both runtime and accuracy are measured.

Bilinear downsampling is a technique that reduces the number of pixels in an image by computing each output pixel as a weighted average of its four nearest input pixels. The weights, represented by interpolation factors α_i and β_i, are determined based on the distances between the target output pixel (x, y) and the neighbouring input pixels (x_i, y_i) in both the horizontal and vertical directions. These factors contribute to a smoothed, downsampled image by interpolating colour values based on the surrounding pixel information. Mathematically, the value of a downsampled pixel $D(x, y)$ can be calculated using the following equation:

$$D(x, y) = \sum_{i=1}^{4} \alpha_i \cdot \beta_i \cdot I(x_i, y_i) \tag{5.1}$$

Where $D(x, y)$ represents the downsampled pixel value at location (x, y) in the downscaled image, and $I(x_i, y_i)$ represents the intensity (colour value) of the neighbouring pixel (x_i, y_i) in the original image. Downsampling reduces the image size between octaves in the 'Gaussian pyramid' construction stage.

5.1.2 Optimisation II: Datatype

Bit width reduction through datatype conversion (*e.g.*, floating-point (FP) to integer) significantly reduces the number of arithmetic operations, resulting in optimised runtime at lower algorithmic accuracy. Whilst quantising from FP to integer representations is common in the software domain, one of the advantages of reconfigurable hardware is the capability to reduce dimensionality to arbitrary sizes (*e.g.*, 7, 6, 5, 4 bits) as a trade-off between accuracy and power/performance.

Consider an image where each pixel has a floating-point value in the range of 0 to 1. A straightforward way to perform quantisation is to map these values to a set of integers. One commonly used formula for this conversion is:

$$Q(x) = \text{round}(x \times (2^k - 1)) \tag{5.2}$$

Here, $Q(x)$ is the quantised value, x is the original floating-point value, and k is the bit-depth (e.g., 8 for an 8-bit image). The function round rounds the value to the nearest integer. This equation multiplies the floating-point value by $2^k - 1$ (255 for an 8-bit image) and rounds it, converting the value into an integer between 0 and $2^k - 1$. This process significantly reduces the data size and computational requirements, albeit with some loss of information due to rounding. The converted integer values can then be used in place of the floating-point values for further processing tasks.

In image processing, most algorithms are inherently developed using floating-point (FP) calculations. Although FP offers higher accuracy, it comes at the expense of computational complexity and, therefore, increased resource and energy consumption. A viable alternative is fixed-point arithmetic, where a fixed location of the decimal point separates integers from fractional numbers. Opting for fixed-point representation can significantly improve computational speed, albeit at the cost of some accuracy. Here, a *datatype conversion* optimisation is proposed, wherein all operational stages are converted from FP to integer arithmetic. This conversion allows for an evaluation of the trade-off between performance and accuracy.

5.1.3 Optimisation III & IV: Convolution

Convolution kernel size optimisation reduces computational complexity, which is directly proportional to the squared size of the filter kernel size, i.e., $\mathcal{O}(n^2)$ or quadratic time complexity. Convolution is a fundamental operation employed in most image processing algorithms that modify the spatial frequency characteristics of an image. Given a kernel and image size $n \times n$ and $M \times N$, respectively, convolution would require $n^2 \times M \times N$ multiplications and additions. For a given image, complexity is dependent on the kernel size, leading to a complexity of $\mathcal{O}(n^2)$. Reducing kernel size significantly lowers the number of computations; for example, replacing a 5×5 kernel with a 3×3 kernel would reduce the computation by a factor of $\times 2.7$. Therefore, this is proposed as an ideal target for optimisation, although it may come at the cost of accuracy.

$$\frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} \quad \times \quad \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} \tag{5.3}$$

Another convolution optimisation strategy is separable filters, which is a type of linear filter that can be broken down into a series of 1D filters shown in Eq.5.3, making it computationally efficient for image processing tasks. The separability property stems from the ability to represent a 2D filter kernel as the outer product of two 1D kernels. This means that instead of directly convolving the image with a 2D kernel, one can first convolve it along the rows with a 1D kernel and then convolve the result along the columns with another 1D kernel. The formula for a separable filter can be expressed as:

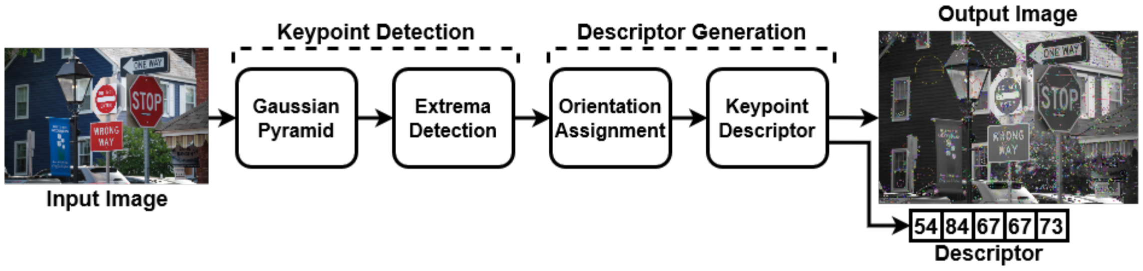


Figure 5.1: SIFT Algorithmic Block Diagram.

$$H(x,y) = F(x) \cdot G(y) \tag{5.4}$$

where $H(x,y)$ is the 2D filter kernel, $F(x)$ is the 1D filter kernel applied along the rows, and $G(y)$ is the 1D filter kernel applied along the columns. By separating the filtering process into 1D convolutions, the number of operations required is significantly reduced, leading to faster image filtering compared to non-separable filters. Common examples of separable filters include Gaussian filters and the Sobel operator for edge detection.

5.2 Case Study Algorithms

In this section, an overview of the representative algorithms targeted for optimisation is presented, as discussed in Section 5.1. Subsequently, the proposed optimisations will be applied to enhance their performance.

5.2.1 SIFT

SIFT [167] is one of the widely used prototypical feature extraction algorithms. To demonstrate the proposed optimisations, various versions of SIFT have been implemented, consisting of two main and several sub-components as shown in Fig. 5.1 and described below.

Scale-Space Construction

Gaussian Pyramid The Gaussian pyramid $L(x,y,\sigma)$ is constructed by taking in an input image $I(x,y)$ and convolving it at different scales with a Gaussian

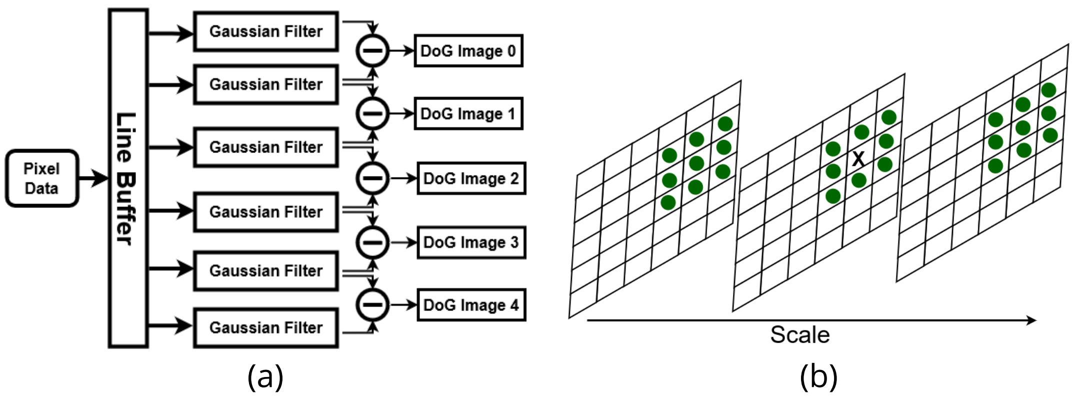


Figure 5.2: a) Scale-Space Hardware Block Diagram b) Extrema Detection in Local Space/Scale Neighbourhood

kernel $G(x, y, \sigma)$:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{5.5}$$

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y), \tag{5.6}$$

Where σ is the standard deviation of the Gaussian distribution. The input image is then halved into a new layer (octave), which is a new set of Gaussian blurred images. The number of octaves and scales can be changed depending on the requirements of the application.

The implemented block design reads pixel data of input images into a line buffer shown in Fig. 5.2(a). The operations in this stage are processed in parallel for maximum throughput. This is due to significant matrix multiplication operations, which greatly impact the runtime. This stage is the most computationally intensive, making it an ideal candidate for optimisation.

The Difference of Gaussian $DOG(x, y, \sigma)$, in Eq.5.7 is obtained by subtracting the blurred images between two adjacent scales, separated by the multiplicative factor k.

$$\textbf{\textit{DOG}}(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma). \tag{5.7}$$

The minima and maxima of the DOG are detected by comparing the pixels between scales shown in Fig. 5.2(b). This identifies points that are best representations of a region of the image. The local extrema are detected by comparing each pixel with its 26 neighbours in the scale space (8 neighbour pixels within the same scale, 9 neighbours within the above/below scales). Simultaneously, the candidate keypoints with low contrast or located on an edge are removed.

Descriptor Generation

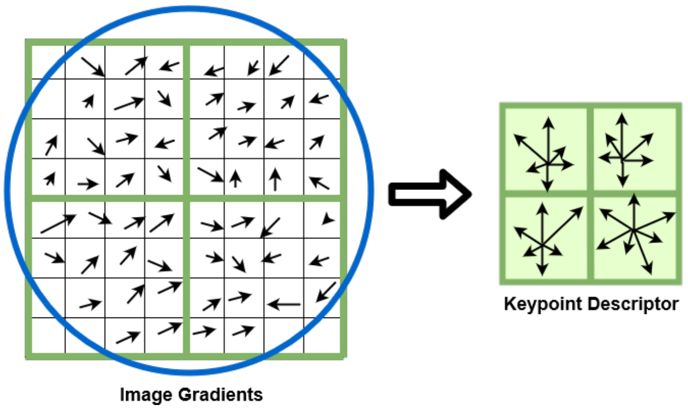


Figure 5.3: Magnitude & Orientation Assignment and Keypoint Descriptor Generation

Magnitude & Orientation Assignment Inside the SIFT descriptor process shown in Fig. 5.3, the keypoint's magnitude and orientation are computed for every pixel within a window and then assigned to each feature based on the local image gradient. Considering L is the scale of feature points, the gradient magnitude $m(x, y)$ and the orientation $\theta(x, y)$ are calculated as:

$$m(x,y) = \sqrt{L_x(x,y) + L_y(x,y)}, \tag{5.8}$$

$$\theta(x,y) = tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right). \tag{5.9}$$

Once the gradient direction is obtained from the result of pixels in the neighbourhood window, a 36 bin histogram is generated. The magnitudes are Gaussian weighted and accumulated in each histogram bin. During the implementation, $m(x, y)$ and $\theta(x, y)$ are computed based on the CORDIC algorithm [169] in vector mode to map efficiently on an FPGA.

Keypoint Descriptor

After calculating the gradient direction around the selected keypoints, a feature descriptor is generated. First, a 16×16 neighbourhood window is constructed around a keypoint and then divided into sixteen 4×4 blocks. An 8-bin orientation histogram is computed in each block. The generated descriptor vector consists of all histogram values, resulting in a vector of $16 \times 8 = 128$ numbers. The 128-dimensional feature vector is normalised to make it robust from rotational and illumination changes.

SIFT Hardware Implementation

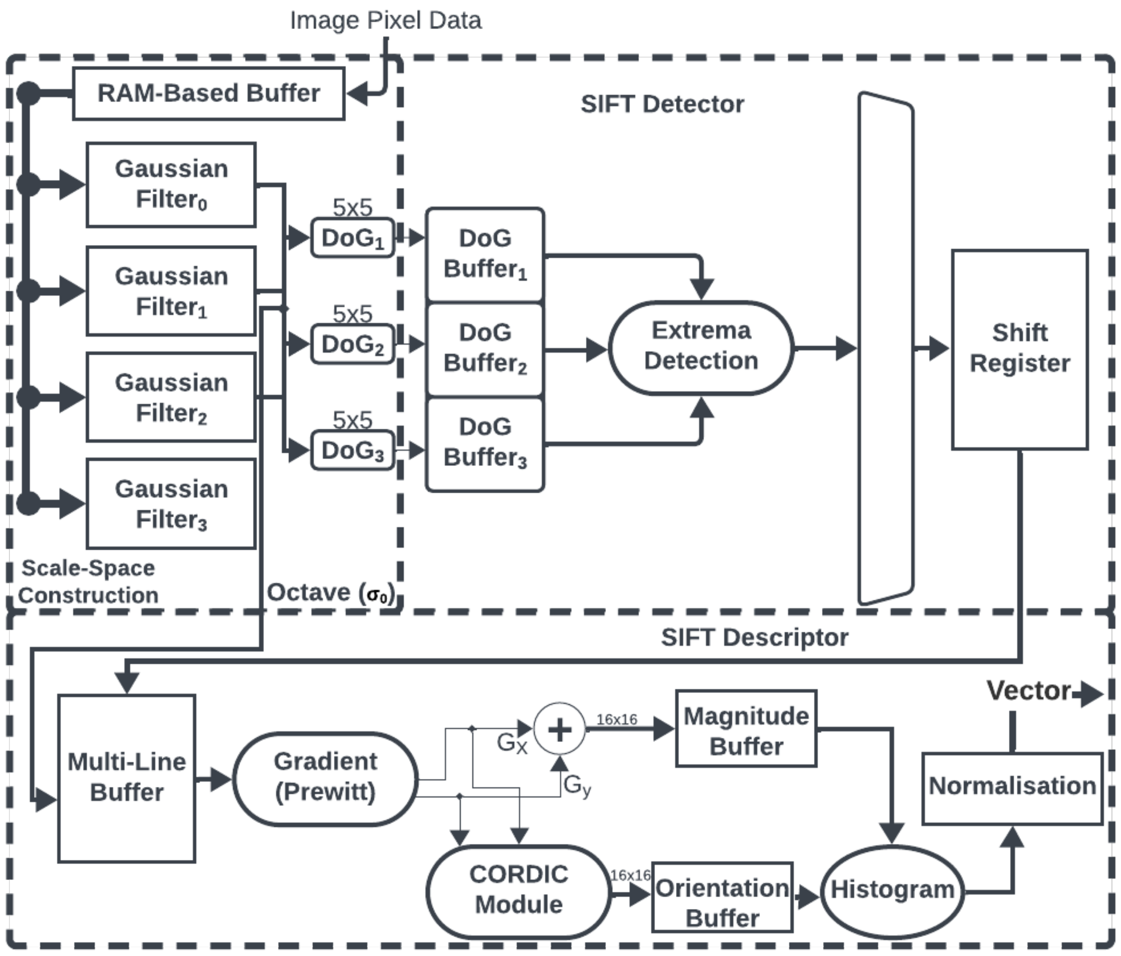


Figure 5.4: High-level block diagram of the SIFT algorithm on FPGA

The high-level hardware block diagram depicted in Fig. 5.4 of SIFT illustrates the individual modules that build up the complete algorithm.

Scale-Space Construction:

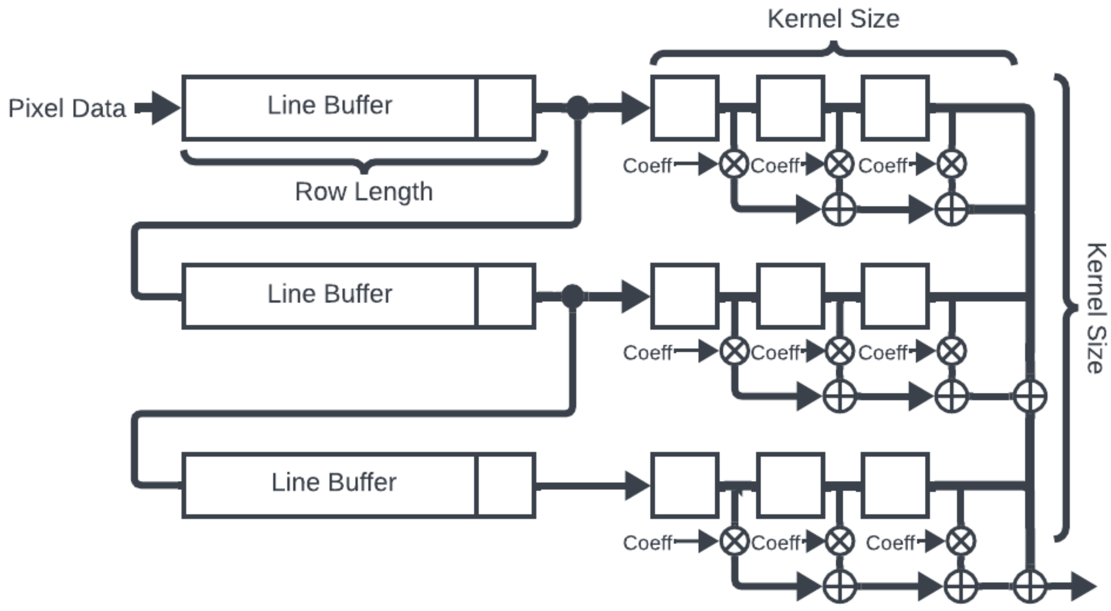


Figure 5.5: Gaussian Convolution Module Block Diagram.

To efficiently process the image data, the Gaussian convolution module observed in Fig. 5.5 uses line buffers to store and process pixel data. As pixel data streams into the FPGA, it is stored in the buffers organised in rows corresponding to image rows. These line buffers hold a portion of the image data needed to compute the convolution operation. Using line buffers, the FPGA can process multiple pixels simultaneously, reducing latency and increasing throughput. Furthermore, pre-calculated coefficients defining the approximated Gaussian weights are applied to the pixel values during filtering and are stored in BRAM registers. The implementation uses separable filters, the module first applies a one-dimensional Gaussian filter in the horizontal direction followed by another in the vertical direction.

In addition, handling boundary pixels is vital to preserve image integrity. However, in this implementation focused on a 1920×1080 image size, boundary pixels are neglected to simplify hardware design.

Subsequently, adjacent levels of the Gaussian pyramid are subtracted from each other to obtain the DoG pyramid. This subtraction operation is preformed in parallel between each scale and the three resulting DoG are buffered. Two row buffers are used for every DoG and form a 3×3 neighbourhood for extrema detection. Fig. 5.6 illustrates the design of maxima detector module

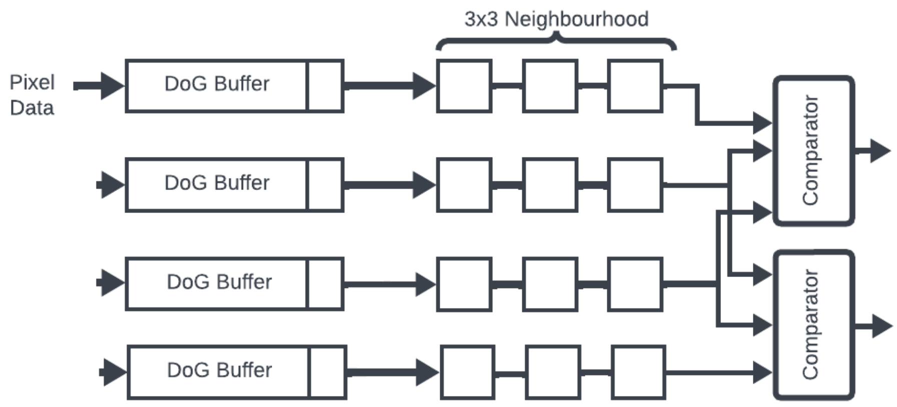


Figure 5.6: Extrema Detection Module Block Diagram.

where each pixel is then compared to its 26 neighbors, and the minimum magnitude is computed to determine the local extremum. Each DoG buffers output consists of three values that constitute one column of a 3x3 neighbouring window. The DoG words are forwarded to a comparator circuit which compares the middle pixel of DoG2 with its neighbours, and an OR gate indicates if it's an extremum. Note that the first row is processed on the fly without requiring buffers, since DoG operation is a single-pixel operation and doesn't affect the boundaries. The width of these buffers depends on the range of the DoG operation results.

CORDIC & Prewitt Mask Module:

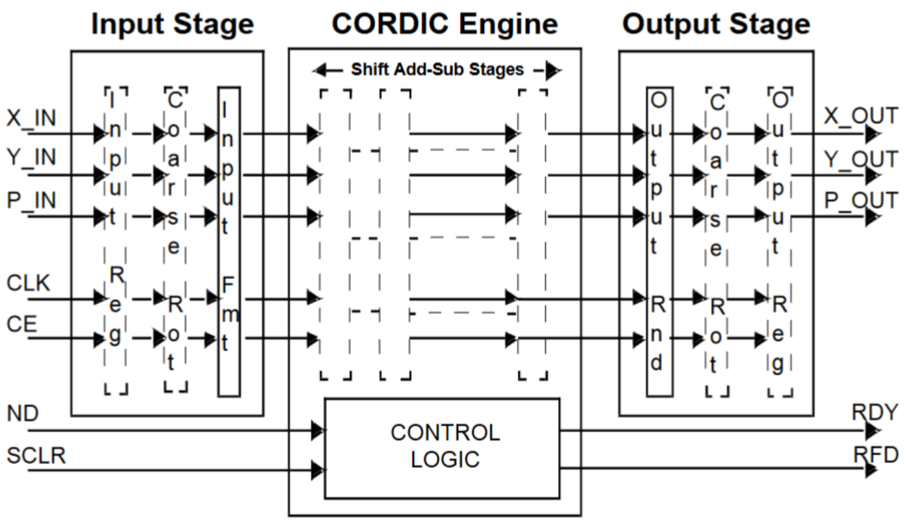


Figure 5.7: CORDIC Module Block Diagram. [1]

In the proposed implementation, the first derivatives of G2, are produced by applying Prewitt mask operator to generates the first-order derivatives of the image with respect to both the x and y directions. The magnitude is efficiently computed using the sum of absolute values of G_x^2 and G_y^2. This approach replaces the square root operation traditionally used in gradient magnitude calculations. After computation, the result is stored as 8 bits, retaining only the integer part of the magnitude for subsequent calculations. The gradient orientation is computed traditionally by using large Look-Up Tables (LUTs) for precomputed arctangents and hardware resources for division operations. However, the implementation uses Xilinx IP CORDIC module, which solves trigonometric equations iteratively and also computes a broader range of equations, including the hyperbolic and square root. The output orientation assignments are stored as a 6-bit integers.

Histogram & Normalisation Module:

In the SIFT descriptor generation stage, each keypoint contributes to histogram entries representing gradient orientations and magnitudes within a local region. Keypoints may influence multiple orientation bins, leading to limited parallelisation. To address this, the descriptor computation utilises eight BRAMs, each dedicated to a specific orientation bin. These arrays independently accumulates data from keypoints associated with a particular orientation, enabling parallel processing. Keypoint coordinates and orientation information are used to distribute magnitude contributions across the arrays. The non-data dependent nature of each array allows for pipelined accumulation and simultaneous processing of multiple keypoints. After processing all the keypoints, an adder tree sums histogram values stored in the array.

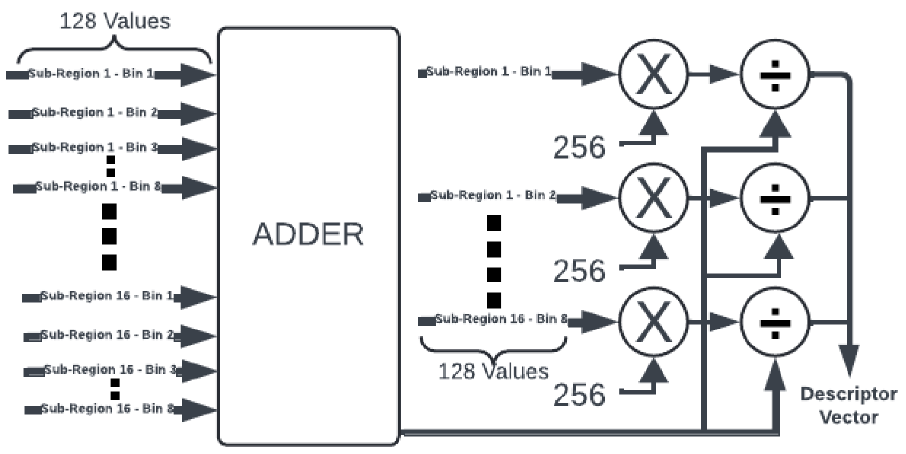


Figure 5.8: Descriptor Normalisation Block Diagram

The unnormalised histogram data is streamed into the final module shown in Fig. 5.8 that performs normalisation. The input data is converted into floating point representation, to ensure accuracy and comparability with other hardware architectures. Once, converted the data is stored into BRAM and L1-norm of the histogram is calculated concurrently using DSPs. Each entry in the histogram is subsequently divided by the computed L1-norm, followed by a square root operation. The resulting normalised values are then converted from floating-point to fixed-point representation (8 bit) to minimise storage space. Furthermore, the outputs from each processing element are accumulated into a unified output vector using a combination of FIFO buffers and a multiplexer.

5.2.2 Digital Filters

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

(a) Box

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

(b) Gaussian

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

(c) Sobel X & Y

Figure 5.9: Example approximated 3×3 image filter kernels.

Digital filters are a tool in image processing to extract useful information from noisy signals. They are commonly used for tasks such as smoothing, edge detection, and feature extraction. Filters operate by applying a kernel, or a small matrix of values, to each pixel of an image. The kernel is convolved with the image, and the resulting output value is placed in the corresponding pixel location of the output image shown in the Eq. (5.10). $I(x, y)$ is the input image and $K(k_x, k_y)$ is the kernel. The convolution result $O(x, y)$ is calculated by:

$$O(x, y) = \sum_{k_x} \sum_{k_y} I(x - k_x, y - k_y) \cdot K(k_x, k_y) \tag{5.10}$$

The indices k_x and k_y correspond to the coordinates of the kernel K, x and y correspond to the coordinates of the output image O.

5.2.3 Convolutional Neural Network

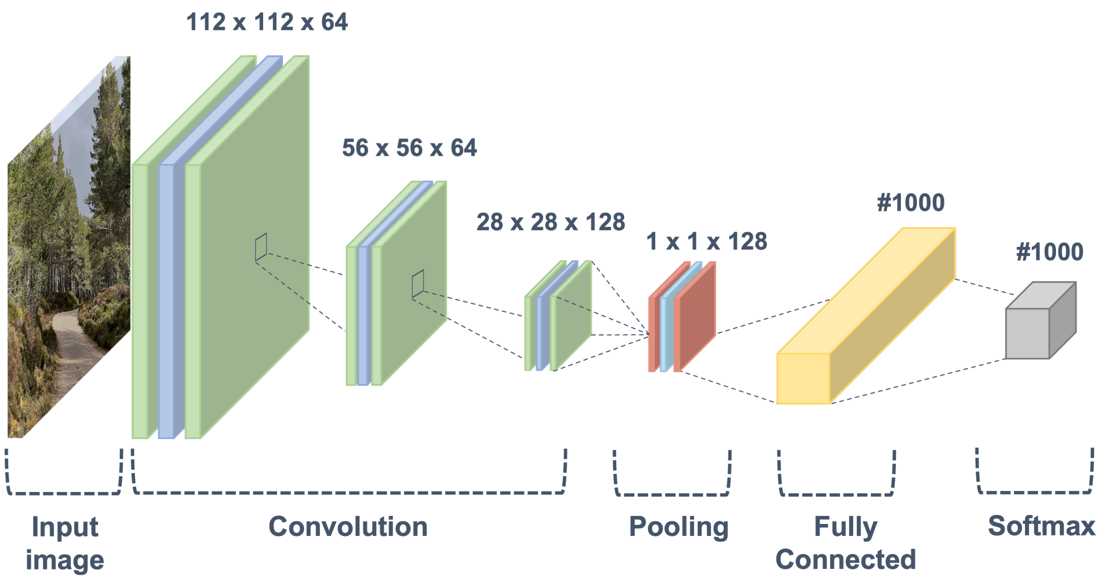


Figure 5.10: Typical layers implemented within CNN Architectures.

Convolutional Neural Networks are a class of deep neural networks typically applied to images to recognise and classify particular features. A CNN architecture typically consists of a combination of convolution, pooling, and fully connected layers shown in Fig. 5.10.

The convolution layers extract features by applying a convolution operation to the input image using a set of learnable filters (also called kernels or weights) designed to detect specific features. The output of the convolution operation is a feature map, which is then passed through a non-linear activation function, such as ReLU, to introduce non-linearity into the network. The convolutional layers can be stacked to form a deeper architecture, where each layer is designed to detect more complex features than the previous one. In addition, it is the most computationally intensive layer because each output element in the feature map is computed by repeatedly taking a dot product between the filter and a local patch of the input, which results in a large number of multiply-add operations.

The pooling layers are responsible for reducing the spatial size of the feature maps while retaining important information. The most common types of pooling are max pooling and average pooling. These layers typically use a small window that moves across the feature map and selects the maximum

Table 5.1: Summary Table: Hardware/Software Environment & Measurement Tools

Architecture	Hardware		Software /Libraries	Power
	Model	Clock		Measurement
CPU	AMD 5900x	4.8 GHz	Pytorch 2.0 [24] / OpenCV	HWMonitor [158]
GPU	Nvidia GTX 3070	1730 MHz	Pytorch 2.0 / OpenCV	Nvidia-smi [159]
FPGA	Xilinx ZCU102	300Mhz	Vivado 2022.2 / Vitis 2020.2	MaxPower-tool [170] / Power Analyser

or average value within the window. This operation effectively reduces the number of parameters in the network and helps to reduce overfitting.

The fully connected layers make predictions based on the extracted features. These layers take the output from the convolutional and pooling layers and apply a linear transformation to the input, followed by a non-linear activation function. The fully connected layer usually has the same number of neurons as the number of classes in the dataset, and the output of this layer is passed through a softmax activation function to produce probability scores for each class. A CNN architecture also includes normalisation layers such as batch normalisation, dropout layers that are used to regularise the network and reduce overfitting, and an output layer that produces the final predictions.

5.3 Experimental Results and Discussion

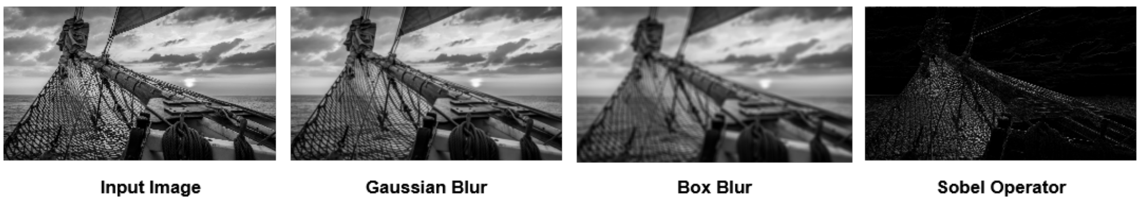


Figure 5.11: Filter Algorithms Applied onto Input Image

We verify the proposed optimisations on 'SIFT', 'Box', 'Gaussian' and 'Sobel' (in Fig. 5.11) algorithms, as well as MobileNetV2 and ResNet50 CNN architectures. This is achieved by creating baseline benchmarks on four target hardware CPU, GPU and FPGA, followed by the realisations of the optimisations individually and combined. The CPU and GPU versions for Filter and SIFT algorithms are implemented using *OpenCV* [23]. Pytorch library is used to implement CNN architectures (ResNet50 & MobileNetV2) and optimisations. Additionally, both architectures are pre-trained on the image-net classification dataset. The FPGA implementation for all algorithms is developed using Verilog (SIFT/Filter) and HLS (CNN). All baseline algorithms and CNN models use floating point 32 (FP32), and an uncompressed grayscale 8-bit 1920×1080 input image is used for the SIFT algorithm, and each sub-operation is profiled. Details of the target hardware/software environments and power measurement tools are given in Table 6.1.

Dataset. The input images used in the CNN and Filter experiments are from LIU4K-v2 dataset [171]. The dataset contains 2000 high resolution 3840×2160 images with various backgrounds and objects.

5.3.1 Performance Metrics

As part of the evaluation process, we measure three different performance metrics, namely, *1) execution time*, *2) energy consumption* and *3) accuracy*.

Execution time

The execution time measured for the CPU and GPU platforms uses time function libraries to count the smallest tick period. Each algorithm/operation is run for 1000 iterations and averaged to minimise competing resources or other processes directly affecting the architecture, especially for the CPU architecture. The GPU has an initialisation time which is taken into account and removed from the results. The timing simulation integrated into Vivado design suite software is used to measure the time for the FPGA platform. The experiments exclude the time of both the image read and write from external memory. We compute the frame per second (FPS) as the inverse of the execution time:

$$\text{FPS} = 1/\text{Execution Time}. \tag{5.11}$$

Power Consumption

Two common methods used for measuring power are software and hardware-based. Accurately estimating power consumption is a challenge using software-based methods, which have underlying assumptions in their models and may not measure other components within the platform. In addition, taking the instantaneous watt or theoretical TDP of a device is not accurate since power consumption varies on the specific workload. Therefore, we obtain the total energy consumed by measuring the power over the duration of the algorithm executed. A script is developed to start and stop the measurements during the execution of the algorithm and extract the power values from the software.

With the use of a power analyser within the Vivado design suite and the MaxPower-tool, the measurement of FPGA power consumption is divided into two parts, *(1)* static power and *(2)* dynamic power. Static power relates to the consumption of power when there is no circuit activity and the system remains idle. Dynamic power is the power consumed when the design is actively performing tasks. The power consumption for the CPU and GPU is obtained using *HWMonitor* and *Nvidia-smi* software. To have a fair comparison across the target hardware for the SIFT algorithm, we normalise it as the energy per operation (EPO):

$$\text{Energy} = (\text{Power} * \text{Execution Time}). \tag{5.12}$$

Additionally, We calculate the energy consumption for the Filter and CNN algorithms:

$$\text{EPO} = (\text{Power} * \text{Execution Time})/\text{Operations}. \tag{5.13}$$

Accuracy

With an expectation that the optimisations impact overall algorithmic accuracy, we capture it by measuring the *Euclidean distance* between the descriptors generated from the CPU (our comparison benchmark) to the descriptor output produced by the FPGA. The Euclidean distance $d(x, y)$ is calculated in

Eq. (5.14) where x and y are vectors, and K is the number of keypoints generated. This accuracy measurement is only used for the SIFT algorithm implementation.

$$d(x,y) = \sqrt{\sum_{i=1}^{K}(x_i - y_i)^2}. \tag{5.14}$$

Subsequently, the accuracy for each Euclidean distance is calculated using Eq. (5.15):

$$\text{Accuracy} = 100 - \left(\left(\frac{\text{Euclidean Distance}}{\text{Max Distance}}\right) \times 100\right) \tag{5.15}$$

The *Euclidean Distance* denotes the distance between the two descriptor vectors being compared, and *Max Distance* represents the maximum Euclidean distance found in the vector. The accuracy is transformed to have 100% indicate identical descriptors, while 0% indicates completely dissimilar descriptors.

We used root mean square error (RSME) to compare the input image to the output images produced by each hardware accelerator to determine the pixel accuracy. RMSE is defined as:

$$RMSE = \sqrt{(\frac{1}{n})\sum_{i=1}^{n}(y_i - x_i)^2} \tag{5.16}$$

Where the difference between the pixel intensity values of output and input (y_i,x_i) images. Divided by N, which is the total number of pixels in the image.

The accuracy of the CNN architecture is measured by taking the number of correct predictions divided by the total number of predictions:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \times 100 \tag{5.17}$$

A high accuracy indicates that the model is making accurate predictions, while a low accuracy suggests room for improvement in the model's performance.

5.3.2 Results and Discussions

The results and discussions section contains the evaluation of algorithms in three categories, feature extraction algorithms (*SIFT*), filter algorithms (*Box, Gaussian, Sobel*) and Convolution Neural Networks (*MobilenetV2, ResNet50*).

Table 5.2: SIFT: Performance against state-of-the-art

	Octave, Scale	Hardware Platform	Image Size	Clock (Mhz)	Frame Rate (FPS)
Chiu [172]	2,4	Virtex-6	640×480	100	30
Mizuno [173]	2,4	65 nm CMOS	1920×1080	N/A	30
Vourvoulakis [174]	1,4	Cyclone IV	640×480	21.7	70
Proposed	2,4	Zynq UltraScale+	1920×1080	300	50
	2,4	Virtex UltraScale+	1920×1080	600	**100**

SIFT

We obtain results for FPGA implementations of the SIFT algorithm, considering various optimisations or combinations of them. Two sets of results are captured for *octave, scale* of (2,4) and (4,5) as they are regularly reported in the literature for SIFT implementation on FPGA. The results are primarily obtained at a target frequency of 300 MHz for various components of SIFT and execution time and accuracy are reported in Table 5.3 along with FPS numbers in Fig. 5.12.

In terms of individual optimisations on the base FPGA implementation, *down sampling* and *integer* optimisations had the most reduction of accuracy but in trade for a greater reduction of runtime. On the other hand, 3×3 *kernel size* (down from default 5×5) had better accuracy results but with a small improvement on the overall runtime. In the case of combined optimisations, both *down sampling* and *integer* combinations greatly reduced the execution times but at a cost of $8 \sim 10\%$ accuracy loss. In the most optimised case, (4,5) and (2,4) configurations achieved 17 and 50 fps, at an accuracy of 90.18% and 89.45%, respectively. The $10 \sim 11\%$ loss in accuracy in both configurations can be attributed to the loss of precision and pixel information resulting in imperfection in feature detection.

The comparison with optimised CPU and GPU implementations is shown in

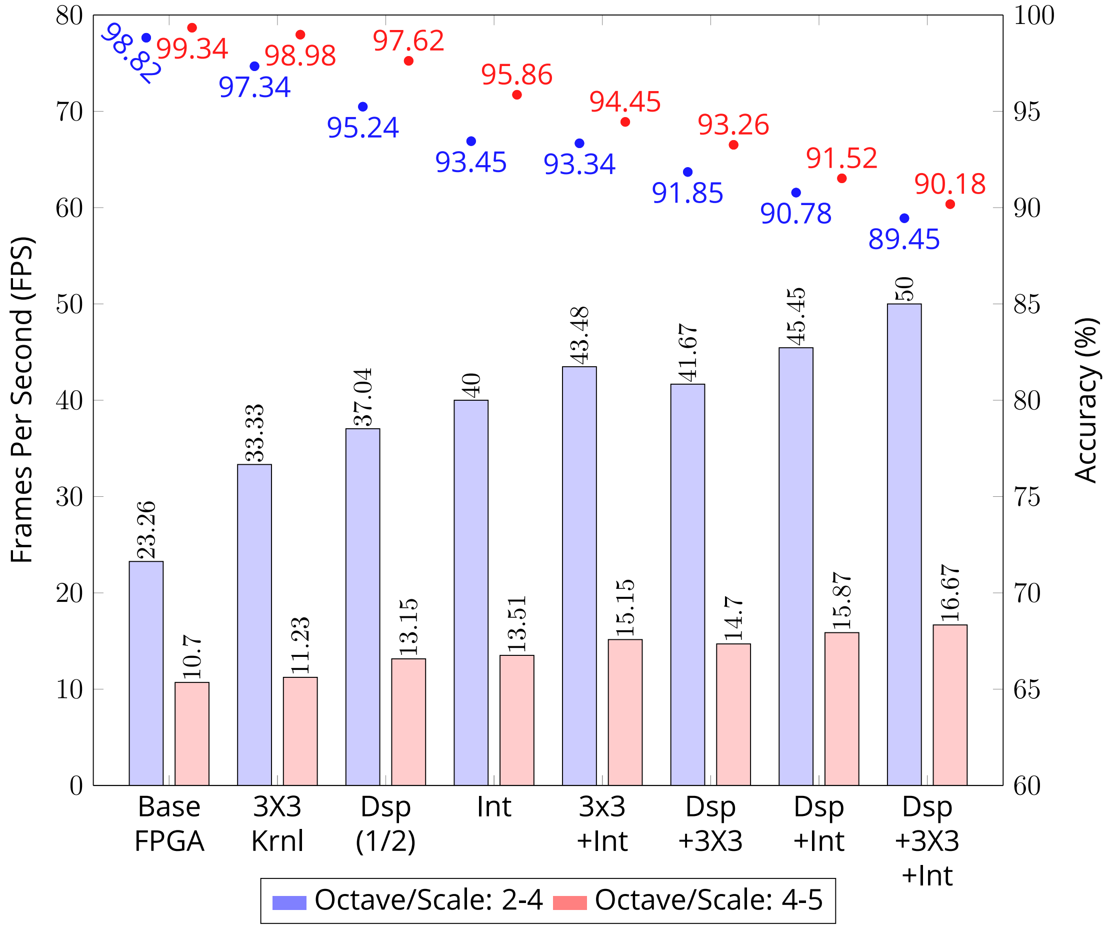


Figure 5.12: SIFT: FPS (Bars) and Accuracy (Dots) for each optimisation on both configurations (octave, scale).

Table 5.5 which includes total execution time and energy consumption per operation (nJ/Op). Results indicate the optimised FPGA implementation achieved comparable GPU runtime at 600 MHz but significantly outperformed them when energy consumption statistics are taken into account. The GPU results excluded the initialisation time, which would add greater latency to the overall runtime. In addition, the power consumption of the GPU is at 12.47nJ/Op, which would make it a difficult choice for real-time embedded systems. On the other hand, optimised FPGA implementations have better performance per watt than the GPU and CPU. The comparison with the state-of-the-art FPGA implementations is reported in Table 5.2, and results show major improvements in the runtime even with larger image size and more or similar feature points (~ 10000). Finally, for completeness, we report the resource and power usage statistics for optimised configurations at 300 MHz in Table 5.4.

Filter Implementations

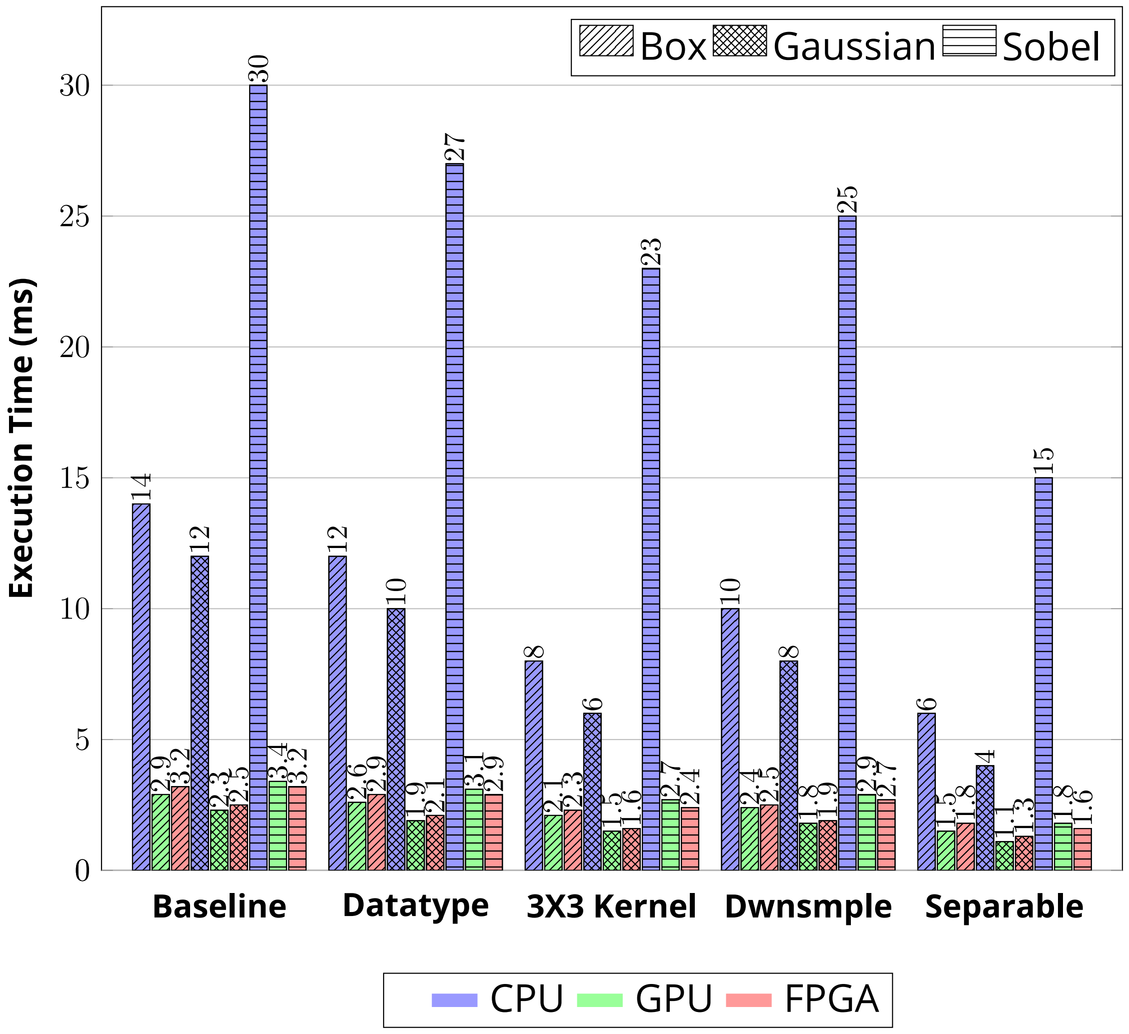


Figure 5.13: Filter: Runtime comparison for optimisations applied on each architecture.

Fig. 5.14 & Fig. 5.13 plots the runtime and energy consumption of three image processing filter algorithms (*Box*, *Gaussian*, and *Sobel*) with various optimisations strategies applied to the baseline algorithm. Comparing the baseline performance, the CPU architecture suffers the most in execution time and energy consumption which can be attributed to the lack of many compute cores. In contrast, GPUs and FPGAs exploit data parallelism and stream/pipeline processing to significantly reduce runtime.

Both Fig. 5.14 & Fig. 5.13 show that the performance of both GPU and FPGA are comparable in both metrics studied. The GPU demonstrated a marginally

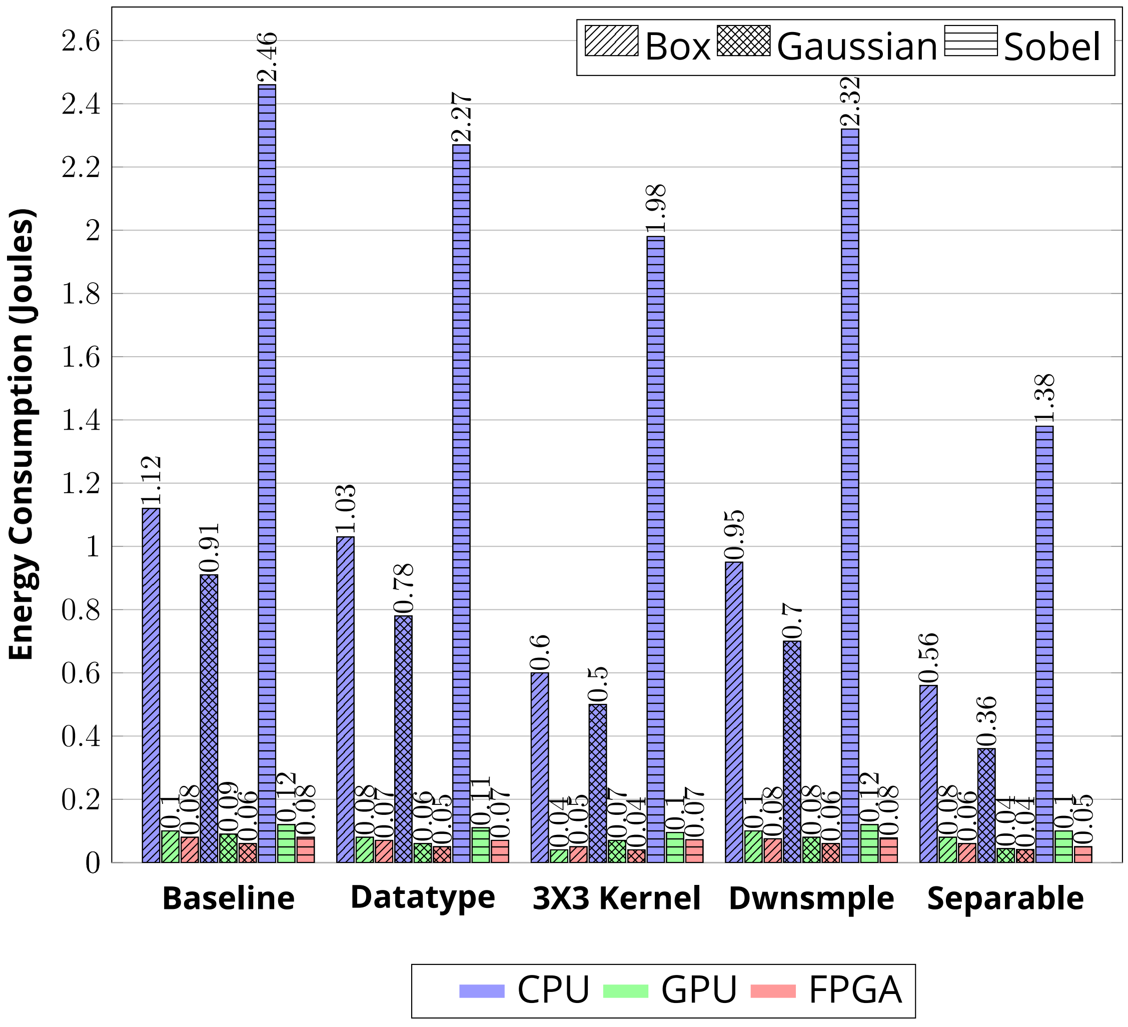


Figure 5.14: Filter: Energy consumption comparison for optimisations applied on each architecture.

better computation speed compared to the FPGA, with an average execution time improvement of 9.45% for *Box* and *Gaussian* algorithms. On the other hand, the FPGA has a 5.88% improvement for *Separable Filter* over GPU.

In the case for *Sobel*, the FPGA achieves a speedup of approximately $1.09\times$ over the GPU across all optimisation strategies. The smaller kernel size allows the FPGA to use its DSP slices to efficiently compute the algorithm, whilst the GPU operations do not fully occupy the compute resources available which results in load imbalance and communication latency. Relative to power consumption, the CPU experiences the most significant impact in all cases by consuming $1.19\times$ Joules on average. The higher energy usage can be attributed to their higher clock, complex memory hierarchy and lack of parallel capability.

The GPU has been observed to consume $\sim 1.36\times$ more Joules than the FPGA. The high energy cost can be derived from the base support/unused logic components consuming static power. However, the FPGA operates on the least energy consumed per clock due to its custom-written nature. The energy consumption for *seperable* optimisation reveals similar results between the GPU and FPGA. This is attributed to the GPU able to compute the algorithm faster to offset the higher clock speed energy cost.

All optimisations, e.g *Datatype*, *Kernel*, *Downsampling*, and *Separable Filters* optimisations had major improvements for each accelerator. Reducing the kernel size to 3×3 *kernel size* and applying *Separable Filters* had the most impact due to lowering the number of operations computed during the convolution operation. The *Datatype* and *Downsampling* optimisations had around on average $12.24 \sim 28.16\%$ decrease in runtime for all algorithms. The optimisation runtime results and accuracies of each filter algorithm are reported in Table 5.6. In terms of image accuracy, downsampling has the most significant difference compared to the original, which can be attributed to the rows that are removed due to the algorithm.

CNN Architecture

Fig. 5.15 displays the runtime performances and classification accuracy of the baseline and optimised CNN algorithms on each hardware architecture. The results show that the CPU, GPU, and FPGA exhibit similar levels of performance, with the GPU having an average improvement of $5.41 \sim 12\%$ over the FPGA for the *Downsampling* optimisation in *MobileNetV2* and the baseline for *ResNet50*, respectively. The FPGA leads in the *Datatype* optimisation over the GPU with a $6.25 - 11.1\%$ reduction in time for both CNNs. The *Datatype* optimisation involves quantisation of the model's weights from FP32 to 8-bit to reduce complexity. The FPGA computes the quantised operations faster on both architectures due to exploiting the DSP blocks and requiring no additional hardware logic for floating-point arithmetic. However, the quantised model weights are unable to represent the full range of values present in the input image, resulting in a $\sim 10\%$ accuracy loss for all platforms. The *Downsampling* strategy has a slight improvement in runtime with minimal impact on the accuracy, with a loss around $\sim 5\%$.

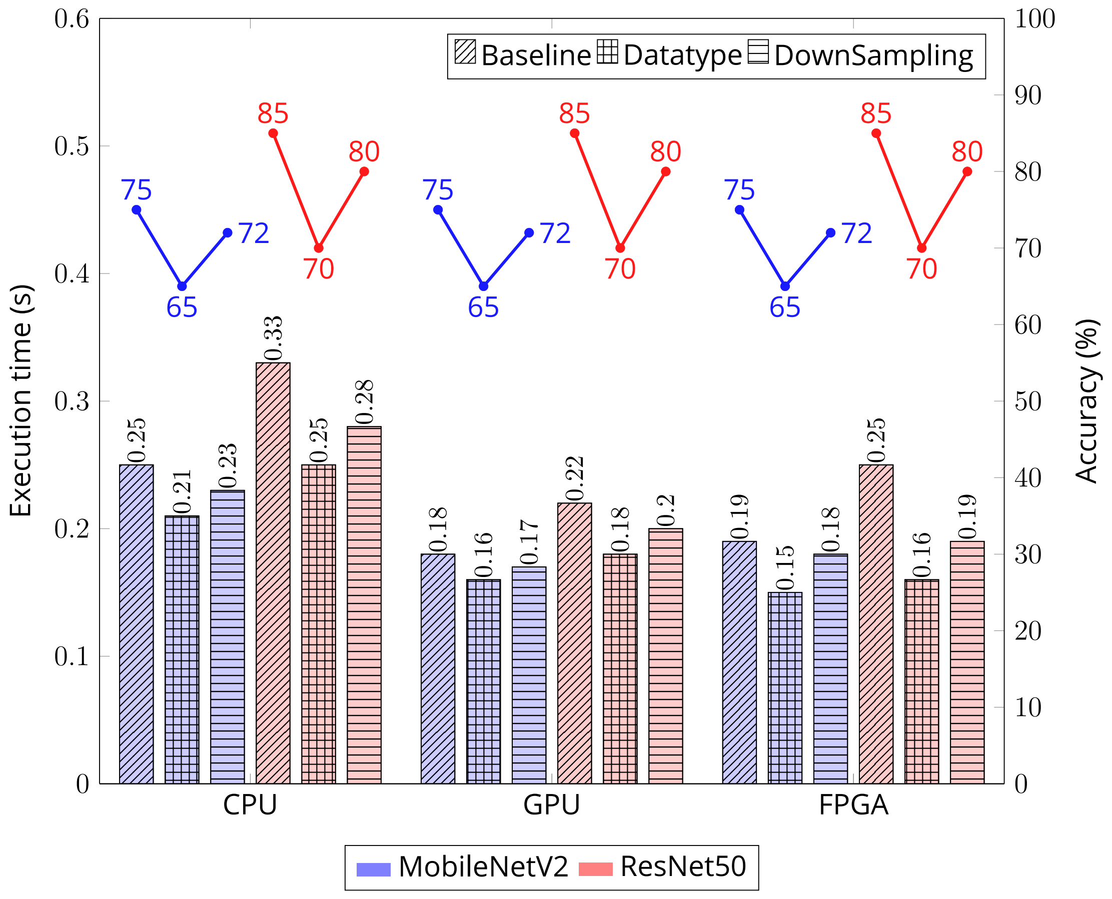


Figure 5.15: CNN: The graph compares execution time and accuracy across three optimisation strategies (**Baseline**, **Datatype**, **DownSampling**) on CPU, GPU, and FPGA for two neural network models (*MobileNetV2* and *ResNet50*).

In Fig. 5.16, the energy consumption graph shows that the CPU consumes on average, $3.14\times$ more energy than the other accelerators for both CNNs. In addition, the *ResNet50* architecture has more layers than *MobileNetV2* and, therefore, contains more operations, resulting in higher energy usage. In all cases, the FPGA consumes the least amount of energy, $1.11 \sim 3.55\times$ less than the CPU and GPU, to compute the image classification. The results show the potential of reducing the computation time of CNNs by further applying the proposed optimisations in a layer by layer basis, but at the cost of slight accuracy loss. The optimisation results of each CNN architecture and accuracy are reported in Table 5.7.

Consequently, larger images or complex networks with many layers and larger filter sizes require more memory to store the weights and activations. This leads to higher memory requirements, especially within real-time em-

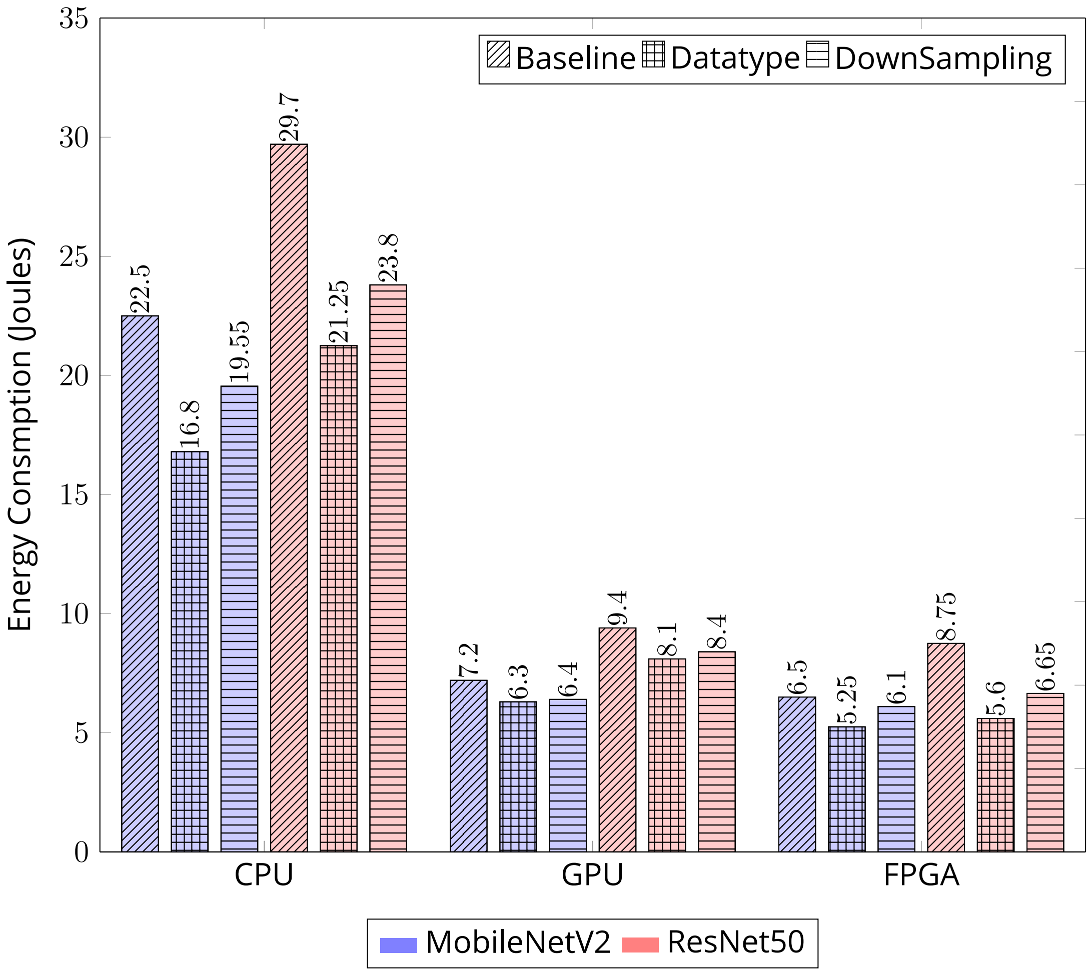


Figure 5.16: CNN: Architecture Energy comparison of Model *Datatype* & Input Image *Downsampling* Optimisations on ResNet50 and MobilenetV2.

bedded systems where space is limited. However, applying optimisations can alleviate the computational load, but careful consideration must be taken to understand the trade-offs between runtime and accuracy depending on the application. The *Kernel* optimisations could not be implemented on the chosen CNN architectures, which would require heavily modifying the standard convolution operation in both networks. This would completely change the network, thus creating a new architecture.

5.4 Conclusion & Future Direction

This chapter proposes new optimisation techniques called *domain specific optimisation* for real-time image processing on FPGAs. Common image processing algorithms and their pipelines are considered in proposing such optimisations, which include down/subsampling, datatype conversation and convolution kernel size reduction. These were validated on the popular image processing algorithms and convolution neural network architectures. The optimisation results for CNN and Filter algorithms vastly improved the computation time for all processing architectures. The SIFT algorithm implementation results significantly outperformed state-of-the-art SIFT implementations on FPGA and achieved runtime at par with GPU performances but with lower power usage. However, the optimisations on all algorithms come at the cost of $\sim 5-20\%$ accuracy loss. Overall, *Downsampling* and *datatype* optimisation resulted in the most significant reductions in execution time and energy consumption.

The results demonstrate that applying domain-specific optimisations to increase computational performance while minimising accuracy loss demands in-depth and thoughtful consideration. Furthermore, it should be noted that the optimisations selected in the experiment are non-exhaustive, leaving room for further exploration.

One proposal for algorithms comprising multiple operation stages is to use adaptive techniques instead of fixed integer downsampling factors, bit-widths, and kernel sizes. These adaptive methods analyse the data and dynamically adjust the level of optimisation based on input characteristics. For instance, adjusting the bit-width and downsampling factor according to the specific input data within each stage can yield better results and strike a more suitable trade-off between performance and accuracy. Several strategies can be employed in the CNN domain to address the challenges. Quantisation-Aware Training (QAT) and mixed-precision training enable the model to adapt to lower precision representations during training, reducing accuracy loss during inference with quantised weights. Additionally, selective downsampling and kernel size reduction of CNN architectures help retain relevant information and preserve accuracy. Channel pruning can further offset accuracy loss by removing redundant or less critical channels. As a result, employing these strategies and considering hardware constraints makes it possible to strike an optimal balance between accuracy and performance, unlocking the full potential of efficient applications.

On the other hand, the drawback of traditional libraries and compilers is that they often struggle to keep pace with the rapid development of deep learning (DL) models, leading to sub-optimal utilisation of specialised accelerators. To address the limitation, adopting optimisation-aware domain-specific languages, frameworks, and compilers is a potential solution to cater to the unique characteristics of domain algorithms (*e.g.*, machine learning or image processing). These tool-chains would enable algorithms to be automatically fine-tuned, alleviating the burden of manual domain-specific optimisation.

Table 5.3: SIFT: Optimisation Result Summary on FPGA, 300 Mhz Configuration (Octave, Scale).

Operations	Gaussian		Extrema		Orientation		Descriptor		Total Runtime (ms)		Overall Accuracy (%)	
	(2,4)	(4,5)	(2,4)	(4,5)	(2,4)	(4,5)	(2,4)	(4,5)	(2,4)	(4,5)	(2,4)	(4,5)
Baseline FPGA (ms)	19	45	10	18	9	16	5	14	43	93	98.82%	99.34%
Downsampling	13	40	4	13	5	10	5	13	27	76	95.24%	97.62%
Integer Arithmetic	11	38	4	14	5	8	5	14	25	74	93.45%	95.86%
3×3 **Kernel**	14	43	6	15	5	14	5	14	30	86	97.34%	98.98%
Downsampling + Integer	9	38	4	8	4	7	5	10	22	63	90.78%	91.52%
Downsampling + 3×3	9	38	5	12	5	8	5	10	24	68	91.85%	93.26%
Integer + 3×3	9	36	5	11	4	9	5	10	23	66	93.34%	94.45%
Downsampling + Integer + 3×3	8	36	3	8	4	6	5	10	**20**	**60**	89.45%	90.18%

Table 5.4: SIFT: Resource Usage Summary on FPGA of all Optimisations *Downsampling*, $3 \times 3 Kernel$ & *Integer Arithmetic* Configuration.

Configuration	LUTs	Registers	BRAM	DSP	Power Usage (Watts) Dynamic/Static
(2,4)	42.11%	14.32%	21.38%	5.36%	10.324/0.97
(4,5)	43.94%	15.38%	23.30%	6.51%	17.343/0.99

Table 5.5: SIFT: Profiling Summary on each Hardware Platform. *Baseline* & Optimised (Octave, Scale).

Operation (ms)	CPU (4,5)	GPU (4,5)	Optimised FPGA (4,5)	Baseline FPGA (4,5)	Optimised FPGA (2,4)	Baseline FPGA (2,4)	Optimised FPGA (2,4) (600Mhz)
Gaussian Pyramid	1118	3	36	45	8	19	4
Extrema Detection	133	2	8	18	3	10	3
Orientation \& Magnitude Assignment	128	1	4	16	4	9	2
Descriptor Generation	50	1	10	14	5	5	1
Total Execution Time (ms)	1429	**7**	60	93	20	43	10
Energy Consumption (nJ/Op)	1620	12.47	4.09	7.34	***2.41***	5.82	4.61

Table 5.6: Image Processing Filters Summary & Accuracy (RSME) Result Summary.

Algorithm	Baseline Runtime (ms)			Optimised Algorithm Runtimes (ms)											
				Datatype (INT)			Kernel (3x3)			Downsampling			Separable Filter		
	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA
Box Filter (50x50)	14	2.9	3.2	12	2.6	2.9	8	2.1	2.3	10	2.4	2.5	6	**1.5**	1.8
Gaussian Filter (31x31)	12	2.3	2.5	10	1.9	2.1	6	1.5	1.6	8	1.8	1.9	4	**1.1**	1.3
Sobel Filter (7x7)	30	3.4	3.2	27	3.1	2.9	23	2.7	2.4	25	2.9	2.7	15	1.8	**1.6**
Accuracy (RSME)															
Box Filter (50x50)	8.11	9.14	10.92	8.78	9.21	9.44	3.37	4.54	5.94	130.21	135.13	145.15	9.34	10.23	10.62
Gaussian Filter (31x31)	6.68	8.45	10.26	6.69	7.59	7.64	4.01	5.27	6.89	145.4	149.42	164.22	7.83	8.16	8.53
Sobel Filter (7x7)	10.25	11.43	12.45	10.28	11.42	11.88	10.16	11.35	12.14	130.41	132.78	148.21	11.32	12.42	12.64

Table 5.7: CNN Optimisation Result Summary: Runtimes and Image Classification Accuracy for Baseline and Optimisations Applied on each Hardware.

Algorithm	Baseline Runtime (s)			Optimisations					
				Datatype (INT8)			Downsampling		
	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA
MobileNetV2	0.25	0.18	0.19	0.21	0.16	**0.15**	0.23	0.17	0.18
ResNet50	0.33	0.22	0.25	0.25	0.18	**0.16**	0.28	0.20	0.19
Energy Consumption (Joules)									
MobileNetV2	22.5	7.20	6.50	16.8	6.30	5.25	19.55	6.40	6.1
ResNet50	29.7	9.4	8.75	21.25	8.1	5.6	23.8	8.4	6.65

6 Image Processing Algorithms on Heterogeneous Platforms

Feature extraction algorithms and Convolutional Neural Networks (CNN) are widely used in various problem domains, such as object detection, image classification, and segmentation. Feature Extraction are typically designed to identify and extract relevant patterns or features from the input data. These algorithms are typically designed and implemented on DSPs or GPUs, which offer a specialised architecture properties which include thousands of general compute cores and coupled with a large amount of high-bandwidth memory.

In the case of CNNs that contain millions of parameters, which in turn require significant memory resources to store their weights, implementing them on low-resource and energy constrained platforms is limited. Heterogeneous platforms solves these constraints by utilising various specialised processors, such as CPUs, GPUs, TPUs, and FPGAs, to process specific operations. Leveraging the advantages of true heterogeneous architectures, run-time and power efficient designs can be realised by exploiting architectures with sufficient resource and processing capacity.

However, partitioning algorithms remain an arduous task, particularly when distributing operations among various accelerators within heterogeneous architectures. This process necessitates a delicate balance, taking into account critical factors such as computational power, memory bandwidth, and communication overhead. Furthermore scheduling tasks presents its own set of challenges. Scheduling involves determining the order in which tasks are executed on different processing units, considering factors such as task dependencies, resource availability, and load balancing. Therefore, a scheduler is required for managing hardware resources efficiently, controlling concurrency, balancing workloads, prioritising tasks while adapting to dynamic conditions. Thus, the scheduler plays a critical role in orchestrating task execution to minimise data transfers and optimise overall system performance

The important contributions of this chapter is the development of a heterogeneous hardware and adaption of algorithms on that hardware. Starting with an extensive analysis of popular feature extraction algorithms such as SIFT and two CNN architectures, namely *ResNet18* [175] and *MobilnetV2* [176]. The feasibility of implementing these algorithms and networks onto heterogeneous systems is investigated by identifying the optimal stage in each network/algorithm to be mapped onto a specific accelerator. A comprehensive benchmarking analysis of the CNNs and SIFT is conducted by performing image classification and feature extraction on a wide range of platforms to discern the layers or stages that exhibit the highest energy consumption, inference, and total runtime. Two new heterogeneous platforms are constructed, one comprising high-performance accelerators and the other an embedded system with power-optimised processors. The algorithms and networks are implemented on both platforms using a fine-grained partitioning strategy and evaluated. Heterogeneous results are compared to the homogeneous accelerator counterparts to determine the best-performing architecture.

The main contributions of this chapter are as follows:

- Efficient deployment of CNNs and SIFT, which are computationally faster and consume less energy, is proposed.
- Partitioning methods on heterogeneous architectures are introduced by studying the features of CNNs and stages of SIFT to identify characteristics used to determine a suitable accelerator.
- Two heterogeneous platforms consisting of two configurations, high-performance and a power-optimised embedded system, are developed.
- Development of a heterogeneous scheduler to allocate tasks onto the suitable accelerator.
- Benchmarking and evaluating runtime, energy, and inference metrics of popular convolution neural networks and SIFT on a wide range of processing architectures and heterogeneous systems.

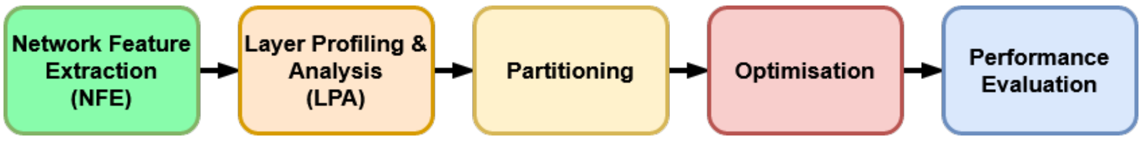

Network Feature Extraction (NFE)
Layer Profiling & Analysis (LPA)
Partitioning
Optimisation
Performance Evaluation

Figure 6.1: Development flow of Implementing CNNs on Heterogeneous Hardware

6.1 Heterogeneous Architecture

In order to implement CNNs efficiently on heterogeneous platforms, A development flow is proposed, shown in Fig. 6.1. The framework identifies the properties of networks and further profiles in accordance with their hardware suitability. Finally, optimise and evaluate the partitioning decisions.

6.1.1 CNN Development Flow:

Network Feature Extraction & Layer Profiling Analysis

Analysing the network architecture is essential in understanding the role of each layer and how they contribute to the overall performance of the model. This is accomplished by running the CNN on various hardware architectures and profiling the execution time and energy consumption of each layer. Once the resource-intensive layers have been identified by profiling, they can be deconstructed into the types of operations they perform. This deconstruction allows for a more granular understanding of the computational requirements.

Partitioning & Optimisation

To efficiently map operations onto hardware, understanding the instruction sets and memory models of different accelerators is necessary. For example, GPUs excel at performing matrix multiplication during high occupancy, while FPGAs are more power and runtime efficient for smaller matrix sizes. CPUs, with their high clock frequency and memory hierarchy, can be advantageous for sequential layers. Optimisations such as quantisation, pruning, and compression can improve performance, but the trade-offs must be carefully considered.

Performance Evaluation

To determine performance, partitioning strategies and hardware choices are benchmarked using various metrics, such as energy consumption, accuracy, inference, and throughput. Evaluating algorithms not only serves as a comparison to other architecture but also identifies areas for improvement.

The following sections, presents an overview of the representative feature extraction and CNN algorithms targeted for partitioning and implementation on heterogeneous platforms.

6.2 Scale-Invariant Feature Transform

The Scale-Invariant Feature Transform (SIFT) algorithm used to detect and describe local features in images. The SIFT algorithm is designed to be robust to changes in scale, rotation, and partial occlusion. It works in several stages: First, it identifies key points in the image through scale-space extrema detection. These keypoints are then localised more accurately and assigned an orientation. Finally, a descriptor is computed for each keypoint, capturing its local image gradient patterns. These descriptors are used for matching keypoints across different images, making SIFT useful for tasks like object recognition, image stitching, and 3D reconstruction. The in-depth algorithm and its operations are explored in the previous chapter, section 5.2.

6.2.1 SIFT Algorithm Analysis

There are several key stages in the SIFT algorithm that vary in computational complexities and hardware implications. This section discusses the complexity, memory footprint and impact of operations on CPU/GPU/FPGA hardware. The full algorithmic description of SIFT is described in Section 5.2.

Gaussian Pyramid Construction:

The Gaussian stage of the SIFT algorithm poses a considerable computational load on hardware, not only due to the intensive convolution operations but also because of the memory requirements involved. The process of generating multiple intermediate images, each corresponding to a different scale, can lead to significant memory consumption, particularly for high-resolution images. For a square image with dimensions of N x N, the complexity of the convolution operation grows quadratically with N, resulting in a substantial computational load. This process is repeated for each of the K scales, further adding to the overall computational demand. Moreover, the hardware would require high memory bandwidth for accessing large filter kernels and image matrices. In addition to the computational complexity, the memory footprint of storing multiple Gaussian-filtered images can become a bottleneck, especially when dealing with large images or a high number of scales. Architectures with higher parallelisation and memory caches are more suitable for this stage.

Extrema Detection:

The computation required to determine these extrema involves a pixel-by-pixel comparison, evaluating whether each pixel qualifies as an extremum. While this comparison is conducted for every pixel in the image. These are $O(1)$ operations for each pixel, and given the nature of the operation, lower compute resources are required. The extrema detection algorithm operates independently on each scale level of the pyramid, comparing pixels within a local neighbourhood to identify potential keypoints. Therefore, this stage can be implemented in parallel which maps well to GPUs and FPGAs.

Orientation and Magnitude Assignment

While square root and arc tangent operations are less computationally intensive compared to convolutions, they still pose a significant computational burden, especially when performed on a large number of pixels. Hardware support for fixed-point or floating-point arithmetic can significantly accelerate these calculations, as these specialised units are optimised for handling complex mathematical operations.

Descriptor Generation:

In the histogram binning and normalization stage of SIFT, local image gradients are quantised into orientation histograms and then normalised to enhance the robustness of feature descriptors. Binning involves multiplication and addition operations while normalisation involves division which requires more resource logic on hardware. However, the operation intensity is relatively lower than the previous stages which makes it suitable for all architectures.

Experimental Design

The selected profiling times for each stage and on each hardware are shown in Fig. 6.2. The runtime profiles of each SIFT stage were collected using a robust experimental methodology. The input is a greyscale 3840×2160 resolution image. The CPU and GPU implementations leveraged the OpenCV SIFT function, while the FPGA was developed using Verilog. The CPU and GPU code is executed 1000 times, and its runtimes are averaged. In the FPGA implementation, the resulting timing graphs from the simulations are used to determine the time taken for each stage. All architectures had 16-bit float precision for their respective designs and algorithms.

6.2.2 SIFT Profiling & Partitioning Strategy

The results reveal that the CPU is substantially slower in execution time than GPU and FPGA by $2\times$ on average. Even though the CPU has the highest clock speed, the lack of many processing cores results in poor for-loop unrolling optimisations for parallelisation. Comparing only GPU and FPGA, the overall *Total* runtime has shown the GPU being $0.83\times$ faster. In the *Gaussian Pyramid* and *Orientation & Magnitude* stage, the GPU outperforms the FPGA by $2\times$. On the other hand, the FPGA outperforms the GPU in the *Extrema Detect* stage by $1.5\times$. The GPU and FPGA architectures are comparable in performance when generating keypoint descriptors due to a lower amount of operations. Overall, the lower GPU runtime in most stages is attributed to having a significantly higher clock speed (*e.g.*, 1725MHz vs 300MHz) and more processing cores than the FPGA to maximise throughput.

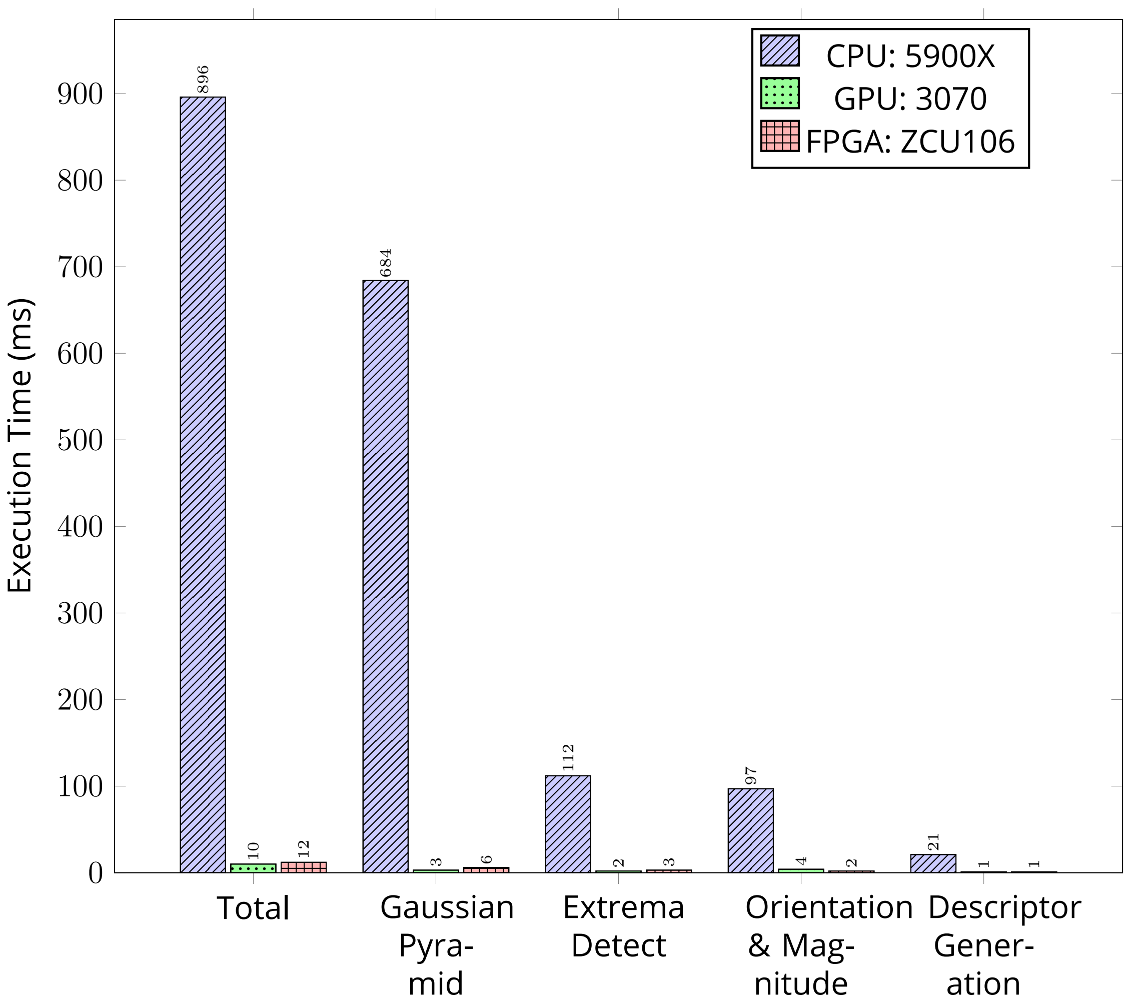


Figure 6.2: Operation Stage Run-time Profiling: SIFT.

The partitioning strategy shown in Fig. 6.3, focuses on striking a balance between potential energy consumption and the execution time of the heterogeneous platform.

The RGB to greyscale conversion is a computationally lightweight image processing operation. It involves a linear combination of the red, green, and blue colour channels with specific weightings to produce a greyscale image. The decision to execute `rgb2gray` algorithm on the CPU instead of the GPU or FPGA is due to minimising the overhead associated with transferring the image data. Although GPUs and FPGAs can perform the RGB to Gray conversion slightly faster due to their throughput processing and pipelining capabilities, the time spent moving the data to and from these platforms can negate the benefits, especially for smaller image sizes where the conversion isn't the bot-

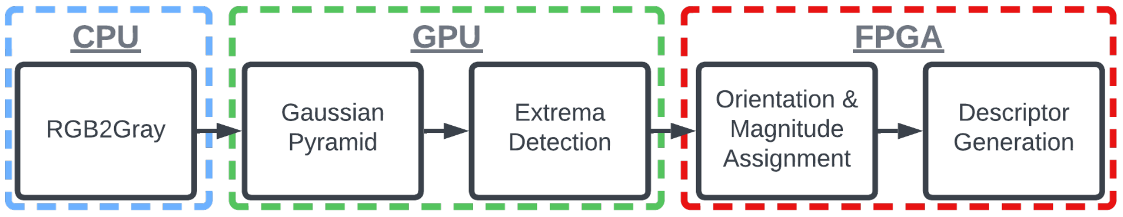


Figure 6.3: SIFT Algorithm & Partitioning Strategy.

tleneck.

The "`Gaussian Pyramid`" executed on GPU is faster than the CPU and FPGA since the vast number of cores and memory resources available can utilise majority of the matrix multiplication operations. To create these scale-space representations, the algorithm needs to access the image data repeatedly for each scale level. The Gaussian filters, used to perform blurring, have larger kernels (in terms of memory requirements) as the scale level increases. Additionally, the image data must be read from memory for each scale level, leading to significant memory access. This heavy memory access is due to the repetitive application of convolution operations with Gaussian filters at multiple scales. The memory bandwidth and efficiency become critical in this stage, especially when working with large images or implementing the algorithm on hardware platforms. Therefore, GPUs can provide an advantage in scenarios where larger memory bandwidth is needed over FPGAs. The "`Extrema Detection`" is allocated to GPU to reduce additional data transfer costs that would offset any performance gain. The "`Orientation & Magnitude`" operations are better suited to the FPGA due to the customised pipelining offered, which allows a higher number of gradient calculation operations per clock cycle. In the final operation stage, "`Descriptor Generation`", the FPGA offers comparable performance to the GPU while consuming less energy per operation and reducing additional data transfers.

6.3 Convolutional Neural Network

This section describes the architecture of two widely used CNNs, *Resnet18* and *MobilenetV2*. Each CNN is profiled layer by layer and a partitioning strategy is developed to execute on the heterogeneous platform. The layer characteristics and profiling results support the partitioning methods.

6.3.1 Convolution Neural Network Architecture:

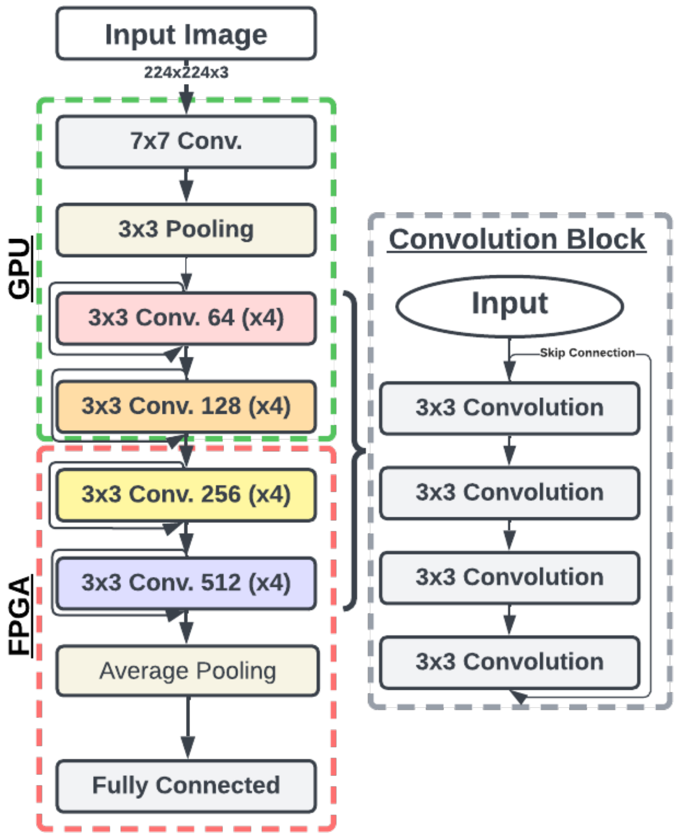


Figure 6.4: ResNet-18 Architecture & Partitioning Strategy.

ResNet-18: is a deep convolutional neural network architecture observed in 6.4, that employs several key techniques to achieve state-of-the-art performance in various computer vision tasks. Its design is driven by the need to train very deep neural networks while mitigating issues like vanishing gradients. The key components and technical details of ResNet-18 are as follows:

1. **Convolutional Layers:** ResNet-18 comprises a total of 18 layers, which are organised into several stages. The first layer is a 7×7 convolutional layer with a stride of 2. This layer is followed by a max-pooling operation with a 3×3 window and a stride of 2. The large kernel size in the initial layer helps capture larger spatial features.
2. **Batch Normalisation and ReLU Activation:** After each convolutional layer, batch normalisation is applied to stabilise and accelerate train-

ing. This is followed by the Rectified Linear Unit (ReLU) activation function, which introduces non-linearity into the network. The combination of batch normalisation and ReLU helps in faster convergence and regularisation.

3. **Residual Connections:** One of the distinctive features of ResNet-18 is its use of residual connections. These connections introduce identity mappings, which allow the network to learn the residual information between layers. Mathematically, the output of a residual block is calculated as follows:
$$F(x) = \mathcal{H}(x) + x \tag{6.1}$$
Where x is the input to the block, $\mathcal{H}(x)$ represents the transformation applied by the block, and $F(x)$ is the output.
4. **Downsampling Blocks:** Each stage in ResNet-18 contains multiple residual blocks. After a series of convolutional layers within a stage, a downsampling block is applied. The downsampling block typically involves a convolutional layer with a stride of 2, which reduces the spatial size of the feature maps while increasing the number of channels. This operation effectively reduces the computational burden while preserving important information.
5. **Softmax Classification Layer:** The final layer of ResNet-18 is a softmax classification layer. It takes the feature map produced by the preceding layers and computes a probability distribution over output classes. The softmax function is applied to the features, and the resulting probabilities indicate the likelihood of the input belonging to different classes.
6. **Shortcut Connections:** To prevent vanishing gradients and facilitate the flow of information, shortcut connections are introduced in residual blocks. These connections skip the first two convolutions within a block and add the input directly to the output of the third convolutional layer. This way, the gradient can flow backwards through the skip connection, making it easier to train very deep networks.

MobileNetV2: shown in 6.5, is an embedded-optimised convolutional neural network architecture that uses a range of techniques to achieve high accuracy with low computational cost. Key details of MobileNetV2 include:

1. **Depthwise Separable Convolution:** MobileNetV2 employs a depthwise separable convolution technique. It divides a standard convolution op-

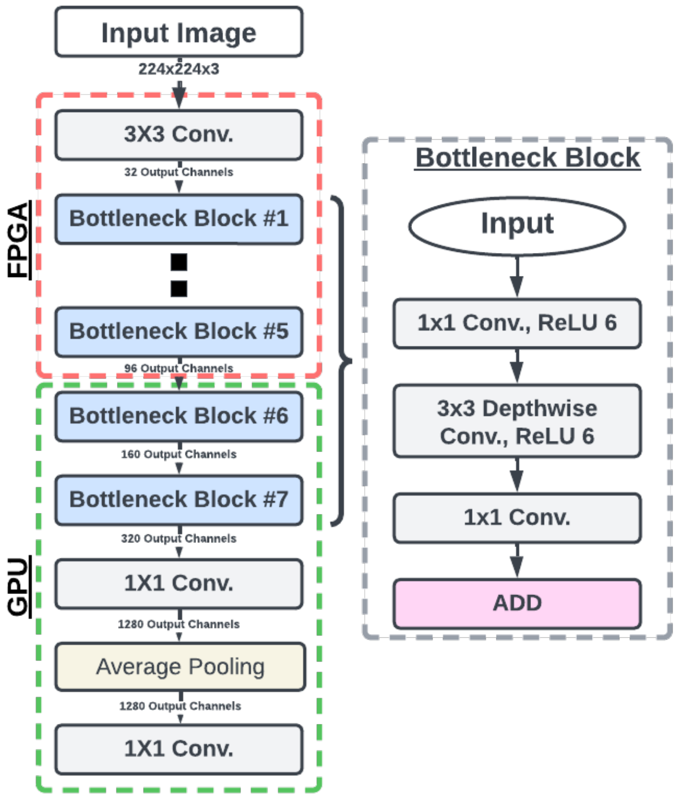


Figure 6.5: MobileNetV2 Architecture & Partitioning Strategy.

eration into two distinct steps: depthwise convolution and pointwise convolution. Depthwise convolution first performs a separate convolution for each input channel using a kernel of size (k, k), where k is the filter size. This reduces the computational load by a significant margin.

The depthwise convolution operation is computed using Eq. (6.2):

$$y = \text{depthwise\_conv}(x, W_d) \tag{6.2}$$

where y is the output, x is the input feature map, and W_d are the depthwise convolution weights.

2. **Pointwise Convolution:** After the depthwise convolution, pointwise convolution is applied using 1×1 kernels. Pointwise convolution combines the results of the depthwise convolution by performing a linear combination of the channels. This step helps to capture complex relationships between channels and is critical for maintaining model accuracy.

The pointwise convolution operation shown in Eq. (6.3):

$$y = \text{pointwise\_conv}(x, W_p) \tag{6.3}$$

where y is the output, x is the result of the depthwise convolution, and W_p are the pointwise convolution weights.

3. **Reduction of Computational Cost and Parameters:** The combination of depthwise separable convolution reduces the computational cost significantly, making MobileNetV2 suitable for resource-constrained embedded devices. The reduction in the number of parameters and the computational requirements is a crucial advantage.
4. **Linear Bottlenecks:** MobileNetV2 also introduces linear bottlenecks, which are inexpensive 1×1 convolutions placed between a ReLU activation function and a 3×3 convolution. These linear bottlenecks are designed to keep the computational cost low while ensuring that the network maintains a high level of accuracy. The ReLU activation helps introduce non-linearity, while the subsequent 3×3 convolution captures more complex features.

   The linear bottleneck operation represented in Eq. (6.4):

$$y = \text{conv\_3x3}(\text{ReLU}(\text{conv\_1x1}(x, W_l)), W_{3x3}) \tag{6.4}$$

   where y is the output, x is the input, W_l are the linear bottleneck weights, and W_{3x3} are the weights for the 3×3 convolution.

6.3.2 CNN Profiling & Partitioning Strategy:

In this section, both CNN architectures are analysed and partitioned onto the appropriate accelerator based on their runtime profiles. The CNN hardware comparison results are displayed in Figure Fig. 6.6 & Fig. 6.7.

Experimental Design

The runtime profiles of each CNN architecture layer were collected using a robust experimental design. Both MobileNetV2 and ResNet18 architectures used a 3840×2160 resolution image dataset. The CPU and GPU implementations leveraged the Pytorch library and its CUDA configuration. A custom

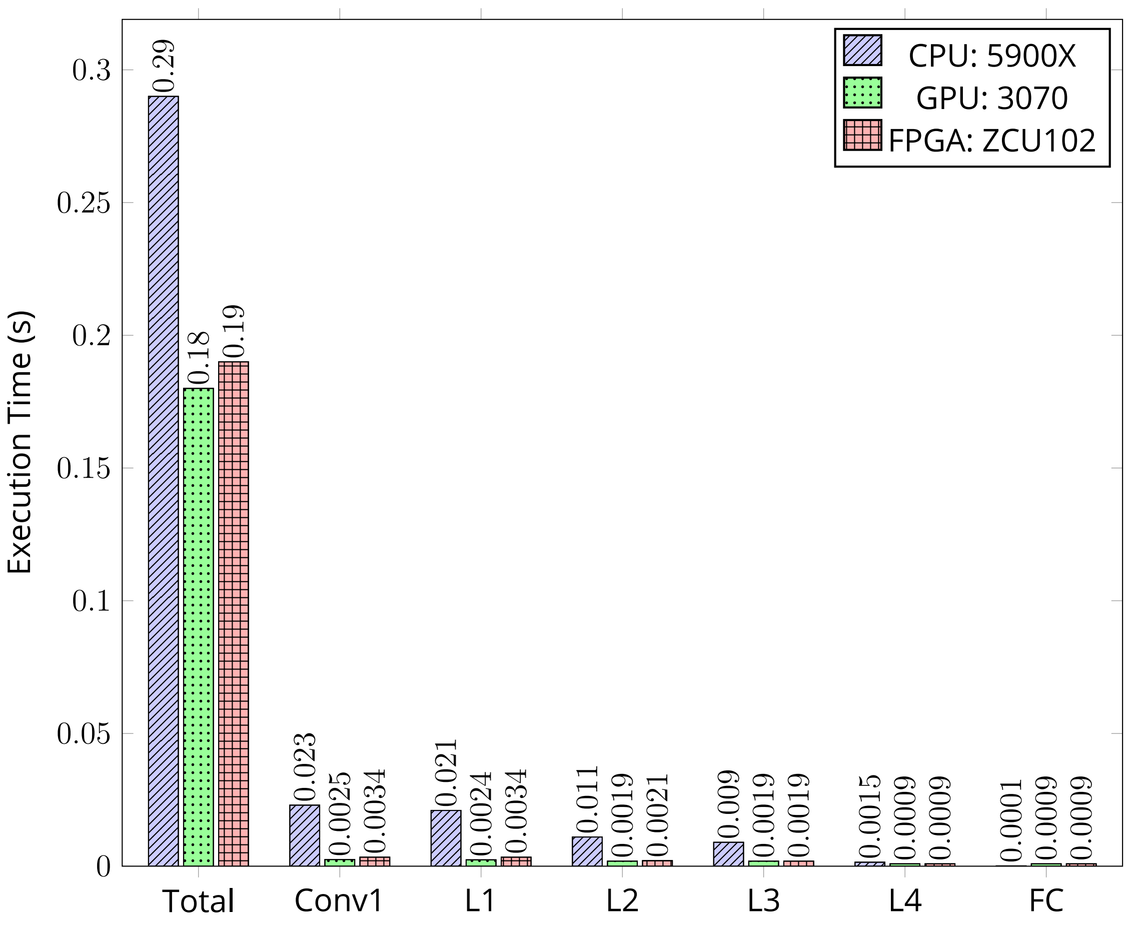


Figure 6.6: ResNet18, Layer Run-time Profiling: First Convolution (Conv1) Layer (L), Fully Connected (FC). Total represents the complete CNN running including initialisation time.

hook function was defined to measure the runtime of each convolutional and linear layer during forward pass execution. For every convolutional and linear layer in the model, the hook function recorded the start and end times, allowing calculation of the layer-wise runtime. Through 1000 code execution iterations, the runtimes collected are averaged. In the FPGA implementation, Xilinx deep learning processing unit IP and their resulting timing graphs are used to determine the time taken by each layer. All architectures had 8-bit precision for both models and implementation.

Resnet18:

The *Resnet18* results in Fig. 6.6 show the fastest hardware for executing the model is the GPU, with the total execution time of 0.18s, while the slowest is the CPU with 0.29s. The FPGA's total execution time is between the two with 0.19s.

The `conv1` layer is computationally intensive for all platforms as it applies 64 filters of size 7×7 to a large input image with multiple colour channels. This results in a large number of multiply-accumulate (MAC) operations. The conv1 layer also performs padding and activation functions, which adds to the overall computational cost. However, the execution time for Conv1 is significantly faster on the GPU, which can parallelise the computations across multiple cores. In layers $L1 \sim L2$, the GPU is $1.36\times$ and $1.10\times$ faster than the FPGA. Therefore, the GPU is the best candidate to allocate Conv1, L1 and L2 for execution.

The $L3 \sim L4$ and Fully Connected (FC) layers take relatively less time to execute. The size of feature maps decreases as they progress through the layers due to downsampling operations like pooling and strides. The $L3 \sim L4$ convolutional and average pool layers can be executed on the FPGA since fewer MAC operations are occurring for the GPU to be fully utilised while taking advantage of power efficient architecture.

MobilenetV2:

The Fig. 6.7 results show that the total execution time for the CNN on the CPU was 0.241s, while on the GPU and the FPGA, it was 0.23s and 0.20s, respectively. The bottleneck layer with the longest execution time for all three devices was BN1, with a time of 0.02202ms on the CPU, 0.015ms on the 3070 GPU, and 0.0034ms on the FPGA.

The runtime for each bottleneck layer decreases as it moves from `Bottleneck 1` to `Bottleneck 7` due to reduced input channels and the amount of computation required to process the data in each layer. Thus, the later bottleneck layers take less time to execute than the earlier ones. The first five bottleneck layers are suitable for execution on the FPGA, as it shows that it performs $\sim 4\times$ faster than the GPU on average. The reason behind the faster execution on FPGA can attributed to multiple factors below:

1. Direct, custom and optimised routing between logic allows efficient dataflow transfer and locality.
2. Separable filters and feature maps have reduced memory footprint which

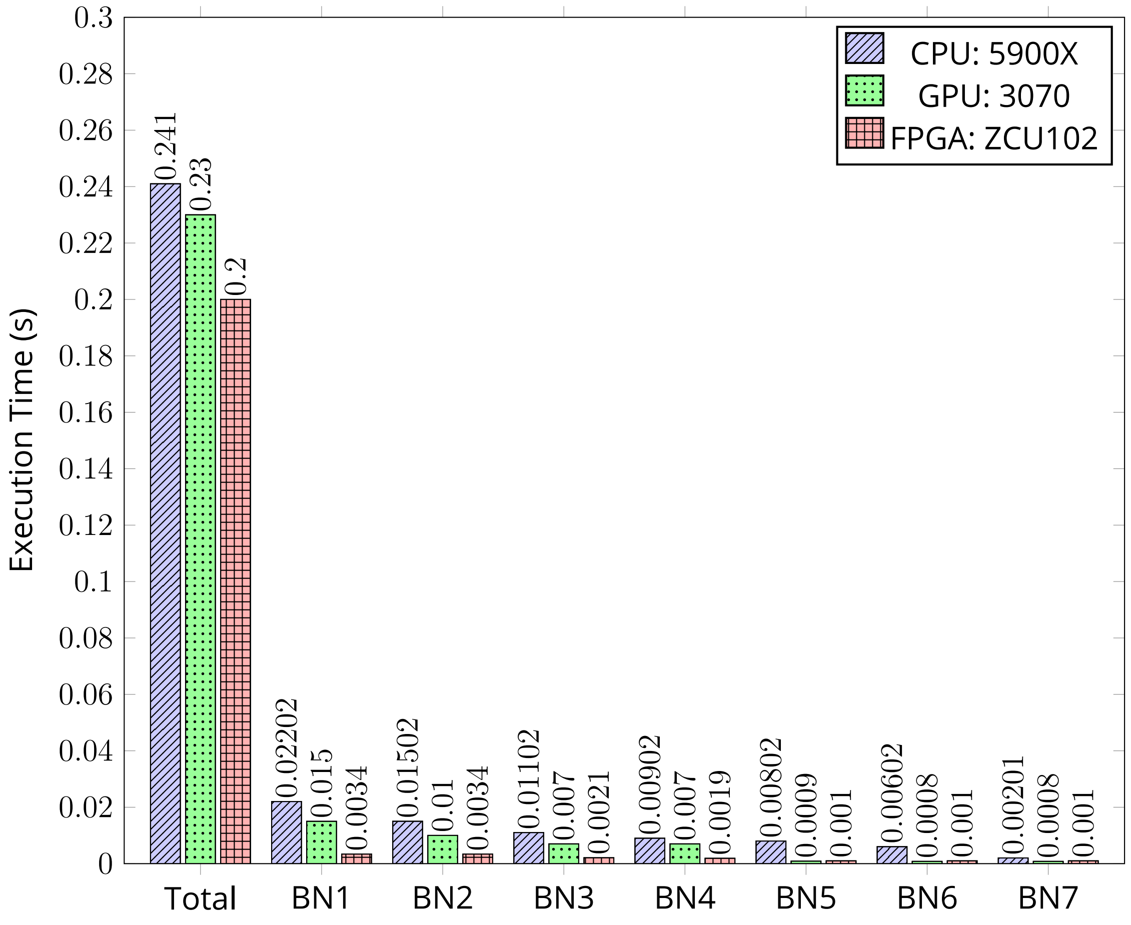


Figure 6.7: MobileNetV2 Layer Run-time Profiling: Bottleneck Layer (BN). Total represents the complete CNN running including initialisation time.

can be efficiently managed.

3. Efficient use of pipelining for convolutional operations and reduced data dependency (*e.g.*, ResNet18 residual connections).

The remaining `Bottleneck 5` ~ 7 layers are suitable for execution on the GPU because of a slight performance advantage and shorter initialisation transfer time back to the host. The `Softmax` and `fully connected` layers are also computed on the GPU since the performance is comparable to the FPGA and reduces data transfer overhead. Both layers are required to transform features into a suitable format, and it assigns a probability score for classification.

Table 6.1: Hardware/Software Environment

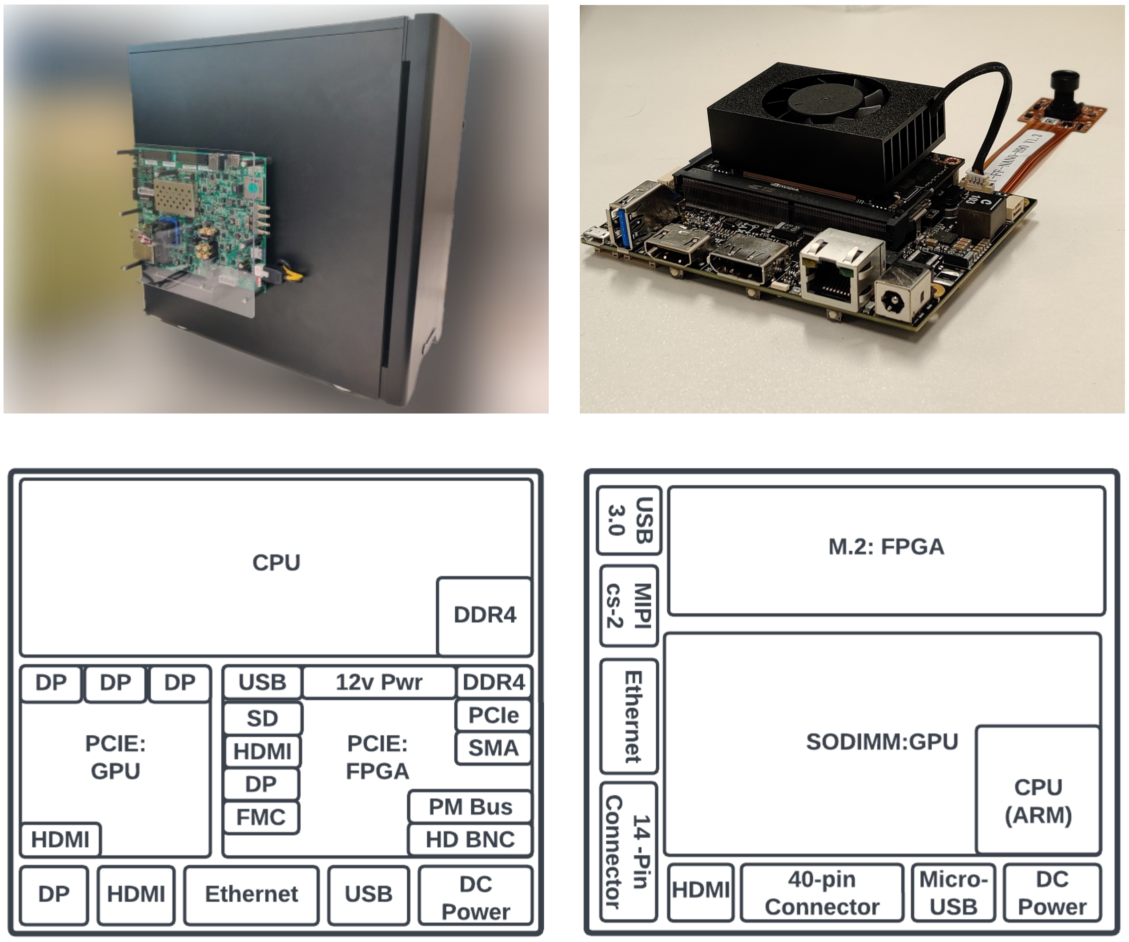
Accelerator	High-Power	Low-Power	Software
CPU	AMD 5900x (4.8 GHz)	ARMv8.2 (1400MHz)	Python / Pytorch 2.0
GPU	Nvidia 3070 (1730 MHz)	Xavier NX (1100MHz)	Python / Pytorch 2.0
FPGA	ZCU106 (300Mhz)	Artix-7 (100MHz)	Verilog / Vivado / Vitis

6.4 Experimental Setup

This section introduces both heterogeneous implementations and their corresponding tools and software used to target those architectures. In addition, a detailed break of the scheduler used to move data between all accelerators while keeping tasks synchronised. Both power and execution time are discussed in detail which are used to evaluate both platforms.The proposed partitioning is tested using two developed heterogeneous platforms containing high-low power components, as shown in Table 6.1, and described below:

Low-Power System: The constructed system consists of a custom carrier board which is equipped with several key components, including an Artix-7 (XC7A200T) FPGA, a Jetson Xavier NX, and an ARM CPU. To provide additional storage space, the Linux image is flashed onto the SD card rather than the 16Gb eMMC. Communication between the FPGA and the Xavier NX is achieved through a PCIe gen2 4-lane interface, which is connected via an M.2 key-M connector.

High-Power System: The system consists of CPU (AMD 5900x), GPU (GTX 3070) and FPGA (Xilinx ZCU106), integrated into a desktop with 32GB 3200Mhz DDR4 Memory. Both devices are interfaced via PCIe Gen3 and the communication to the host CPU uses direct memory access (DMA), allowing the movement of data between host memory and subsystems. The GPU and FPGA drivers are used to program the DMA engine and DMA/bridge subsystem IP. Idle CPU is frequency scaled to reduce power consumption.



(a) (b)

Figure 6.8: (a) Heterogeneous High-Power System, (b) Low-Power Embedded System

Dataset. The test images used in the experiments are from LIU4K-v2 dataset [171], which is a high resolution data-set that includes 2000 3840×2160 images. The images contain a variety of backgrounds and objects.

Heterogeneous Scheduler Architecture

A heterogeneous scheduler architecture in Fig. 6.9 is developed, which is responsible for distributing and managing tasks or workloads across different types of processing units or resources. The scheduler is written in C/C++ to manage memory transfers and pass the feature map and operation stage data from the CPU and into the GPU or FPGA.

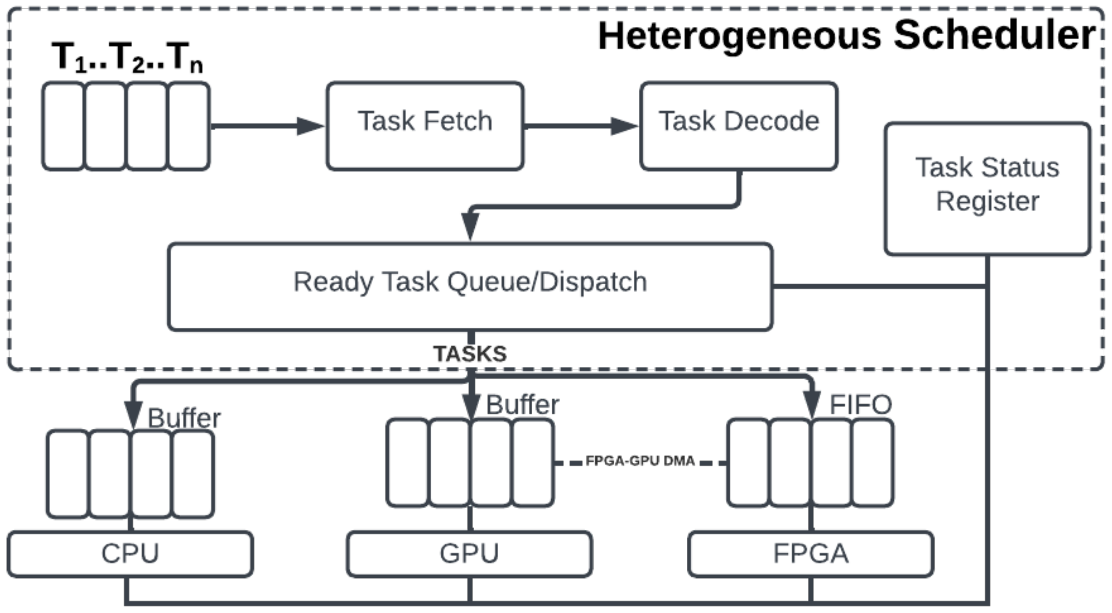


Figure 6.9: Scheduler Architecture for Heterogeneous Platform: Allocating tasks to processors depending on the partitioning strategy.

The scheduler uses a *First-Come, First-Served* (FCFS) algorithm due to its relatively simple construction and minimal runtime overhead. The operation stages in imaging algorithms are executed sequentially. Therefore, tasks are partitioned in order based on their dependency on previous task data, which makes it suitable for this type of scheduling algorithm. The pre-partitioned ordered tasks are fetched from the queue and decoded to prepare for execution. As tasks are dequeued, they are placed in the task ready queue, a staging area where tasks await execution. The task ready queue ensures that tasks are readily available for the execution stage. Each task contains a predefined pragma which tells the scheduler which architecture to execute the operation on. Additionally, a task status register is used to keep track of the current state and progress of each task. This register provides real-time information about the task's status, whether it is waiting, executing, or completed. The task status register is essential for maintaining task synchronisation and ensuring that tasks progress through the execution pipeline smoothly. It allows the scheduler to monitor and manage the execution of tasks and make informed decisions based on the tasks' statuses, ensuring optimal resource utilisation and order of execution.

The pseudocode presented in Algorithm 1 outlines the process of the Het-

Algorithm 1 Heterogeneous Scheduler

```
 1: $fpga\_output \leftarrow$ None
 2: $gpu\_output \leftarrow$ None
 3: $task\_queue \leftarrow \emptyset$                          ▷ FCFS task queue
 4: for $i \leftarrow 1$ to Length($tasks$) do
 5:     $task \leftarrow$ tasks[$i$]
 6:     $task\_queue \leftarrow task\_queue \cup \{task\}$        ▷ Add task to FCFS queue
 7: end for
 8: while $task\_queue$ is not empty do
 9:     $task \leftarrow$ Dequeue($task\_queue$)               ▷ Dequeue the first task
10:     if ExecuteOnFPGA($task, fpga\_partition$) then
11:         $fpga\_output \leftarrow$ ExecuteOnFPGA($task$)
12:     else if ExecuteOnGPU($task, gpu\_partition$) then
13:         $gpu\_output \leftarrow$ ExecuteOnGPU($task$)
14:     end if
15: end while
16: if $fpga\_output$ is not None then
17:     $final\_output \leftarrow$ TransferData($fpga\_output, "FPGA", "CPU"$)
18: else
19:     $final\_output \leftarrow$ TransferData($gpu\_output, "GPU", "CPU"$)
20: end if
21: $final\_output \leftarrow$ PostProcess($final\_output$)
22: DisplayResult($final\_output$)
```

erogeneous Scheduler. This algorithm iterates over the tasks in the workload and employs an FCFS queue to execute tasks in the order they were received. The algorithm initialises two functions, `ExecuteOnFPGA()` and `ExecuteOnGPU()`, which are used to execute the task on a certain device based on a partitioning strategy. The FCFS queue ensures that tasks are executed in the order of arrival, maintaining fairness and accuracy in task execution. Once all tasks have been executed, the algorithm transfers the output data from the FPGA or GPU to the CPU, depending on which device the output is stored on. This transfer facilitates the consolidation of results for further processing. Lastly, the algorithm post-processes the output to generate the final result of the workload.

6.4.1 CPU-GPU-FPGA Data Communication

The data transfer process controlled by the scheduler follows a series of steps described below:

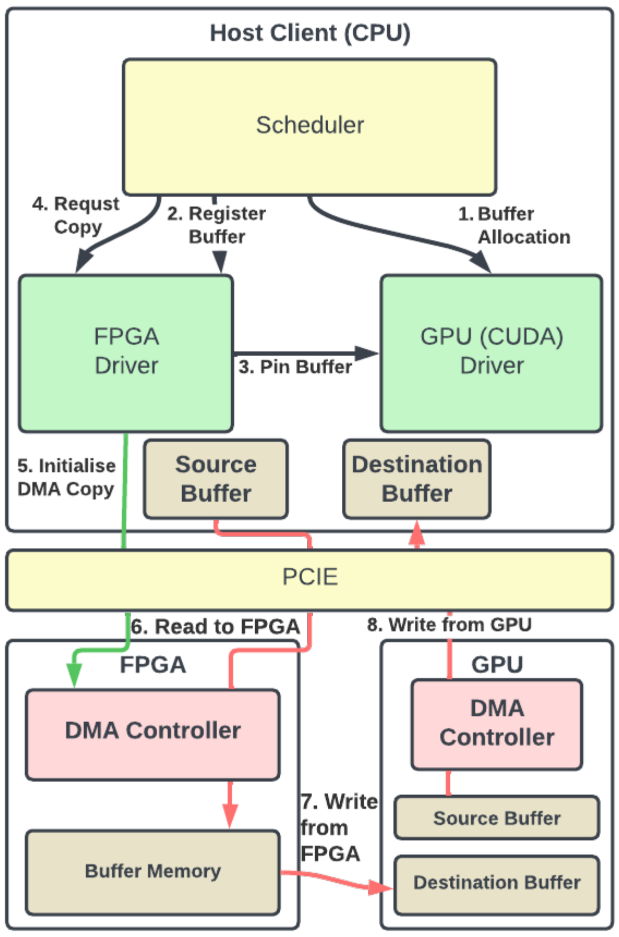


Figure 6.10: RDMA Direct Memory Access Data-transfer process steps for communication between FPGA-GPU.

1. Allocate Buffer: Allocate memory buffers on both the GPU and FPGA to hold the data to be transferred.
2. Register Buffer: Register the GPU buffer with the FPGA using PCIe Base Address Register mapping or GPUDirect Remote Direct Memory Access to enable direct access.
3. Pin Buffer: Pin the GPU buffer to prevent it from being swapped out of physical memory, ensuring consistent and efficient access by the FP-GA/Host.
4. Request Copy: Initiate the data transfer from the GPU buffer to the FPGA buffer using a Direct Memory Access controller.
5. Program DMA Copy: Configure the DMA engine's source and destination addresses, transfer size, and control parameters to transfer data autonomously.

6. Handle DMA Completion: Monitor the DMA engine's status flags or interrupts to detect transfer completion.
7. Process Data on FPGA/GPU: Access the transferred data from the destination buffer.

In the case of FPGA, the scheduler uses the drivers to initialise the DMA to allocate buffer pointers and then pass the pointer to the write function. After receiving the input data and its size, the driver creates a descriptor and initialises the DMA process by providing the descriptor's start address. The driver writes a control register to start the DMA transfer, which reads the descriptor and fetches the feature map data to be processed on the FPGA.

On the GPU side, the CPU host code allocates device pointers using CUDA functions like `cudaMalloc` to specify the locations in the GPU's memory where the data will be placed. Then, the CPU host code invokes CUDA API functions such as `cudaMemcpy` to request the data transfer. The GPU driver, which manages GPU resources, sets up the transfer, allocates GPU memory, and configures the data transfer channels. It ensures that the FPGA's memory is correctly mapped to the GPU's address space to enable efficient transfer. Subsequently, the GPU driver issues commands to the GPU to initiate the data transfer. The actual transfer is executed by the GPU hardware using Direct Memory Access (DMA). Once completed, the GPU returns a status to the driver. The CPU host code can then access and process the data in the GPU's memory. Synchronisation mechanisms, like CUDA events or `cudaStreamSynchronize`, may be employed to ensure that the GPU doesn't process the data prematurely.

The systems exploit GPUDirect RDMA shown in Fig. 6.10 to facilitate low-latency communication without involving the host CPU by retrieving the bus address of buffers in GPU memory. Traditionally, BAR windows are mapped to the CPU's address space using memory-mapped I/O (MMIO) addresses. However, current operating systems lack mechanisms for sharing MMIO regions between drivers. Therefore, the NVIDIA kernel driver provides functions to handle address translations and mappings. This means that data can be moved directly between the GPU and the host without the need for intermediate buffers or copies in CPU memory. This approach significantly improves data transfer efficiency, particularly for large images or data, as it eliminates unnecessary data movement and reduces memory overhead.

6.4.2 Execution time

The evaluation of the overall system performance considers both latency and compute factors, reporting performance metrics for total time, inference, and other significant layers while using floating point 16 precision. Other devices, such as i9-11900KF (5.30GHz), are also benchmarked for additional insight. The run-time is measured using the host platform's built-in time libraries. The network performance is estimated by executing and averaging the results of 100 images. The frame per second (FPS) metric is computed using Eq. 6.5:

$$\text{FPS} = 1/\text{Execution Time}. \tag{6.5}$$

6.4.3 Power Consumption

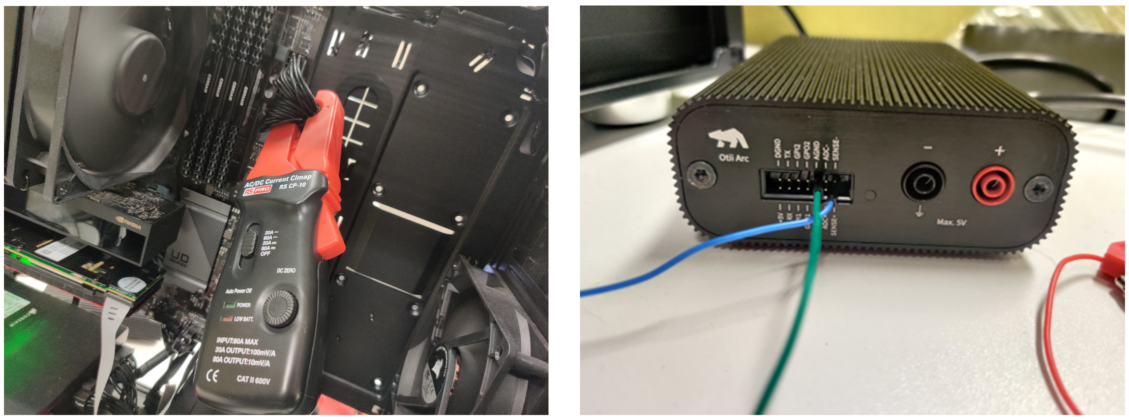


Figure 6.11: (a) Power Measurement using Current Clamp, (b) Connected to a Data Logger at 4 Kilo Samples Per Second.

Two common methods used for measuring power are software and hardware based. Accurate power estimation is always challenging for software tools because they have to assume various factors in their models. Additionally, Taking the instantaneous power or TDP of a device is not accurate since power consumption varies on the specific workload. Therefore, measuring power over the time it takes for the algorithm to execute improves accuracy as opposed to using just fixed Wattage. The hardware measurement approach uses a current clamp meter shown in 6.11, which outputs a voltage for every Amp measured. The *Otii Arc Pro* [177] data-logger captures the time series data from the current clamp and generates a graph showing the current consumption over time. A script is developed to start and stop the measurements during the algorithm's execution. The mean current is averaged and multiplied by the input voltage to determine the energy consumed in Joules (J).

The energy consumption is obtained using Eq. (6.6) where *E* is energy, *P* represents power and *t* time.

$$E = P \times t \tag{6.6}$$

6.5 Experimental Results

In this section, results of both CNN and SIFT algorithms implemented on a heterogeneous platform are observed and discussed in-depth.

6.5.1 Heterogeneous SIFT Results

In achieving that, a custom pipeline was created by targeting various algorithm components on different hardware based on their suitability obtained from the benchmarking framework. This includes the latency of transferring image data between memory and the accelerators. The heterogeneous architecture empowers the ability to pick and execute operations within the image processing algorithms on each architecture to meet the speed and power target. However, within the scope of this work, only an initial configuration of the SIFT algorithm is reported, which establishes preliminary steps toward future work on finding the most optimal configuration for the algorithms.

Heterogeneous SIFT Runtime & Energy Consumption

Table 6.2 shows the execution time of the SIFT algorithm on a heterogeneous platform. The table includes memory transfer latency between the host and devices, an aspect frequently overlooked in similar analyses. The results reveal that the heterogeneous platform (Excl. Memory Transfer) outperforms

Table 6.2: Execution Time (Excl. Memory Latency) on Individual Hardware and Heterogeneous Platform (CPU: 5900X, GPU: RTX-3070, FPGA: ZCU102). Baseline excludes memory latency. Incl. Memory Latency as additional data.

Algorithm	**Baseline (ms)**			**Heterogeneous Architecture (ms)**		
	CPU	**GPU**	**FPGA**	**Accelerator**	**Excl. Memory Latency**	**Inc. Memory Latency**
RGB2Gray	0.80	0.54	0.40	CPU	0.64	0.78
Gaussian Pyramid	684	3	6	GPU	3	115
Extrema Detection	112	2	3	GPU	2	101
Orientation \\& Magnitude	97	4	2	FPGA	2	2
Descriptor Generation	21	1	1	FPGA	1	1
Total Runtime	896.8	10.54	12.4	CPU+GPU+FPGA	8.64	219.78

all discrete architectures, CPU, GPU and FPGA by $103\times$, $1.21\times$ and $1.43\times$, respectively. However, taking data transfer into account, the heterogeneous architecture increases in execution time due to host task scheduling. On the other hand, As accelerators are fabricated on the same silicon die, memory latency would significantly be reduced.

The energy consumption results in Fig. 6.12 reveal that the CPU consumes the most energy for all operation stages in *SIFT* and `RGB2Gray` while the FPGA used the least. The `Gaussian Pyramid` stage stands out as the most energy intensive due to the substantial number of operations it requires. In contrast, the low resource requirements of `RGB2Gray` and `Descriptor Generation` stages reflected lower energy usage.

The heterogeneous architecture, which combines CPU, GPU and FPGA resources, strikes a balance between power consumption and execution time. The "*Total*" power consumption is notably lower than the CPU but between both GPU and FPGA since the static power of other accelerators is taken into account.

6.6 Heterogeneous CNN Results

The results in Fig. 6.14 & Fig. 6.15 show the total run-time and energy consumption of Resnet18 and MobilenetV2 on each architecture and heteroge-

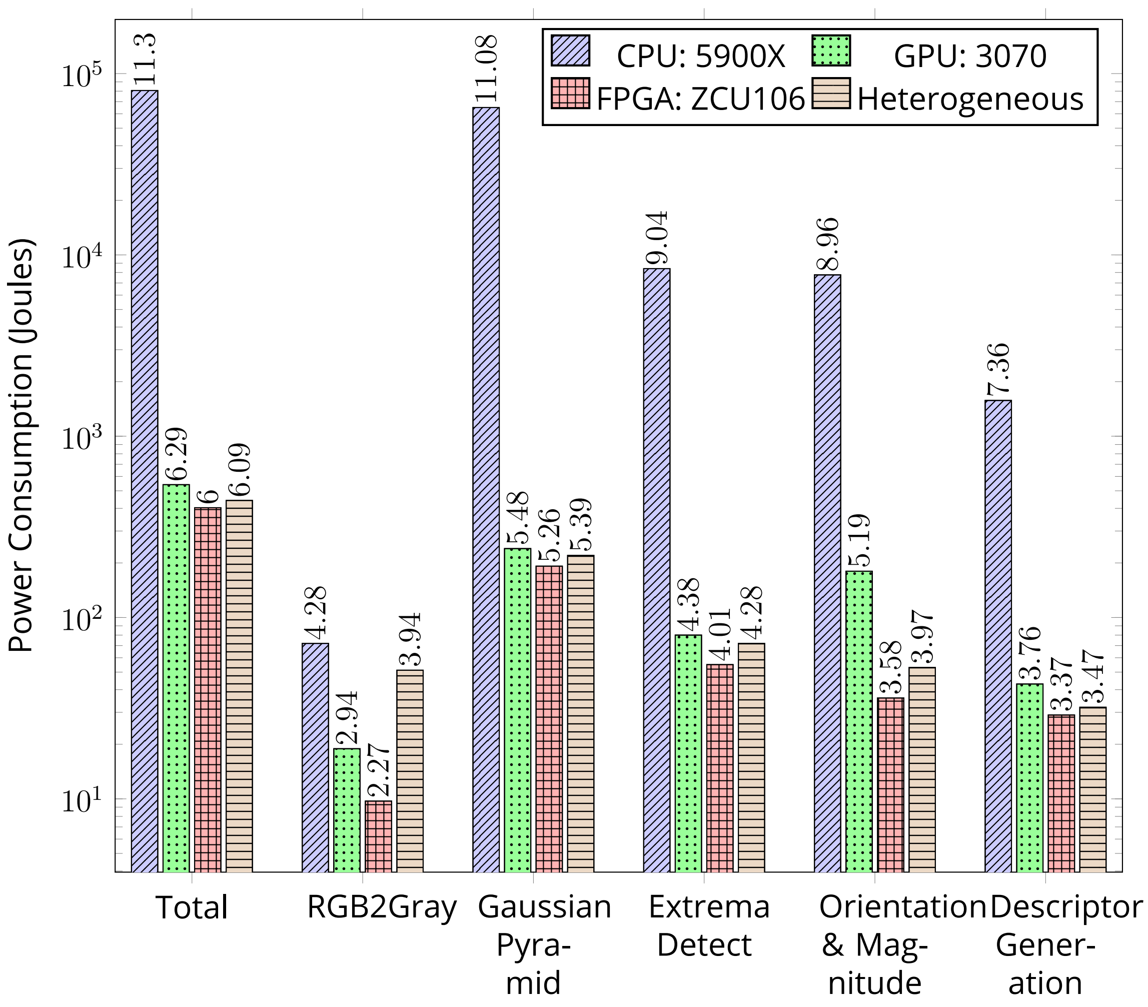


Figure 6.12: SIFT Power Consumption, Baseline Homogeneous and Heterogeneous Implementation Comparison.

neous platform while Fig. 6.13 shows the inference in *Frames per Second*. The tables 6.4 & 6.3, summarises the results for the *Total Execution Time*, *Inference*, *Convolution*, *Fully Connected*, *Total Energy*, *CPU Energy*, *Device Energy* & *Datalogger*

6.6.1 Inference

According to Fig. 6.13, *Resnet18* CNN had the highest FPS value at 270 in contrast to *MobileNetV2* 243 FPS on high and low power heterogeneous architecture. This difference in FPS can be attributed to the network depths and parameters, with *ResNet-18* having 18 layers and *MobileNetV2* having 53 layers, leading to differences in computational complexity. Considering individual

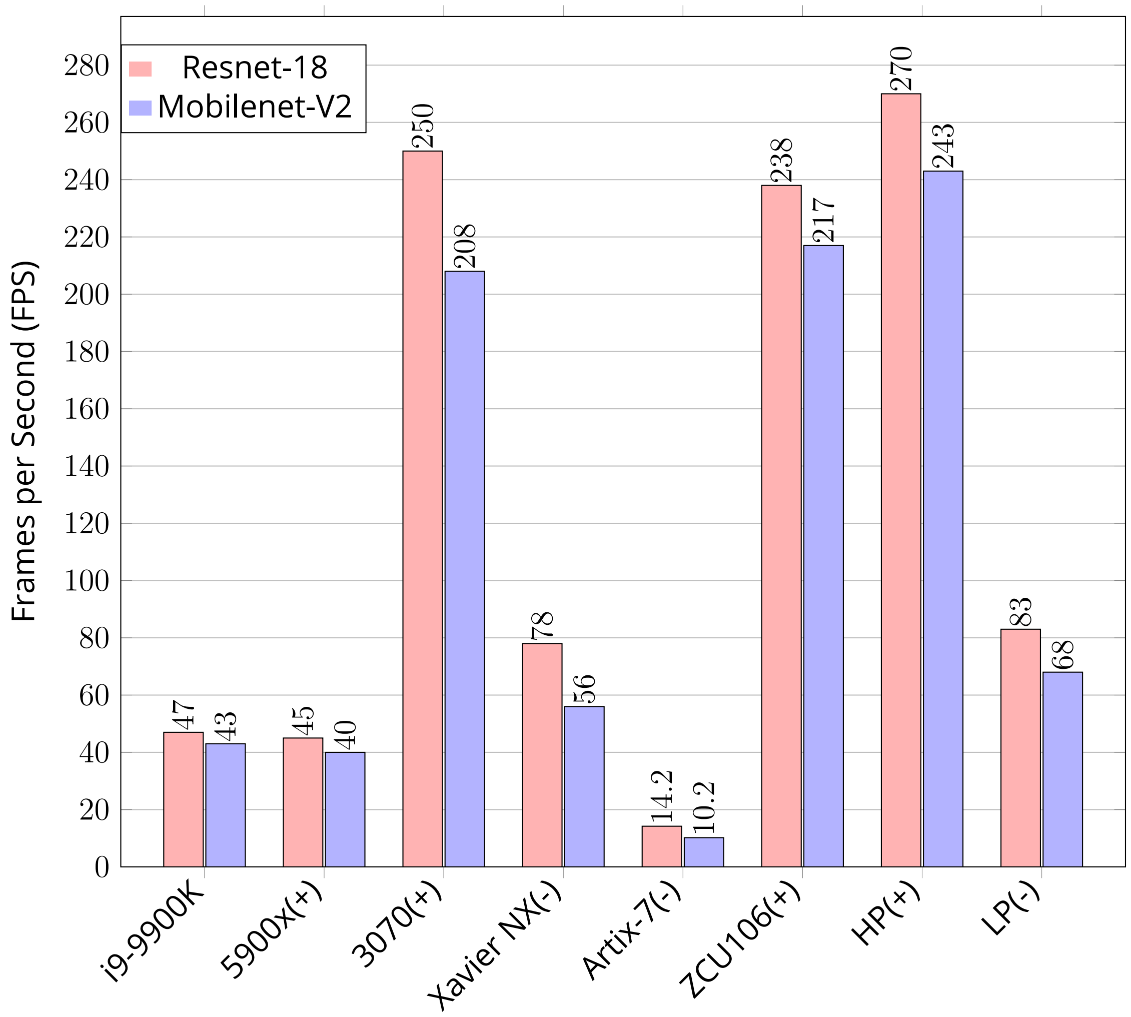


Figure 6.13: Frames per Second (FPS) for Inference on CPU:(I9-9900K, 5900X) GPU:(GTX 3070, Xavier NX) FPGA:(Artix-7, ZCU106), High-Power (HP), Low-Power (LP). (+) Denotes components in HP Platform, (-) Denotes Components in LP Platform.

hardware only, the '3070' GPU achieved the highest FPS on *Resenet18* and the 'ZCU106' FPGA for *MobileNetv2*. On the other hand, the Artix-7 has the lowest FPS in both architectures due to limited resources and clock speed. In the case of both heterogeneous systems, *HP* & *LP* architectures achieve higher FPS than their individual counterparts.

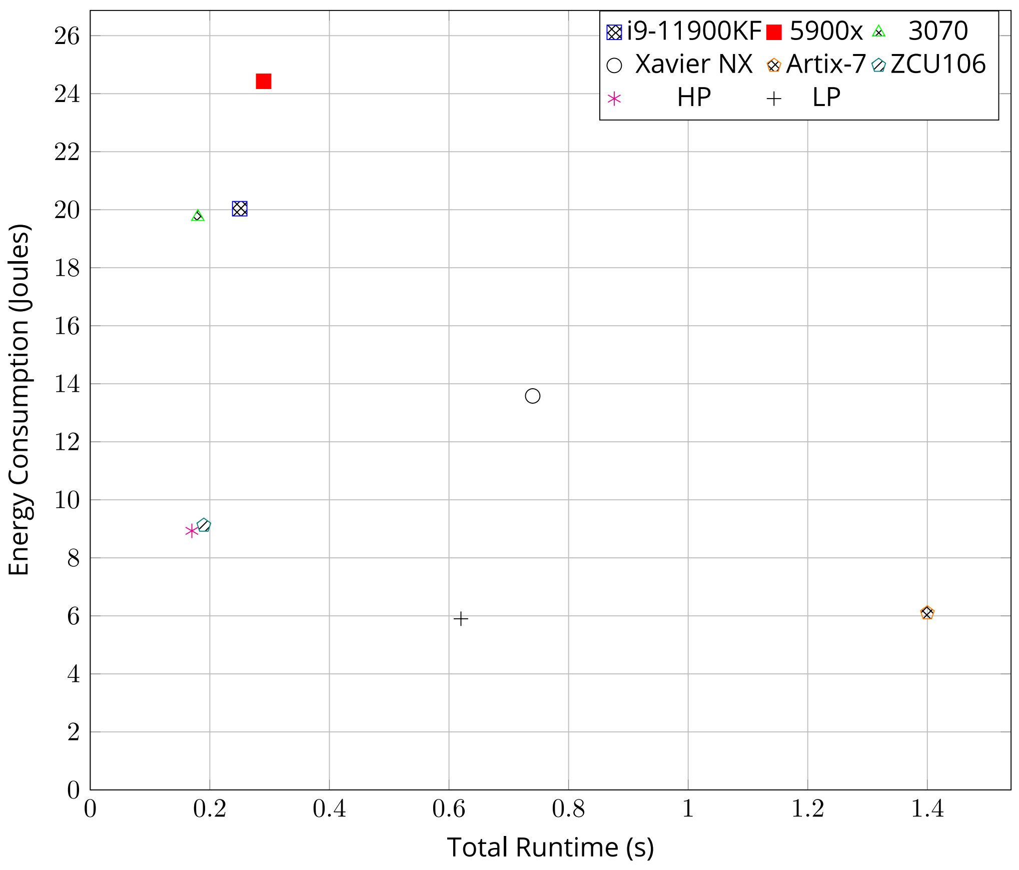


Figure 6.14: ResNet18 Energy Consumption & Total Runtime Comparison; CPU:(I9, 5900X), GPU:(3070, Xavier NX), FPGA:(Artix-7, ZCU106), High-Power (HP), Low-Power (LP).

6.6.2 Total Execution Time

Regarding Fig. 6.14 for Resnet18, it can be observed that the Artix-7 exhibits the highest total execution time, taking approximately 1.1 seconds to complete the task. Conversely, the 'GPU: 3070' has the lowest real execution time, both completing the task at approximately 0.18s. As for MobileNetV2 in 6.15, the 'FPGA: Artix-7' platform also leads with the highest execution time at 1.4s, while the ZCU102 classifies the image in 0.19s and GPU at 0.23s. It is noteworthy to mention that the higher runtimes observed for the GPUs may be attributed to the communication and transfer of data from the CPU and lower core utilisation.

Total Execution time speedups of high (HP) and Low (LP) power systems are compared against their fastest discrete components within each system. The 'HP' system demonstrated a speedup of $1.05\times$ over the 'GPU:3070' for

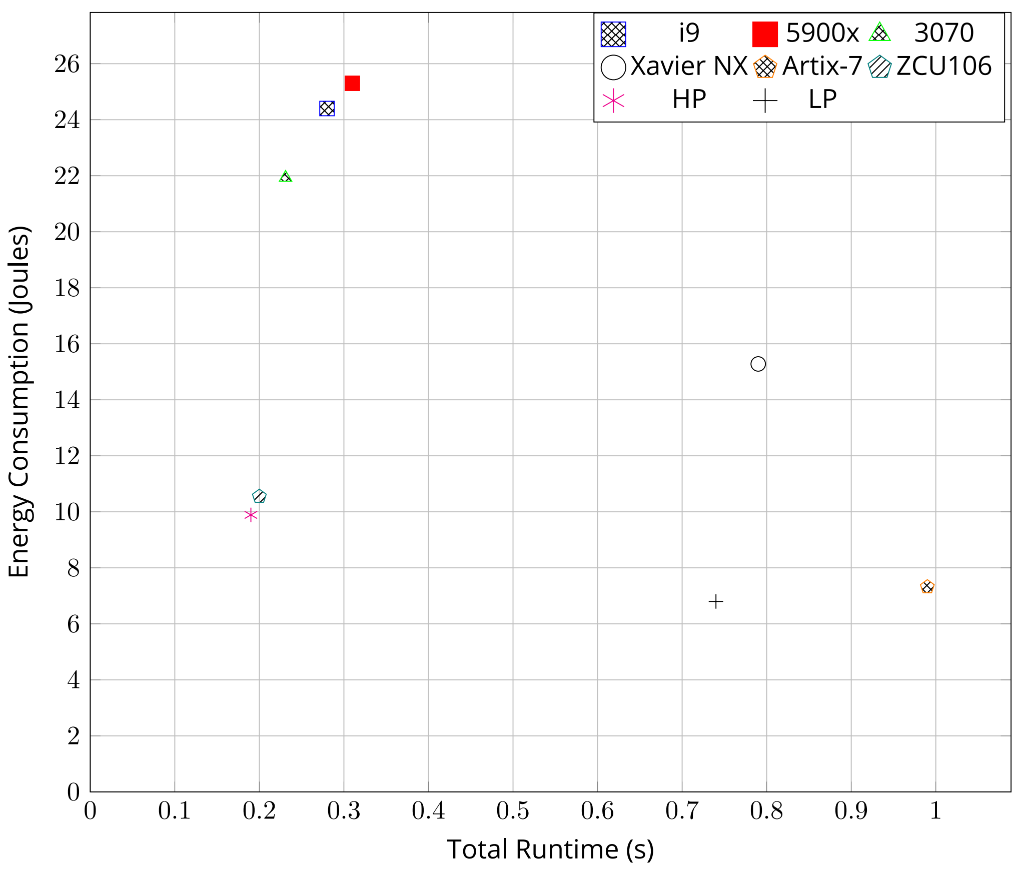


Figure 6.15: MobileNetV2 Energy Consumption & Total Runtime Comparison; CPU:(I9, 5900X), GPU:(3070, Xavier NX), FPGA:(Artix-7, ZCU106), High-Power (HP), Low-Power (LP).

Resnet18 and $1.05\times$ for*MobileNetv2*. In the case for 'LP' system, it exhibited a speedup of $1.21\times$ over the 'GPU: Xavier NX' for *Resnet18* and $1.06\times$ for *MobileNetv2*. The `Convolution` layer results show that the most time is spent performing convolution operations. However, taking in account data/memory transfer latency, both heterogeneous architectures implementations have interconnect (PCIe) and distance bottlenecks. This bottleneck reduces the FPS of HP & LP systems by $10 \sim 25$ which in turn allows the 'GPU:3070' to marginally edge out on the HP system. The disparity between the `Total Wallclock` and `Inference` runtimes are due to the overhead and initialisation time which include model/weight loading and data preprocessing. In addition, the `Fully Connected` layer revealed that all architecture had comparable performance due to the layer's simple MAC operations.

Table 6.3: ResNet-18: Result Summary of Energy Consumption and Execution Time on Each Architecture. (Bold: Best Runtime Performance)

Accelerator	**Execution time (s)**				**Sum CPU + Device**	**Total CPU**	**Total Device**	**Datalogger**
	Total Execution time	Inference	Convolution	Fully Connected	**(Joule)**	**(Joule)**	**(Joule)**	**(Joule)**
CPU: (i9-11900KF)	0.25	0.021	0.019	0.0009	20.035	20.04	N/A	18.23
CPU: (5900X)	0.29	0.022	0.018	0.0009	24.43	24.4267	N/A	22.48
GPU: (GTX 3070)	0.18	0.004	0.0028	0.008	19.08	8.7	10.38	9.11
Jetson (Xavier NX)	0.74	0.0128	0.0109	0.001	13.58	N/A	13.5716	10.56
FPGA: (ARCTIX7)	1.1	0.070	0.062	0.0009	6.10	N/A	6.10	5.8
FPGA: (ZCU106)	0.19	0.0042	0.0033	0.0009	9.12	N/A	9.12	8.07
5900x + 3070 + ZCU106	**0.17**	0.0037	0.0027	0.0009	8.93	$\sim$	$\sim$	8.5
ARM + Xavier + Artix-7	**0.62**	0.012	0.011	0.0009	5.90	$\sim$	$\sim$	5.44

Table 6.4: Mobilenet-V2: Result Summary of Energy Consumption and Execution Time on Each Architecture. (Bold: Best Runtime Performance)

Accelerator	**Execution time (s)**			**Sum CPU + Device**	**Total CPU**	**Total Device**	**Datalogger**
	Total wallclock time	Inference	Convolution	**(Joule)**	**(Joule)**	**(Joule)**	**(Joule)**
CPU: (i9-11900KF)	0.28	0.023	0.02	24.4	24.4	N/A	20.23
CPU: (5900X)	0.31	0.025	0.022	25.3	25.3	N/A	21.48
GPU: (GTX 3070)	0.231	0.0048	0.0045	21.945	9.24	12.70	19.43
Jetson (Xavier NX)	0.79	0.018	0.0125	15.28	N/A	15.28	14.92
FPGA: (ARCTIX7)	1.4	0.098	0.088	7.32	N/A	7.32	6.24
FPGA: (ZCU106)	0.20	0.0046	0.0036	10.55	N/A	10.55	9.65
5900x + 3070 + ZCU106	**0.19**	0.0041	0.0029	9.89	$\sim$	$\sim$	9.01
ARM + Xavier + Artix-7	**0.74**	0.0145	0.015	6.80	$\sim$	$\sim$	5.86

6.6.3 Energy Consumption

Concerning energy consumption, the discrete '5900 & I9' CPUs consume the most energy in both CNN architectures, around $20 \sim 26$ Joules. The least amount of energy consumed is from both FPGA architectures, 'FPGA: Artix' and 'FPGA: ZCU106', which used less than 15 Joules for both networks. Taking CPU idle energy usage into account results in GPUs having comparable energy usage statistics with both CPUs which is linked to higher CPU-GPU data transfer and initialisation cost. However, HP and LP systems consume $1.02\times$ and $1.03\times$ less energy, respectively, compared to the single ZCU106 and Artix architectures for *Resnet18*. As for *MobileNetV2*, there is a $1.06\times$ and $1.07\times$ reduction in energy consumption for the 'HP' and 'LP' systems, respectively. The idle accelerators within heterogeneous systems had their clocks lowered to save on static energy consumption. However, a small increase in idle energy usage was observed during execution. If idle energy is taken into account, then the energy consumption results of both HP & LP systems would be greater than discrete FPGA but lower than GPU.

6.7 Conclusion

In this chapter, partitioning strategies are introduced to map the layers of two widely used convolutional neural networks, namely, `Resnet18` and `Mobilnetv2`, along with a feature extraction algorithm (`SIFT`), onto a heterogeneous architecture. Two new CPU-GPU-FPGA systems, one designed for high performance and the other for low-power consumption, are proposed. The experiments demonstrate that when layer/per-operation partitioning methods are applied, both high-power and low-power systems outperform homogeneous accelerators in terms of energy efficiency and execution time. Furthermore, the results suggest that partitioning networks based on their layer profiles holds the potential for efficient deployment on heterogeneous architectures, offering a viable alternative to GPU/FPGA-only applications.

7 Discussion, Conclusions and Future Work

The aim of this thesis is to present domain-specific optimisation techniques for image processing algorithms on heterogeneous hardware. In section 7.1, the research problems are identified and discussed. Furthermore, in section 7.2, the contributions to addressing the problems are presented. Lastly, in section 7.3, future research directions extending the work are suggested.

7.1 Discussion

The thesis focuses on achieving two main research aims, "Which is the best method of partitioning and implementing algorithms on heterogeneous hardware" and "Identifying domain-specific optimisations and understanding the performance and accuracy trade-offs".

In the case of implementing algorithms on heterogeneous platforms, the primary challenge is navigating the route to hardware. Designers often face a complex terrain of diverse hardware architectures, each with unique constraints and optimisation opportunities. This problem can lead to a time consuming and error-prone process of manually configuring and optimising algorithms for specific accelerators. Moreover, the need for standardised tools and interfaces across platforms further exacerbates the difficulty of seamless integration. Designers must grapple with the intricacies of synchronisation and resource allocation, which can be particularly daunting in complex, real-time image processing applications. These challenges underscore the need for a more streamlined and systematic approach to ensure efficient deployment of algorithms on heterogeneous platforms.

The other challenge for heterogeneous systems is efficiently managing data transfers between various processing units. Data movement between processors introduces latency and consumes substantial computational resources and memory bandwidth. In image processing, operations usually form a pipeline often subject to data dependencies, making it difficult to optimise the scheduling of tasks. Therefore, inefficient data transfers can lead to a significant performance bottleneck.

Optimisation is an essential step towards extracting the best performance out of systems. Traditional optimisations are not domain-aware and, therefore, cannot exploit the unique properties of specific problem domains to improve performance. Additionally, understanding the trade-offs between domain-specific optimisations and energy consumption or accuracy has not been fully considered.

Finally, implementing deep-learning algorithms such as CNNs or feature extraction algorithms on heterogeneous architectures necessitates the development of fine-grained partitioning strategies. Applying these strategies requires a thorough understanding of hardware architectures and profiling. In addition, developed metrics should be used to effectively evaluate the performance of heterogeneous implementations based on the type of algorithm deployed. Addressing these challenges is critical to unlocking the full potential of heterogeneous architectures.

7.2 Conclusions

To achieve the first objective of efficient deployment of algorithms on heterogeneous platforms, the characteristics of image processing algorithms were decomposed into fundamental operations. A benchmarking framework is presented in Chapter 4 to understand the features of algorithms found in the image-signal pipeline and to determine their suitability for specific accelerators in a heterogeneous environment. This modular framework, termed the Heterogeneous Architecture Benchmarking Framework on Unified Resources (HArBoUR), provides an in-depth analysis and set of metrics for imaging algorithms, which in turn enables the identification of the most efficient processing unit. To support the proposed framework, low and high complexity image processing pipelines are evaluated on each architecture using various tools and libraries. This gives a comprehensive insight into their design choices and optimisations. Different evaluation metrics are proposed such as throughput, energy per operation and clock cycles per operation.

The following chapter 5, explores domain-specific optimisation techniques within image processing. Several optimisations were proposed and validated on CNNs, feature extraction and filter algorithms. The results for CNN and filter algorithms had significantly reduced computation times on all processing architectures. In the case of the optimised SIFT algorithm implementation, it had outperformed the state-of-the-art on FPGAs. Additionally, it achieved runtime at par with GPU performances while consuming less power. However, these optimisations come at the expense of reduced accuracy, highlighting the need for thoughtful consideration when aiming to enhance performance through domain-specific optimisations.

In the concluding Chapter 6, the development of two low and high-power heterogeneous systems using commercial off-the-shelf hardware is discussed. Moreover, two convolutional networks and a feature extraction algorithm are profiled and analysed to identify performance hotspots and assess their hardware compatibility. The algorithms are then partitioned onto the most efficient hardware using a fine-granular strategy, involving the separation of CNNs layer by layer and operations within the feature extraction algorithm. The implemented heterogeneous algorithms are evaluated against their discrete hardware counterparts, resulting in notable speedups in performance and reduced energy consumption.

7.3 Limitations & Future work

In this section, the limitations and potential future directions of this research are discussed below:

7.3.1 Heterogeneous Benchmark Framework

The heterogeneous benchmarking framework proposed in Section 4 can be extended by including additional performance metrics that consider communication latency and scheduling of algorithms to determine the true performance. Further developing a tool-chain to support designers by highlighting key areas of code that can be accelerated by a particular processor and automatically partitioning algorithms without requiring additional designer input. This automation not only saves time but also ensures that the resulting code is fine-tuned on the target hardware.

7.3.2 Domain-Specific Optimisations

The methods described in Section 5.1 mainly centre on applying optimisations uniformly across the entire algorithm, resulting in the same optimisation being applied to every operation stage within the algorithms. This coarse-grained approach may potentially impact accuracy while having no significant impact on the overall runtime. Therefore, adopting a fine-grained approach by applying optimisation strategies to specific parts of an algorithm. An example of a granular approach involves applying optimisations to individual layers of a CNN or each stage of the SIFT algorithm, where each operation is fine-tuned independently. Additionally, the development of a domain-specific compiler capable of dynamically adapting and optimising algorithms based on runtime conditions, accuracy requirements, energy considerations, and data patterns can further enhance the efficiency and effectiveness of domain-specific optimisations on a heterogeneous platform.

7.3.3 Heterogeneous Implementations

In section 6, future directions of this chapter involve improving the hardware, scheduler and measurement accuracy. Initially, The heterogeneous scheduler on the high performance system transfers the output of one operation or layer onto the pinned memory of the GPU or buffer. The transfer is indirect since it has to go through the CPU, but can be shortened through direct memory transfer between accelerators. The scheduling algorithm can be improved by dynamically scheduling workloads for image processing if requirements change.

To address the challenges with data transfer latency can be separated into two categories, Hardware and Software listed below:

- Hardware: Using a faster interface between accelerators than PCIe or integrating the cores onto a single compute chip to reduce the data transfer distance.
- Hardware: Utilising in-memory processing which integrates computation within memory modules. This design reduces data movement overhead and latency by processing data where it's stored rather than shuttling it between memory and processors.
- Software: Latency-aware compilers predict anticipated data usage by different processors and proactively optimise data placement to improve data locality.

7.3.4 Domain-Specific Language

All the previously mentioned future research directions can be unified within the framework of an image processing domain-specific language targeting image processing. The language would allow users to streamline the development of customised image pipelines by mapping algorithms together using a data-flow model. The result of abstracting away from hardware using a high-level signal flow diagram would simplify the route to hardware while removing the burden of partitioning and manual tuning from the designer.

Bibliography

[1] [Online]. Available: https://docs.amd.com/v/u/en-US/cordic_ds249

[2] C. Zhang, Y. Meng, and V. Prasanna, "A framework for mapping drl algorithms with prioritized replay buffer onto heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1816–1829, 2023.

[3] K. Wei, K. Honda, and H. Amano, "Fpga design for autonomous vehicle driving using binarized neural networks," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 425–428.

[4] E. A. Papatheofanous, P. Tziolos, V. Kalekis, T. Amrou, G. Konstantoulakis, G. Venitourakis, and D. Reisis, "Soc fpga acceleration for semantic segmentation of clouds in satellite images," in *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, 2022, pp. 1–4.

[5] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–4.

[6] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *IEEE Int'l Conf on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[7] W. Aspray, "The intel 4004 microprocessor: what constituted invention?" *IEEE Annals of the History of Computing*, vol. 19, no. 3, pp. 4–15, 1997.

[8] S. Mazor, "Intel 8080 CPU chip development," *IEEE Annals of the History of Computing*, vol. 29, no. 2, pp. 70–73, 2007.

[9] [Online]. Available: https://www.eecis.udel.edu/~cavazos/cisc879/papers/Intel-AMD/quad-core-06.pdf

[10] G. S. Alliance, *Assembly Pricing Surveys and Reports*. GSA, 2022.

[11] G. E. Smith, "The invention and early history of the CCD," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 607, no. 1, pp. 1–6, 8 2009.

[12] E. R. Fossum, "CMOS active pixel image sensors," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 395, no. 3, pp. 291–297, 8 1997.

[13] L. C. P. Gouveia and B. Choubey, "Advances on CMOS image sensors," *Sensor Review*, vol. 36, no. 3, pp. 231–239, 6 2016.

[14] "Color imaging array," Patent, 1975.

[15] AIA, *Camera Link Specification*, v2.1 ed. CameraLink, 2018.

[16] PCI-SIG, *PCI Express® Base Specification Revision*, v2.1 ed. PCI-SIG, 2023.

[17] "Ieee standard for ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, 2018.

[18] A. V. amp; Imaging, *Video Streaming and Device Control Over Ethernet Standard*, 2nd ed. A3 Vision amp; Imaging, 2022.

[19] C. W. Group, *MIPI CSI-2 Specification*, v4.0.1 ed. Camera Working Group, 2022.

[20] [Online]. Available: https://www.intel.com/content/www/us/en/content-details/788851/meteor-lake-architecture-overview.html

[21] J. Eker and J. W. Janneck, "Cal language report specification of the cal actor language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, Dec 2003. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html

[22] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[23] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates Inc., 2019, pp. 8024–8035.

[25] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

[26] Y. Ma, F. Rusu, and M. Torres, "Stochastic gradient descent on modern hardware: Multi-core cpu or gpu? synchronous or asynchronous?" in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 1063–1072.

[27] E. Wszola, C. Mendler-Dunner, M. Jaggi, and M. Puschel, "On linear learning with manycore processors," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Dec. 2019. [Online]. Available: http://dx.doi.org/10.1109/HiPC.2019.00032

[28] L. Cheng, P. Pan, Z. Zhao, K. Ranjan, J. Weber, B. Veluri, S. B. Ehsani, M. Ruttenberg, D. C. Jung, P. Ivanov, D. Richmond, M. B. Taylor, Z. Zhang, and C. Batten, "A tensor processing framework for cpu-manycore heterogeneous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 6, pp. 1620–1635, 2022.

[29] Z. Jia, A. Zlateski, F. Durand, and K. Li, "Optimizing n-dimensional, winograd-based convolution for manycore cpus," *SIGPLAN Not.*, vol. 53, no. 1, p. 109–123, feb 2018. [Online]. Available: https://doi.org/10.1145/3200691.3178496

[30] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis, "Performance evaluation of image processing algorithms on the GPU," *Journal of Structural Biology*, vol. 164, no. 1, pp. 153–160, 2008.

[31] Y. E. Wang, G.-Y. Wei, and D. M. Brooks, “Benchmarking tpu, GPU, and CPU platforms for deep learning,” *ArXiv*, vol. abs/1907.10701, 2019.

[32] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim, “Design and performance evaluation of image processing algorithms on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 91–104, 2011.

[33] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, “Real-time optical flow calculations on FPGA and GPU architectures: A comparison study,” in *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 173–182.

[34] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback, “A multi-resolution FPGA-based architecture for real-time edge and corner detection,” *IEEE Transactions on Computers*, vol. 63, no. 10, pp. 2376–2388, 2014.

[35] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, “Accelerating compute-intensive applications with GPUs and FPGAs,” in *2008 Symposium on Application Specific Processors*, 2008, pp. 101–107.

[36] T. Saegusa, T. Maruyama, and Y. Yamaguchi, “How fast is an FPGA in image processing?” in *2008 International Conference on Field Programmable Logic and Applications*, 2008, pp. 77–82.

[37] W. MacLean, “An evaluation of the suitability of FPGAs for embedded vision systems,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Workshops*, 2005, pp. 131–131.

[38] D. Baumgartner, P. Rossler, and W. Kubinger, “Performance benchmark of dsp and FPGA implementations of low-level vision algorithms,” in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, 2007, pp. 1–8.

[39] AMD, “Microblaze processor reference guide,” Jun 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug984-vivado-microblaze-ref

[40] Jun 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683632/23-1/overview.html

[41] Jul 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview

[42] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu, "Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 79–90. [Online]. Available: https://doi.org/10.1145/3297663.3310305

[43] R. Rajesh, S. J. Darak, A. Jain, S. Chandhok, and A. Sharma, "Hardware–software co-design of statistical and deep-learning frameworks for wideband sensing on zynq system on chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 1, pp. 79–89, 2023.

[44] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," ser. CASES '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 202–212. [Online]. Available: https://doi.org/10.1145/1086297.1086325

[45] F. Siddiqui, S. Amiri, U. I. Minhas, T. Deng, R. Woods, K. Rafferty, and D. Crookes, "FPGA-based processor acceleration for image processing applications," *Journal of Imaging*, vol. 5, no. 1, 2019. [Online]. Available: https://www.mdpi.com/2313-433X/5/1/16

[46] M. Che and Y. Chang, "A hardware/software co-design of a face detection algorithm based on FPGA," in *2010 International Conference on Measuring Technology and Mechatronics Automation*, vol. 1, 2010, pp. 109–112.

[47] D. Honegger, H. Oleynikova, and M. Pollefeys, "Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 4930–4935.

[48] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "Neuraghe: Exploiting CPU-FPGA synergies for efficient and flexible cnn inference acceleration on zynq socs," *ACM Trans.*

Reconfigurable Technol. Syst., vol. 11, no. 3, dec 2018. [Online]. Available: https://doi.org/10.1145/3284357

[49] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 35–44. [Online]. Available: https://doi.org/10.1145/3020078.3021727

[50] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on CPU-FPGA heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 255–265. [Online]. Available: https://doi.org/10.1145/3373087.3375312

[51] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High performance binary neural networks on the xeon+FPGA™ platform," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[52] H. Cho, J. Lee, and J. Lee, "Farnn: FPGA-GPU hybrid acceleration platform for recurrent neural networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1725–1738, 2022.

[53] M. Hosseinabady, M. A. B. Zainol, and J. L. Núñez-Yáñez, "Heterogeneous FPGA+GPU embedded systems: Challenges and opportunities," *ArXiv*, vol. abs/1901.06331, 2019.

[54] Y. Tu, S. Sadiq, Y. Tao, M.-L. Shyu, and S.-C. Chen, "A power efficient neural network implementation on heterogeneous FPGA and GPU devices," in *2019 IEEE 20th International Conference on Information Reuse and Integration for Data Science (IRI)*, 2019, pp. 193–199.

[55] W. Carballo-Hernández, M. Pelcat, and F. Berry, "Why is FPGA-GPU heterogeneity the best option for embedded deep neural networks?" *ArXiv*, vol. abs/2102.01343, 2021.

[56] N. Sumeet, K. Rawat, M. Nambiar, and R. Singhal, "Hetero-vis: A framework for latency optimized heterogeneous deployment of convolu-

tional neural networks," in *Euro-Par 2022: Parallel Processing Workshops*. Springer Nature Switzerland, 2023, pp. 171–183.

[57] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A high-throughput and power-efficient FPGA implementation of yolo cnn for object detection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, 2019.

[58] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[59] S. Rivas-Gomez, A. J. Pena, D. Moloney, E. Laure, and S. Markidis, "Exploring the vision processing unit as co-processor for inference," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 589–598.

[60] A. Kyriakos, E.-A. Papatheofanous, B. Charalampos, E. Petrongonas, D. Soudris, and D. Reisis, "Design and performance comparison of cnn accelerators based on the intel movidius myriad2 soc and FPGA embedded prototype," in *2019 International Conference on Control, Artificial Intelligence, Robotics Optimization (ICCAIRO)*, 2019, pp. 142–147.

[61] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool, "Ai benchmark: All about deep learning on smartphones in 2019," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 3617–3635.

[62] W. Fang, Y. Zhang, B. Yu, and S. Liu, "FPGA-based ORB feature extraction for real-time visual slam," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 275–278.

[63] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, "Imaging: In-memory algorithms for image processing," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4258–4271, 2018.

[64] S. Song, S. Lee, J. P. Ko, and J. W. Jeon, "A hardware architecture design for real-time gaussian filter," in *2014 IEEE International Conference on Industrial Technology (ICIT)*, 2014, pp. 626–629.

[65] W. Fang, Y. Zhang, B. Yu, and S. Liu, “FPGA-based orb feature extraction for real-time visual slam,” in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 275–278.

[66] H. Zhang, M. Xia, and G. Hu, “A multiwindow partial buffering scheme for FPGA-based 2-d convolvers,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 2, pp. 200–204, 2007.

[67] S. Asano, T. Maruyama, and Y. Yamaguchi, “Performance comparison of FPGA, GPU and CPU in image processing,” in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 126–131.

[68] J. Jiang, X. Li, and G. Zhang, “Sift hardware implementation for real-time image feature extraction,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 7, pp. 1209–1220, 2014.

[69] L. Bai, Y. Zhao, and X. Huang, “A cnn accelerator on FPGA using depthwise separable convolution,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018.

[70] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, “An FPGA-based cnn accelerator integrating depthwise separable convolution,” *Electronics*, vol. 8, no. 3, 2019.

[71] J. Ryu and T. H. Nishimura, “Fast image blurring using lookup table for real time feature extraction,” in *2009 IEEE International Symposium on Industrial Electronics*, 2009, pp. 1864–1869.

[72] M. Mese and P. Vaidyanathan, “Look-up table (lut) method for inverse halftoning,” *IEEE Transactions on Image Processing*, vol. 10, no. 10, pp. 1566–1578, 2001.

[73] E. Kadric, D. Lakata, and A. Dehon, “Impact of parallelism and memory architecture on FPGA communication energy,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 4, aug 2016.

[74] E. Kadric, D. Lakata, and A. DeHon, “Impact of memory architecture on FPGA energy consumption,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 146–155.

[75] R. Tessier, V. Betz, D. Neto, A. Egier, and T. Gopalsamy, “Power-efficient ram mapping algorithms for FPGA embedded memory blocks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 278–290, 2007.

[76] P. Garcia, D. Bhowmik, R. Stewart, G. Michaelson, and A. Wallace, “Optimized memory allocation and power minimization for FPGA-based image processing,” *Journal of Imaging*, vol. 5, no. 1, p. 7, Jan 2019.

[77] N. Zhang, X. Wei, L. Chen, and H. Chen, “Three-level memory access architecture for FPGA-based real-time remote sensing image processing system,” in *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)*, 2019, pp. 1–6.

[78] S. Mittal, “A survey of techniques for approximate computing,” *ACM Comput. Surv.*, vol. 48, no. 4, mar 2016.

[79] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A survey of quantization methods for efficient neural network inference,” *CoRR*, vol. abs/2103.13630, 2021.

[80] “Review of deep learning: concepts, cnn architectures, challenges, applications, future directions,” *Journal of Big Data*, vol. 8, no. 1, pp. 1–74, 2021.

[81] R. Reed, “Pruning algorithms-a survey,” *IEEE Transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.

[82] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” 2016.

[83] T.-J. Yang, Y.-H. Chen, and V. Sze, “Designing energy-efficient convolutional neural networks using energy-aware pruning,” 2017.

[84] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015.

[85] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1398–1406.

[86] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, p. 519–530, jun 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462176

[87] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," ser. SPAA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 117–128. [Online]. Available: https://doi.org/10.1145/1989493.1989508

[88] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *CGO 2021*, 2021.

[89] W. Lin and L. Dong, "Adaptive downsampling to improve image compression at low bit rates," *IEEE Transactions on Image Processing*, vol. 15, no. 9, pp. 2513–2521, 2006.

[90] S. Sinha and W. Zhang, "Low-power FPGA design using memoization-based approximate computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 8, pp. 2665–2678, 2016.

[91] Y. Zeng, L. Cheng, G. Bi, and A. Kot, "Integer dcts and fast algorithms," *IEEE Transactions on Signal Processing*, vol. 49, no. 11, pp. 2774–2782, 2001.

[92] S. Niklaus, L. Mai, and F. Liu, "Video frame interpolation via adaptive separable convolution," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 261–270.

[93] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded fpga," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 163–1636.

[94] W. Wang, J. Yan, N. Xu, Y. Wang, and F.-H. Hsu, “Real-time high-quality stereo vision system in FPGA,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 10, pp. 1696–1708, 2015.

[95] F. Steinbrücker, J. Sturm, and D. Cremers, “Volumetric 3d mapping in real-time on a CPU,” in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 2021–2028.

[96] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, “A fast and efficient sift detector using the mobile GPU,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 2674–2678.

[97] O. Rioul and P. Duhamel, “Fast algorithms for discrete and continuous wavelet transforms,” *IEEE Transactions on Information Theory*, vol. 38, no. 2, pp. 569–586, 1992.

[98] O. Fialka and M. Cadik, “Fft and convolution performance in image filtering on GPU,” in *Tenth International Conference on Information Visualisation (IV’06)*, 2006, pp. 609–614.

[99] D. Zhang, X. Shen, and Y. Song, “The implementation of large fft convolution on heterogeneous multicore programmable system,” in *2016 International Conference on Integrated Circuits and Microsystems (ICICM)*, 2016, pp. 349–353.

[100] B. Qiao, O. Reiche, F. Hannig, and J. Teich, “From loop fusion to kernel fusion: A domain-specific approach to locality optimization,” in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 242–253.

[101] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, and P. Sadayappan, “Domain-specific optimization and generation of high-performance GPU code for stencil computations,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, 2018.

[102] O. Reiche, K. Häublein, M. Reichenbach, M. Schmid, F. Hannig, J. Teich, and D. Fey, “Synthesis and optimization of image processing accelerators using domain knowledge,” *Journal of Systems Architecture*, vol. 61, no. 10, pp. 646–658, 2015.

[103] V. STRASSEN, “Gaussian elimination is not optimal.” *Numerische Mathematik*, vol. 13, pp. 354–356, 1969. [Online]. Available: http://eudml.org/doc/131927

[104] Y. Zhao, D. Wang, L. Wang, and P. Liu, “A faster algorithm for reducing the computational complexity of convolutional neural networks,” *Algorithms*, vol. 11, no. 10, 2018. [Online]. Available: https://www.mdpi.com/1999-4893/11/10/159

[105] S. Winograd, *Arithmetic Complexity of Computations*, ser. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1980.

[106] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4013–4021, 2015.

[107] J. Yepez and S.-B. Ko, “Stride 2 1-d, 2-d, and 3-d winograd for convolutional neural networks,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 853–863, 2020.

[108] K. E. Batcher, “Sorting networks and their applications,” in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS ’68 (Spring). New York, NY, USA: Association for Computing Machinery, 1968, p. 307–314.

[109] M. Kim, D. Kim, M. Sung, and W. W. Ro, “An accelerated separable median filter with sorting networks,” in *2015 IEEE International Conference on Image Processing (ICIP)*, 2015, pp. 803–807.

[110] H. Kim, H. Nam, W. Jung, and J. Lee, “Performance analysis of cnn frameworks for GPUs,” in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 55–64.

[111] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, “Energy-efficient cnn implementation on a deeply pipelined FPGA cluster,” *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016.

[112] Xilinx, *Vitis Unified Software Platform Documentation*, ug1400 (v2019.2) ed., Xilinx, San Jose, California, United States, 3 2020.

[113] Mathworks, “Simulink - simulation and model-based design.” [Online]. Available: https://www.mathworks.com/products/simulink.html?s_tid=hp_products_simulink

[114] Intel, “High-level synthesis compiler - intel® hls compiler.” [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html

[115] Janestreet, “janestreet/hardcaml.” [Online]. Available: https://github.com/janestreet/hardcaml

[116] Cadence, “Stratus high-level synthesis.” [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[117] AUGH, “Augh.” [Online]. Available: http://tima.imag.fr/sls/research-projects/augh/

[118] U. of illnois, “Shang.” [Online]. Available: https://github.com/etherzhhb/Shang

[119] P. Solod, N. Jindapetch, K. Sengchuai, A. Booranawong, P. Hoyingcharoen, S. Chumpol, and M. Ikura, “Memory optimization for accelerating hough transform on FPGA using high level synthesis,” in *2019 IEEE International Circuits and Systems Symposium (ICSyS)*, 2019, pp. 1–4.

[120] J. Cong, B. Liu, R. Prabhakar, and P. Zhang, “A study on the impact of compiler optimizations on high-level synthesis,” in *LCPC*, 2012.

[121] J. Cong, M. Huang, and Y. Zou, “Accelerating fluid registration algorithm on multi-FPGA platforms,” 09 2011, pp. 50–57.

[122] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, “High level synthesis of stereo matching: Productivity, performance, and software constraints,” in *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–8.

[123] Y. Liang, K. Rupnow, Y. Li, D. Min, M. Do, and D. Chen, “High-level synthesis: Productivity, performance, and software constraints,” *Journal of Electrical and Computer Engineering*, vol. 2012, 02 2012.

[124] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for FPGAs," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 89–96.

[125] C. Li, Y. Bi, Y. Benezeth, D. Ginhac, and F. Yang, "High-level synthesis for FPGAs: Code optimisation strategies for real-time image processing," *Journal of Real-Time Image Processing*, vol. 14, 10 2017.

[126] K. W. N. Corp., "C-based behavioral synthesis and verification analysis on industrial design examples." *ASP-DAC '04 Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, 2004.

[127] A. Ishikawa, N. Fukushima, A. Maruoka, and T. Iizuka, "Halide and genesis for generating domain-specific architecture of guided image filtering," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[128] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, and A. Wallace, "Ripl: A parallel image processing language for FPGAs," vol. 11, no. 1, 2018. [Online]. Available: https://doi.org/10.1145/3180481

[129] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2016.

[130] J. Serot, F. Berry, and S. Ahmed, "Implementing stream-processing applications on FPGAs: A dsl-based approach," in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 130–137.

[131] J. Janneck, M. Mattavelli, M. Raulet, and M. Wipliez, "Reconfigurable video coding: A stream programming approach to the specification of new video coding standards," *MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, 01 2010.

[132] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An opencl FPGA benchmark suite," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 141–148.

[133] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, “GpuCV: An opensource GPU-accelerated framework for image processing and computer vision,” in *Proceedings of the 16th ACM International Conference on Multimedia*, ser. MM ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1089–1092.

[134] 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.

[135] C. Qi, Y. Wang, H. Wang, Y. Lu, S. S. Subramanian, F. Cahill, C. Tuohy, V. Li, X. Qian, D. Crews, L. Wang, S. Roy, A. Deidda, M. Power, N. Hanrahan, R. Richmond, U. Cheema, A. Raha, A. Palla, G. Baugh, and D. Mathaikutty, “Vpu-em: An event-based modeling framework to evaluate npu performance and power efficiency at scale,” 2023.

[136] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, “Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels,” in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[137] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, “Mlperf inference benchmark,” in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA ’20. IEEE Press, 2020, p. 446–459.

[138] M. Mazumder, C. Banbury, X. Yao, B. Karlaš, W. G. Rojas, S. Diamos, G. Diamos, L. He, A. Parrish, H. R. Kirk, J. Quaye, C. Rastogi, D. Kiela, D. Jurado, D. Kanter, R. Mosquera, J. Ciro, L. Aroyo, B. Acun, L. Chen, M. S. Raje, M. Bartolo, S. Eyuboglu, A. Ghorbani, E. Goodman, O. Inel, T. Kane, C. R. Kirkpatrick, T.-S. Kuo, J. Mueller, T. Thrush, J. Vanschoren, M. Warren, A. Williams, S. Yeung, N. Ardalani, P. Paritosh, C. Zhang, J. Zou, C.-J.

Wu, C. Coleman, A. Ng, P. Mattson, and V. J. Reddi, “Dataperf: Benchmarks for data-centric ai development,” 2023.

[139] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[140] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, “The scalable heterogeneous computing (shoc) benchmark suite,” in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010, p. 63–74.

[141] B. Hu and C. J. Rossbach, “Mirovia: A benchmarking suite for modern heterogeneous computing,” *ArXiv*, vol. abs/1906.10347, 2019.

[142] M. Blott, L. Halder, M. Leeser, and L. Doyle, “Qutibench: Benchmarking neural networks on heterogeneous hardware,” *J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 4, dec 2019.

[143] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Survey and benchmarking of machine learning accelerators,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2019, pp. 1–9.

[144] [Online]. Available: https://xilinx.github.io/Vitis_Libraries/dsp/2022.1/index.html

[145] [Online]. Available: https://www.xilinx.com/support/documents/ip_documentation/xfft/v9_1/pg109-xfft.pdf

[146] M. Meyer, T. Kenter, and C. Plessl, “Evaluating fpga accelerator performance with a parameterized opencl adaptation of selected benchmarks of the hpcchallenge benchmark suite,” in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. IEEE, 2020, pp. 10–18.

[147] [Online]. Available: https://docs.amd.com/r/en-US/pg286-v-demosaic

[148] [Online]. Available: https://docs.amd.com/r/3.2-English/pg338-dpu/Overview

[149] M. Geier, F. Franzen, and S. Chakraborty, "Hardware-accelerated data acquisition and authentication for high-speed video streams on future heterogeneous automotive processing platforms," in *IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018, pp. 1–6.

[150] F. Dias, F. Berry, J. Serot, and F. Marmoiton, "Hardware, design and implementation issues on a FPGA-based smart camera," in *2007 First ACM/IEEE International Conference on Distributed Smart Cameras*, 2007, pp. 20–26.

[151] F. Bruhn, K. Brunberg, J. Hines, L. Asplund, and M. Norgren, "Introducing radiation tolerant heterogeneous computers for small satellites," in *IEEE Aerospace Conference*, 2015, pp. 1–10.

[152] Xilinx. Ug1137 - zynq ultrascale+ mpsoc: Software developers guide (v2020.1). [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1137-zynq-ultrascale-mpsoc-swdev.pdf

[153] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, 1999, pp. 1150–1157 vol.2.

[154] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417.

[155] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571.

[156] J. MCCALPIN, "Stream : Sustainable memory bandwidth in high performance computers," *http://www.cs.virginia.edu/stream/*, 2006.

[157] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," 10 2006.

[158] “Hwmonitor,” Mar 2023. [Online]. Available: https://www.cpuid.com/softwares/hwmonitor.html

[159] “Nvidia system management interface,” Mar 2023. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface

[160] “xbutil,” Mar 2023. [Online]. Available: https://xilinx.github.io/XRT/master/html/xbutil.html

[161] D. Bhowmik and K. Appiah, “Embedded vision systems: A review of the literature,” in *International Symposium on Applied Reconfigurable Computing*. Springer, 2018, pp. 204–216.

[162] J. Vourvoulakis, J. Kalomiros, and J. Lygouras, “Fully pipelined FPGA-based architecture for real-time SIFT extraction,” *Microprocessors and Microsystems*, vol. 40, pp. 53–73, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933115001921

[163] G. Chaple and R. D. Daruwala, “Design of Sobel operator based image edge detection algorithm on FPGA,” in *2014 International Conference on Communication and Signal Processing*, 2014, pp. 788–792.

[164] P. Leyva, G. Doménech-Asensi, J. Garrigós, J. Illade-Quinteiro, V. M. Brea, P. López, and D. Cabello, “Simplification and hardware implementation of the feature descriptor vector calculation in the SIFT algorithm,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[165] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[166] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[167] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[168] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[169] R. Andraka, “A survey of cordic algorithms for FPGA based computers,” in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, 1998, pp. 191–200.

[170] “Usb-to-pmbus interface,” Mar 2023. [Online]. Available: https://www.stg-maximintegrated.com/en/products/power/switching-regulators/MAXPOWER.html

[171] J. Liu, D. Liu, W. Yang, S. Xia, X. Zhang, and Y. Dai, “A comprehensive benchmark for single image compression artifacts reduction,” in *arXiv*, 2019.

[172] L.-C. Chiu, T.-S. Chang, J.-Y. Chen, and N. Y.-C. Chang, “Fast SIFT design for real-time visual feature extraction,” *IEEE Transactions on Image Processing*, vol. 22, no. 8, pp. 3158–3167, 2013.

[173] K. Mizuno, H. Noguchi, G. He, Y. Terachi, T. Kamino, T. Fujinaga, S. Izumi, Y. Ariki, H. Kawaguchi, and M. Yoshimoto, “A low-power real-time SIFT descriptor generation engine for full-HDTV video recognition,” *IEICE Transactions*, vol. 94-C, pp. 448–457, 04 2011.

[174] J. Vourvoulakis, J. Kalomiros, and J. Lygouras, “Fully pipelined FPGA-based architecture for real-time SIFT extraction,” *Microprocessors and Microsystems*, vol. 40, pp. 53–73, 2016.

[175] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[176] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.

[177] “Otii arc pro,” Mar 2023. [Online]. Available: https://www.qoitech.com/otii-arc-pro/