Designing Programmable Domain-Specific Overlay Accelerators on FPGAs

Thesis submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Engineering

by

Ziaul Haque Choudhury
20162150
ziaul.c@research.iiit.ac.in

International Institute of Information Technology, Hyderabad
(Deemed to be University)
Hyderabad - 500 032, INDIA
June 2024

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled **"Designing Programmable Domain-Specific Overlay Accelerators on FPGAs"** by **Ziaul Haque Choudhury**, has been carried out under my supervision and is not submitted elsewhere for a degree.

_____ _____

Date Adviser: Prof. Venkata Suresh Reddy Purini

Dedicated to

Syeda Sehnaz Ahmed

&

Alveena Choudhury

To the pillars of my life, my beloved wife Sehnaz and my precious daughter Alveena, your love and support have been my constant inspiration.

Acknowledgments

I extend my deepest gratitude to my advisor, Dr. Suresh Purini, whose guidance, support, and expertise have been invaluable throughout the journey of this thesis. His insights and encouragement have shaped the direction of my research and significantly contributed to its success.

I would like to express my appreciation to my dedicated lab mates, Anish, Shashwat Kandelwal, and Shashwat Srivastava, for their collaborative spirit, shared enthusiasm, and valuable discussions. Their camaraderie has made the research environment enjoyable and productive.

The learning curve for this thesis was exceptionally steep, given my background in theory and data structures from my master's thesis. I am grateful for the challenges and growth this experience has provided, expanding my knowledge and skills in unforeseen ways.

I would like to acknowledge and thank the institution, IIIT Hyderabad, for the efficient and timely processing of the formalities related to my thesis. Their administrative support has streamlined the academic aspects of this research.

To my parents, Mainul Haque Choudhury, Mother Rezina Begum, and Brother Aliul Haque Choudhury, I express my heartfelt gratitude for their unwavering support during the initial phases of my thesis. Their encouragement and understanding have been the bedrock of my academic pursuits.

This thesis stands as a culmination of the collective efforts and support from these individuals and institutions. I am truly thankful for the guidance, collaboration, and encouragement that have shaped this research endeavor.

Abstract

For about fifty years, hardware designers have been relying on different types of semiconductor scaling laws, like Moore's Law and Dennard scaling, to achieve gains in performance. The industry became used to processor performance per watt, doubling approximately every 18 months. Over the past decade, we have seen the breakdown of these scaling predictions. With the old certainties of scaling silicon geometries gone forever, the industry is already changing. The number of cores has increased in a single die. SoCs like mobile phone processors are combining application-specific co-processors, GPUs, and DSPs in different configurations to maintain the performance scaling trend. However, in a post-Dennard, post-Moore world, further processor specialization will be needed to achieve performance improvements. Emerging applications such as artificial intelligence and vision are demanding heavy computational performance that cannot be met by conventional architectures. This inevitably leads to creating special purpose, domain-specific accelerators.

A domain-specific accelerator (DSA) is a processor or set of processors that are optimized to perform a narrow range of computations. They are tailored to meet the needs of the algorithms required for their domain. For example, an AI accelerator might have an array of elements, including multiply-accumulate functionality, to efficiently undertake matrix operations. Google's Tensor Processing Unit (TPU), Neural Engine in Apple's M1 processor, and Xilinx Vitis-AI Engine are popular ASIC-based DSAs. ASIC bases DSAs provide significant gains in performance and power efficiency. However, due to the long design cycles and high engineering costs, they may not cope with the ever-evolving computation landscape. Field-Programmable Gate Arrays (FPGAs) offer advantages over Application-Specific Integrated Circuits (ASICs) in certain scenarios due to their flexibility and quicker development times. FPGAs are programmable hardware, allowing users to configure their functionality after manufacturing. In contrast, ASICs are hardwired for specific tasks. This flexibility makes FPGAs suitable for prototyping, testing, and adapting to changing requirements without the need for costly chip redesigns. Developing an ASIC is a complex and time-consuming process, often taking several months to years. FPGAs can be programmed and deployed much faster, making them advantageous when speed-to-market is crucial. ASICs can be expensive to design and manufacture, especially for low-volume or rapidly changing applications. FPGAs have a lower initial cost, as they don't require custom chip fabrication. This cost-effectiveness is notable for small production runs or research projects.

Therefore, FPGAs are rapidly evolving towards becoming an alternative to custom ASICs to design DSAs because of their low power consumption and a higher degree of parallelism. Designing DSAs on

an FPGA requires carefully calibrating the FPGA compute and memory resources to achieve optimal throughput from a given device. Hardware Descriptive Languages (HDL) like Verilog have been traditionally used to design FPGA hardware. HDLs are generic and not geared towards any domain. Also, the user has to put in much effort to describe the hardware at the register transfer level using the HDL. A recent trend is emerging wherein existing HDLs are used to create carefully handwritten templates suiting a specific domain. A compiler framework weaves together these templates to generate the DSA for accelerating the domain computations. This approach requires expensive design synthesis and FPGA re-flashing for accelerating different algorithms from the domain. In many edge and deeply embedded applications, this may not be feasible. Further, these days cloud companies are offering FPGA bases acceleration as a service. A large cluster of custom accelerators supports these services at the backend.

In contrast to this fixed-function hardware approach, where the DSA gets tied with a specific function, an alternative design approach of overlay accelerators is gaining prominence. Overlays are DSAs resembling a processor, which is synthesized and flashed on the FPGA once but is flexible enough to process a broad class of computations soft reconfiguration. Over the last couple of years, a few design approaches have developed for overlay accelerators. Overlay designs exist that resemble a processor controlled through an instruction set; the Xilinx Vitis-AI Engine is a prime example of such a design. However, such a *homogeneous* approach often leads to inefficiencies arising out of the fetch-decode-execute model of processor design. Instruction-based overlays spend significant energy on instruction overhead rather than actual computation. To solve this problem of homogeneous overlay accelerators, a heterogeneous design methodology is proposed. A heterogeneous overlay accelerator contains multiple small homogeneous units. These units have a simple, non-instruction-based interface optimized specifically for specific workload characteristics. A heterogeneous design mainly aims to optimize throughput by concurrently processing multiple inputs over the different homogenous accelerator units in a pipelined fashion. While this approach reduces overhead, the latency will be longer. Also, as the variations in domain workloads grow, so does the complexity of these heterogeneous architectures.

This thesis proposes the creation of uniform, non-instruction-based overlays, introducing two distinct overlays: FlowPix and FlexNN. FlowPix is specialized for image processing pipelines, while FlexNN is optimized for applications employing Convolutional Neural Networks (CNNs). We believe overlays represent a promising path toward achieving domain-specific acceleration, presenting adaptable and efficient solutions to address the ever-changing requirements of contemporary computing tasks. The concepts presented herein are expected to provide valuable insights for crafting efficient overlays with a superior performance-to-area ratio.

FlowPix is a DSL-based overlay accelerator for image processing applications. The DSL programs are expressed as pipelines, with each stage representing a computational step in the overall algorithm. We implement 15 image-processing benchmarks using FlowPix on a Virtex-7-690t FPGA. The benchmarks range from simple blur operations to complex pipelines like Lucas-Kande optical flow. We compare FlowPix against existing DSL-to-FPGA frameworks like Hetero-Halide and Vitis Vision library that generate fixed-function hardware. On most benchmarks, we see up to 25% degradation in la-

tency with approximately a 1.7x to 2x increase in the FPGA LUT consumption. Our ability to execute any benchmark without incurring the high costs of hardware synthesis, place-and-route, and FPGA re-flashing justifies the slight performance loss and increased resource consumption that we experience. FlowPix achieves an average frame rate of 170 FPS on HD frames of 1920x1080 pixels in the implemented benchmarks.

The **FlexNN** overlay efficiently processes CNNs and can be scaled based on the available compute and memory resources of the FPGA. The overlay is configured on the fly through control words sent by the host on a per-layer basis. Unlike current overlays, our architecture exploits all forms of parallelism inside a convolution operation. A constraint system is employed at the host end to find out the per-layer configuration of the overlay that uses all forms of parallelism in the processing of the layer, resulting in the highest throughput for that layer. We studied the effectiveness of our overlay by using it to process AlexNet, VGG16, YOLO, MobileNet, and ResNet-50 CNNs targeting a Virtex7 and a bigger Ultrascale+VU9P FPGAs. The chosen CNNs have a mix of different types of convolution layers and filter sizes, presenting a good variation in model size and structure. Our accelerator reported a maximum throughput of 1200 GOps/sec on the Virtex7, an improvement of 1.2x to 5x over the recent designs. Also, the reported performance density, measured in Giga operations per second per KLUT, is 1.3x to 4x improvement over existing works. Similar speed-up and performance density is also observed for the Ultrascale+VU9P FPGA.

Contents

List of Figures

List of Tables

Related Publications

Journals

- **Ziaul Choudhury**, Anish Gulati, and Suresh Purini. 2023. **FlowPix: Accelerating Image Processing Pipelines on an FPGA Overlay using a Domain Specific Compiler**. *ACM Transactions on Architecture and Code Optimization (ACM TACO)* (October 2023). https://doi.org/10.1145/3629523

- **Ziaul Choudhury**, Shashwat Shrivastava, Lavanya Ramapantulu, and Suresh Purini. 2022. **An FPGA Overlay for CNN Inference with Fine-grained Flexible Parallelism.** *ACM Transactions on Architecture and Code Optimization (ACM TACO)* 19, 3, Article 34 (September 2022), 26 pages. https://doi.org/10.1145/3519598

- Biji George, Om Ji Omer, **Ziaul Choudhury**, Anoop V, and Sreenivas Subramoney. 2022. **A Unified Programmable Edge Matrix Processor for Deep Neural Networks and Matrix Algebra.** *ACM Trans. Embed. Comput. Syst* (ACM TECS). 21, 5, Article 63 (September 2022), 30 pages. https://doi.org/10.1145/3524453

Conferences

- Shashwat Shrivastava, **Ziaul Choudhury**, Shashwat Khandelwal, Suresh Purini: **Accelerating local laplacian filters on fpgas**, *2020 30th International Conference on Field-Programmable Logic and Applications (FPL), August 2020, pp: 304-309*

- Shashwat Khandelwal, **Ziaul Choudhury**, Shashwat Shrivastava, Suresh Purini: **FPGA accelerator for stereo vision using semi-global matching through dependency relaxation**, *2020 30th International Conference on Field-Programmable Logic and Applications (FPL), August 2020, pp: 109-114*

- Suresh Purini, Vinamra Benara, **Ziaul Choudhury**, Uday Bondhugula: **Bitwidth customization in image processing pipelines using interval analysis and SMT solvers**, *Proceedings of the 29th International Conference on Compiler Construction (CC), Februray 2020, pp: 167-178*

- Shivani Maurya, **Ziaul Choudhury**, Suresh Purini: **Accuracy Configurable FPGA Implementation of Harris Corner Detection**, *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2022)*

Chapter 1

Introduction

Moore's Law remained steadfast for many decades but started to exhibit signs of slowing down around the year 2000. By the time 2018 rolled around, a noticeable disparity had emerged, with Moore's original predictions differing from the current technological capabilities by a factor of roughly 15, an observation Moore himself had acknowledged as inevitable back in 2003 [64]. Alongside Moore's Law, there was another concept known as "Dennard scaling," introduced by Robert Dennard in 2007 [11], which posited that as transistor density increased, the power consumption per transistor would decrease, resulting in a relatively constant power usage per square millimeter of silicon. However, Dennard scaling also started to experience a significant slowdown in 2007 and had nearly faded away by 2012. With Dennard scaling on the wane and Moore's Law showing reduced effectiveness, the era of rapid performance improvements was replaced by a more modest annual improvement rate of just a few percent, as illustrated in Figure 1.1. Achieving higher rates of performance enhancement, akin to what was witnessed in the 1980s and 1990s, will necessitate the development of novel architectural approaches that leverage integrated circuit capabilities more efficiently.

Emerging technologies and innovative design paradigms are currently reshaping the computing landscape, and Field-Programmable Gate Arrays (FPGAs) have emerged as a pivotal driving force for ongoing advancements. As the once-dominant Moore's Law and Dennard scaling have gradually lost momentum, the traditional path to enhancing performance has become obsolete. FPGAs, highly adaptable hardware platforms that can be reconfigured on-the-fly to execute specialized tasks, present a compelling alternative to the conventional General-Purpose Graphics Processing Units (GPUs). While GPUs excel in handling massive parallelism and data-intensive tasks, FPGAs provide developers with the ability to tailor hardware solutions directly to their specific applications, resulting in unparalleled levels of efficiency and performance [24]. Take, for example, the acceleration of real-time data analytics. Historically, GPUs, with their parallel processing capabilities, have been the preferred choice for such tasks. However, FPGAs can be meticulously crafted to efficiently handle specific data manipulation routines, leading to significant reductions in latency and power consumption. This advantage becomes particularly pronounced in edge computing scenarios where FPGAs' lower energy footprint and adaptability play a crucial role.

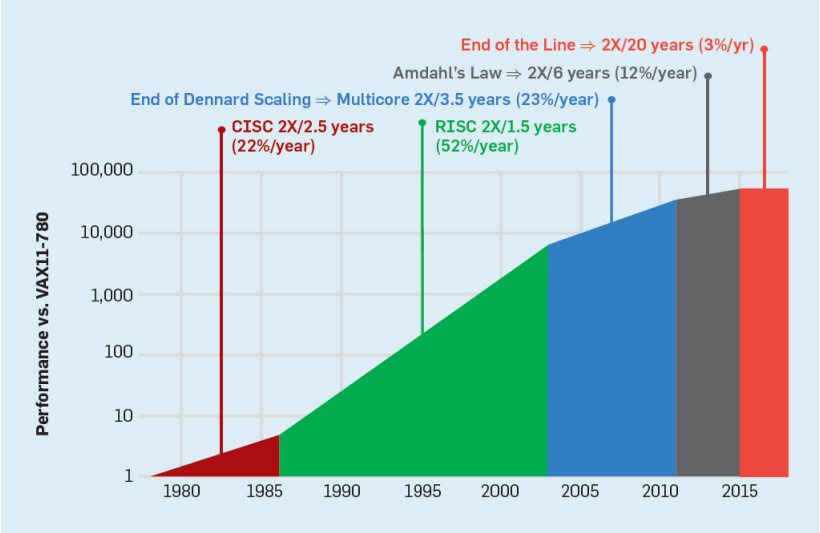


Figure 1.1: Growth of computer performance using integer programs (SPECintCPU). Source [36]

An equally noteworthy example lies within the realm of machine learning. Historically, GPUs have been the workhorses for deep learning training, primarily due to their capacity to process extensive data batches concurrently. In contrast, FPGAs present the potential to design tailored hardware architectures specifically for individual neural network layers. This approach, often referred to as "hardware customization," has the potential to yield significant performance improvements, especially in tasks like accelerating inference, where the hardware can be precisely matched to the neural network's requirements. Furthermore, FPGAs serve as a bridge between hardware and software design. While GPUs demand expertise in CUDA or OpenCL programming languages, FPGAs empower developers to create hardware accelerators using high-level languages such as OpenCL or even Python, rendering them more accessible to a broader range of programmers. In the broader context, this transformation signifies more than just a shift in hardware; it marks a paradigm shift in computing. FPGAs, with their adaptability, efficiency, and capacity to cater to specialized workloads, do not merely substitute for GPUs but rather open up a new frontier capable of unlocking unprecedented performance enhancements.

Table 1.1 presents a comprehensive comparison between FPGAs, CPUs, and GPUs. Each technology is assessed based on key attributes. FPGAs stand out for their reconfigurable logic, allowing high customization to tailor hardware to specific tasks. This adaptability enables FPGAs to excel in real-time analytics and machine learning acceleration, achieving low latency and energy efficiency for targeted applications. In contrast, CPUs offer general-purpose computing and moderate customization, making them suitable for a wide range of tasks. GPUs shine in parallel processing, making them ideal for graphics-intensive workloads and deep learning. However, their customization is limited compared to FPGAs. FPGAs employ hardware description languages for programming, CPUs use high-level languages, and GPUs are often programmed with CUDA or OpenCL [46]. FPGAs are favored for specialized tasks, CPUs for versatility, and GPUs for parallelism. Energy efficiency varies among the

Table 1.1: Comparison of FPGAs, CPUs, and GPUs

Aspect	FPGAs	CPUs	GPUs
Functionality	Reconfigurable logic	General-purpose	Parallel processing
Customization	High customization	Limited customization	Limited customization
Parallelism	High potential	Moderate	Very high
Power Efficiency	Varies based on design	Moderate	Varies based on workload
Latency	Low latency for specific tasks	Moderate	Can be high for certain tasks
Specialization	Tailored for specific tasks	Versatile	Specialized for graphics
Programming	Hardware description languages	High-level languages	CUDA, OpenCL
Applications	Real-time analytics, ML acceleration	General computing	Graphics, deep learning
Energy Footprint	Configurable based on design	Moderate	Can be high for intensive tasks

three, with FPGAs' footprint determined by design, CPUs offering moderate efficiency, and GPUs varying based on workload.

The concept of domain-specific architecture (DSA) has gained significant traction in recent years as a strategy to enhance efficiency and performance for specific tasks or applications. General purpose computing architectures, like CPUs and GPUs, are versatile but may not be optimized for every workload. We discuss more about DSAs in the following section.

1.1 Domain-Specific Architectures

A more hardware-centric approach towards increasing the performace/watt is to design architectures tailored to a specific problem domain and offer significant performance (and efficiency) gains for that domain, hence, the name "domain-specific architectures" (DSAs). DSAs are often called accelerators, since they accelerate an application when compared to executing the entire application on a general purpose CPU. Moreover, DSAs can achieve better performance because they are more closely tailored to the needs of the application; examples of DSAs include graphics processing units (GPUs), neural network processors used for deep learning, and processors for software-defined networks (SDNs). DSAs can achieve higher performance and greater energy efficiency for the following reasons:

1. **Task-specific Optimization:** DSAs are designed from the ground up to perform a specific task or set of tasks. This allows architects to optimize every aspect of the hardware, from the instruction set to the memory hierarchy, to match the specific needs of the domain. This level of specialization results in hardware that's finely tuned for the targeted workload, leading to better performance.

2. **Reduced Overhead:** General-purpose architectures are burdened with overhead that comes from being versatile enough to handle various tasks. They include instructions, hardware components, and features that might not be necessary for a specific domain. For example, NVIDIA CUDA

hardware (GPUs) is better at image processing than x86-based CPU hardware because GPUs have thousands of smaller cores that can process many image pixels in parallel, whereas CPUs have fewer, more powerful cores optimized for sequential tasks. GPUs offer higher memory bandwidth, allowing for quicker handling of large image datasets, and utilize Single Instruction, Multiple Data (SIMD) architecture, ideal for operations like convolution. DSAs eliminate these unnecessary components and instructions, which reduces overhead and results in a leaner, more efficient design.

3. **Parallelism and Pipelining:** DSAs can incorporate parallelism and pipelining techniques that are tailored to the specific task's requirements. This allows for better utilization of hardware resources and more efficient execution of tasks. For instance, if a certain task requires heavy vector operations, the DSA can have a highly efficient vector processing unit.

4. **Memory Hierarchy Optimization:** Many tasks have specific memory access patterns that can be exploited for better performance. DSAs can optimize their memory hierarchies, including cache sizes, memory bandwidth, and memory organization, to match these patterns, reducing data access latencies and improving overall performance.

5. **Custom Programming:** Many domains require specific computational patterns that can be expressed using domain-specific languages (DSLs). A DSL make it easier to map the application efficiently to a domain-specific processor, enabling significant speedups for the targeted tasks.

6. **Less Hardware Complexity:** While DSAs are optimized for specific domains, they might have less overall complexity compared to GPUs, which need to accommodate a wide range of tasks. This reduced complexity can lead to better reliability, lower manufacturing costs, and easier design verification.

1.1.1 Example: Google TPU-v1

As an example DSA, consider the Google TPU version 1, see Figure 1.2, which was designed to accelerate neural net inference [41]. The TPU organization is radically different from a general-purpose processor. The main computational unit is a matrix unit, a systolic array structure that provides 256×256 multiply-accumulates every clock cycle. The combination of 8-bit precision, highly efficient systolic structure, SIMD control, and dedication of significant chip area to this function means the number of multiply-accumulates per clock cycle is approximately 100× what a general purpose single-core CPU can sustain.

TPUs demonstrated impressive performance metrics. A single TPUv3 chip was reported to deliver around 420 teraflops of computation power, while the newer TPUv4 was anticipated to surpass 700 teraflops. TPUs excelled in both training and inference speeds, with TPUv3 pods capable of training

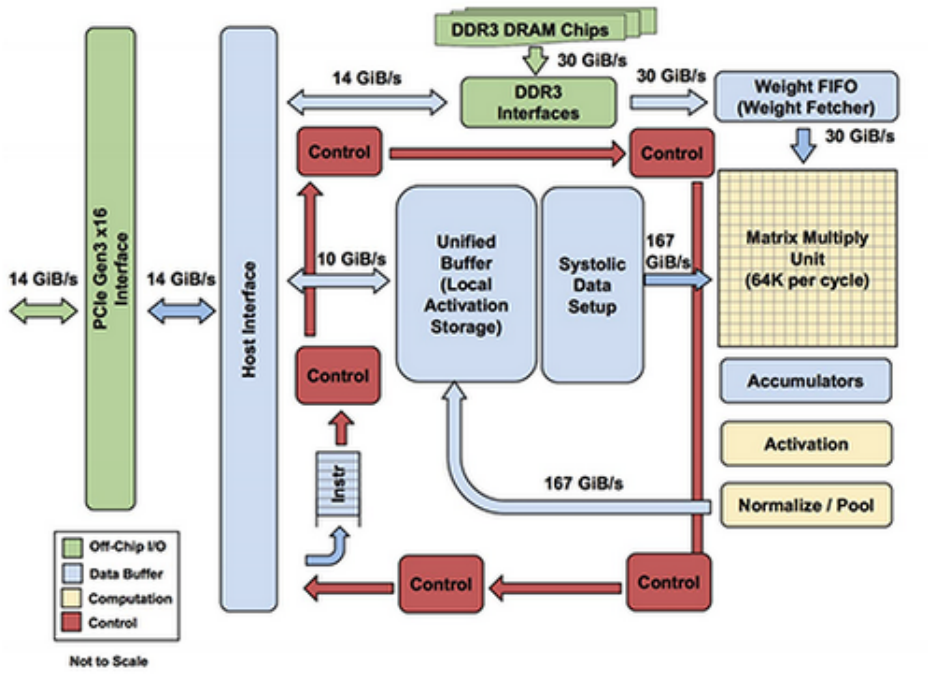


Figure 1.2: Tensor Processing Unit (TPU) for Processing Deep Learning Workloads. Source [41]

large-scale models like BERT in a matter of hours, a task that could take days on other hardware. Inference tasks also benefited from TPUs' rapid processing, with Edge TPUs enabling real-time on-device inference. Notably, TPUs often outperformed GPUs in terms of training speed and energy efficiency, particularly in distributed training scenarios. Google Cloud offered TPUs through its services, facilitating accessibility, and TensorFlow integration streamlined the utilization of TPUs for machine learning tasks. However, it's important to note that performance metrics might have evolved since that time, and the most current information should be sourced from Google's official documentation [42].

1.1.2 Example: AMD AI Engine

AI Engines are designed as 2D arrays (https://www.xilinx.com/products/technology/ai-engine.html), see Figure 1.3, comprising multiple tiles that offer a highly scalable solution across the Versal portfolio. This scalability ranges from 10s to 100s of AI Engines within a single device, catering to the diverse compute requirements of various applications. The advantages encompass software programmability, featuring C programmability with quick compilation and a library-based design for ML framework developers.The deterministic aspects of AI Engines include dedicated instruction and data memories, along with specific connectivity paired with DMA engines for scheduled data movement between AI Engine tiles. In terms of efficiency, these engines provide up to 8 times better silicon area compute density compared to traditional programmable logic DSP and ML implementations. This results in a nominal 40% reduction in power consumption. Each AI Engine tile is

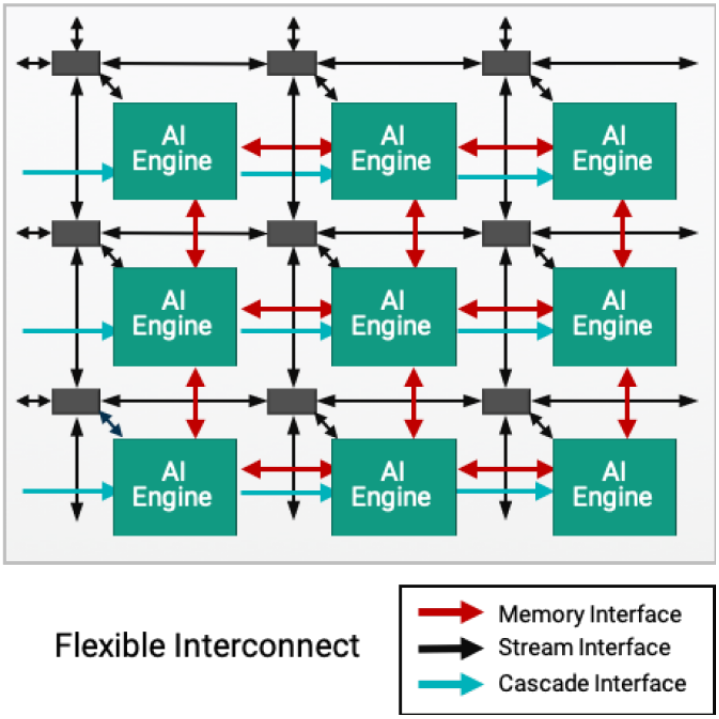


Figure 1.3: The AI Engine Processor from AMD.

composed of a VLIW (Very Long Instruction Word), SIMD (Single Instruction Multiple Data) vector processor optimized for machine learning and advanced signal processing applications. Operating at up to 1.3GHz, the AI Engine processor ensures efficient, high throughput, and low latency functions. Additionally, each tile includes program memory, local data memory, a RISC scalar processor, and various modes of interconnect for handling different types of data communication. AMD offers two types of AI Engines: AIE and AIE-ML (AI Engine for Machine Learning). Both offer substantial performance improvements over previous-generation FPGAs. AIE supports a balanced set of workloads, including ML inference applications and advanced signal processing tasks like beamforming, radar, FFTs, and filters. AIE-ML, with enhanced AI vector extensions and shared memory tiles, outperforms AIE in ML inference-focused applications, while AIE may offer superior performance for certain types of advanced signal processing. AI Engines boast support for diverse workloads and applications, such as advanced DSP for communications, video and image processing, and machine learning inference. They provide native support for real, complex, and floating-point data types, including INT8/16 fixed point, CINT16, CINT32 complex fixed point, and FP32 floating data point. Dedicated hardware features for FFT and FIR implementations include 128 INT8 MACs per tile.

1.1.3 Domain Specific Languages

Domain-specific accelerators need to target high-level operations from the domain to their specific architecture. However, extracting this structure and information from general-purpose languages, which are often low-level in terms of semantics, like Python, Java, C, or Fortran is extremely challenging.

Domain specific languages (DSLs) enable this process and make it possible to program DSAs efficiently. For example, DSLs can make vector, dense matrix, and sparse matrix operations explicit, enabling the DSL compiler to map the operations to the processor efficiently. Examples of DSLs include Matlab, a language for operating on matrices, TensorFlow, a dataflow language used for programming DNNs, P4, a language for programming SDNs, and Halide, a language for image processing specifying high-level transformations. The challenge when using DSLs is how to retain enough architecture independence that software written in a DSL can be ported to different architectures while also achieving high efficiency in mapping the software to the underlying DSA.

1.1.4 FPGAs and Domain Specific Architectures

ASIC-based DSAs such as Scaledeep[92] and Eyeriss[14] can achieve extremely high performance and energy efficiency under certain constraints, but there exist problems of long design cycle and high cost. Reconfigurable logic devices are usually used to verify the necessary performance in the early stages of ASIC design, or in applications requiring rapid iteration. Compared with AISC, it has advantages in cost and flexibility due to the programmable properties, while the performance is competitive. Reconfigurable logic devices include Coarse-grained reconfigurable architecture (CGRA)[95] and Field-programmable gate array (FPGA), which are different in the granularity of customizable circuit components. Although CGRA has certain advantages in terms of reconstruction speed and energy efficiency, it has long design cycle because of the incomplete development ecology.

With its good development ecology, FPGA is a representative ideal carrier for current research on DSA [87]. Through the fine-grained custom architecture, parallel computing can be maintained through the spatial structure. Hence the performance is comparable with ASIC-based custom architectures. In addition, with the development of high-level design tools such as Vitis [44], FPGA has a shorter development cycle. It simplifies the algorithm deployment and the implementation process of computing architecture. At present, FPGA-based custom computing architecture becomes a representative design method of DSA.

1.2 FPGA Overlays and DSAs

Many works cite10.1145/3563946 have proposed various FPGA based DSAs which outperform CPUs and GPUs on the performance-per-watt metric. These architectures can be broadly categorized into two classes. In the first class, for a given algorithm within the domain, a custom architecture is synthesized using one of the available parametric templates [30]. This requires expensive design synthesis

and FPGA re-flashing. In many edge and deeply embedded applications, for example in autonomous driving and robotics, this may not be feasible. Further, these days cloud companies are offering FPGA acceleration[1] as a service like Intel Devcloud and Amazon AWS. These services are supported by a large cluster of custom accelerators at the backend. The algorithm specific hardware accelerators severely constrains the scheduling of different types of workloads and results in hardware resource under utilization.

In contrast to the algorithm specific architectures, the second class consists of **overlay accelerators** [102] which is synthesized and flashed on the FPGA once, but is flexible enough to process a broad class of algorithms belonging to the domain. This addresses the aforementioned problems due to custom accelerators. There are a few design approaches for overlay architectures. For example, we can have an overlay architecture which resembles a processor controlled through an domain specfic instruction set [81]. The same architecture can process the alogrithms belonging to the domain using the instruction set, without the need for re-synthesis. The AI Engine from Xilinx is a good example of this design philosophy (https://www.xilinx.com/products/technology/ai-engine.html). However, such a *homogeneous* approach to process different algorithms can result in limited acceleration, due the pitfalls of the fetch-decode-execute pipeline structure of the hardware.

To solve this problem of homogeneous overlay accelerators, a heterogeneous design methodology is proposed in several previous works [96]. A heterogeneous overlay accelerator contains multiple homogeneous units. Each of these is optimized specifically for a set of workload characteristics. A heterogeneous design mainly aims to optimize throughput by concurrently processing multiple input streams over the different accelerator units in a pipelined fashion. While this approach increases the throughput, the latency will be longer. Also, as the variations in domain algorithms grow, so does the complexity of these heterogeneous architectures.

1.3 Problem Statement

This thesis proposes a new class of homogeneous overlay accelerators that eliminate the reliance on instruction sets for domain-specific computations. These accelerators enhance various algorithms in image processing and convolutional neural networks, outperforming existing HLS-based FPGAs and overlays.

1.4 Contributions

The thesis proposes homogeneous overlay accelerators, which operate without relying on specific instruction sets, for domain specific computations. These proposed accelerators have the capacity to significantly enhance a wide range of algorithms within the realms of image processing and convo-

[1] In this work we use the words accelerator and architecture interchangably

lutional neural networks. These homogenous accelerators outperform existing HLS-based FPGAs and pre-existing overlays. We extensively validate this claim through rigorous testing with micro-benchmark suites.

More precisely, we do the following contributions.

- In contrast to existing homogeneous overlays that depend on scheduled instructions, we take a different approach. We manage computations by configuring the overlay dynamically through control words. This streamlined control design minimizes the complexity of our overlays in terms of FPGA resource consumption.
- Our proposed overlays possess the capability to harness various types of parallelism inherent in the domain-specific algorithms. Consequently, the targeted accelerations yield substantially higher throughput when compared to existing alternatives.
- Finally, we place emphasis on maintaining a relatively straightforward Host-to-FPGA interface. This approach aims to simplify the design of front-end DSLs, making it more accessible and user-friendly.

The remainder of the thesis is structured into three distinct parts, spanning seven chapters. In **Part 1**, our focus is on laying the groundwork for overlay design, including an overview of the existing work in this field. **Part 1** encompasses Chapters 2 and 3. **Part 2** of the thesis introduces both of our overlay designs, which are detailed across Chapters 4 and Chapter 5. Finally, in **Part 3**, we delve into miscellaneous topics, such as optimizing an overlay module for FPGA area efficiency and integrating the overlay with fixed function designs. These aspects are elaborated upon in Chapters 6 and 7.

PART I

Background and Motivation

Chapter 2

Concepts in Overlay Design

This chapter lays the groundwork for the upcoming chapters, especially chapters 4 and 5. It's essential for readers to understand the content in this chapter to grasp the design choices discussed in this thesis and to appreciate better the complexities involved in overlay design. We begin this chapter by explaining FPGAs in the context of accelerator design. Then, we discuss overlay accelerators and when they might be better than the traditional hardware design approach using FPGAs. Later in the chapter, we discuss the key principles for creating efficient overlays on FPGAs, which we use when making design decisions for the overlays in this thesis.

2.1 FPGA Basics

A Field-Programmable Gate Array (FPGA), see Figure 2.1, is a complex integrated circuit designed to be highly flexible and customizable regarding digital logic functions. Unlike traditional Application-Specific Integrated Circuits (ASICs) that have fixed hardware configurations, FPGAs consist of an array of configurable logic blocks (CLBs), interconnection resources, and programmable input/output (I/O) pins.

The CLBs in an FPGA can be likened to small logic cells, each containing Lookup Tables (LUTs) and flip-flops. LUTs are programmable to implement various combinational logic functions, and flip-flops allow for the creation of sequential logic elements. These CLBs can be interconnected using a matrix of programmable routing resources, enabling the designer to create custom digital circuits by defining the connections between these blocks.

What sets FPGAs apart is their reprogrammability. Unlike ASICs, which have fixed circuitry, FPGAs can be configured and reconfigured to perform specific tasks, allowing rapid prototyping and adaptation to changing requirements. The configuration of an FPGA is typically done using a hardware description language (HDL) like Verilog or VHDL. When you load a configuration file onto the FPGA, it becomes the hardware you've described, allowing it to execute specific logic operations in real time.

FPGAs are widely used in industries such as telecommunications, aerospace, automotive, and digital signal processing due to their adaptability and performance. They are valuable tools for hardware de-

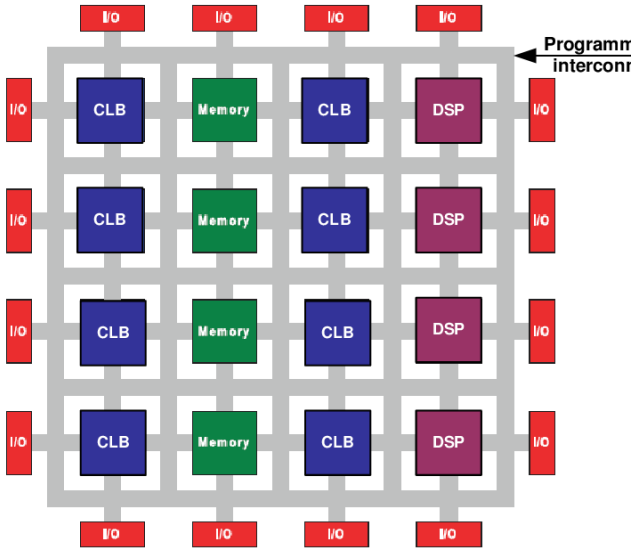


Figure 2.1: Description of a typical FPGA.

signers and engineers who need to create custom, application-specific digital circuits without the time and cost associated with developing custom ASICs. Additionally, FPGAs play a crucial role in tasks like hardware acceleration for AI algorithms, cryptographic processing, and high-speed data processing, where speed, flexibility, and customization are paramount.

2.1.1 FPGA Compute

Table 2.1 outlines different compute resources commonly found on FPGAs, including programmable logic cells, DSP blocks, hard processor cores, high-speed transceivers, and custom hardware accelerators. Programmable Logic Cells, often called PLDs (Programmable Logic Devices), are a fundamental component of FPGA architecture. These cells provide the building blocks for implementing custom digital logic circuits on the FPGA. Within Programmable Logic Cells, Configurable Logic Blocks (CLBs) are the primary components, consisting of Lookup Tables (LUTs), multiplexers, and flip-flops. LUTs allow designers to implement combinational logic functions, while flip-flops enable the creation of sequential logic elements. Carry chains are used to implement arithmetic operations like addition and subtraction efficiently, facilitating the propagation of carry signals through the FPGA fabric. These cells are versatile and can be used for various tasks, including combinational and sequential logic, arithmetic operations, custom state machines, and building hardware accelerators.

Digital Signal Processing (DSP) Blocks, on the other hand, are dedicated hardware blocks within the FPGA designed to accelerate digital signal processing tasks. They are optimized for operations commonly found in applications such as telecommunications, audio processing, image processing, and more. DSP Blocks typically include multiple Multiply-Accumulate (MAC) units, which excel at performing multiply-and-accumulate operations rapidly. These blocks support both fixed-point and

Compute Type	Description
Programmable Logic Cells	Configurable Logic Blocks (CLBs) for logic functions.
Digital Signal Processing (DSP) Blocks	Dedicated blocks for complex math operations.
Hard Processor Cores	Embedded CPUs (e.g., ARM Cortex-A9) for general-purpose computing.
High-Speed Transceivers	Handling high-speed serial protocols (e.g., PCIe, Ethernet).
Custom Hardware Accelerators	User-defined specialized hardware for specific tasks.

Table 2.1: Compute Availability on typical FPGAs

floating-point arithmetic and are often pipelined, allowing for high-throughput processing of data. DSP Blocks find application in filtering, Fast Fourier Transforms (FFTs), modulation and demodulation in communication systems, audio and speech processing, radar and sonar processing, as well as image and video processing, making them essential for various signal processing tasks in FPGA-based systems.

2.1.2 FPGA Memory Hierarchy

Understanding the FPGA memory hierarchy is essential when designing overlays. At the lowest stratum of the hierarchy, the registers provide ultra-fast access times, making them ideal for housing frequently used data and transient values within the FPGA's logic elements. Ascending the hierarchy, on-chip memory, also known as local memory, steps in with a larger storage capacity and relatively swift access. This tier serves as a repository for data necessitating rapid retrieval by multiple processing units within the overlay. Moving further up, block RAMs (BRAMs) offer a balance between capacity and speed. These configurable memory blocks are employed to store more substantial datasets, intermediate computations, and lookup tables. Finally, specific overlay configurations incorporate cache memory as a bridge between on-chip and off-chip memories. Caches preserve frequently accessed data from external memory, reducing access latencies. They employ sophisticated strategies to enhance data locality and alleviate cache misses. In quantifiable terms, registers offer sub-nanosecond access times with a minute capacity, on-chip memory provides access times in the range of a few nanoseconds with a larger storage capacity, BRAMs strike a balance between them, offering access times in the tens of nanoseconds and a more substantial capacity. In contrast, cache memory introduces latency in the tens of nanoseconds but significantly enhances effective memory access speed. Given below are the typical access latencies in cycles and sizes for different memory types in the Virtex-7 XC7V690T FPGA:

1. **Registers**

    - Typical Access Latency: 1 clock cycle
    - Size: Small (a few kilobytes), located within FPGA logic elements.

2. **Block RAMs (BRAM)**

    - Typical Access Latency: 2-3 clock cycles
    - Size: Varies depending on FPGA model, but XC7V690T includes over 4.8 megabits of BRAM in total.

3. **FPGA DRAM (External Memory - DDR3)**

    - Typical Access Latency: 30-100+ clock cycles
    - Size: Virtex-7 XC7V690T supports external DDR3 SDRAM with capacities up to several gigabytes.

Please note that these access latency values are approximate and can vary based on the specific FPGA model, memory technology, clock frequencies, and memory access patterns. For precise information, it's important to refer to the Virtex-7 XC7V690T FPGA's documentation and specifications here https://docs.xilinx.com/v/u/en-US/ds183_Virtex_7_Data_Sheet.

Optimizing data movement strategies is crucial to minimize stalls and ensure efficient utilization of compute units within an overlay. A typical data movement strategy involves a combination of techniques aimed at reducing data access latencies and maximizing the overlap of computation and data transfer. Following are the common data movement strategies typically used in overlay designs.

1. **Double Buffering:** Double buffering involves using two sets of memory buffers—one for data input and another for output. This allows compute units to work on one buffer while the other is read or written, minimizing stalls by overlapping computation and data movement.

2. **Prefetching:** Prefetching proactively loads data into memory before it's needed. This technique reduces memory access latencies by predicting access patterns and fetching data in advance, optimizing spatial and temporal locality.

3. **Local Memory Utilization:** Leveraging on-chip memory (local memory) with lower access latencies, compute units can cache frequently accessed data. Local memory acts as a staging area for data movement, enabling faster access and reducing stalls.

4. **DMA and Pipelining:** Direct Memory Access (DMA) engines offload data movement tasks from compute units. By pipelining transfers, successive movements overlap, reducing stalls and freeing compute units for computation.

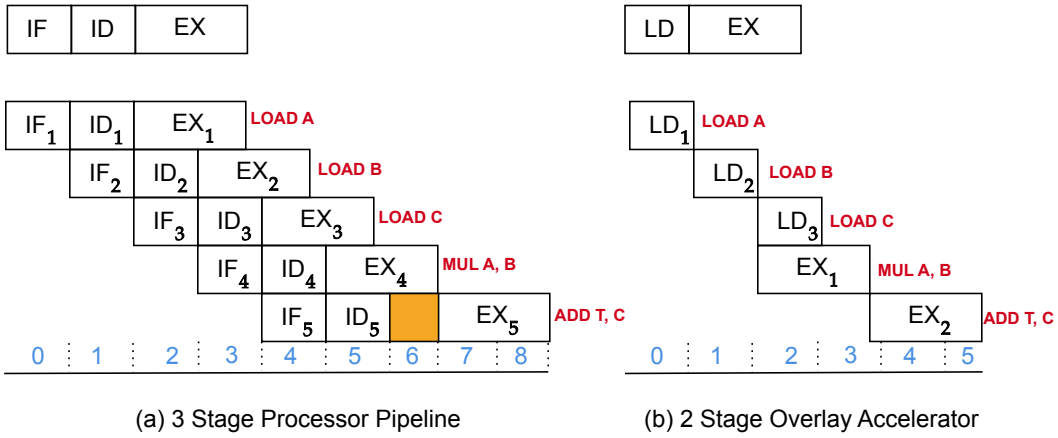


Figure 2.2: Difference between processor pipeline and an Overlay pipeline

5. **Overlap of Computation and Communication:** Task scheduling and coordination should maximize the overlap between compute tasks and data transfers. Ensuring data is available before compute units require it minimizes delays and prevents unnecessary stalls.

6. **Memory Hierarchy Awareness:** Tailoring the data movement strategy to the FPGA's memory hierarchy optimizes stalls. Smaller, faster memory levels house frequently accessed data, while larger levels store data, efficiently utilizing the hierarchy's characteristics.

2.2 Overlay Accelerators

Definition: Overlay accelerator is a specialized hardware component designed to accelerate specific computational tasks or workloads in a flexible and reconfigurable manner. This type of accelerator is created by overlaying a custom hardware design onto the programmable fabric of an FPGA. Below, we explain two scenarios where overlays are a better alternative than the contemporary solutions.

2.2.1 Overlays and General Processors

Let's analyze a computational task expressed as $a \times b + c$ within the context of a hypothetical three-stage processor consisting of fetch, decode, and execute stages. This computation breaks down into five instructions: three for loading data and two for performing computations. You can see this breakdown in Figure 2.2. In this processor model, the fetch and decode stages have a single-cycle latency, while the execute stage requires two cycles. As a result, the entire computation takes nine cycles, as depicted in the figure. Notably, a stall occurs due to the dependency of the second compute operation on the output of the previous computation.

Now, let's consider an overlay design with two stages: one for loading data and the other for executing operations on the fetched data, as shown in Figure 2.2 (b). In this design, the execute stage can

be dynamically configured to perform different operations. The execution pipeline, in this case finishes earlier compared to the processor pipeline. During the first three cycles, the load stage is active and loads the necessary data into registers. The first operation is executed in the third cycle, and the execution unit is configured to perform the second operation in the same cycle. At the end of the fourth cycle, the first operation is completed. Finally, in the fifth and sixth cycles, the final operation is executed using the data from the previous step. This clearly illustrates that the overlay design saves cycles by eliminating the need for the fetch-decode steps of the processor pipeline.

2.2.2 Overlays and Fixed-function Hardware

Consider a real-world scenario where a computer vision system is tasked with applying a 3x3 Gaussian blur operation to image frames in real time. This operation is a fundamental step in image processing, often used for various purposes such as noise reduction and creating artistic effects. We have two design approaches in this scenario: a fixed-function design and an overlay design. The fixed-function approach involves crafting a hardware accelerator using Hardware Description Language (HLS) tools, where the filter weights are hardcoded into the design, and the kernel size remains fixed at 3x3. On the other hand, the overlay design offers greater flexibility. The overlay design's scalability is of importance. In scenarios where larger kernel sizes or higher-resolution images are needed, the overlay design adapts seamlessly. This adaptability future-proofs the system, allowing it to handle evolving processing demands without hardware redesigns. Real-time adjustments to the Gaussian blur operation are essential in applications like video conferencing. Users can customize the blur level dynamically during a call for privacy or quality reasons, showcasing the responsiveness of the overlay design. Development time is often a critical factor. The overlay design significantly shortens this phase due to its adaptability. Quick iterations lead to faster time-to-market, making products featuring this image-processing accelerator more competitive. In summary, the overlay design's dynamic flexibility, superior performance, efficient resource utilization, ease of maintenance, scalability, adaptability to dynamic applications, and shorter development time make it the superior choice compared to the fixed-function design.

2.2.3 CGRAs and Overlays

Overlay accelerators and Coarse-Grained Reconfigurable Arrays (CGRAs) serve as hardware accelerators designed to improve the performance of specific computing tasks. However, they differ in various important aspects. Overlay accelerators typically work at a higher level of granularity, intended to be overlaid onto existing hardware, often based on FPGAs. They utilize pre-designed functional units, which are interconnected to create a customized accelerator for a specific application. CGRAs operate at a finer level of granularity, featuring a grid of configurable processing elements (PEs) that can be dynamically reconfigured to perform various operations. This fine-grained adaptability allows CGRAs to handle a broader spectrum of tasks but may require more intricate programming. Programming overlay accelerators often involves providing a high-level description of the application's functionality, with

Aspect	Overlay Accelerators
Granularity of Reconfigurability	Higher (task-level)
Programming Model	High-level description
Resource Utilization	More efficient for specific tasks
Flexibility	Specialized for specific applications
Aspect	**CGRAs**
Granularity of Reconfigurability	Finer (PE-level)
Programming Model	Low-level, data flow and operation specification
Resource Utilization	More versatile, potentially higher resource usage
Flexibility	Adaptable to a wide range of tasks

Table 2.2: Comparison between Overlay Accelerators and CGRAs

the overlay tool responsible for generating the necessary hardware configurations automatically. This approach makes programming overlays more accessible, especially for software developers. CGRAs usually demand a more low-level programming approach, where developers specify how data flows through the array of processing elements and explicitly define operations. This can be more demanding and time-consuming. Overlay accelerators are more resource-efficient for specific applications since they are customized for the task at hand. However, they may lack versatility in accommodating different workloads. CGRAs are generally more versatile but may consume more resources for a given task due to their fine-grained nature. They can potentially support a broader range of applications without requiring reconfiguration. Overlay accelerators are designed to accelerate specific applications or a set of applications. While they may lack versatility, they offer superior performance for targeted workloads. CGRAs inherently possess more flexibility and are capable of adapting to a wide range of tasks. They are suitable for applications with varying requirements. In summary, overlay accelerators are specialized, user-friendly accelerators designed for integration into existing hardware, delivering high performance for specific tasks. Conversely, CGRAs offer finer-grained adaptability and versatility but may involve more programming and resource management effort. The choice between the two depends on specific application requirements and the trade-offs between performance, programming ease, and flexibility.

2.2.4 Microprogrammed Processors and Overlays

The overlay designs presented in this thesis resemble the design of a microprogrammed processor. Let's start by providing a concise overview of a microprogrammed processor. We will then outline the commonalities and distinctions between overlays and the microprogrammed processor.

A micro-programmed processor design is an architecture where the control unit of the processor is implemented using a microcode. In this approach, the control unit generates microinstructions stored in a control memory (often ROM) to orchestrate the execution of machine-level instructions. Each

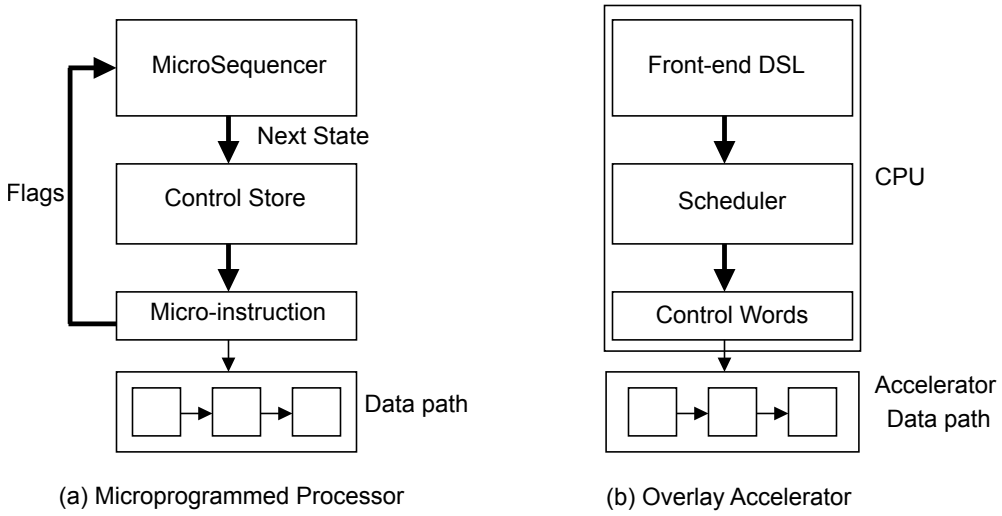


Figure 2.3: Difference between a micro-programmed processor and an overlay accelerator.

microinstruction corresponds to a specific step in the instruction execution process, including fetching operands, performing operations, and updating registers. A microprogram sequencer fetches microinstructions from the control memory in a prescribed sequence, and these microinstructions control the processor's operation. Microinstructions typically include control fields that enable or disable components, specify data paths and set flags. This design offers flexibility, making it easy to modify the processor's behaviour without changing its hardware. However, due to their speed advantages, they tend to be slower than processors with hardwired control units, which are more common in high-performance general-purpose processors. Micro-programmed designs are often used in embedded systems, specialized processors, or research-oriented architectures. In a microprogrammed processor design, the `NEXT` state equation describes how the control unit of the processor determines the next state based on the current state and control signals. This equation is typically implemented using a combinational logic circuit. To create a next-state equation, we must define the state variables and the control signals influencing the transition between states. Assume n state variables $(S^1, S^2, \cdots, S^n)$ representing the current state and m control signals $(C^1, C^2, \cdots, C^m)$ that affect the state transitions.

We will have a next state logic function $F(S^1, S^2, \cdots, S^n, C^1, C^2, \cdots, C^m)$ that determines the next state. The next state equation can be expressed as follows:

$$
\begin{aligned}
S^1_{next} &= F(S^1, S^2, \ldots, S^n, C^1, C^2, \ldots, C^m) \\
S^2_{next} &= G(S^1, S^2, \ldots, S^n, C^1, C^2, \ldots, C^m) \\
&\vdots \\
S^n_{next} &= H(S^1, S^2, \ldots, S^n, C^1, C^2, \ldots, C^m)
\end{aligned}
$$

Here, $S^1_{next}, S^2_{next}, ..., S^n_{next}$ represent the next state values for each state variable $S^1, S^2, ..., S^n$. $F, G, H \cdots$ are combinational logic functions that take the current state variables $(S^1, S^2, ..., S^n)$ and

control signals $(C^1, C^2, ..., C^m)$ as inputs and produce the next state values. To create a specific next-state equation for the microprogrammed processor design, we need to define the states, control signals, and their relationships based on the architecture and functionality of your processor.

Overlays bear a resemblance to a microprogrammed processor in terms of their programmable data path, as illustrated in Figure 2.3. However, a crucial distinction lies in the generation of control words, which occurs externally to the overlay by the host CPU. During runtime, these control words are transmitted to the accelerator on the FPGA through high-speed interfaces, such as PCIe. The specific content of these control words, produced by the CPU, is contingent on the computational tasks mapped onto the accelerator. The scheduler, a component coded on the host side, takes the DSL specification as input and produces the sequence of control words. Once the accelerator is correctly configured with these control words, the host then supplies the actual data for processing by the accelerator.

2.3 Host-Accelerator Interface

The design of the host stack must be meticulous to enhance the usability of the accelerator. This stack comprises key components, including a front-end DSL, a scheduler, and driver code. To ensure optimal usability and efficiency of the accelerator, it is crucial to carefully craft the host stack, prioritizing the optimization of interactions with the accelerator. The first component, the front-end Domain-Specific Language (DSL), plays a pivotal role in enabling seamless communication between the host and the accelerator. It acts as a bridge, providing a specialized interface that abstracts the complexity of the accelerator's architecture, making it more accessible to programmers. The DSL simplifies the process of writing code for the accelerator and enhances the overall user experience. The scheduler is another vital component within the host stack. It is responsible for efficiently managing and distributing tasks to the accelerator, ensuring that computational workloads are optimally balanced and executed in a timely manner. A well-designed scheduler can significantly enhance the overall system's performance by effectively utilizing the accelerator's resources. The driver code forms the backbone of the host stack, serving as the intermediary layer that handles low-level read/write data transfers. The driver code varies between the overlay designs.

2.3.1 Domain Specific Language Characteristics

A Domain-Specific Language (DSL) in the context of overlay accelerators refers to a specialized programming language or framework designed to streamline use of a custom hardware accelerator. An ideal DSL for an overlay accelerator can be characterized as follows.

- **Specialization:** The DSL must be tailored to the specific domain, making it easier for developers to express the domain computations that requires acceleration. The low-level hardware details must be abstracted away to provide a higher-level and more intuitive interface to express the computations.

- **Integration:** DSLs should be composable with the development environment and APIs of the larger computing system, ensuring seamless interaction between software running on the host system and the overlay accelerator. This integration allows data to be efficiently transferred between the CPU and the accelerator and enables developers to call accelerator functions from their software applications.
- **Customization and Flexibility:** The DSL should allow for a degree of customization and specialization within the specific domain. Developers may have the flexibility to fine-tune the accelerator for their unique requirements.

2.3.2 Scheduler

The overlay scheduler, as defined abstractly in Equation (2.1), plays a crucial role in conjunction with the driver. Its primary function is to generate a control word matrix M, and in doing so, it considers two key inputs: the compute DAG denoted as G and a set of constraints designated as C. The compute graph, created by the compiler's front-end parsing of the programmer's DSL code, contains metadata detailing compute dependencies and types. It's worth noting that the scheduler must also possess an abstract model of the overlay, with the constraints playing a pivotal role in this regard. These constraints serve as guidelines to ensure that when generating the compute scheduler, certain conditions are met. For instance, if the overlay features a single divider unit, the scheduler should create schedules in which compute nodes with only a single division operation are executed on the accelerator simultaneously. Beyond just resource limitations, these constraints encompass a wide array of factors. These may include the size of compute nodes, the available host-to-FPGA bandwidth, and other relevant considerations, all aimed at optimizing the functionality and efficiency of the accelerator.

$$Scheduler(G, C) = M = \begin{bmatrix} c_{11} & c_{12} & c_{13} & \dots & c_{1m} \\ c_{21} & c_{22} & c_{23} & \dots & c_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{r1} & c_{r2} & c_{r3} & \dots & c_{rp} \end{bmatrix} \tag{2.1}$$

2.4 Accelerator Data path

The data path of the accelerator refers to the portion of the accelerator's hardware design responsible for performing the actual data processing or computation. The datapath includes the components responsible for executing the specific computations or operations that the accelerator is designed for. This may include arithmetic and logic units (ALUs), memory units, registers, and interconnects. The datapath is designed to efficiently process data, and it operates in parallel with the host CPU or the main processor, offloading specific workloads to improve performance. The design of the datapath is highly dependent on the specific application and requirements of the overlay accelerator. It is optimized to execute the targeted computations efficiently, often utilizing parallelism and other techniques to speed up

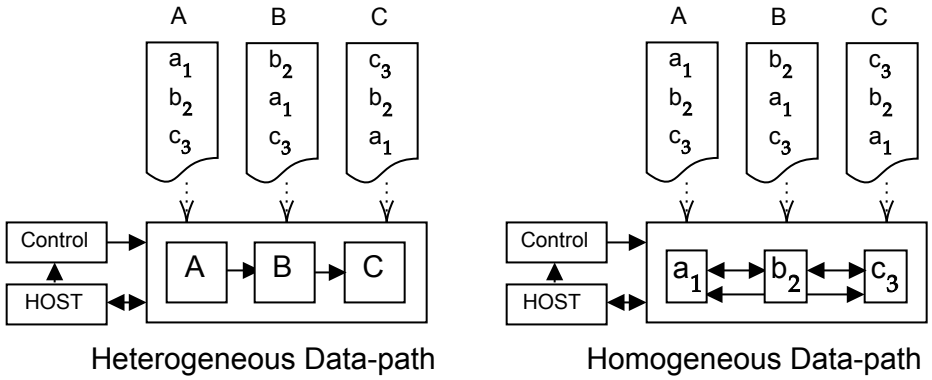


Figure 2.4: Principle 1: A shared or common data path design between functional units.

processing. Below we list some of the key principles that we followed to design the data path for the accelerators proposed in this thesis. These principles are instrumental in guaranteeing that the accelerator achieves exceptional performance while optimizing the use of FPGA real estate.

2.4.1 Principle 1: Common Data path Design

Consider a data path comprising three functional units, denoted as A, B, and C, each capable of performing three distinct micro-operation types: a_1, b_2, and c_3. These functional units are constructed using various permutations of these micro-operations, as depicted in Figure 2.4. There are two alternative approaches to designing this data path. The first approach involves creating a separate module for each functional unit, with the micro-operations arranged in a pipeline according to the required configuration. In contrast, the second approach seeks to extract the individual micro-operations and establish a shared data path. In this case, the interconnections between the stages become more intricate. The three functional units can then be processed over this shared data path by configuring the connections in a specific permutation according to the function unit's requirements.

In the context of designing the data path, we will now explore the advantages and disadvantages of two approaches. The first approach entails the creation of distinct modules for each functional unit, offering several benefits. This method simplifies the design process, improving the manageability and comprehension of each unit's operation. Furthermore, it enables a straightforward pipeline arrangement, enhancing the overall efficiency of the data path. When dealing with a computation involving three functional units - A, B, and C, this approach yields lower latency compared to the second approach. However, it does introduce increased hardware redundancy, as micro-operations are replicated within each functional unit. This redundancy can result in hardware underutilization when not all functional units are engaged. The second approach involves establishing a shared data path, which has its own merits. By pooling resources and connections, it can potentially reduce the hardware footprint and cost. This approach is particularly advantageous when functional units have overlapping micro-operation requirements. It also minimizes the risk of hardware underutilization since the data path will always execute at least one functional unit. Nonetheless, it does come with the trade-off of more complex

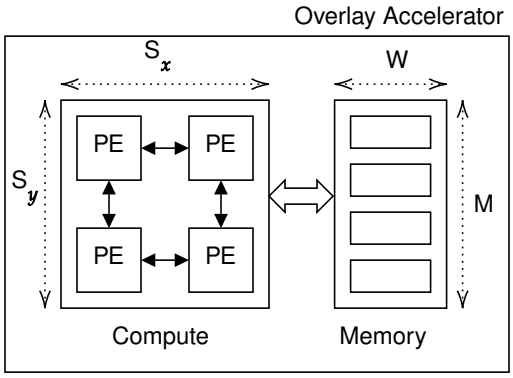


Figure 2.5: Principle 2: A scalable data path using a parametric design.

interconnections between stages. Additionally, if a computation involves more than one functional unit, this approach results in increased latency since the data path can process only one functional unit at a time. In the context of the overlays proposed, we have opted for the second approach of a shared/common data path. This choice is driven by its ability to yield a more compact design.

2.4.2 Principle 2: Parametric Design

A parametric data path design refers creating components that can be easily customized or adjusted by varying parameters during synthesis. It allows for easy adjustments to meet different requirements. Instead of designing entirely new hardware for each variation, parametric design enables you to reuse the same basic design and adjust it as needed. With parametric hardware, you can make the most of available resources. Components that may not be needed in one configuration can be used in another, reducing waste and improving resource utilization. Most importantly, a parametric design can be applied to scale the data path up or down as needed.

Refer to Figure 2.5. The data path is composed of two main sections: a compute section and a memory section. The compute section is defined by the parameters S_x and S_y, which determine the size of the processing element (PE) matrix. On the other hand, the memory section is characterized by the word size, denoted as W, and the number of words, represented as M. If you have an FPGA with specific computational and memory capacities, indicated as F_c and F_m respectively, the function $P(F_c, F_m)$ is used to configure the scaling parameters to appropriately fit the data path within the constraints of the given FPGA.

2.4.3 Principle 3: Maximize Data Reuse

Data reuse in accelerators refers to the practice of efficiently utilizing on-chip memory and compute resources to minimize the need for off-chip memory access, which can be a significant bottleneck in terms of both latency and power consumption. By minimizing data transfers between on-chip and off-chip memory, data reuse can greatly enhance the overall performance of the accelerator.

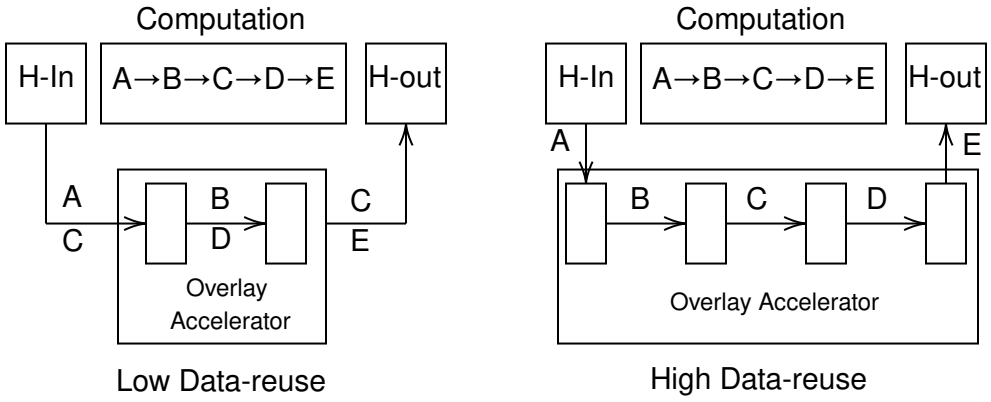


Figure 2.6: Principle 3: maximizing data reuse.

Refer to Figure 2.6. The computation consists of five distinct steps: A, B, C, D, and E. The process begins at step A, and it concludes at step E, where the output is generated. There are two different data path designs in consideration. In the first data path design, both compute stages are responsible for executing all five computation steps in a cyclical manner. Initially, the data path handles input A, produces an intermediate value at step B, and generates the output C at the second compute stage. Subsequently, the value of C is transmitted to the host, which then streams it back to the first compute stage. During this second phase, the data path generates the final output, D, which is subsequently returned to the host. While this design minimizes resource utilization, it suffers from low data reusability. This limitation arises because the intermediate data isn't entirely processed within the data path; instead, it must be redirected back through the host CPU. Consequently, this design experiences high external memory traffic, a significant bottleneck, as the external memory bandwidth is significantly slower than the compute bandwidth. An alternative data path design employs five computing stages. This design prioritizes data reuse, ensuring that all intermediate data remains within the accelerator. The host is only required to input A once and retrieve the output E from the output stage of the data path. However, it's important to note that this design does consume more hardware space. We have specifically engineered our overlay accelerator to maximize data reuse while adhering to our resource budget.

2.4.4 Principle 4: Distributed Control Memory

The dynamic configuration of the data path is managed through control words stored in the control memory. There are two primary design approaches for this control memory: linear and hierarchical, as depicted in Figure 2.7. In the linear configuration, control words move sequentially through the processing stages. To explain further, control words enter the first Processing Element (PE) and are shifted across subsequent PEs until they reach the intended PE. This design resembles a typical bus interconnect design and offers the advantage of minimal resource overhead. However, it comes with the drawback of higher configuration latency. For instance, if there are N stages in the data path and

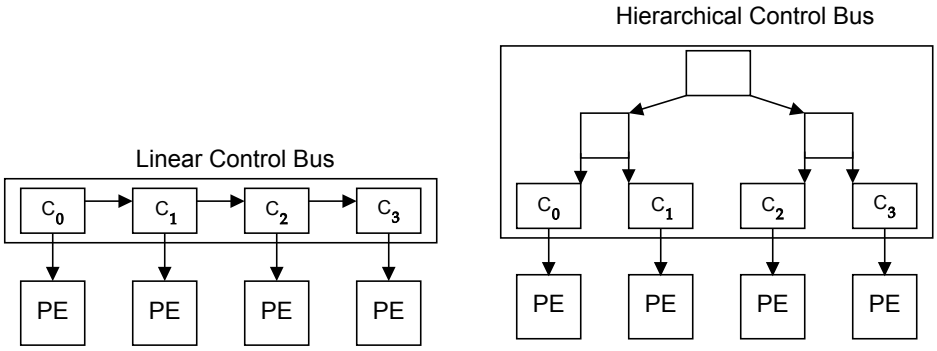


Figure 2.7: Principle 4: hierarchical control memory design.

we need to configure the last stage, it would require at least N-1 configuration cycles before the correct stage is initialized.

The second approach adopts a hierarchical design. In this setup, the control memory is spread out across the compute stages, with each stage possessing dedicated registers for storing control words required during its operation. This configuration guarantees that there are no communication bottlenecks when modules access their respective control words. These control words include an embedded index that serves as a guide for routing values through the hierarchical tree structure of the control bus. While this design comes with a higher hardware cost, it offers improved configuration latency. For example, if there are N stages in the data path and the goal is to configure the last stage, it would take approximately log(N) configuration cycles before the correct stage is initialized. In the overlay proposed in this work, we use the hierarchical control memory design to optimize the configuration latency.

2.5 Chapter Conclusion

In this chapter, we have delved into essential concepts related to overlay accelerators. The design principles we've elucidated are pivotal when crafting overlays, serving as guiding principles during the design process. These principles are interwoven to achieve a balanced approach in overlay creation, even though some of them may appear contradictory at first. Take, for instance, the trade-off between maximizing data reuse, which typically entails higher hardware costs, and the common data path design principle, which seeks to reduce hardware costs. Our goal is to harmonize these principles, striking a balance that ultimately leads to an optimal overlay design.

Chapter 3

Related Work on Accelerator Design

3.1 Classification of Accelerator Designs

In this section, we provide an insight into the accelerator ecosystem with respect to its applicability, design methodology, and performance. The applicability to an end user is investigated based on the host interface and portability. The design methodology is examined based on the hardware architecture. Finally, the performance is analyzed based on the throughput achieved. We categorize the accelerators as fixed-function hardware or overlays. Fixed-function accelerators refer to hardware generated by HDL languages like Verilog. Overlays, on the other, resemble generic processor-like designs. We discuss these two categories in the upcoming sections.

3.1.1 Fixed-Function Hardware

Hardware generated by traditional HLS flows is called fixed-function hardware. Here, an algorithm is accelerated on the FPGA by developing a source program in languages like Verilog or Bluespec. The source code is then taken through tools like Vivado or Quartus to generate the RTL and, finally, the hardware IP. Accelerators designed using this methodology are fixed to accelerate only the given algorithm and have little or no flexibility to tackle new algorithms. These accelerators have high throughput and a low FPGA resource footprint. From the usability perspective, these accelerators are hard to program, since the user has to know low-level hardware descriptive languages to accelerate the algorithms. Also, the host-to-accelerator interfaces are non-uniform across designs.

A recent trend in the design of fixed-function hardware is the use of template libraries. In this design flow, the user expresses the algorithm of interest using a set of predefined APIs or a DSL. A compiler then maps these top-level APIs to low-level hardware templates designed using HDL languages like Verilog. Finally, the templated code is processed and converted to a hardware IP using HLS tools. This design flow enables the user to use the same framework to accelerate various algorithms, only limited by the template library. Also, the host-to-accelerator interface remains uniform across various designs.

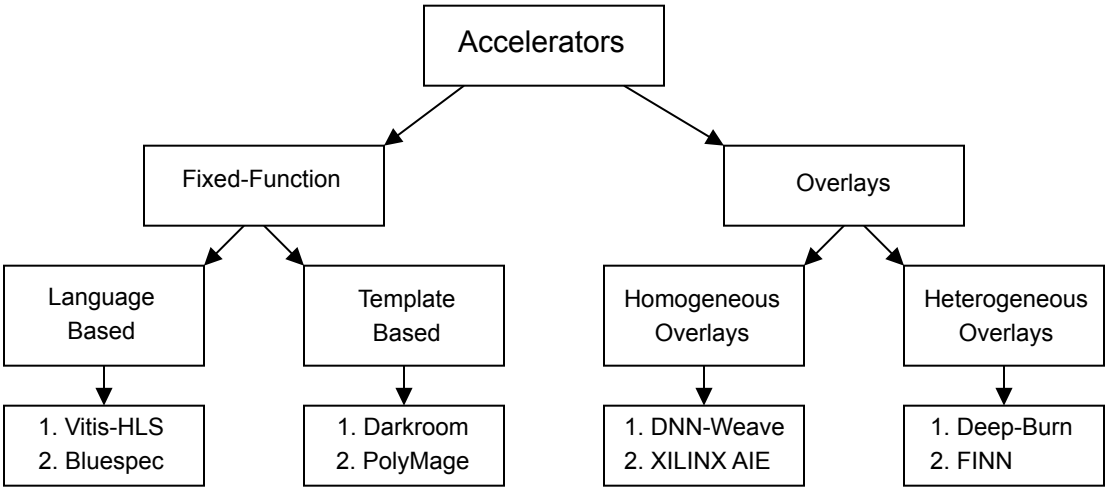


Figure 3.1: Broad Classification of the Accelerator Ecosystem.

3.1.2 Overlay Architectures

Overlays are hardware accelerators resembling co-processors that can accelerate a range of algorithms without needing FPGA resynthesis. In contrast to fixed-function static hardware, overlays are dynamic in terms of functionality. Overlays can be classified further as homogenous or heterogeneous.

A heterogeneous design consists of one distinct hardware block for macro-operation, where each block is optimized separately. All the heterogeneous blocks are chained to form a pipeline. The data proceed through the different parts of the pipeline as they are streamed through the architecture. As a result, this design approach exploits the parallelism between hardware blocks using pipelining and enables their concurrent execution. The hardware blocks are selectively included/excluded from the data path at runtime depending on the algorithm that is being accelerated.

The homogenous design approach favors flexibility over customization. Such an architecture comprises a single computation engine, for example, a systolic array of processing elements or a matrix multiplication unit. The software performs the control of the hardware and the scheduling of operations through instructions, resembling a CPU-type architecture. Consequently, each algorithm translates to a sequence of microinstructions that are executable by the hardware. Despite the flexibility gains, inefficiencies are introduced due to control mechanisms that resemble those of a processor

3.2 Image Processing Accelerators

High-level Synthesis (HLS) through C based languages [12, 72, 31], OpenCL/CUDA-based ones [73], high-level language frameworks [6, 34], and model-based frameworks such as NI LabVIEW and MATLAB HDL coder [22] are various tools that can be used to design FPGA hardware. A comprehensive survey of all these tool flows can be found in [8]. HLS suites, for example, Xilinx Vivado, Cadence C-to-silicon compiler etc., are significantly advanced to allow the programmer to use a C/C++ based

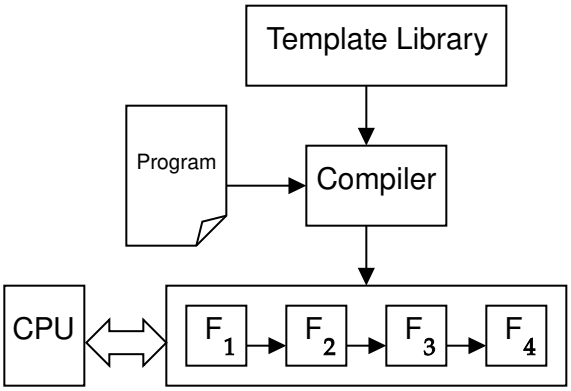


Figure 3.2: Fixed-function design approach of image processing accelerators.

interface for programming FPGAs compared to low-level hardware description languages like VHDL or Verilog.

Several recent efforts have designed fixed-function hardware through domain-specific languages and the corresponding compilation frameworks for image processing pipelines targeting FPGAs, see Figure 3.2. Darkroom [34], Rigel [35], Halide [75], Hipacc [37] and PolyMage [16] are some examples of DSL-to-FPGA compiler infrastructures.

Halide [77] and PolyMage [17], see Figure 3.2. DSL compilers translate a source program to HLS C; Hipacc generates either OpenCL or HLS C depending on the target FPGA; Darkroom and Rigel generate Verilog code directly. While generating HLS C or OpenCL code is relatively easy compared with Verilog, controlling the architecture of the target design and resource allocation on FPGAs is challenging. It requires directing the HLS compilers by including appropriate pragmas in the generated code, which is often a cumbersome job. On the other hand, using Verilog gives us fine-grained control of the architecture and resource allocation but makes code generation and design space exploration complex.

In the realm of high-performance hardware design, High-Level Synthesis (HLS) has emerged as a powerful tool, akin to using C for software development. However, the inherent limitations of HLS necessitate innovative solutions. This challenge is met by the introduction of domain-level abstractions through Domain-Specific Languages (DSLs) with micro-architectural templates. These DSLs generate hardware blocks tailored to specific domains, facilitating a more efficient synthesis process. This paragraph delves into various DSLs, exploring their capabilities and applications in high-performance hardware synthesis, with a particular focus on image processing. Additionally, it discusses efforts to enhance pre-existing DSLs for FPGA architectures, providing insights into the ongoing research landscape. Furthermore, the section introduces the concept of FPGA-based softcore processors and the unique approach of overlay processors, which forego traditional instruction-based architectures in favor of direct hardware control through control words. The subsequent exploration aims to bridge gaps in research, particularly in the context of image processing applications.

Creating high-performance hardware through High-Level Synthesis (HLS) [62, 19] is analogous to employing C for the development of high-performance software [75]. To overcome this limitation, domain-level abstractions are introduced atop the high-level language, manifesting as Domain-Specific Languages (DSLs) with micro-architectural templates. These templates generate hardware blocks tailored to the specific domain [1, 66, 31, 47]. Aetherling [21] is a DSL that compiles high-level image processing algorithms into hardware, focusing on exploring resource-vs-throughput trade-offs. DSAGEN[98] annotates algorithms using pragmas, conducting an automated search within a vast architecture design space for diverse algorithms. FLOWER[5] stands out as a comprehensive compiler infrastructure, offering automatic canonical transformations for high-level synthesis from a domain-specific library. This enables programmers to concentrate on algorithm implementations, rather than low-level optimizations for dataflow architectures. AnyHLS[71] proposes a modular and abstract approach to synthesize FPGA designs, elevating the abstraction level through programming language features such as types and higher-order functions. Image processing DSLs based on the line-buffered pipeline model have been developed, with Darkroom [34] transforming stencil and pointwise computations into custom hardware using a synchronous data flow model [51]. Rigel[35], an extension of Darkroom, supports multi-rate pipelines, allowing specification of pipeline stages running at throughputs other than one pixel/cycle. FlowPix employs a DAG formulation and data relaying techniques for intuitive line buffer size optimization.

In contrast to designing new DSLs, efforts have been made to enhance existing DSLs for CPU and GPU architectures with FPGA back-ends. The `Halide` HLS framework [75] extends the Halide compiler [77], enabling FPGA acceleration. The Halide compiler is a domain-specific language (DSL) and toolset developed by researchers at MIT and Stanford University for efficient creation of high-performance image and signal processing pipelines. It provides a functional programming language tailored for image processing, allowing developers to express algorithms independently of low-level details. Halide separates algorithm description from scheduling, promoting experimentation with optimizations. It achieves performance portability across different hardware platforms by automatically parallelizing and vectorizing code. With a focus on CPUs, GPUs, and accelerators, it enables target-independent algorithm development. Being open source and integrable with existing code, Halide facilitates concise and expressive descriptions of image processing algorithms while generating optimized, high-performance code. PolyMageHLS [17] extends the PolyMage [65] DSL for FPGA targeting, leveraging coarse-grain parallelism to process image tiles in parallel. O. Reiche et al. [79], [108] explore image processing using a source-to-source compiler called Hipacc, capable of producing highly efficient hardware accelerators. FPGA-based softcore processors, like GraphSoC for graph algorithms [43] and IPPro for image processing [83], address domain-specific computations, aiming to balance programmability and performance. Overlay processors, proposed in this work, break away from conventional 5-stage RISC designs, utilizing control words for direct hardware control, eliminating the need for traditional instructions. This approach, particularly relevant to image processing, aligns with the overarching goal of addressing the gap in research on overlay processors in this domain.

3.3 Convolutional Neural Network Accelerators

Many works have proposed various FPGA-based CNN accelerators which outperform CPUs and GPUs on the performance-per-watt metric. An extensive survey of the same can be found in [30]. Most of the existing accelerators limit their focus on the automated mapping of CNN inference, with `FINN` focusing on the more specific field of Binarised Neural Networks (BNNs) [40]. The most common types of layers in a CNN are the convolutional (CONV), nonlinear (NONLIN), pooling (POOL) and fully-connected (FC) layers [50]. All existing accelerators support these layers, with ALAMO, DeepBurning, DnnWeaver, and AutoCodeGen also supporting Local Response Normalisation (NORM) layers [48].

Heterogeneous CNN overlays have a streaming architecture and typically consist of one distinct hardware block for each layer of the target CNN. fpgaConvNet [90] employs a streaming architecture that assigns one processing stage per layer. It also employs specialized hardware blocks to map networks with irregular data flow. In a similar approach to fpgaConvNet, DeepBurning's [94] core consists of a library of building blocks that follow the functionality of standard neural network components. Haddoc2 [3] generates its architecture by modeling the target CNN as a data flow graph of actors and directly mapping each actor to a dedicated compute unit. AutoCodeGen [55] includes parametrized hardware blocks at the layer level, supporting CONV, POOL, NORM, and FC layers.

Homogenous CNN accelerators have also been reported, where the control of the hardware and the scheduling of operations is performed by software. The design principle behind the Angel-Eye [28] framework is based on having a single flexible computation engine that can be programmed and controlled by software. Unlike Angel-Eye, ALAMO [61] customizes the generated computation engine to the input CNN. DnnWeaver's [81] hardware is based on a parametrized architectural template. The template comprises an array of coarse Processing Units (PUs). Caffeine's [103] hardware consists of a systolic array of PEs that perform multiplication operations. Snowflake's [26] hardware design employs a hierarchical structure controlled by software. At the top level, the architecture comprises several hardware compute clusters, organized as an array of tunable sizes. The chart in Figure 3.3 compares the CNN accelerators across several aspects. In the same Figure we also show how our CNN accelerator fares with the others across the different aspects of usability and performance.

CNNs demand substantial computational power, prompting the exploration of FPGA technology for acceleration. CNN overlay accelerators on FPGAs employ a customizable architecture to enhance specific operations like convolution and pooling. Key components include dedicated units for convolution and pooling, optimized memory interfaces, configurability for diverse CNN models, and efficient host system communication. The benefits encompass improved performance through parallel processing, energy efficiency, low latency, and adaptability to evolving neural network architectures. Overall, CNN overlay accelerators on FPGAs offer an efficient and flexible solution, particularly valuable for real-time processing applications.

The hardware generated by the existing CNN-to-FPGA frameworks can be categorized as either a streaming architecture or a single computation engine-based architecture [91]. A streaming architecture typically consists of dedicated hardware modules for each layer of the CNN, connected in a pipeline.

All the layers are processed simultaneously by streaming data across the pipeline. FpgaConvNet [90] is designed on this principle. It supports multi-bit-stream design via complete FPGA reconfiguration, where different hardware architectures, matching the layer workloads, are used to process different layers of the CNN. DeepBurning [94] used a library-based approach. Based on the layer functionality, hardware building blocks are instantiated from a repository and interconnected to form the network. Each block is configured using fixed tiling parameters, calculated from a heuristic search, and is time-shared across the network layers. Haddoc [4] generates its architecture by modeling the target CNN as a data flow graph of actors and directly mapping each actor to a dedicated compute unit. AutoCodeGen [56] includes parameterized hardware blocks for each CNN layer, instantiated using a high-level analytical performance and resource model, with the convolution blocks executing in a fully unrolled manner. The streaming design principles favor customization over flexibility, where a single accelerator gets tightly coupled with a specific CNN. Also, it becomes hard to map all the CNN layers to resource-constrained FPGAs, which is when the CNN is processed in a time-multiplexed fashion over a generic accelerator architecture leading us to the idea of single-engine architectures. Single engine architectures comprise a single computation engine that executes the CNN layers sequentially. The accelerator processes each layer at its maximum throughput. This design is a derivative of the well-studied systolic array structures [63]. AngelEye [76] comprises an array of homogeneous processing elements (PEs). Each contains a bank of convolvers, a summing tree, and a pooling logic that are instantiated using a throughput maximization heuristic, which uses a set of loop unroll factors. DnnWeaver [81] contains parametric hardware templates arranged in a similar array of PEs. The configuration of each PE is found through a search-based heuristic, which is later used to synthesize the accelerator. Once synthesized, this configuration remains the same throughout the processing of the network. Caffeine [103] consists of a systolic array of PEs that perform multiplication operations configured using a roofline model on the hardware design space. Snowflake [26] employs a hierarchical hardware structure that is designed to be controlled by software, with complex control logic and is CNN agnostic depending only on the available FPGA resources. Despite the flexibility gains in this design principle, inefficiencies are introduced due to control mechanisms that resemble those of an instruction-based processor [32]. Moreover, the uniform unroll factors applied to the entire processing array can reduce the performance of CNNs with varying workload characteristics.

In another classification, the state-of-the-art CNN accelerator designs can be divided into two categories: uniform/homogeneous and heterogeneous. The homogeneous design methodology uses a uniform architecture to process all layers of a CNN model in order. For a single layer's inference, a homogeneous design firstly divides an input feature map into tiles. Then it repeatedly loads the tiles one after another from off-chip memory to on-chip memory and then processes the tiles in sequence by a jointly optimized accelerator design for all convolutional layers. Such designs have been described in [7, 60, 23, 105, 25]. However, different layers in a CNN model have different input data shapes. As a result, the same tiling factors in a homogeneous design may cause dynamic resource inefficiency for some layers. To solve this problem, a heterogeneous design methodology is proposed in several previ-

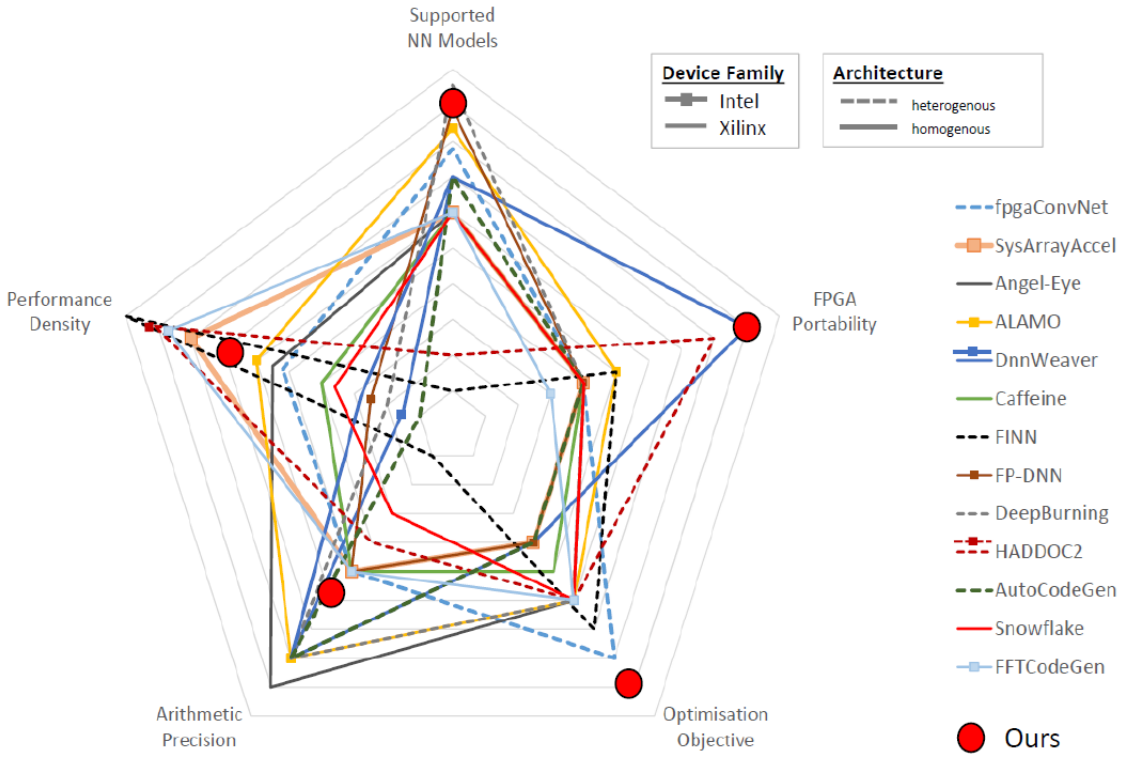


Figure 3.3: Classification of the CNN accelerators considered in this work and how our design fares comparatively. Figure source [91]

ous works [52, 82, 88, 104, 82]. A heterogeneous design incorporates multiple accelerators on a single FPGA. Each of them is optimized specifically for one or a set of layers. These architectures concurrently process multiple input images, by pipelining them on the different accelerators. The work [97], supported pipelined execution of different tiles from a single input image on multiple heterogeneous accelerators. As the range of supported networks grows so does the complexity of managing the different accelerators on the FPGA. Overlay architectures[85, 13, 58, 2], combine ideas from both heterogeneous and homogeneous designs. They operate as a single uniform architecture with the flexibility to adjust to different tile sizes during runtime. The work in [101] proposed an FPGA overlay with software-like programmability for CNN end users. The overlays operate via an ISA with complicated functions with variable runtimes but a uniform length. The granularity of instruction is optimized to provide good performance and sufficient flexibility. Overlay designs tend to consume more FPGA logic resources and often fail to exploit all forms of parallelism with a CNN layer. In this work, the proposed overlay architecture operates with fewer FPGA LUTs, which can be attributed to a simple processing engine structure and exploits all forms of parallelism within a CNN layer.

PART II

Proposed Overlay Designs

Chapter 4

FlowPix: An Overlay for Image Processing Pipelines

This chapter introduces FlowPix, an innovative overlay architecture illustrated in Figure 4.1, specifically engineered to enhance the efficiency of image processing pipelines. An image processing pipeline is conceptualized as a directed acyclic graph (DAG) comprising various computational stages. These stages involve elementary data-parallel operations applied to all image pixels, including point-wise and stencil computations. The complexity of the compute DAG varies across image processing algorithms, encompassing anything from straightforward blurring tasks to intricate pyramid systems. Moreover, the operations within the DAG exhibit significant diversity. To overcome the intricacies associated with accelerating image processing pipelines, the FlowPix overlay architecture confronts two pivotal challenges:

- **Challenge 1: Designing a Simple Front-End DSL -** The first challenge revolves around creating an intuitive front-end Domain Specific Language (DSL) that empowers users to articulate the specifications of their image processing pipeline seamlessly. This involves the development of a user-friendly interface that abstracts the underlying complexity of the computational stages, ensuring accessibility for a broad spectrum of users with varying levels of expertise.

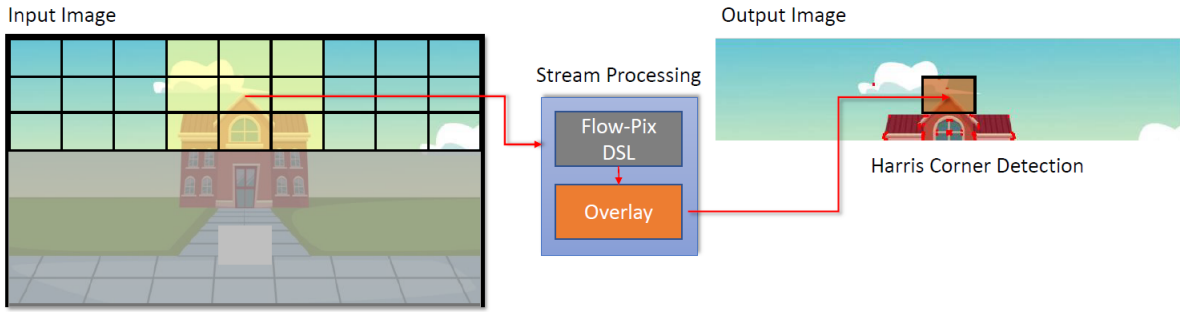


Figure 4.1: The input image is streamed through the FlowPix framework. At any point the overlay processes an input window of pixel to generate an output pixel.

- **Challenge 2: Flexibility in Mapping DAG Structures -** The second challenge pertains to endowing the overlay architecture with the flexibility to map diverse DAG structures and their corresponding operations. Given the variability in size, shape, and complexity of DAGs across different image processing algorithms, the overlay must possess adaptability and scalability. It should be capable of efficiently accommodating both simple and intricate DAGs, ensuring optimal performance regardless of the specific image processing task at hand.

In addition to tackling these two fundamental challenges, this chapter aims to align with the overarching design principles outlined in Chapter 2, further contributing to the evolution and refinement of overlay architectures for image processing acceleration.

4.1 Image Processing Pipelines

In this segment, we delve into the fundamental aspects of image processing pipelines and explore the execution of computations within a dynamic stream processing paradigm. The essence of an image processing pipeline lies in its ability to metamorphose an input image into an output image through the intricate orchestration of stages forming a directed acyclic graph (DAG). The links between stages in the DAG manifest as edges, embodying a vital producer-consumer relationship between the source and destination stages. Every node in the DAG signifies either a stencil or a point-wise computation. A stencil stage, characterized by a singular predecessor, executes a stencil operation on the incoming intermediate image streamed from the preceding stage along the connecting edge. While commonly associated with convolution, a stencil operation transcends this boundary and can encompass diverse filter operations, exemplified by the likes of a 'max' filter, employed for the downsampling of images. Contrastingly, a point-wise stage is more versatile, permitting multiple predecessor stages. Here, each output pixel is meticulously calculated by assimilating point-pixels from the incoming intermediate images streamed through the interconnected edges. The execution of stages within the DAG can unfold concurrently, contingent upon dependency constraints and data availability. This synergy epitomizes a captivating streaming data flow model of computation, a model impeccably suited for realization within the realm of FPGAs.

In an example of an image processing pipeline, we initiate the process with an input stage where the raw image is introduced. Subsequently, a stencil operation is applied in the first stage, utilizing convolution to detect horizontal edges and generating an intermediate image. The following point-wise computation stage calculates the gradient of the intermediate image, with each pixel in the output reflecting the gradient magnitude. Moving forward, the second stencil stage employs a 'max' filter to downsample the image, showcasing the versatility of stencil operations beyond traditional convolution. The ensuing point-wise stage executes non-maximum suppression on the downsampled image, emphasizing the preservation of significant edges. Ultimately, the output stage produces the final image after the complete transformation, illustrating the dynamic stream processing paradigm and the concurrent

Operation	Example
Point-wise	f(x, y) = g(x, y) + h(x, y)
Stencil	f(x, y) = $\sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(x+\sigma_x, y+\sigma_y)$
Upsample	f(x, y) = $\sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(\frac{x+\sigma_x}{2}, \frac{y+\sigma_y}{2})$
Downsample	f(x, y) = $\sum_{\sigma_x=-1}^{+1} \sum_{\sigma_y=-1}^{+1} g(2x+\sigma_x, 2y+\sigma_y)$

Table 4.1: Computational patterns explored in FlowPix overlay.

execution of stages based on dependency constraints and data availability within the directed acyclic graph framework.

4.2 Overview of FlowPix Framework

FlowPix comprises three essential parts: a DSL front-end, a compiler, and a backend hardware overlay. The FlowPix overlay is structured with Compute Units (CUs), each having an array of Processing Engines (PE) linked in a pipeline through FIFO interfaces. Both parallelism with a single PE and across PEs in the form of pipelined parallelism is exploited within a CU, as shown in Figure 4.3. The compute Directed Acyclic Graph (DAG) defined by the front-end is divided into clusters before processing, with each cluster scheduled for execution on a single CU. PEs are assigned to topologically sorted DAG clusters, enabling the execution of streaming dataflow computations within and between PEs within the same CU. If there aren't enough available PEs, time multiplexing occurs through PE reconfiguration. Intermediate data is internally buffered, eliminating the need for off-chip DDR memory accesses. Subsequent clusters are soft-reconfigured to process this intermediate data, producing the final processed image. Details on this process are discussed in Section 4.

The FlowPix DSL is integrated into Scala, allowing FlowPix DSL code to run as a Scala program on the CPU. This is useful for developers during development, enabling them to use CPU execution mode for debugging without expensive hardware execution costs. The FlowPix compiler builds a DAG from the DSL program, mapping it to processing engines in the overlay and creating a sequence of control words. The overlay processes DAG computations based on these control words, with a C++ driver streaming control words and image frames to the overlay. The overlay persists in performing computations until the driver initiates a reconfiguration using an alternative control word sequence.Figure 4.2 gives an overview of the compilation and execution flow of a FlowPix DSL program.

Our implementation leverages the robust capabilities of Bluespec System Verilog (BSV) for the overlay, where we emphasize a parametric design approach. BSV's distinctive features include parametric modules and interfaces, allowing for the creation of a single, concise source code that seamlessly adapts to various architectures. During hardware synthesis, configurability becomes a key strength, as both the number of CUs and the PE array size within each CU can be dynamically adjusted based on FPGA

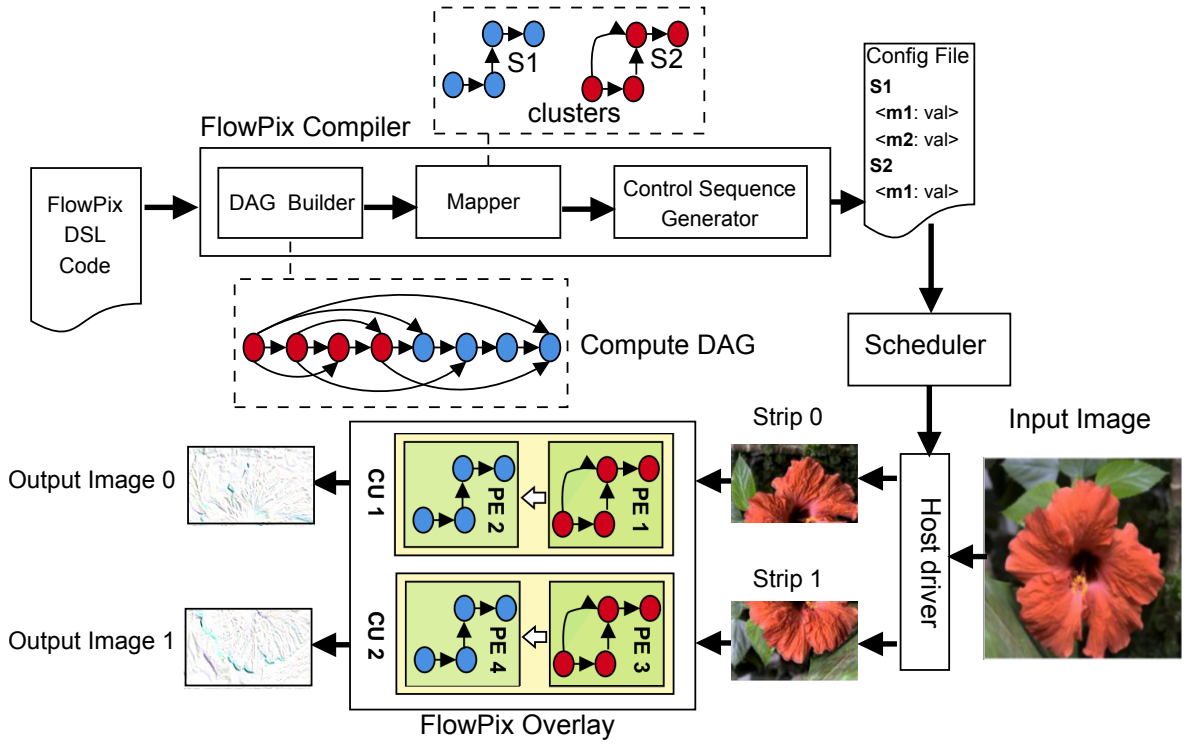


Figure 4.2: The diagram depicted above illustrates the comprehensive process of program compilation and execution within FlowPix. DSL code is inputted into the compiler, generating a series of control words. The directed acyclic graph (DAG) is partitioned into distinct clusters, each subsequently mapped to a compute unit (CU). Utilizing the control words, the driver configures the overlay and initiates the streaming of the input image. This image is then divided into smaller strips, allowing for simultaneous processing across the available CUs. Importantly, the input image is streamed only once, and the overlay internally buffers the partial data produced from cluster executions.

specifications and the characteristics of the workload. The versatility of BSV enables the replication of computations across multiple CUs, a strategy employed to efficiently handle large-scale image processing tasks. This involves vertically dividing the image into strips, with each CU dedicated to processing a specific strip. The size of each strip is determined by the overlay's internal memory, facilitating data parallelism as multiple CUs simultaneously process distinct strips. Control words, integral to the coordination of parallel computations, are placed in dedicated registers distributed across the PEs. A sophisticated distribution logic ensures incoming control words are efficiently directed to their respective locations within the memory architecture. This structured approach optimizes the overall processing efficiency and coordination within the overlay. Once the image processing is completed, the resulting image is streamed back to the driver and seamlessly integrated into the application logic. The parallel nature of CUs allows for the concurrent handling of diverse workloads, enhancing the overall throughput and performance.

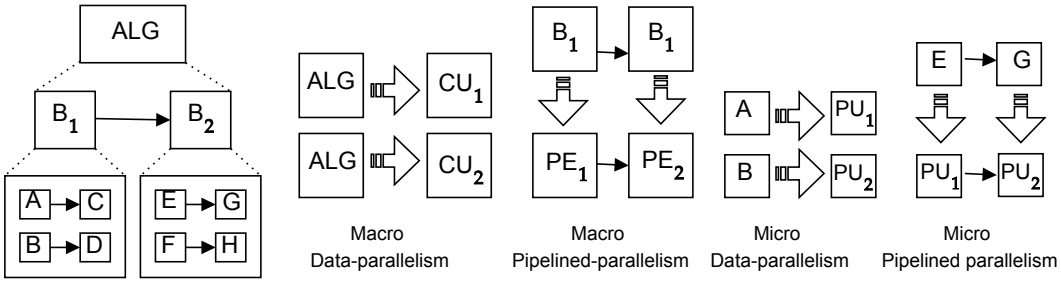


Figure 4.3: The diagram elucidates the diverse parallelism types harnessed during the execution of Algorithm A, an image processing algorithm, on the FlowPix overlay. Algorithm A is segmented into logical compute blocks, each encompassing multiple computing steps. The architecture of Algorithm A is intricately aligned with the compute hierarchy of the overlay, featuring CUs composed of Processing PEs, and PEs comprising PUs. At a macro level, CUs and PEs leverage both data and pipelined parallelism, while the PU network within a PE exploits these parallel forms at a more granular micro level.

4.3 FlowPix Language

This section provides a brief background on the image processing pipelines and then shows how they can be expressed using our FlowPix DSL.

4.3.1 Image Processing Pipelines

An image processing pipeline transforms an input image into an output image by passing it through a sequence of stages connected in DAG form. Each node in the DAG denotes a stage in the pipeline that performs computations on input image pixels and generates output frame pixels. Each node in the DAG can be either a stencil or a point-wise stage computation. A stencil stage has only one predecessor stage. It applies a stencil operation on the intermediate input image streamed by the predecessor stage along the edge. Usually, a stencil operation refers to convolution on a small pixel window. However, generally, it can be any filter operation, such as `max` filter, used for downsampling images. A point-wise stage can have multiple predecessor stages. Each output pixel is computed using the corresponding input pixels from the intermediate input images streamed along incoming edges. The stages in the DAG can be executed in parallel, subject to dependency constraints and data availability. This leads to a *streaming data flow model* of computation suitable for realization in FPGAs.

Pointwise stages do not require buffering because the input required, a single pixel is exactly what the prior stage produces. In contrast, stencil operations need multiple input pixels from the previous stage. For example, with a 3x3 Gaussian operator, the blurring stage in the unsharp mask pipeline requires 3 pixels in vertical and horizontal directions of the input image. The input pixels arrive in the scan-line order. Therefore, the stage has to buffer at least two rows of the input frame before performing the computation. This is referred to as line buffering as it buffers lines of the input image. The line

$Program$	$::=$	$C\ Stage \mid Stage$
$Stage$	$::=$	$Stage\ **\ Filter \mid Stage\ OP\ Stage \mid Stage\ ?\ Stage\ :\ Stage \mid \epsilon$
$Filter$	$::=$	$max \mid min \mid up \mid down$
OP	$::=$	$+ \mid - \mid *$
C	$::=$	$Scala\ Construct$

Table 4.2: Representative grammar for the FlowPix DSL.

buffers' presence between the stages ensures that the intermediate values are fed from the producer to a consumer without any external memory intervention, which is a considerable energy saving.

4.3.2 FlowPix Front-end

Our DSL is integrated into the Scala language, allowing DSL programmers to leverage all Scala features while utilizing embedded DSL constructs for defining image processing pipelines. In these computational pipelines, input and intermediate images are represented as Scala objects of the class type `Stage`. Operator overloading is used to implement Point-wise computations on images. For stencil stages involving filter operations on images, programmers can instantiate the `Filter` class with appropriate parameters. The DSL currently supports essential filter operations such as `convolution`, `max`, `min`, `sum`, and `average`. Expanding the repertoire of filter operations in our DSL compiler infrastructure is easily achievable. Note that the listings in this section are only part of the code expressing the computation. A boilerplate code defines the stage class and the overloaded operators.

Our approach simplifies the programming process by relieving the programmer of the need to specify image dimensions at each computation step. Instead, our compiler autonomously deduces these dimensions through DAG (Directed Acyclic Graph) analysis.

In this section, we use the Harris Corner Detection (HCD) to explain the DSL. HCD algorithm is a widely used method in computer vision for identifying key points or corners in an image. The algorithm operates based on the principle that corners exhibit significant variations in intensity when an image is shifted in any direction. The Harris Corner Detection algorithm is known for its simplicity and effectiveness in detecting corners in images. It is a foundational component in many computer vision applications, such as feature matching, object recognition, and image stitching. Adjustments to parameters, such as the threshold and filter sizes, allow for customization based on specific image characteristics and application requirements. Figure 4.4 depicts the computational DAG associated with the Harris corner benchmark, with stencil and point-wise stages represented by diamond and circle symbols, respectively.

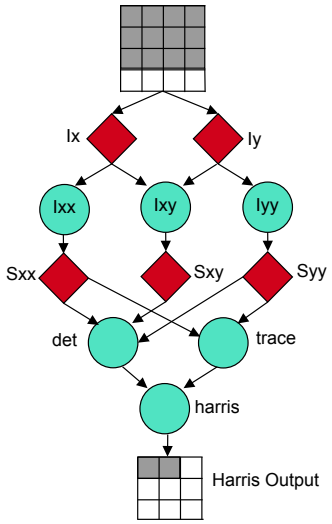
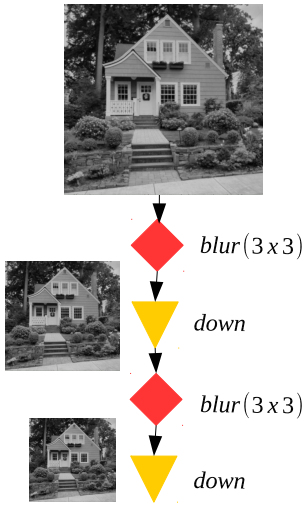


Figure 4.4: The diagram illustrates the computational DAG associated with the Harris Corner (HCD) benchmark. Stencil and point-wise stages are denoted by diamond and circle symbols, respectively.



Pyramid DAG

Figure 4.5: The diagram show the DAG and the associated computations for the 2-level Gaussian pyramid benchmark. This benchmark performs a downsampling operation using the `max` filter. The downsampling stage is represented using an inverted pyramid symbol in the DAG.

Listing 4.1 essentially implements the Harris Corner Detection algorithm in FlowPix DSL, identifying corners in the input image based on local intensity changes in multiple directions. The resulting Harris response image is then stored for further analysis or visualization. Adjustments to filter parameters and the threshold can be made to fine-tune the corner detection based on specific requirements. The code can be decomposed as follows.

- **Filter Definitions (Lines 1-3)**: Three filters (vec, vec2, vec3) are defined for computing image gradients and applying a Gaussian filter. The first two parameters in a filter instantiation denote the filter dimensions and filter strides, respectively. The filter dimension is a tuple specifying the number of rows and columns, while the filter stride is a tuple representing the row and column strides as the filter moves over the image. The last parameter designates the weights for the convolution operation. These filters are essential for extracting the directional changes in intensity across the image.
- **Input Image Stage (Line 4)**: An input image stage (img) is instantiated by providing a valid image path to the `Stage` class. This image will undergo Harris Corner Detection to identify points of interest or corners. Other stage objects in the program— like `Ix` (line 5), `Ixx` (line 7), `Sxx` (line 10), and `harris` (line 17)—correspond to different stages in the DAG.
- **Compute Image Gradients (Lines 5-6)**: Image gradients in the x and y directions (Ix and Iy) are computed using convolution with the defined filters. These gradients represent the rate of intensity change in the horizontal and vertical directions. Compute image gradients are calculated using convolution (** operator).
- **Compute Products of Image Gradients (Lines 7-9)**: The products of image gradients (Ixx, Iyy, Ixy) are computed. These products are crucial for calculating the structure tensor components, which are necessary for Harris Corner Detection.
- **Apply Gaussian Filter to Products (Lines 10-12)**: A Gaussian filter (vec3) is applied to smooth the products of image gradients. This step helps in reducing noise and refining the structure tensor components for more accurate corner detection.
- **Compute Determinant and Trace (Lines 13-16)**: The determinant (det) and trace (trace) of the structure tensor matrix are computed. These values are used to calculate the Harris response at each pixel, indicating the likelihood of a corner.
- **Harris Response Calculation (Line 17)**: The Harris response is calculated based on a threshold condition. If the response at a pixel exceeds a certain threshold, it is considered a corner and assigned a white colour (255); otherwise, it is assigned a black colour (0).

Point-wise operations on lines 7 through 9 and lines 13 through 17 are expressed as operations on image objects. The output Stage object, `harris`, is stored as an image by providing an image path to the `store_image` method offered by the `Stage` class.

```
1  var vec = Filter((3, 3), (1, 1), filter_data = ((-1, 0, 1), (-2, 0, 2), (-1, 0, 1)))
2  var vec2 = Filter((3, 3), (1, 1), filter_data = ((-1, -2, -1), (0, 0, 0), (1, 2, 1)))
3  var vec3 = Filter((3, 3), (1, 1), filter_data = ((1, 1, 1), (1, 1, 1), (1, 1, 1)))
4  var img = Stage("Images/chess.png")
5  var Ix = img ** vec
6  var Iy = img ** vec2
7  var Ixx = Ix * Ix
8  var Iyy = Iy * Iy
9  var Ixy = Ix * Iy
10 var Sxx = Ixx ** vec3
11 var Syy = Iyy ** vec3
12 var Sxy = Ixy ** vec3
13 var SxxSyy = Sxx * Syy
14 var SxySxy = Sxy * Sxy
15 var det = SxxSyy - SxySxy
16 var trace = Sxx + Syy
17 var harris = (det - trace * trace * 0.004 >= 300) ? (255, 0)
```

Listing 4.1: FlowPix code for HCD pipeline.

```
var img = Stage("Path/to/Image")
var down = Filter((2, 2), (2, 2), "max")
var kernel = Filter((3, 3), (1, 1),
                    ((1/32, 1/16, 1/32),
                    (1/16, 1/8, 1/16),
                    (1/32, 1/16, 1/32))
                    )
(i in 0 to 2){
 var blur = img ** kernel
 var reduce = blur ** down
 img = reduce
}
img.store_image("gaussian_pyramid")
```

Listing 4.2: Listing shows a two level Gaussian pyramid

Figure 4.5 illustrates the Directed Acyclic Graph (DAG) and the corresponding computations for the 2-level Gaussian pyramid benchmark. In this benchmark, a downsampling operation is performed using the `max` filter, and the downsampling stage is denoted by an inverted pyramid symbol in the DAG. The corresponding FlowPix code is presented in Listing 4.2. The image undergoes a stepwise process, commencing with a convolution utilizing a 3×3 filter `kernel` (Line 3), succeeded by passage through a 2×2 `max` filter `down` (Line 2). The `max` filter identifies the pixel with the highest intensity within the 2x2 pixel window, and the window moves with a row and column stride of 2, resulting in a 4x reduction

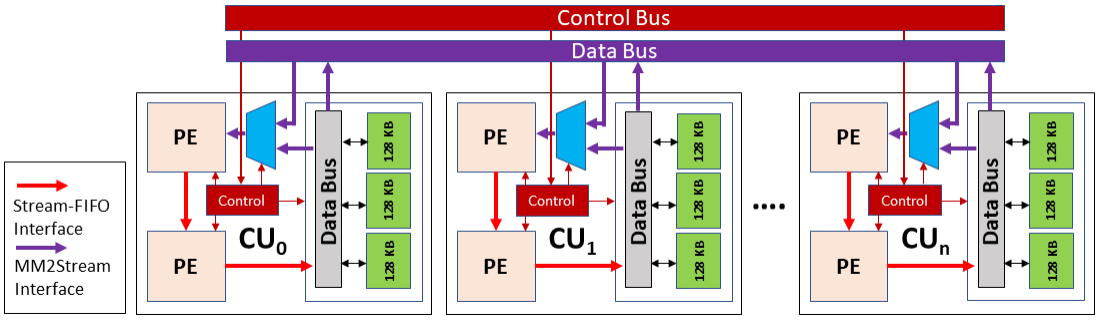


Figure 4.6: The FlowPix overlay is a collection of control units (CUs). The CUs operate independently and receive control and data input from a common control and data bus.

in image size. In Line 2, the first tuple designates the filter dimensions, while the second tuple specifies the row and column strides. This convolution and max-pooling sequence iterates to produce the final output image. The series of computations, involving convolution and down-sampling, is enclosed within a loop using Scala's loop construct. Within this loop, the output image from one iteration becomes the input for the next. The image progressively undergoes down-sampling, diminishing to $\frac{1}{16}^{th}$ of its original dimension after two loop iterations.

4.3.3 Semantic Checks

Semantic checks are critical to ensuring that the DSL program adheres to the rules, thereby ensuring program correctness. Here are the primary types of semantic checks done by the FlowPix compiler. As DSL is embedded in the Scala language, these checks are enabled by the Scala semantics.

- **Type Compatibility**: This check ensures that the operators in the DSL are performed on compatible objects. For example, a filter operator can only be applied to a single stage object. Another example could be that a comparison operator can not be applied between two stage objects. The errors would manifest as syntax errors during compilation.

- **Variable Declaration**: Ensures that a stage object is defined before it is used in the computation DAG.

- **Name Conflicts**: Multiple stages can not have the same name in the compute DAG.

- **Inference Rules**: Scala supports type inference. The Scala compiler deduces the types of expressions based on the types of their constituent parts and the operations applied to them. For example, the Scala compiler automatically infers each compute expression as a stage object.

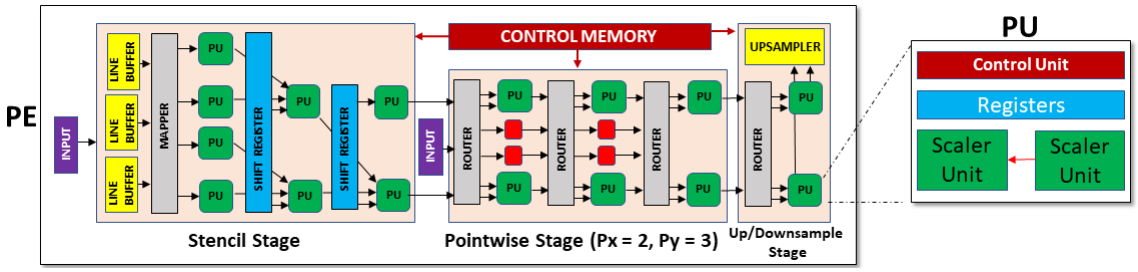


Figure 4.7: Architecture of a Processing Engine (PE) inside the FlowPix overlay. The PE control unit is part of the overall control memory of the overlay. The relaying PUs are shown in red inside the figure.

4.4 FlowPix Overlay Architecture

The structure of the overlay consists of a group of Compute Units (CUs), as shown in Figure 4.6. To transfer control and data words from the host CPU to the overlay, a shared control and data-bus is utilized. The host input is initially stored in FPGA DDR memory, then streamed to the overlay via an AXI DMA MM2S (Memory Mapped to Stream) channel. The CUs are made up of processing engines (PEs) that are linked in a pipeline, with data exchange occurring through a streaming FIFO interface. The number and size of CUs are pre-determined during synthesis based on FPGA resources. Local control registers that form part of the overlay's overall control memory dictate the runtime behaviour of a CU. A limited number of BRAM memory banks are utilized to buffer the output from a CU locally. The output is then transmitted back to the CPU through an AXI DMA S2MM (Stream to Memory Mapped) channel. Depending on the computation schedule, the stored output can also be looped back to the CU.

The AXI DMA (Advanced eXtensible Interface Direct Memory Access) MM2S (Memory Mapped to Stream) and S2MM (Stream to Memory Mapped) channels are components commonly used in FPGA designs to efficiently transfer data between different parts of a system. The MM2S channel is used to transfer host input data from FPGA DDR memory to the overlay. This data is then streamed to the Compute Units (CUs) for processing. The S2MM channel is used to transmit the processed output from the Compute Units back to the CPU. The data is buffered locally using Block RAM (BRAM) memory banks before being transmitted through the AXI DMA S2MM channel.

4.4.1 Processing Engine

The overlay's primary execution unit is the Processing Engine (PE), see Figure 4.7, which comprises processing units (PUs) that carry out computations. Each PU accepts three inputs and has two scalar units that can perform simple operations such as addition, subtraction, multiplication, and comparison. Both scalar units can be used in a cascaded manner when utilizing the three input values. The PU also contains registers that store control words and constants. Apart from the computing PUs, there are additional relaying PUs positioned between the computing PUs. These relaying PUs are responsible for

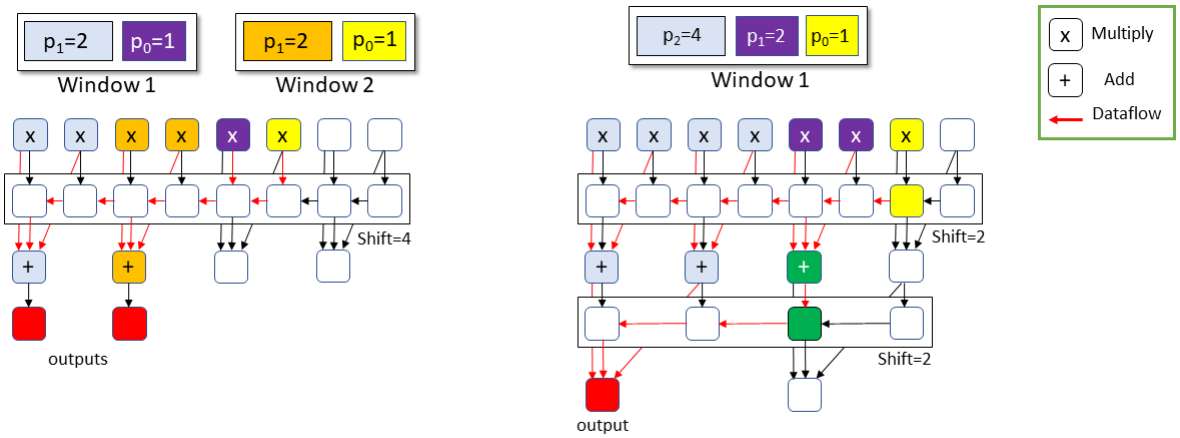


Figure 4.8: Figure shows two dynamic configurations of the PE. In part (a) the PE executes two 1x3 stencils. In part (b) the PE executes a single 1x7 stencil.

forwarding data without performing any processing. We will delve into the specific functions of these relay PUs at a later point. The PE has a parametric design and its shape and size is set during synthesis by two parameters: P_x and P_y. The PE architecture is divided into four pipelined stages, one input buffering stage, and three compute stages. The compute stages consist of pipelined sequences of PU arrays. The number of PU arrays and their interconnections varies across the stages.

Input to the PE is in the form of a vector. The output from the PE is streamed to the next PE inside the CU. The output from the final PE in the PE sequence is stored in the CU memory banks. The first compute stage performs stencil computations and is designed as a multiply-and-accumulator unit. The PUs in this stage are arranged in a tree structure with the base of the tree having $4 \times P_x$ PUs. The next stage computes pointwise operations and consists of P_y arrays with P_x PUs in each array. The final stage has a single array with P_x PUs and processes upsample or downsample operations. This stage can perform a maximum of P_x downsample operations or a single upsample operation. The input to the PE can be new data from the host or partial data stored in the CU memory banks from previous executions. Input buffering is done using an array of line buffers, which are storage structures built using the FPGA BRAM that buffer a minimum number of rows from the input image to produce square windows. The line buffer design used in FlowPix is from [15] and can be dynamically programmed to generate any window size moving with a stride (the default stride is set at 1) over the input. The ordering of the compute stages inside a PE aligns with the compute pattern of the benchmarks analyzed. To illustrate, the majority of these benchmarks apply stencils to an input image, then perform pointwise operations to combine the stencil outputs. This is followed by an up-sampling or down-sampling operation on the intermediate output image. When the stage ordering differs, it results in under-utilization of resources, as certain stages must be skipped during the benchmark mapping process. The three compute stages of the PE are discussed in more detail below.

Stencil Stage:- The stencil operation generally involves a computation that multiplies and accumulates values. The processing units (PUs) within the stencil stage are arranged in a reduction tree-like pattern. This stage comprises $m+1$ PU arrays, where m is equal to $\log(4 \times P_x)$. Each PU within array i reads two input values from the adjacent PUs in the array $i-1$. The first PU array performs the multiplication, while the subsequent arrays handle the accumulation step. A binary reduction tree is efficient for computing stencil windows whose size is a power of 2 but this window size is uncommon in image processing algorithms. Typically, window sizes are 3×3, 5×5, etc. To address the issue of odd sizes, the window can be zero-padded to make its size a power of 2, but this approach under utilizes the PUs. Optimal throughput is achieved when this stage processes the maximum possible number of stencils in parallel. Therefore, we use a unique data layout that rearranges the line buffer windows across the first PU array. This layout is as follows.

The proposed approach is to break each stencil window into smaller partitions whose size is a power of 2. For instance, a 3×3 window is partitioned into two smaller partitions, with sizes 8 and 1, respectively. In general, a window of size k^2 is broken down into at most w partitions, denoted as p_0 through p_w, where $w = \lfloor \log_2(k^2) \rfloor - 1$. In the new data layout, the same-sized partitions from all windows are positioned next to each other. These partition groups are ordered from left to right, based on decreasing size. The mapper module located within the stencil stage has access to all windows created by the line buffers. It is a multiplexer array that maps a value from a line buffer window to an input port of a PU. The multiplexers are configured to implement this data layout.

In order to illustrate the data layout and operation of the stencil stage, we provide an example in Figure 4.8. In part (a), the stencil stage processes two 1×3 stencil windows. The window is first partitioned into two smaller partitions of sizes 2 and 1, respectively. The partitions are then rearranged using the data layout as [2,2,1,1] over the first 6 processing units (PUs), with the filter coefficients stored in the PU registers. Similarly, in part (b), the stencil stage processes a single 1×7 window which is partitioned into three smaller partitions of sizes 4, 2, and 1, respectively, and arranged over the first 7 PUs. In part (a) of Figure 4.8, the 6 partial products produced by the first PU array are accumulated by the rest of the PU arrays. This is achieved by first reducing the partitions corresponding to a window to single values inside the tree. The reduced value r_i^m corresponds to a partition p_i^m of size m, belongs to window i, and is generated at the $\log m^{th}$ array. For example, in part (a) of Figure 4.8, the 2 sized partitions p_1^2, p_2^2 are reduced to the single values r_1^2, r_2^2 by the second ($\log 2 = 1$) PU array. All the reduced values corresponding to a window must be combined into a single value, which is not possible using the level-to-level PU interconnections since the reduced values lie at different PU arrays. Therefore, the reduced values are aligned and forwarded using the shifter and the vector register between the arrays. The vector register at level i stores the output of the $i-1^{th}$ PU array. The reduced values r_1^1 and r_2^1 of size 1 are stored at positions 5 and 6 inside the first vector register, respectively. These values needs to be added with r_1^2, r_2^2, produced by the first and second PUs of the second PU array. Since r_1^1 and r_2^1 are produced early, they are moved to positions 1 and 2 by shifting the first vector register by 4 units. Subsequently, these two values are propagated. The first and second PUs adds the forwarded value with

the inputs received from its predecessor PUs in the tree structure to produce the final output. The stencil stage also supports other reduction operators, such as max or min over a window. When using max or min operators over a stencil window, the stencil coefficients are set to 1 and the PUs are configured to perform the reduction operation using the maximum or minimum function instead of multiplication and accumulation. This is because the max and min operators do not require multiplication with filter coefficients, but rather involve comparing values to find the maximum or minimum. Therefore, the PUs are set up to accumulate the partial results using the max or min function, depending on the desired operation.

Pointwise Stage:- The pointwise stage can obtain input from either the stencil stage or directly from the input vector, bypassing the stencil stage. The PU arrays within this stage are all of equal length, and data exchange occurs through an all-to-all routing network between the arrays. This network consists of multiplexers that connect the output of a PU at level i to the inputs of the PUs at level $i+1$. The inter-level multiplexers are configured by control words generated by the host, which also determine the operation of the scalar units within each PU. In cascaded mode, the PU utilizes both the scalar units and three input values. The first scalar unit processes the first two inputs, while the second scalar unit operates on the output of the first unit and the third input value. A ternary operator of the form $E_1 \odot E_2 ? E_3 : E_4$ is treated as a pointwise operation. Here E_1 through E_4 are pointwise expressions, and $\odot$ is a relational operator. This operation is also processed by the pointwise stage as follows. Assume the expressions E_1 though E_4 have already been processed. Two PU units, execute the operation $R_1 = E_1 \odot E_2$ and $R_2 = \neg(E_1 \odot E_2)$. The value of R_1 and R_2 is either a 1 or a 0. Following this, two more PUs from the successive level computes $S_1 = R_1 \times E_3$ and $S_2 = R_2 \times E_4$. The value of S_1 is either E_3 or 0, and the value of S_2 is either E_4 or 0. Finally, S_1 is added to S_2 to get the final value. In summary, after computing the four pointwise expressions inside the ternary operator, the pointwise stage utilizes 5 PUs spread across 3 arrays to produce the final output.

Upsample-Downsample Stage:- In this stage multiple images can be simultaneously downsampled or a single image can be upsampled by a factor of two. In the case of downsampling, a single Processing Unit (PU) is responsible for this operation. To perform the downsampling, a scalar unit inside the PU is configured. A PU register is set up with the row length of the input image that needs to be downsampled. As the input data is received, the scalar unit increments a counter to keep track of the row and column index of the image. The scalar unit then outputs data from every other row and column, effectively reducing the image size by half.

The upsample operation is processed by multiple PUs along with the upsampler module. If the image to be upsampled is produced earlier, it is forwarded to this stage through intermediate PUs in the pointwise stage. Image is upsampled by generating a 2×2 matrix for every single input received. More precisely, For every pixel $w_{r,c}$ belonging to the r^{th} row and c^{th} column of the input image, a matrix $W = [0, 0, 0, w_{r,c}]$ is generated. A total of four PUs generates W. The first 3 PUs are configured to produce a 0 in the output, and the fourth PU forwards the received input $w_{r,c}$. The matrix W is read by the upsampler module that stores it across four internal memory banks U_0 through U_3 in a

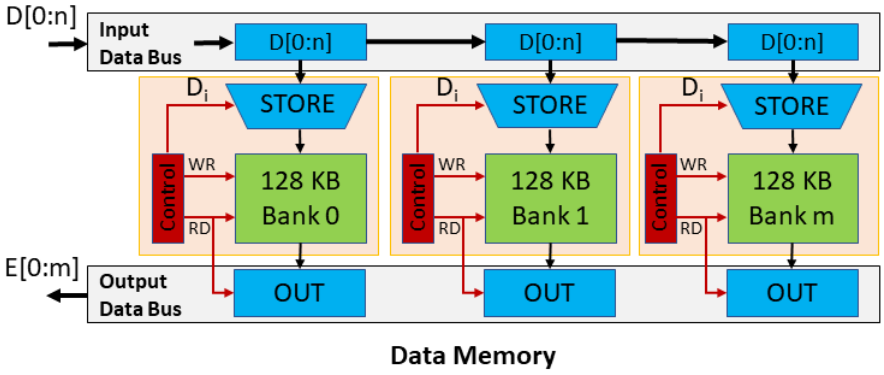


Figure 4.9: The figure displays how the data and control memory are organized within the overlay. The data memory is specific to a CU and is structured as a series of 128 KB memory banks. On the other hand, the control memory is built using registers and is distributed across all the overlay modules.

striped fashion. Note that these banks are separate from the CU memory banks and are exclusive to the upsampler module. The data in the four banks is collated and stored sequentially as a single upscaled image inside one of the CU memory banks by the upsampler module. This is done by emptying the banks U_0 through U_3 in an interleaved fashion. U_0 contains data from rows $0, 2, 4, 6...$ and column $0, 2, 4, 6....$ U_1 contains data from the same row indices but odd columns. U_2 and U_3 contain odd rows and even and odd columns, respectively. The drain sequence starts by reading a single value from U_0 followed by the other value from U_1, alternating between both until row 0 is filled in B; here, B is a CU memory bank. Then the same alternating sequence is repeated with U_2 and U_3 until row 1 is filled in B. At this point, the upsampler module again switches back to the first two banks. This interleaved sequence is repeated until all the rows of the upscaled image are buffered in B, marking the end of the upsample operation.

4.4.2 Control Memory

The dynamic configuration of the overlay is stored in the control memory as control words, which can be seen in Figure 4.10. The control memory is distributed across all modules of a CU, with each module having its own set of dedicated registers to store control words for its runtime functioning. This design ensures no communication bottleneck is created when the modules read their respective control words. The control words are 32-bit (16-bit index and 16-bit value) and transmitted over a hierarchical tree-structured control bus. The 16-bit index is used to route the value through the hierarchical tree structure of the control bus. For example, the top levels of the tree route the control word to the correct CU, followed by the next levels routing it to a PE and the lower levels routing it to a module inside the PE.

Before every computation phase, a control phase is initiated to set the control words. Minimizing the time spent setting the control words is essential to reduce stall cycles. The host sets a bit-vector

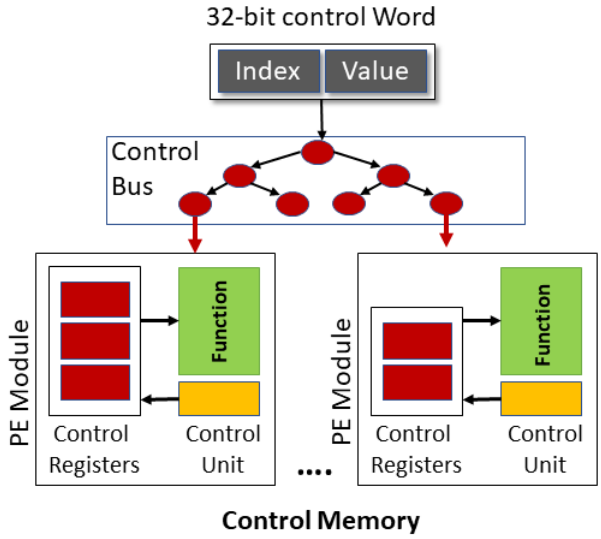


Figure 4.10: The figure displays how the data and control memory are organized within the overlay. The data memory is specific to a CU and is structured as a series of 128 KB memory banks. On the other hand, the control memory is built using registers and is distributed across all the overlay modules.

register E before the control phase begins. Each overlay module that depends on control words has a corresponding bit representation in the vector. Only modules with a set bit in E are reconfigured during the control phase, while others continue to execute the same functionality as defined by their previous control word setting.

4.4.3 Data Memory

The output of a CU is stored in the data memory, as shown in Figure 4.9. An input data bus transfers the output data stream from a PE to a set of 128 KB memory banks. Each bank has its control unit generating the read-write signals and can be configured to store data from any index of the vector data stream. The input vector $D[0 : n]$ is relayed from bank to bank. When the control unit activates the store signal at a bank, the bank reads the $D[i]$ data value from the input vector and stores it in memory. The index value i is stored in the control memory of the bank. Banks that do not have their store signal activated forward the input vector to the following bank in the sequence through the data bus. When the write signal is activated, the bank transfers its output on the output data bus. The output data bus is parallel and can be accessed by all the banks simultaneously. The output from all the banks is concatenated into a vector that can be sent back to the host CPU or looped back to the CU, depending on the behaviour determined by the CU control unit. The looped back data behaves as the input for the future CU computations.

Consider a practical scenario within the described computational framework. Assume an input vector $D[0 : n]$ with values [10, 20, 30, 40, 50]. As the PE performs a vector operation, let's denote the

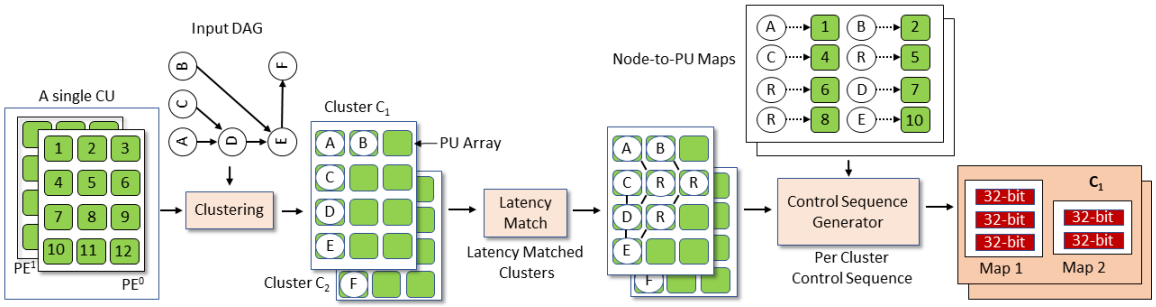


Figure 4.11: The depiction showcases different phases of the compilation process, where the FlowPix compiler produces a series of control words corresponding to the input DAG.

computed result as $R[0:n]$. The CU activates the store signal at a specific memory bank, say Bank 3, instructing it to store the computed value $R[3]$ at a designated index, like index 3. Meanwhile, other banks not activated for storage forward the input vector to the subsequent bank through the data bus. This sequential movement allows each bank to process its specified index. Subsequently, when the CU decides to retrieve results, it activates the write signal, prompting Bank 3 to transfer its output on the output data bus. Simultaneously, all banks contribute their outputs to the parallel output data bus, resulting in a concatenated output vector O, e.g., $O = [R[0], R[1], R[2], R[3], R[4]]$.

4.5 FlowPix Compiler

The FlowPix compiler undergoes a three-phase process, as illustrated in Figure 4.2, to generate a control word sequence from the DSL code. In the initial phase, known as the *DAG Builder*, the DSL code is processed to construct a Directed Acyclic Graph (DAG) representation, where nodes signify computations, and edges denote producer-consumer relationships. The resulting DAG is stored internally and serves as the foundation for the subsequent phases. Following this, the compiler conducts a range inference pass on the DAG to compute image dimensions at each node, crucial for generating accurate control words and ensuring precise hardware execution.

The second phase, termed the *Mapper*, establishes a mapping between DAG nodes and compute units. This phase comprises two sub-phases: clustering and latency matching. During clustering, the DAG is partitioned into smaller sub-graphs, or clusters, representing the maximum number of DAG nodes a single Compute Unit (CU) can execute simultaneously, adhering to specific constraints. Subsequently, in the latency matching phase, the formed clusters are adjusted to accommodate variable latency nodes.

The *Control Sequence Generator* operates in the concluding phase, generating a control word sequence for every mapped node within the cluster. At this point, the compiler holds detailed information about each DAG node, encompassing its type, parent nodes, and CU mapping. For example, if dealing with a stencil node, the compiler possesses knowledge regarding the stencil size, stride, coefficients, and

dimensions of input/output images. The effective collaboration among these three phases empowers the compiler to convert DSL code into a control word sequence. This process enhances the precision and efficiency of executing an image processing pipeline.

$$\texttt{Ordering constraint} \rightarrow \; L(x) < L(y) \implies P(x) < P(y) \tag{4.1}$$

$$\texttt{Location constraint} \rightarrow \; P_{low}(t(x)) \leq P(x) < P_{high}(t(x)) \tag{4.2}$$

4.5.1 DAG Clustering

The clustering phase in the FlowPix compiler involves partitioning DAG into clusters, which are smaller compute blocks processed by a CU. In this phase, the multiplicity of Compute Units is not initially considered. All image strips mapped to Compute Units undergo processing through the same cluster. The input DAG, denoted as S, is subdivided into distinct clusters S_1, S_2, and so on, up to S_n. Nodes within a cluster are then mapped to Processing Units (PUs) inside the Processing Elements (PEs), assuming the entire CU is accessible to execute the cluster.

During the initial treatment of S as a single cluster, the compiler assigns PUs to the nodes in S while adhering to specified constraints. The assigned nodes constitute the first cluster, S_1. This clustering process recurs on the remaining DAG ($S - S_1$) to create subsequent clusters (S_2, and so forth) until S is empty, signifying convergence.

To assign PUs, DAG nodes are first sorted in topological order and grouped into different levels, denoted as L_0 through L_n. In Figure 4.11, for example, the DAG is categorized into four levels: $L_0 = [A, B, C]$, $L_1 = [D]$, $L_2 = [E]$, and $L_3 = [F]$. These DAG levels are then matched with available PU arrays, represented as P_0 through P_m. PU arrays within a PE are distributed across stencil, pointwise, and upsample/downsample stages. The type of a node x, denoted as $t(x)$, dictates which PE stage or PU arrays can process it. The permissible range of PU arrays to process a given node type is determined by $P_{low}(t(x))$ to $P_{high}(t(x))$. Notably, only the PU array at the base of the multiply-and-accumulate tree is considered for stencil nodes.

Ensuring correct mapping involves satisfying two constraints: the ordering constraint (expressed in equation (4.1)) and the location constraint (articulated in equation (4.2)). The ordering constraint guarantees that DAG nodes are assigned to PU arrays in level order. If nodes x and y belong to DAG levels $L(x)$ and $L(y)$ respectively, and $L(x) < L(y)$, then node x must be allocated to a PU array, denoted as $P(x)$, with a lower index. This constraint ensures that data flows within the CU in the same direction as in the DAG.

Restrictions imposed by the location constraint limit the assignment of a node of a specific type to a subset of PU arrays. Upon identifying the appropriate PU array, the assignment process concludes by designating the next accessible PU within that array. The compiler keeps track of PU availability through a map encompassing all PUs within the PE. In cases where a suitable PU cannot be identified for a given node, the compiler investigates the next accessible PE within the CU. Should all PEs be

utilized, the compiler consolidates the existing cluster, resets mapping data structures, and then advances to establish the subsequent cluster using the remaining DAG nodes.

4.5.2 Latency Matching

Consider a scenario in which a CU is mapped to a cluster consisting of three nodes: x, v, and w. Data transmission from nodes v and w to node x is involved, and they are assigned to the i^{th} and j^{th} PU arrays, respectively, with $j-i > 2$. Node x is specifically mapped to the $j+1^{th}$ PU array within the CU. During execution, direct data exchange takes place between the PUs assigned to nodes w and x since they are located in adjoining PU arrays. Conversely, there is no direct connection between nodes v and x. To ensure correct execution, a buffer of size $j - i$ needs to be inserted along the route between v and x. Regrettably, the PE lacks buffering capacity between PUs, leading to the introduction of additional PUs that relay data from the i^{th} to the j^{th} array. The compiler addresses this by introducing relay nodes into the cluster, distributing them with one PU per array across the intervening $j - i - 2$ arrays.

The relaying mechanism is employed to address variable latency across the input edges of a mapped node. For successful relaying, each PU array inside the PE reserves a set of PUs as forwarding units. These lightweight forwarding units do not perform computations and match the length of the array. The compiler configures relay nodes based on the difference in edge latencies across DAG nodes. In Figure 4.11, node E mapped to the fourth PU array receives input from node B at the first PU array, requiring two relay nodes between them. However, no relaying is needed for the other input to node E since the data originates from the preceding PU array. The same principle applies to node D with inputs from nodes A and C.

In certain situations, a lack of relay nodes can impede data transmission across all edges within a cluster. Consider a scenario where a PU consists of four compute levels, each with a single relay node. Cluster C_1 (depicted in Figure 4.11) cannot be mapped to the PU due to an insufficient number of relay nodes, particularly in the second level requiring two relay nodes. To address this, the compiler divides the cluster, ensuring smaller clusters receive an adequate number of relay nodes. In the example, C_1 is split into two clusters: one with nodes A, C, and D, and the other with nodes B and E. The cluster division involves removing nodes from C_1 to create two new clusters, C_1^a (the reduced cluster with adequate relay connectivity) and C_1^b (containing the deleted nodes). This process is recursively applied to C_1^b. Nodes are eliminated in reverse topological order (starting with node E), and all dependent nodes of the deleted node are removed. Additionally, any node whose output is no longer utilized by any other node within the cluster is eliminated (e.g., node B).

4.5.3 Control Word Generation

In the final phase, the compiler produces 32-bit control words for each PU-to-node mapping. Generating these control words involves a series of steps, and the quantity of control words varies depending on the type of node being mapped. Templates for each mapping type, containing a predefined sequence

of control words, are stored in a lookup table. During compilation, specific values are assigned to these templates. For instance, when mapping a PU to an addition operation, three control words are necessary. The initial control word configures the scalar unit for the addition operation, while the subsequent two words establish the input ports of the PU with corresponding input source indices. These sources may be other PUs for intra-cluster dependencies or the PE input vector for inter-cluster dependencies.

When dealing with mapping stencil nodes, the compiler generates a unified control word sequence to set up the reduction tree for processing all stencils collectively. Initially, the line buffer array is configured with the stencil size, stride, and image dimensions. The lookup table contains line buffer settings for a pre-defined set of stencil sizes, and processed stencils must share the same shape. Following this, the mapper module is configured to determine the data layout (as discussed in Section 4.4.1) based on the number and size of the stencils. The stencil coefficients are loaded into the registers of the PUs located at the base of the reduction tree. PUs in the stencil stage are pre-set to execute multiply and accumulate operations, and shifter units between PU arrays are configured with the appropriate shift amounts. Additionally, PUs can be configured with min/max operations to process min/max filters, respectively. Finally, the output module of the stencil stage is configured to transmit the output correctly downward to the pointwise stage.

In configuring a processing unit (PU) for a 3x3 stencil convolution with a stride of 2, the compiler generates a series of control words. Initially, Control Word 1 specifies the stencil size (3x3) and stride (2), guiding the PU to traverse the input data accordingly. Control Word 2 configures the line buffer array to accommodate the stencil, while Control Word 3 sets up the mapper module for correct data layout based on the specified stencil and stride. Control Words 4-12 are dedicated to loading the nine stencil coefficients into the PU registers. Following this, Control Word 13 configures the PU for convolution operations, and Control Word 14 adjusts shifter units to optimize computation for strided convolutions. Depending on the architecture, additional control words may be included for optional configurations like mix/max operations. Together, these control words tailor the PU to efficiently perform a 3x3 stencil convolution with a stride of 2, ensuring precise processing of input data in line with the convolution parameters.

4.5.4 Correctness

The ordering (Equation (4.1)) and location (Equation (4.2)) constraints in the FlowPix compiler ensure the correct mapping of DAG nodes to processing units (PUs) within compute units (CUs) by maintaining the logical and functional structure of the computations. Here's how each constraint contributes to the correctness of the DAG mapping:

Ordering Constraint Guarantees

1. **Topological Order Maintenance**: Ensures that the nodes are processed in a topological order, meaning each node is processed only after all its dependencies (parent nodes) have been processed. This is crucial for maintaining the data flow integrity, as nodes at a lower level (which

represent earlier computations) must be executed before nodes at a higher level (which represent subsequent computations).

2. **Data Dependency**: By enforcing $P(x) < P(y)$ when $L(x) < L(y)$, the constraint ensures that the data produced by node x (at level $L(x)$) is available before it is needed by node y (at level $L(y)$). This guarantees that data dependencies are respected, and no computation attempts to use data that hasn't been produced yet.

3. **Sequential Execution**: Within the CU, the data flows in the same direction as in the DAG, ensuring that the computation sequence in hardware mirrors the logical sequence in the DAG. This prevents race conditions and ensures that all intermediate results are correctly computed and available when needed.

Location Constraint Guarantees

1. **Node Type Suitability**: Ensures that each node is assigned to a PU array capable of executing the specific type of computation it requires. Different types of nodes (e.g., stencil, pointwise, upsample/downsample) may have different computational requirements and optimizations. By restricting the PU assignment to a specific range, the compiler ensures that the appropriate hardware resources are used for each node type.

2. **Resource Allocation**: Prevents overloading or underutilization of PUs by distributing the nodes across the available PU arrays according to their types. This ensures a balanced workload and optimal use of the hardware resources.

3. **Functional Correctness**: Enforces that nodes are not arbitrarily placed in any PU array but rather in those specifically designed to handle their computations. This minimizes the risk of functional errors and inefficiencies in the execution.

Together, the ordering and location constraints ensure that data dependencies are respected and intermediate results are correctly produced and consumed.

4.6 FlowPix Scheduler

The FlowPix scheduler, leveraging the driver API, takes on the responsibility of coordinating the real-time scheduling of clusters generated by the compiler onto the overlay. Initially, a cluster is formed with the assumption that the entire memory banks of the Compute Unit (CU) are available to it. However, during runtime, the CU memory banks are shared among all cluster executions. The scheduler's goal is to create an optimal or acceptable execution schedule that minimizes external memory traffic between the host and FPGA. This is accomplished by streaming the input image only once and retaining partial data generated by clusters within the CU memory banks. If there are insufficient memory banks to

meet the schedule's needs, the scheduler triggers an exception, prompting the user to re-synthesize the overlay with an increased number of memory banks.

To create an execution schedule, the scheduler relies on the cluster graph and the per-cluster control word sequence as input. The cluster graph outlines interdependencies among clusters, where each cluster may have one or more parents. This graph, essentially a Directed Acyclic Graph (DAG), can be envisioned as a tree, with the cluster generating the final output acting as the root. Each cluster serves as the root of a subtree, and for a cluster to be eligible for execution, all clusters within its corresponding subtree must be processed. The scheduler maintains a dynamic map to track the mapping of compute nodes within a cluster to specific memory banks.

Before diving into the scheduling algorithm, let's establish a few terms that will be referred to throughout this discussion.

Definitions: The number of banks needed to store the output of cluster M is denoted by $B_o(M)$, while the number of banks from which cluster M reads its inputs is indicated by $B_i(M)$. Considering all the clusters belonging to the subtree rooted at cluster M, the maximum value of B_i is referred to as $B_{imax}(M)$. Essentially, B_{imax} identifies the highest number of input bank requirements needed to compute all the clusters in the subtree. The bank utilization factor $U(M)$ for a cluster is given by $B_o(M) - B_{imax}(M)$. This value approximates the effective number of banks used by a cluster. Clusters with a negative value of $U(\cdot)$ indicate a situation where the number of banks consumed for computation exceeds the number of banks needed to store the output. In such cases, the number of CU banks currently in use decreases.

Theorem 1. *Given a set of clusters eligible for execution, the overall bank utilization is minimized by processing the clusters in the increasing order of their bank utilization factor.*

Proof. Assume a cluster C with two parent clusters, M and N. We can make this assumption without losing the generalization of the problem. It is necessary to compute M and N before computing C. We prove that if $U(M) > U(N)$, then the bank utilization is lesser if we compute N first followed by M.

We calculate the maximum number of banks utilized when M is computed first followed by N. $B_{imax}(M)$ is the maximum number of input banks required to compute M. The output of M is stored in $B_o(M)$ banks. Next, to compute N, $B_{imax}(N)$ banks are utilized. Therefore the maximum number of banks utilized in this compute sequence is $\max\{B_{imax}(M), B_o(M) + B_{imax}(N)\}$ which is equal to $B_o(M) + B_{imax}(N)$. This can be proved by simple substitution using our initial assumption that

$U(M) > U(N)$.

$$
\begin{aligned}
& B_o(M) - B_{imax}(M) > B_o(N) - B_{imax}(N) && \text{Initial assumption} \\
\implies & B_o(M) + B_{imax}(N) > B_o(N) + B_{imax}(M) && \\
\implies & B_o(M) + B_{imax}(N) > B_{imax}(M) && \text{since } B_o(N) > 0
\end{aligned}
$$

Now, when N is computed first followed by M, the maximum number of banks in use will be

$$\max\{B_{imax}(N), B_o(N) + B_{imax}(M)\} \tag{4.3}$$

There are two case to consider here.

1. $\mathbf{B_{imax}(N) > B_o(N) + B_{imax}(M)}$: Therefore maximum banks utilized is $B_{imax}(N)$. If executing M first leads to a lower bank utilization, then $B_o(M) + B_{imax}(N)$ must be less than $B_{imax}(N)$. This implies that $B_o(M) < 0$ which is not possible.

2. $\mathbf{B_{imax}(N) < B_o(N) + B_{imax}(M)}$: Therefore maximum banks utilized is $B_o(N) + B_{imax}(M)$. If executing M first leads to a lower bank utilization, then $B_o(M) + B_{imax}(N)$ must be less than $B_o(N) + B_{imax}(M)$. By simple rearrangement, $B_o(M) - B_{imax}(M) < B_o(N) - B_{imax}(N)$, implying that $U(M) < U(N)$. This contradicts out initial assumption.

Therefore, we conclude that if $U(M) > U(N)$, then N must be computed first to achieve a lower overall bank utilization. □

Scheduling Algorithm: The FlowPix scheduler employs a systematic approach to generate a schedule that meets certain criteria, striving to select clusters with the lowest bank utilization factor throughout the process. The algorithm initiates its operations from the root of the cluster DAG, a data structure that represents the relationships and dependencies among clusters. This initiation signals the beginning of a recursive exploration of subtrees, focusing on those whose root nodes exhibit the lowest utilization factors. This recursive strategy ensures that the scheduler consistently prioritizes clusters with optimal resource usage, specifically targeting the judicious utilization of Compute Unit (CU) memory banks. The outcome of this meticulous process is an intelligently sequenced execution of clusters, resulting in a schedule that not only adheres to the constraints imposed by the lowest bank utilization factor but also aligns with the overarching objective of efficient CU memory bank utilization, as asserted in Theorem 1.

In the above example, the scheduling order of the nodes A, B, C, and D is determined by the FlowPix scheduler's algorithm. Node A is the root of the Directed Acyclic Graph (DAG), initiating the scheduling process. As it has no dependencies, it is scheduled first. Nodes B and C are dependent on Node A.

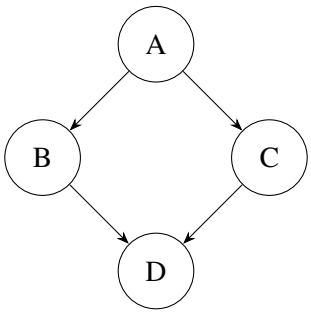


Figure 4.12: Node DAG

Node	**Utilization Factor**
A	5
B	3
C	4
D	6

Table 4.3: Utilization Factors

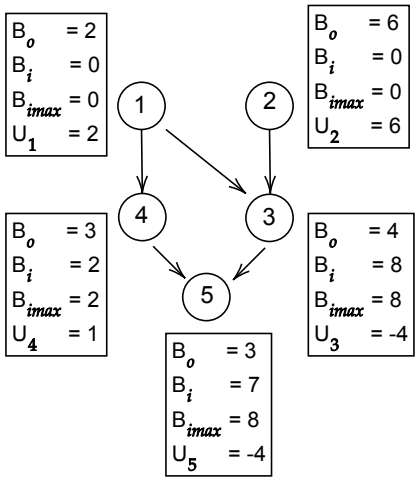


Figure 4.13: Figure shows a graph on 5 clusters being executed on 8 memory banks.

The FlowPix scheduler prioritizes nodes based on their utilization factors. In this case, Node B has a utilization factor of 3, and Node C has a utilization factor of 4. The scheduler strategically prioritizes Node B and then Node C, starting with the node that has the lower utilization factor. Node D has dependencies on both Nodes B and C. The FlowPix scheduler, in its recursive and strategic approach, schedules Node D after Nodes B and C. Node D has a utilization factor of 6, which is higher than Nodes B and C. However, the scheduler takes into account the dependencies and strategically schedules Node D after ensuring that Nodes B and C have been executed.

Figure 4.13, shows a example where $B_i \neq B_{imax}$. The clusters execute on 8 banks. The scheduling order of the clusters is $1- > 2- > 3- > 4- > 5$.

4.6.1 Examples

We employ a dummy benchmark, illustrated in Figure 4.14, to exemplify the clustering and execution. The overlay is configured with a single CU with one PE. The PE is set with parameters $P_x = 6$ and $P_y = 4$. The CU can concurrently process a maximum of two 3×3 stencil nodes. The clustering phase yields the formation of six clusters, denoted as S_1 through S_6. Nodes 1, 2, and 7 coexist within the same cluster, as 1 and 2 represent parallel stencil nodes that fit within the same cluster. Pointwise node 7 relies on the outputs from nodes 1 and 2. Clusters S_2 and S_3 are structured similarly. However, stencil nodes 10 and 11 cannot be accommodated in clusters S_1 or S_2 due to their dependence on a pointwise node, and there is no internal data path in the PE to transfer data from a pointwise to a stencil node. Additionally, upsample nodes 14 and 15 are assigned to separate clusters due to the PE's limitation in processing a single upsample operation.

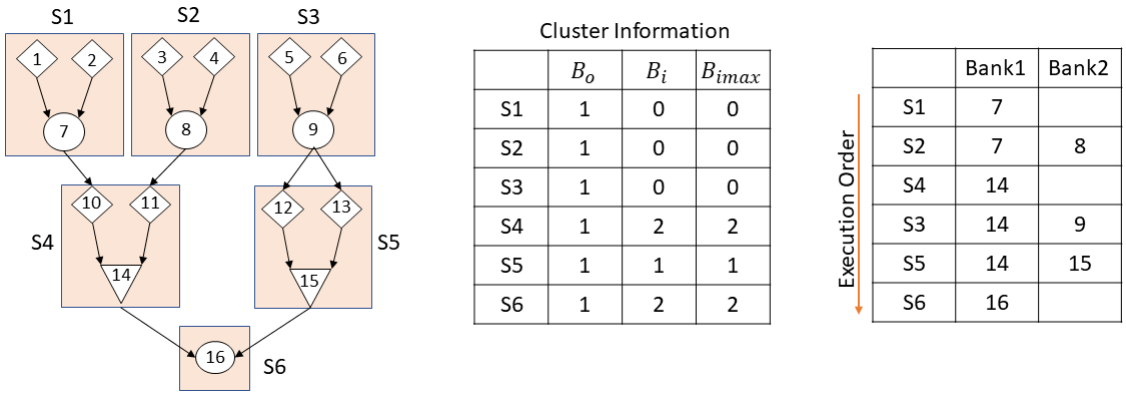
The scheduler-generated cluster execution sequence starts with S_6 as it is the root of the cluster DAG. S_6 depends on S_4 and S_5. Between S_4 and S_5, the scheduler selects S_4 since it exhibits a lower bank utilization factor (-1 ¡ 0) than S_5. S_4 in turn relies on S_1 and S_2, with both clusters having identical

utilization factors. The scheduler executes the source cluster S_1 followed by S_2. Subsequently, S_4 is processed. A similar execution pattern is applied within the subtree rooted at S_5, resulting in the execution of clusters S_3 and S_5. Finally, S_6 is processed.

Pyramid blending is a sophisticated image processing technique designed to achieve seamless integration between two images. The process starts by constructing Gaussian pyramids for each input image. A Gaussian pyramid is created by successively applying a Gaussian filter and downsampling the image, resulting in a series of images at different scales. Subsequently, Laplacian pyramids are derived from the Gaussian pyramids. Each level of the Laplacian pyramid represents the difference between the corresponding levels of the Gaussian pyramid and an upsampled version of the next lower level. This effectively captures both low-frequency and high-frequency details in the images. The blending operation takes place at each level of the Laplacian pyramid, involving techniques like weighted averaging. This step ensures that the blending process considers information at multiple scales, avoiding visible artifacts and producing a visually appealing result. Finally, the blended image is reconstructed by summing up the levels of the blended Laplacian pyramid. Pyramid blending is extensively utilized in computer graphics, computer vision, and image processing applications, providing an effective solution for tasks such as image stitching, panorama creation, and seamless layer blending in image editing. The DAG for the pyramid blending benchmark and the resulting clusters are depicted in Figure 4.15. Nodes 1, 2, 3, and 4, representing 3×3 stencils, are distributed across clusters 1 and 2. This distribution is due to the constraint that each cluster can accommodate only two stencils simultaneously. The downsample node 5 relies on the outputs from nodes 1 and 2. However, the upsample node 9 cannot be placed in cluster C1, as each cluster can only house one type of upsample or downsample node. Node 7, representing the pointwise operation, doesn't fit in either C1 or C2 because it receives input from a downsample node, and there is no direct data path within the processing element (PE) to transfer data from a downsample node to a pointwise node. Similarly, pointwise node 14 cannot be accommodated in cluster C5 since it receives input from an upsample node, and there is no corresponding data path within the PE. Following these constraints, nodes 9 and 10 are allocated to their individual clusters. Nodes 11, 12, 13, and 14, being pointwise nodes with an overall depth of 3, are grouped together in a single cluster.

4.7 Experimental Results

On a Virtex-7-690t FPGA, we conducted an evaluation involving 15 image-processing benchmarks using FlowPix. These benchmarks cover a spectrum of structures and complexities, ranging from straightforward pipelines like Unsharp Mask to more intricate ones such as Optical Flow with multiple input images. Additionally, we implemented iterative pyramid systems like Gaussian Pyramids and Up-Down sampling. In our analysis, we compared FlowPix with several existing FPGA-based image processing frameworks, namely Polymage [65], DarkRoom [34], HeteroHalide [53], and the Vitis-Vision library [18]. These frameworks generate fixed-function hardware from a DSL specification.



	B_o	B_i	B_{imax}
S1	1	0	0
S2	1	0	0
S3	1	0	0
S4	1	2	2
S5	1	1	1
S6	1	2	2

	Bank1	Bank2
S1	7	
S2	7	8
S4	14	
S3	14	9
S5	14	15
S6	16	

Figure 4.14: The figure depicts a dummy application with 3x3 stencil, pointwise, and upsample nodes, as well as its clustering and execution schedule. The DAG clusters are executed over a CU with two memory banks. The number on each bank corresponds to the node whose output it presently stores.

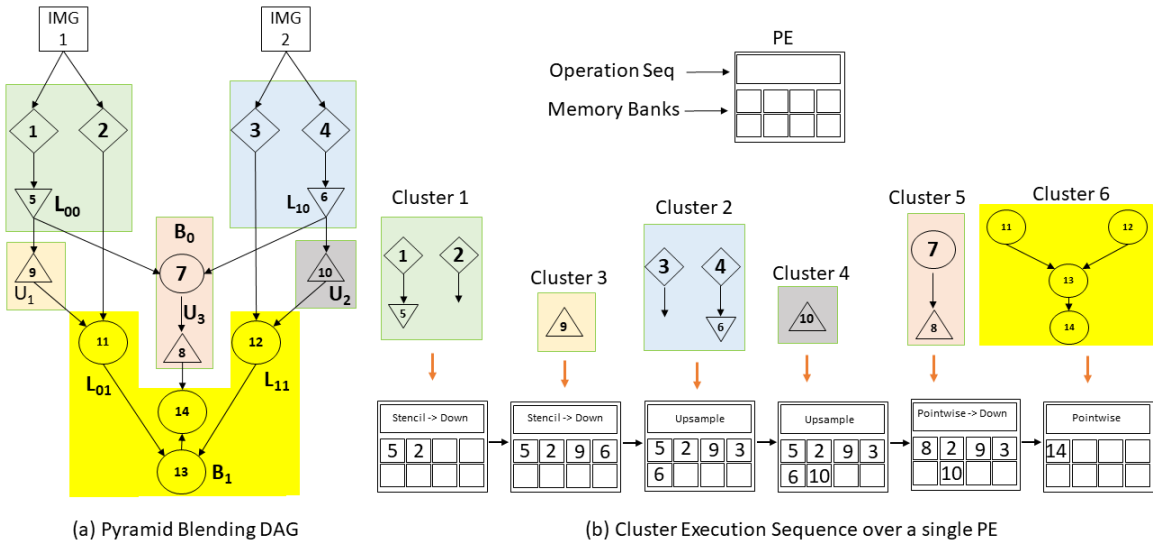


Figure 4.15: The figure depicts the Pyramid Blending application with 3x3 stencil, pointwise, and upsample nodes, as well as its clustering and execution schedule.

Details about our benchmark suite can be found in Table 4.4. Furthermore, we conducted a comparative analysis between FlowPix and IPpro [83], which adopts an overlay-based approach.

Experimental set-up: Our overlay is synthesized on a Virtex-7 690t FPGA featuring 3600 DSP blocks and 6.4 MB of on-chip BRAM (2940 18Kb blocks). This FPGA card is linked to an Intel Core-i5 processor via a PCIe-8x link. The host driver, responsible for streaming image frames in row-major order, is written in C++ and executes on the CPU. We utilize the Xillybus PCIe core (available at `http://xillybus.com/doc/revision-b-xl`) to connect the host input with our overlay, as illustrated in Figure 4.16. The Xillybus core can achieve an ideal data bandwidth of 6.4 GB/s in each direction (read from and write to the host) when operating on the Virtex-7 device with 8x Gen3 lanes, utilizing the Gen3 Integrated Block for PCI Express v3.0. Further details can be found at `http://xillybus.com/doc/xillybus-bandwidth`.

Our hardware design is implemented using Bluespec System Verilog (BSV) [69]. BSV enables us to precisely specify the desired hardware using a set of transactions or rules in an atomic manner, expressing system behavior in all-or-none transactional semantics. The reported hardware characteristics are obtained using the Vivado Design Suite version 2022.1 Post Place and Route.

To ensure compatibility with various benchmarks across different frameworks, we created four variants of the Processing Element (PE) by modifying both the datatype and the PE size. The characteristics of these PE variants are listed in Table 4.6. Although other design variants were possible, we specifically chose these four. The number of Compute Units (CUs) and PEs within a CU were determined based on the benchmark and framework being used. To minimize comparative latency and LUT consumption, we synthesized the overlay using one of the PE variants and increased the number of CUs to match the pixel throughput. For example, when comparing FlowPix with a framework that generates 32 pixels per cycle for a given benchmark, we synthesized the overlay with a CU count of 8, considering a single PE within the CU generates 4 pixels per cycle. After matching the pixel throughput, we increased the PE count within a CU until the relative LUT increment was within a 50% (1.5x increase) threshold. This increase in PEs aimed to investigate if pipelined parallelism could reduce the achieved latency while slightly increasing LUT resource consumption.

To accurately measure the processing latency of a benchmark, we subtract the loop-back time from the total execution time. The loop-back time refers to the time required for the hardware to stream in and stream out an image without undergoing any processing. It includes factors such as the PCIe transfer time and other overhead introduced by the Xillybus framework. To establish the hardware in loop-back mode, we remove the FlowPix overlay and connect the write FIFO to the read FIFO, as shown in Figure 4.16. It's important to note that the loop-back time remains constant at 20 milliseconds regardless of the benchmark. However, the time required for transferring the control words is not included in the loop-back time calculation, as this value varies depending on the benchmark being processed.

Driver Code: The code in Figure 4.16 utilizes the OpenCV library for video processing and interacts with an FPGA overlay through device files. It initializes a video capture object to capture frames from the default camera, allocates memory to store data received from the FPGA, and opens device files

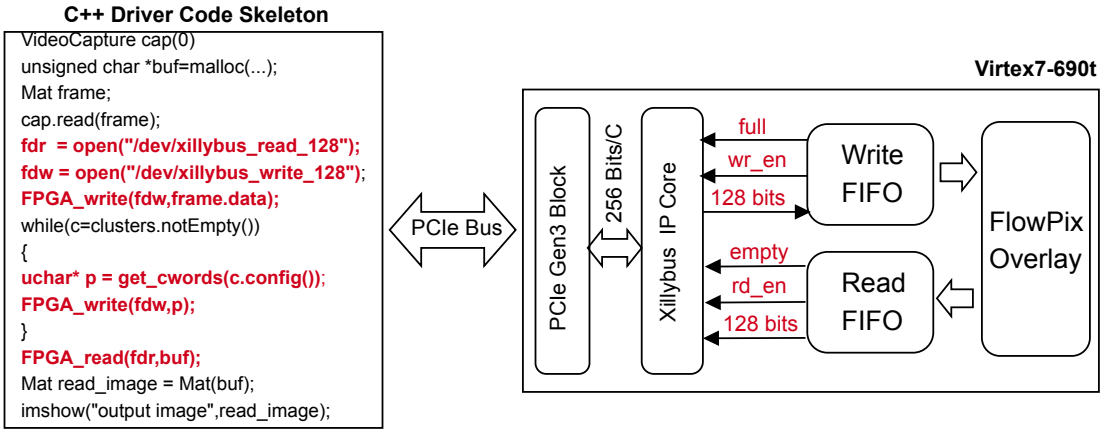


Figure 4.16: The diagram depicts the integration of the FlowPix overlay through the utilization of the Xillybus IP with read and write FIFO interfaces. This integration allows for the host to effortlessly stream input to the overlay, as well as receive output from it, via simple read and write file interfaces. The input image is streamed only once and the output image is read after all the clusters are processed.

associated with the FPGA for reading and writing. Frames captured from the camera are written to the FPGA using a custom function `FPGA_write`, and a loop processes clusters by obtaining configuration words and sending them to the FPGA. Subsequently, data is read from the FPGA using a custom function `FPGA_read`. The received data is converted into an OpenCV Mat object and displayed using the `imshow` function.

4.7.1 Performance Comparison

In this section, we present a comparative analysis of FlowPix alongside other frameworks, emphasizing their throughput and FPGA resource consumption across various benchmarks. Resource utilization details for Look-Up Tables (LUTs) and Flip-Flops (FFs) are provided, and the total consumption of BRAM and DSP can be calculated based on the number of Processing Elements (PEs) employed. For example, utilizing a 16 Compute Units (CUs) overlay, each with an A4 type PE, would result in total DSP block utilization of $16 \times 176 = 2816$ and BRAM utilization of $1 \times 16 \times 144 = 2304$ (refer to Table 4.6). Table 4.5 compares the Lines of Code of FlowPix with other frameworks. To ensure a fair comparison, we process each benchmark under the same settings as reported by the respective frameworks, considering specifications such as image width (W), height (H), data type, and pixels produced per cycle (P/C).

Table 4.7 presents a comparison between FlowPix and the HeteroHalide framework across reported benchmarks, all processed at a pixel throughput greater than 1. FlowPix demonstrates comparable latency to HeteroHalide for 8-bit cases, with an average 1.7x increase in FPGA LUT consumption. In other benchmarks, latency increases by 25%, accompanied by an average 1.6x increase in FPGA LUT consumption. Notably, the `BLU` benchmark exhibits the highest increase in LUT consumption,

SL No	Benchmark	Acronym	Description
1	Harris Corner	HCD	Corner Detection Using 3 x 3 Stencil and Point-wise Operations.
2	Canny Edge	CED	Edge Detection Using 5 x 5 Stencil and Point-wise Operations.
3	Gaussian Filter	GAF	Application of a 3 x 3 Gaussian Filter over an Image.
4	Blur	BLU	Average operation over a 3 x 3 window of pixels.
5	Linear Blur	LB	Blur operation followed by 2 linear transformation operation.
6	Pyramid	PYR	Pyramid of Up-sampling or Down-sampling Gaussian Filters
7	Down-Up	DUS	Single Down-sampling followed by Up-sampling operation.
8	Erosion	ERS	Minimum operation over a 3 x 3 window of pixels.
9	Median	MED	Median operation over a 3 x 3 window of pixels.
10	Dilation	DIL	Maximum operation over a 3 x 3 window of pixels.
11	Convolution	CONV	A 8 x 8 convolution operation over an Image.
12	Lucas Kanade	LK	Single iteration of Lucas-Kanade Optical flow Algorithm.
13	Stencil Chain	SC	3x3 Stencil Operation repeated 3 times in a pipeline.
14	Gaussian Diff	GAD	Difference of a single and double Gaussian Blur.
15	Pyramid Blend	PYRB	Blending of two images using 2 level Image Pyramids.

Table 4.4: A short description of the benchmarks implemented using FlowPix on a Virtex-7 FPGA.

Framework	HCD	CED	GAF	USM	BLU	LB	SC	DIL	ERS	MED	CONV	PYR	LK	DUS
Flowpix	15	8	13	6	1	11	3	2	2	2	1	19	23	10
Halide	15	-	15	7	2	9	3	-	-	-	-	16	-	-
Polymage	43	-	-	16	-	-	-	-	-	-	-	71	75	50
Hetero Halide	26	-	8	13	2	11	15	2	2	2	-	-	-	-

Table 4.5: Lines of code comparison of the different frameworks for the benchmarks implemented.

Arch	Datatype	Px	LUT	FF	DSP	BRAM (18Kb)	Frequency MHz
A1	16 bit	8	9970	16984	176	144	200
A2	8 bit	8	5062	8673	88	80	250
A3	16 bit	16	13681	22111	352	272	200
A4	8 bit	16	5721	9971	176	144	250

Table 4.6: FPGA resource consumed by the different architecture variants of our overlay. P_y is set to 8 in all the PE designs.

	Input				Hetero Halide			FlowPix					Relative Performance		
	W	H	Data Type	P/C	LUTs	FFs	Latency (ms)	LUTs	FFs	Latency (ms)	PEs , CUs	PE	xLUTs	xFF	xLatency
HCD	2448	3264	UInt8	32	55198	64427	1.00	91536	159536	1.00	1,16	A4	1.66	2.48	1
GAF	2160	3840	UInt8	32	67298	41496	1.04	45768	79768	1.04	1,8	A4	0.68	1.92	1
USM	2448	3264	UInt8	32	47683	33114	3.00	91536	159536	3.00	1,16	A4	1.92	4.82	1
BLU	648	482	UInt16	16	6821	8209	0.08	54724	88444	0.10	1,4	A3	8.02	10.77	1.25
LB	768	1280	Float32	8	31049	39369	1.47	27362	44222	1.84	1,2	A3	0.88	1.12	1.25
SC	1536	2560	UInt16	16	61230	46174	0.98	109448	176888	1.23	1,8	A3	1.79	3.83	1.25
DIL	6480	4820	UInt16	32	13046	12114	3.90	54724	88444	4.88	1,4	A3	4.19	7.3	1.25
MED	6480	4820	UInt16	32	14388	10066	3.90	54724	88444	4.88	1,4	A3	3.83	8.78	1.25

Table 4.7: Comparing FlowPix with HeteroHalide.

	Input				Vitis Vision			FlowPix					Relative Performance		
	W	**H**	**Data Type**	**P/C**	**LUTs**	**FFs**	**Latency (ms)**	**LUTs**	**FFs**	**Latency (ms)**	**PEs, CUs**	**PE**	**xLUT**	**xFFs**	**xLatency**
HCD	1080	1920	UInt8	8	13222	9330	1.7	22884	39884	2.07	1,4	A4	1.73	4.27	1.22
GAF	1080	1920	UInt8	8	2791	3641	7	5062	8673	8.28	1,1	A2	1.81	2.38	1.18
PYR-D	1920	1080	UInt8	1	1171	1238	6.99	5062	8673	8.29	1,1	A2	4.32	7.01	1.19
PYR-U	1920	1080	UInt8	1	1124	1199	27.82	5062	8673	33.14	1,1	A2	4.50	7.23	1.19
LK	3840	2160	UInt8	1	7730	11984	28.01	11442	19942	32.18	2,1	A4	1.48	1.66	1.14
CED	1080	1920	UInt8	8	6518	4899	8.5	11442	19942	2.07	1,2	A4	1.76	4.07	0.24

Table 4.8: Comparing FlowPix with Vitis Vision library implemented using Vitis-HLS.

	Input			DarkRoom+Rigel		FlowPix				Relative Performance	
	W	**H**	**P/C**	**LUTs**	**Latency (ms)**	**LUTs**	**Latency (ms)**	**PEs, CUs**	**PE**	**xLUTs**	**xLatency**
HCD	1080	1920	1	**12208**	13.83	13681	20.60	1,1	A3	1.12	1.49
CED	1080	1920	1	15696	15.10	13681	40.20	1,1	A3	0.87	2.66
LK	1080	1920	1	222000	11.91	27362	10.34	2,1	A3	0.12	0.87
CONV	1080	1920	4	20748	4.39	22884	2.50	1,4	A4	1.10	0.57
PYR-D	384	384	4	45220	0.55	22884	0.20	1,4	A4	0.51	0.36

Table 4.9: Comparing FlowPix with Darkroom and Rigel.

	Input				PolyMage			FlowPix					Relative Performance		
	W	**H**	**Size**	**P/C**	**LUTs**	**FFs**	**Latency**	**LUTs**	**FFs**	**Latency**	**PEs, CUs**	**PE**	**xLUTs**	**xFFs**	**xLatency**
HCD	1920	1080	24 bits	1	11314	5422	1.83	9970	16984	20.71	1,1	A1	0.88	3.13	11.32
USM	1920	1080	24 bits	3	6883	2500	5.85	9970	16984	10.36	1,1	A1	1.45	6.79	1.77
DUS	1920	1080	24 bits	3	6115	2352	5.39	9970	16984	10.36	1,1	A1	1.63	7.22	1.92

Table 4.10: Comparing FlowPix with PolyMage.

		FlowPix			IPPro
Micro Benchmark	**Description**	**Control Cycles**	**Compute Cycles**	**Latency** μs	**Latency** μs
CONV	A single 3x3 Gaussian filter	168	22	0.76	0.14
POLY	Degree-2 polynomial with non-zero constants	68	5	0.29	3.29.
FIR	5-tap Finite Impulse Response Filter	116	18	0.53	5.34

Table 4.11: Comparison of FlowPix with IPPro.

computed with 16-bit precision. Conversely, the GAF benchmark achieves better LUT utilization than HeteroHalide, as it employs the 8-bit PE version. Achieving desired latency and throughput involves scaling PEs for pipelined parallelism and CUs for data parallelism.

Table 4.8 compares FlowPix with the Vitis Vision library implemented using Vitis-HLS, an industry-standard providing a software API interface for FPGA-accelerated computer vision functions. FlowPix exhibits approximately 20% higher latency than Vitis Vision, attributed to different synthesis frequencies. For pyramid benchmarks, FlowPix's relative LUT consumption is over 4x that of Vitis for a single-level pyramid using 5×5 filters. However, Vitis excels at creating an area-efficient design for this benchmark. For LK optical flow, FlowPix employs pipelined parallelism, achieving 1.25x better latency than Rigel. For CED edge detection, FlowPix outpaces Vitis by 4x when processed with two 3×3 filters, utilizing the A4 PE type.

Table 4.9 compares FlowPix with Darkroom and Rigel frameworks. FlowPix exhibits 1.5x to 2.7x higher latency for HCD and CED benchmarks than Darkroom at single-pixel throughput. The increase in latency is due to using a single PE, resulting in more clusters time-multiplexed over the same PE. For CED, FlowPix's LUT consumption is 23% better than Darkroom. FlowPix outperforms Rigel in latency and LUT consumption for various benchmarks, achieving 1.25x better latency for LK and 50% fewer LUTs for PYR-D. However, Rigel outshines FlowPix in CONV with 10% less LUT usage.

Table 4.10 compares FlowPix with the PolyMage framework. For HCD, FlowPix exhibits 11 times higher latency due to processing in two clusters over a single PE. For USM and DUS, FlowPix leverages channel parallelism, achieving lower latency than PolyMage. However, the latency increase for DUS is considerable due to processing over two clusters.

Transitioning from fixed-function hardware frameworks, we compare FlowPix with the IPPro instruction set-based processor in Table 4.11. While the control cycles are 7x-10x higher than compute cycles, FlowPix achieves considerably lower latency for POLY and FIR functions compared to IPPro.

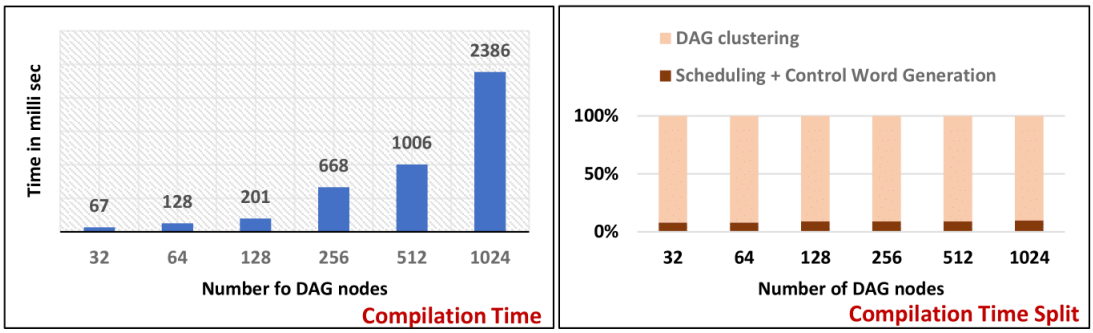


Figure 4.17: Analysis of the FlowPix Compiler.

4.7.2 Framework Analysis

In this section, we initiate our exploration by evaluating the FlowPix compiler's capability to handle larger designs comprising hundreds of nodes. Subsequently, our focus shifts to the overlay as we delve into the consequences of processing benchmarks using a unified design, contrasting our approach in the previous section where diverse PE designs were employed for different benchmarks. The evaluation outlined in the paragraph is significant, It addresses the critical need to assess the scalability and performance of the FlowPix compiler in handling larger designs comprising hundreds of nodes. This examination is pivotal for understanding the practical utility of the compiler in real-world applications where system designs are often complex.

In our first experiment, we generated Directed Acyclic Graphs (DAGs) with varying numbers of compute nodes, ranging from 32 to 1024. The maximum 1024 is an inflated value as the typical image processing pipelines are limited to 100 stages. To create these DAGs, we utilized a random graph generator that labelled nodes as stencil, pointwise, upsample, or downsample nodes. The labelling process adhered to a set of constraints. For instance, only nodes with a single input were eligible for labelling as stencil, downsample, or upsample, while nodes with two input edges could be labelled as pointwise. These constraints ensured that the generated DAGs could effectively process an input image to produce an output image. Figure 4.17 visually illustrates that the compilation time scales proportionally with the size of the input DAG. In other words, the complexity of the compiler exhibits an $O(n)$ relationship, where n represents the number of DAG nodes. Notably, the clustering phase constitutes approximately 90% of the compilation time, with the remaining 10% dedicated to generating the control word and determining the scheduling order of the clusters. The significance of the clustering phase becomes apparent as the size of the DAG increases, leading to a higher number of clusters that need evaluation.

In the second experiment, as detailed in Table 4.12, we employed a unified 16-bit overlay design to execute all benchmarks. This overlay comprises a single Compute Unit (CU) equipped with one Processing Element (PE), where P_x = 16 and P_y = 8. The chosen input image size for a benchmark is determined as the maximum dimension among all dimensions specified for that benchmark in section

Benchmark	Compile Time	Control Words	Run Time (ms)	FPS	Benchmark	Compile Time	Control Words	Run Time (ms)	FPS
BLU	1 ms	16	0.44	108.70	LB	5 ms	37	4.14	241.55
MED	1 ms	16	21.23	47.11	GAF	6 ms	52	8.74	114.47
DIL	1 ms	16	21.72	46.05	CED	12 ms	73	4.76	210.04
DUS	2 ms	25	12.95	77.22	PYR-U	13 ms	92	39.77	25.15
CONV	2 ms	127	10.95	91.32	HCD	106 ms	130	16.45	60.79
SC	4 ms	48	10.31	97.02	LK	339 ms	261	73.69	13.57

Table 4.12: A single design to process all the benchmarks.

8.1. Compilation times ranged from 1 to 300 milliseconds. When considering both the compilation time and the maximum control word transfer time of approximately 10 ms, the total time is notably shorter than the duration required for writing an FPGA bitstream, typically taking seconds. An overlay design proves highly advantageous when users need to switch between processing pipelines without incurring the overhead of writing a new bitstream each time. Additionally, it is evident that similar benchmarks exhibit comparable compilation times. For instance, both the `BLU` and `MED` benchmarks boast a 1-millisecond compilation time as they involve a single stencil operation.

The increase in runtime is directly proportional to the number of Processing Elements (PEs) and Compute Units (CUs). To illustrate this point, consider the `HCD` benchmark, where the latency is 16.45 milliseconds, representing a 16-fold increase compared to the data reported in Table 4.7. Similarly, the `PYR-U` benchmark demonstrates a latency of 39.77 milliseconds, nearly identical to the latency reported in Table 4.8, given the identical PE and CU count used in both cases. However, a slight increment is observed due to the difference in bit width (16-bit vs. 8-bit), directly affecting the operating frequency of the design.

4.8 Chapter Summary

Our comprehensive evaluation of the FlowPix framework on a Virtex-7-690t FPGA reveals its robust performance across a diverse set of image-processing benchmarks. The experimental results showcase FlowPix's competitiveness when compared to other FPGA-based image processing frameworks, such as Polymage, DarkRoom, HeteroHalide, and the Vitis-Vision library. Our careful analysis involves comparisons of throughput, FPGA resource consumption, latency, and Lines of Code (LoC) with each framework, providing insights into FlowPix's strengths and areas for improvement. The performance comparison section highlights FlowPix's adaptability and efficiency in handling various benchmarks.

Through systematic adjustments in the number of Compute Units (CUs) and Processing Elements (PEs), FlowPix demonstrates competitive latency and resource utilization. Notably, the framework excels in specific benchmarks, outperforming alternatives like HeteroHalide, Vitis-Vision, DarkRoom, Rigel, and PolyMage under certain conditions. This adaptability is facilitated by the flexibility of FlowPix's design, allowing for efficient scaling and optimization based on the characteristics of each benchmark.

Additionally, the experiment employing a unified overlay design for all benchmarks demonstrates the practical advantage of FlowPix in terms of shorter compilation times and ease of switching between processing pipelines. In summary, FlowPix emerges as a versatile and competitive FPGA-based image processing framework, offering a balance between adaptability, efficiency, and ease of use. The framework's ability to handle diverse benchmarks with efficient resource utilization positions it as a promising solution for real-world image processing applications on FPGAs. Our architecture simplifies the process of scheduling image processing (ImP) workloads on the cloud, which in turn enables *ImP-as-a-Service*.

Chapter 5

Overlay Design for Convolutional Neural Networks

In this chapter, we propose, **FlexNN**, an FPGA overlay for efficient processing of Convolutional Neural Networks that can be scaled based on the FPGA's available compute and memory resources. In contrast to the image processing overlay from the previous chapter, the DAG structure in the CNN pipelines are more regular and the DAG operations are relatively more straightforward. The proposed overlay addresses the following two primary challenges.

- **Challenge 1: Exploiting Parallelism in CNN Pipeline**: In addressing the first challenge, the FPGA overlay must possess the capability to harness all forms of parallelism inherent CNN pipeline. Convolutional Neural Networks typically exhibit various levels of parallelism, including data parallelism and model parallelism. Data parallelism involves processing different input data in parallel, while model parallelism involves parallelizing the computation of different layers of the neural network. The challenge lies in efficiently identifying and utilizing these parallelism opportunities to enhance the overall processing speed of the CNN on the FPGA. To meet this challenge, the FPGA overlay needs to implement strategies that effectively distribute the computational workload across the available FPGA resources, ensuring that parallel operations within the CNN pipeline can be executed concurrently. This may involve optimizing the mapping of neural network layers to FPGA components, managing data dependencies, and dynamically adjusting the level of parallelism based on the characteristics of the input data and network architecture.

- **Challenge 2: Maximizing Per-Layer Throughput**: The second challenge focuses on maximizing the throughput achieved on the hardware for each individual layer of the CNN. Unlike generic processing, CNNs often consist of layers with distinct characteristics, such as convolutional layers, pooling layers, and fully connected layers. Each of these layers requires specific optimizations to achieve maximum throughput on the FPGA. Addressing this challenge involves developing tailored strategies for optimizing the execution of each layer. This may include exploring specialized hardware architectures for convolutional operations, implementing efficient memory access patterns, and leveraging FPGA-specific features to accelerate specific layer types. The goal is to ensure that the overlay extracts the highest possible performance from the FPGA

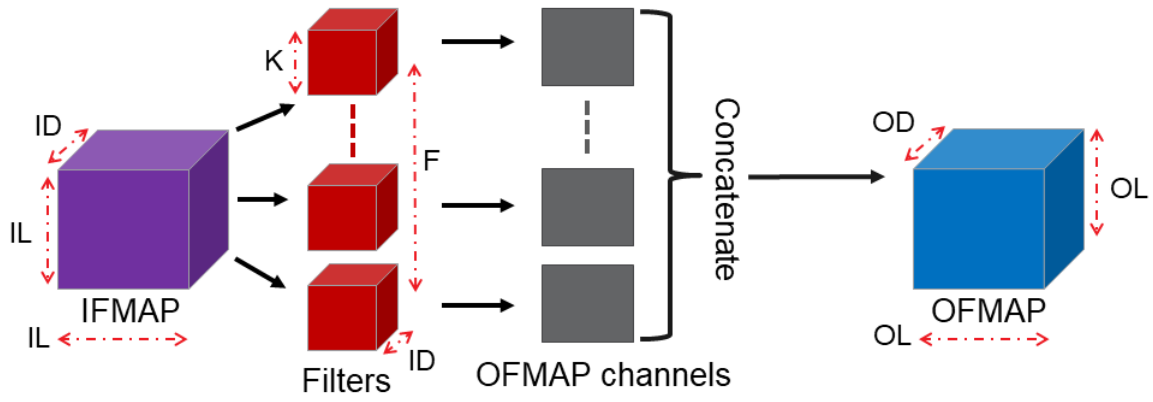


Figure 5.1: A schematic of the convolution operation.

for each layer, collectively resulting in an optimized and efficient overall processing throughput for the entire CNN.

Unlike the image-processing overlay we do not define a new DSL. We use the pre-existing python based Tensorflow specification as the front-end DSL for the CNN overlay. The python API calls are translated by the `C++` driver into FPGA calls to the overlay. More details can be found in the following sections.

5.1 Background on Convolutional Neural Networks

This section provides a brief introduction to the structure of Convolutional Neural Networks (CNNs). The anterior part of a CNN consists of a series of convolutional (CONV) and pooling (POOL) layers, whereas the posterior part can contain zero, one, or multiple Fully Connected (FC) layers.

At each convolutional layer of a CNN, Figure 5.1, an input volume[1] V of dimensions (IL, IL, ID) is convolved with a set of F filters $\{F_i \mid 1 \leq i \leq F\}$, each of dimensions (K, K, ID), to generate an output volume V' of dimensions (OL, OL, OD). If S is the stride of the filter application, then $OL = \frac{IL-K+1}{S}$. The depth slices of the input volume are called input feature maps and are denoted by $IFMAP_i, 1 \leq i \leq ID$. The dimensions of an input feature map $IFMAP_i$ are $IL \times IL$. Similarly, the depth slices of the output volume are called output feature maps and are denoted by $OFMAP_i, 1 \leq i \leq OD$. Note that $OD = F$. The dimensions of an output feature map $OFMAP_i$ are $OL \times OL$.

An output feature map $OFMAP_i$ is rendered by applying a 2D convolution of filter F_i with the input volume V. We call this a 2D convolution since the filter moves only in the length and breadth directions but not along the depth. Each three dimensional filter F_i is an array of two dimensional kernels $F_i^j, 1 \leq j \leq ID$. Then the surface convolution between an input feature map $IFMAP_j$ and a

[1] In this work, input volume refers to the input of the first CNN layer and subsequent intermediary layers too.

filter kernel F_i^j is defined as

$$OFMAP_i^j = IFMAP_j \circledast_s F_i^j. \tag{5.1}$$

Given that, the 2D convolution of the input volume with a filter is algebraically equivalent to the summation of the surface convolutions between their corresponding depth slices. Thus we have the following.

$$OFMAP_i = \sum_{j=1}^{ID} IFMAP_j \circledast_s F_i^j = \sum_{j=1}^{ID} OFMAP_i^j \tag{5.2}$$

An activation function such as Rectified Linear Unit (ReLU) is applied on each pixel of the output volume before it is fed as input to the next layer. Usually, a pooling layer follows a convolution layer. In a pooling layer, `max` or `average` filters are applied on each surface of the input volume. So, when a 2×2 max-pooling filter with a stride 2 is applied on an input volume, then the ouput volume dimensions are $OL = IL/2$ and $OD = ID$. Thus the pooling layers help in reducing the surface dimensions of an input volume. Notice that there is no change in the depth dimension.

The fully connected layers resemble the conventional neural networks in which every neuron from a layer is connected with every other neuron from the previous layer. Due to this, the number of weights per layer will be huge. So as to contain them, fully connected layers start after the input volume dimensions are substantially reduced by the preceding convolutional and pooling layers.

Apart from the canonical full-depth convolution described above, there are other novel convolutions such as point and depth-separable convolutions like in YOLO and MobileNet. In point convolutions, the filter dimensions are $(1, 1, ID)$, which leaves the output volume's surface dimensions the same as the input volume. In a depth-separable convolution layer, the depth of each filter is only 1, and hence the output feature map $OFMAP_i$ is obtained by doing a mere surface convolution between the input feature map $IFMAP_i$ and filter f_i, i.e., $OFMAP_i = IFMAP_i \circledast_s f_i$.

5.1.1 Parallelism and Data-Reuse

From the equation (5.2), we can infer the following four kinds of available parallelism while rendering an output volume, see Figure 5.2.

1. **Filter Parallelism (FP):** Each OFMAP can be calculated in parallel. This is because an OFMAP depends only on the input volume and corresponding filter bank. Let the parameter FP denote the number of OFMAPs generated in parallel.

2. **Surface Parallelism (SP):** While applying a surface convolution, it is possible to compute each value in the OFMAP in parallel, and there are OL^2 such values. Let the parameter SP denote the number of surface convolutions computed in parallel.

3. **Channel Parallelism (CP):** From equation (5.2), we can see that an OFMAP is obtained by the summing of the ID surface convolutions. Each of these surface convolutions can be computed in

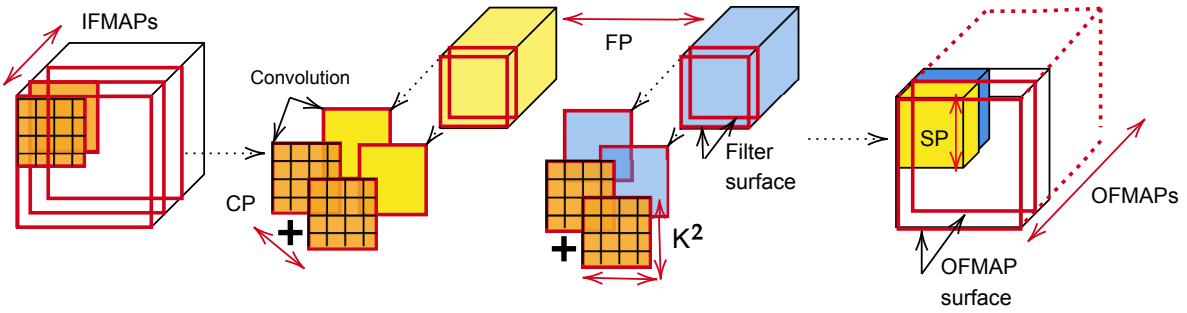


Figure 5.2: An schematic overview of the types of parallelism exploited in our overlay.

parallel. Let the parameter CP denote the number of surface convolutions computed in parallel across the input volume's ID channels.

4. **Kernel Parallelism ($\mathbf{K^2}$):** A single value of an OFMAP is created by applying a filter with a kernel of size K over an IFMAP. A total of K^2 multiply-and-accumulate operations are required to compute this value and all of them can be executed in parallel.

Data reuse of an input data point refers to the number of computations it contributes to in the overall processing of a layer. Each filter weight, in any filter, is used to generate OL^2 pixels in an OFMAP, and hence the corresponding reuse factor is OL^2. While rendering an OFMAP, each value in an IFMAP is used $(K/S)^2$ times, where S is the stride. There are in total F OFMAPs. Therefore the overall reuse factor for a single IFMAP value is $F \times \left(\frac{K}{S}\right)^2$.

5.2 Proposed Approach

It is possible to exploit all the aforementioned kinds of parallelism by suitably orchestrating computations on our overlay. The type of parallelism exploited has a direct impact on the data reuse. We usually exploit filter, surface and kernel parallelism to saturate the DSP utilization. Utilizing channel parallelism puts stress on the available memory bandwidth. The canonical approach we adopt is to fetch a filter only once from off-chip memory and completely reuse it before discarding it. Those fetches are equivalent to compulsory misses in the cache parlance of CPUs. This means we achieve maximal data reuse with respect to filter coefficients. An input volume pixel, once fetched, is completely reused while rendering FP OFMAPs and then discarded, to be re-fetched again while rendering the next batch of FP OFMAPS. Thus a value from an IFMAP is fetched F/FP times from off-chip memory leading to a reuse of $FP \times (K/S)^2$. In the next section, we present how we orchestrate the layer computations over our overlay to exploit different parallelism types while maximizing data reuse.

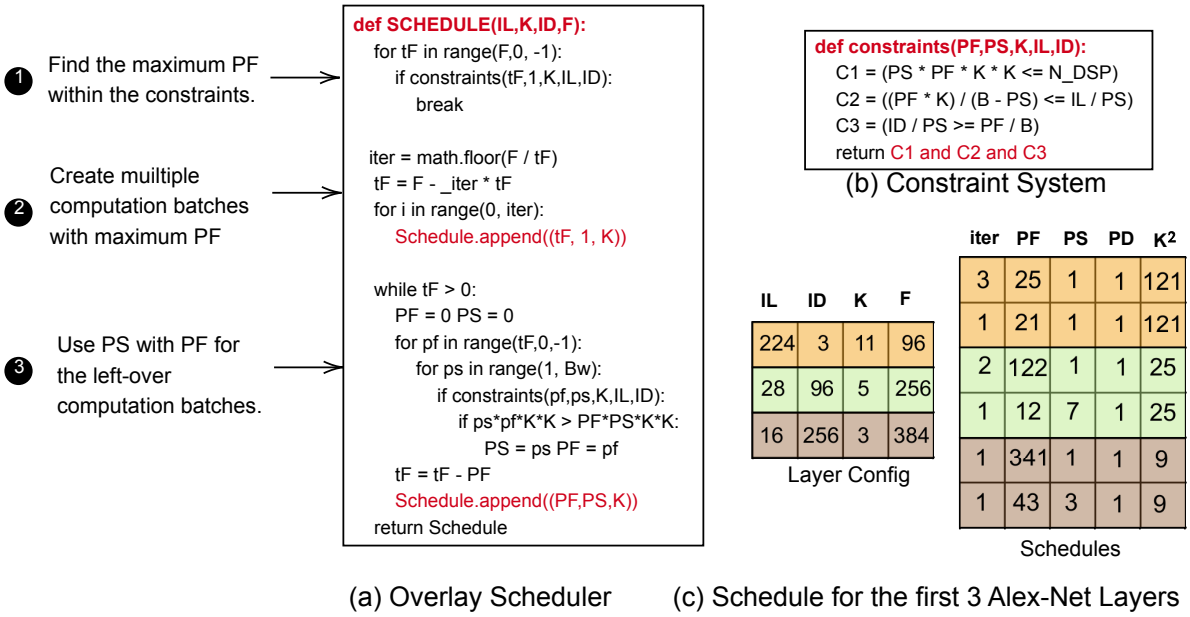


Figure 5.3: The scheduler logic and the compute schedule generated for the first three layers of AlexNet. The scheduler code is written in python.

5.3 Scheduling Computations on Our Overlay

Multiply-and-accumulate (MAC) operations dominate the computations in a CNN. Let M^{mac} be the total number of MAC operations from all the CNN layers put together. These operations are executed on the physical FPGA DSP blocks. If there are N^{dsp} blocks on the FPGA, then the peak achievable compute throughput, measured in MACs/cycle, is N^{dsp}. If hardware is synthesized at a certain clock speed, then the effective compute throughput achievable depends on two factors: the off-chip memory bandwidth and the number of MAC operations performed for every input fetched from the off-chip memory (data reuse factor). A single inference pass of a CNN requires at least $\frac{M^{mac}}{N^{dsp}}$ clock cycles. To approach this theoretical lower bound, and in general, to minimize the end-to-end latency of a design, we have to arrive at an overlay architecture and a suitable way to schedule the layer computations on the overlay such that the data reuse is maximized.

In the rest of this section, we first describe the execution model of our overlay architecture. Then we show how we batch the computations within a layer using a system of constraints and schedule them on the overlay while maximizing parallelism and data reuse to achieve peak DSP utilization.

5.3.1 Execution Model

We use a single-engine accelerator design approach for our overlay. A single processing pipeline computes all the layers of the network. The output volume rendered while processing layer i is streamed out to external memory and later loaded as input when computing the next layer $i + 1$. With respect

to the Equation (5.2), rendering the output volume is equivalent to computing the output feature maps $OFMAP_i$ for $1 \leq i \leq F$. These output feature maps are computed in batches iteratively. The number of OFMAPs computed in each batch, which is equal to FP, can vary and is determined by the scheduling algorithm described in Section 5.3.3.

We compute convolutions in a streaming fashion, wherein the input volume is streamed, and the necessary filter coefficients are pre-fetched and stored in registers. The pre-fetch of filter coefficients overlaps with computation to avoid stall cycles.

Each input feature map $IFMAP_j$ of the input volume is surface convolved with the corresponding depth slice F_i^j where filter F_i belongs to the current batch. This computation starts only after the corresponding depth slices of all the current batch filters are available on the FPGA registers. Then each input feature map $IFMAP_j$ is streamed in a row major fashion to compute the corresponding surface convolution with respect to each filter. The computed surface convolution is accumulated in respective output buffers. The number of surface convolutions done in parallel depends on the parallelism factor SP. While the surface convolutions are being computed, the next depth slice's filter coefficients are streamed using a double buffering technique.

Once the minimum number of input pixels required for carrying out the filter convolutions have accumulated in the input buffers, surface convolutions can be computed at a steady throughput determined by FP and SP. At any point, the accelerator stores a set of rows from an input slice and multiple partially rendered output volume slices. The line buffering of the input ensures that only a working set of rows from the input slice is required to sustain a fixed number of surface convolutions every cycle.

Once a filter batch is done, i.e., all the filter depths have convolved with the corresponding input volume depths, the output slices stored in the buffers are drained out to external memory one at a time. This drainage of output corresponding to a filter batch is overlapped by executing parallel convolutions corresponding to the next batch of filters. To do this, a second set of buffers are used for accumulating the output of the next batch. Note that the input volume has to be streamed again at this point. This double buffering scheme ensures maximum compute to memory overlap during the processing of multiple filter batches.

$$\texttt{DSP constraint} \rightarrow \quad FP \times SP \times K^2 \leq N^{dsp} \tag{5.3}$$

$$\texttt{DRAM bandwidth constraint} \rightarrow \quad SP \times CP \leq \frac{B_e}{S^2} \tag{5.4}$$

$$\texttt{Number of BRAM blocks constraint} \rightarrow \quad 2 \times FP \times SP \leq N^{bram} \tag{5.5}$$

$$\texttt{Compute/Memory overlap constraint} \rightarrow \quad \frac{OL^2}{SP} \geq \frac{FP \times K^2}{B_w} \tag{5.6}$$

$$\texttt{Output flush constraint} \rightarrow \quad \frac{OL^2 \times ID}{SP} \geq \frac{FP \times OL^2}{B} \tag{5.7}$$

5.3.2 Constraint System

The runtime configuration of our overlay is characterized by the type of operation (convolution, max pooling, etc.) and the parallelism mix with which it is executing that operation. We set constraints on the parallelism factors, such that the processing pipeline in our overlay operates with zero or minimum stalls. The constraint system is discussed in the rest of the section. At first, we recall the CNN layer and FPGA parameters used in the constraint system.

- **Layer Parameters:** At each layer of a CNN, an input volume of dimensions (IL, IL, ID) is convolved with a set of F filters, each of dimension (K, K, ID), to generate an output volume of dimensions (OL, OL, F).
- **FPGA Parameters:** Let the number of DSP and BRAM blocks on the FPGA be N^{dsp} and N^{bram}, respectively, and $2B$ bytes/cycle be the combined read and write memory bandwidth to external memory.

If $FP \times SP$ surface convolutions are computed per cycle and the filter dimensions are (K, K), then the number of MACs per cycle are $FP \times SP \times K^2$. The maximum number of MACs is bound by the number of available DSPs (N^{dsp}) on the FPGA yielding the DSP constraint (5.3). Our overlay is configured to process different types of convolutions. One such type are depth-wise convolutions prevalent in sparse CNNs like MobileNet. These special convolutions are computed with extra DSP resources. The DSPs used for computing the MACs in the depth-wise computations are very few compared to the DSPs used during the normal convolutions. Because of the low number, these DSPs have not been taken into the DSP constraint (5.3).

If B_e is effective off-chip memory bandwidth, since surface convolutions are computed in a streaming fashion, every surface convolution requires S^2 additional inputs, corresponding to an IFMAP, from off-chip memory, where S is the filter stride. Hence, to compute SP convolutions in parallel, $SP \times S^2$ inputs have to be fetched from off-chip memory. For parallel convolutions across the channel dimension, every cycle CP inputs corresponding to an CP different IFMAPs have to be fetched. Combining the above two factors, leads to the DRAM bandwidth constraint (5.4). Notice that the off-chip memory bandwidth has to be split between the surface and channel type of parallelism.

The number of available BRAM blocks on the FPGA impacts the parallelism factors FP and SP. To facilitate SP parallel convolutions for each of the FP output volume slices from the current batch, we store each output volume slice in SP BRAM blocks in an interleaved fashion. Thus we need $FP \times SP$ BRAM blocks. Since we are doing double buffering to overlap computation and communication, we need BRAM blocks twice that number. This leads to the BRAM size/bandwidth constraint (5.5). The input buffer utilizes very few BRAM blocks compared to the output buffers. Empirically, we use a $16 \times L$ line buffer, where L is the row length of the largest IFMAP. The buffer just utilizes 16-20 BRAM blocks. Therefore, we keep the BRAM count from the input buffer out of the constraint system. The channel parallelism factor CP does not contribute towards BRAM usage since all the CP convolutions reduce to a single value. We resort to channel parallelism only for point-wise convolutions where $K = 1$.

We view a BRAM block as an array of fixed-point numbers of type (IB, FB) where IB and FB denote integral and fractional bit widths. When we say that a BRAM block's size is A, it can hold A pixels of a given fixed-point type. For many CNNs, in the first few layers, the output dimension OL^2 could be much greater than $A \times SP$ limiting the available filter parallelism. This limitation is relaxed by tiling the input volume across its surface into vertical cubes of dimensions $IL \times OL' \times ID$.

As we move from one input channel to another, we have to pre-fetch the corresponding filter weights for the respective channel. During these cycles, there is no DSP utilization. Alternatively, we can use part of the bandwidth B to overlap fetching of filters from the next channel while computing convolutions on the current channel,i.e. $B_e = B - B_w$. Here B_w is the part of the input bandwidth that is used for fetching of filter weights and it varies across layers. In our accelerator design, we use this approach, which leads to a small decrease in the available bandwidth B for the input volume pixels, but at the same time, prevents the DSPs from stalling for filter data. For a given SP, $\frac{OL^2}{SP}$ is the number of cycles it takes to render FP partial output surfaces, for a filter batch, in parallel. To hide filter pre-fetch latency, $FP \times K^2$ weights corresponding to the next batch of a filter from the respective channel should also be fetched within this interval using a part of the DRAM bandwidth. This compute/memory overlap constraint to hide the communication latency is given in (5.6).

As the current batch of OFMAPs is computed, the previous batch of OFMAPs is written to off-chip DRAM. The number of cycles required for computing a batch of FP OFMAPs is given by $(FP \times OL^2 \times ID)/(FP \times SP)$ which is equal to $(OL^2 \times ID)/SP$. The size of the output volume generated in each step is $FP \times OL^2$. Assuming that the write bandwidth is equal to the read bandwidth of the DRAM, the number of clock cycles required to spill the entire output buffer is $(FP \times OL^2)/B$. To minimize the interference between computing and communication and reduce the potential stall cycles as the output buffer is not yet completely spilled, the output flush constraint (5.7) has to hold good and as tight as possible. If this constraint gets violated, the correctness may not be affected, but there will be an imbalance between computation and communication cycles, due to which stalls will be incurred. In other words, there will be cycles where the DSP blocks are idle.

The constraint system is part of the CPU host logic written in a high level language like python. These constraints are used by a scheduler, that generates per layer accelerator configurations for processing the current layer.

5.3.3 Layer Scheduling

Figure 5.4 shows the overall workflow of executing a CNN on the overlay. The host side code for CNN inference can be written using any of the available libraries such as TensorFlow, PyTorch, etc. In this work, we used TensorFlow-Python to test our overlay. The host code communicates with the overlay running on the FPGA through a C++ based driver code.

The host code pre-processes each CNN layer to identify the feasible parameters (FP, SP) satisfying the set of Constraints (5.3) to (5.7). Note that if the overlay is configured to compute FP output feature maps in parallel with a surface parallelism factor SP, then $FP \times SP$ convolutions will be computed

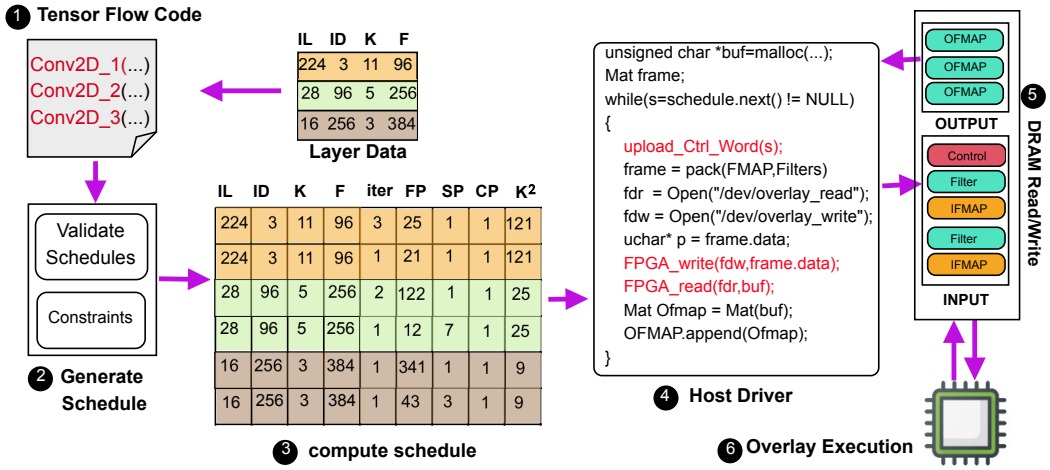


Figure 5.4: The workflow for processing a CNN using our overlay. The tensor flow specification is lowered to a set of accelerator calls using a high-level language (Python). Communication with the overlay is done through a driver API written in C++. Computations of a layer are broken into multiple batches/iterations, in the form of a compute schedule, and executed on the overlay. The hardware is configured on the fly towards processing a compute batch using control words. A compute schedule for three layers of AlexNet is shown in the figure.

per cycle. Hence, in order to maximize the DSP utilization, among the feasible parameters, those parameters (FP, SP) which maximize $FP \times SP$ are filtered. From among them, a parametric pair with maximal FP is chosen as it minimizes the number of times the FPGA is invoked from the host side. The F output feature maps are computed in $\lceil \frac{F}{FP} \rceil$ batches with each batch computing FP output feature maps in parallel. The last batch may not be full and hence to increase the DSP utilization we increase the surface parallelism factor SP subject to the satisfaction of the constraints. The input volume will be re-streamed from DRAM while computing a batch of output feature maps. Our strategy not only maximizes DSP utilization but also maximizes data reuse at the same time.

The host configures the overlay to compute a batch of output feature maps by sending a set of suitable control words (refer Figure 5.4). The overlay uses the control words for hardware reconfiguration. The overlay acknowledges the completion of the configuration phase to the host. At this point, the host packs the input feature maps and filter weights into a data stream. This data stream is communicated to the overlay through the host driver. The processed OFMAPs from overlay are transferred back to the host code via the driver. The OFMAPs are stored in the host memory to be used as input to the subsequent layers.

Overall, the host processes a CNN layer-by-layer. Each layer is further divided into batches. This batching is done so as to maximize DSP utilization, data reuse and reduce the number of FPGA invocations. The overlay is soft-reconfigured using control words to process a batch with a certain filter and surface parallelism parameters.

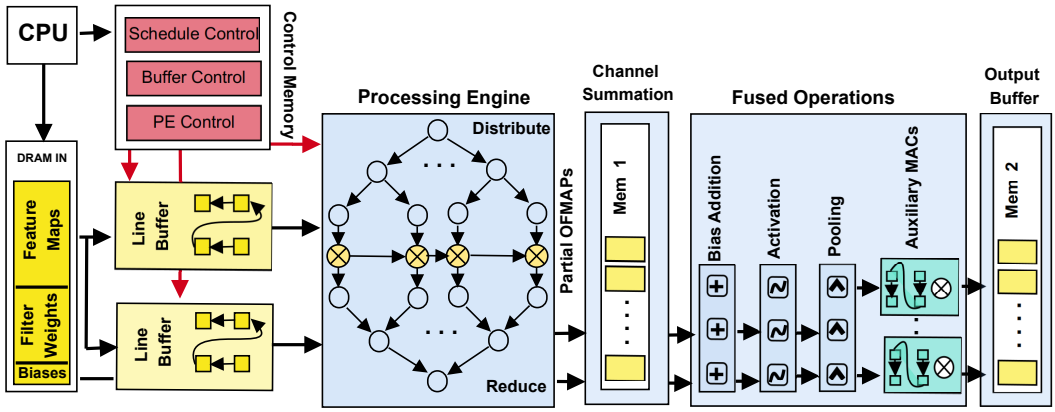


Figure 5.5: The block diagram of our overlay. The control words dynamically configure the memory and the compute modules. These configurations determine the runtime behaviour of the overlay. The control words for each module are stored in a register-based memory.

5.4 Overlay Micro-Architecture

This section describes the micro-architecture of our overlay as shown in Figure 5.5. The host operates the overlay through control words. A register-based control memory contains control words that define the runtime configuration of the overlay. A dedicated set of control words configures the overlay's memory and processing engine modules depending on the compute schedule. Loading up of the control memory marks the beginning of a new computation batch. Input to the overlay is made up of filter weights and feature map values. The host streams this input data on a PCIe-DRAM interface using the C++ driver. IFMAPs are buffered inside the overlay using line buffers. Filter weights are stored using registers. In a steady-state, the line buffer generates SP surface windows over an IFMAP. These windows are convolved with FP pre-fetched filter windows. The resulting $FP \times SP$ convolutions are computed in parallel over the overlay's Processing Engine (PE). The output data from PE is packed into a $SP \times FP$ sized vector.

The partial OFMAPs rendered by the PE are accumulated in a memory block (Mem 1 in Figure 5.5), built from the FPGA BRAMs. The channel summation module uses this memory to sum up, all the partial OFMAPs corresponding to the input volume channels to produce the final output feature map. The final OFMAP is passed through a set of stages representing the operations that typically follow a convolution. The fusing of these operations greatly cuts down the need to re-fetch the processed output volume. Depending on the network specification, these stages can be selectively disabled, in which case they pass the input to the next stage without any modification. The final output is stored in a memory block (Mem 2 in Figure 5.5) from where it is written to the FPGA DRAM to be read by the CPU host driver.

In the following sections, we discuss the line buffer and processing engine modules in our overlay. We focus on the reconfigurability aspect of each of these modules, adding to the reconfigurability of the overall design.

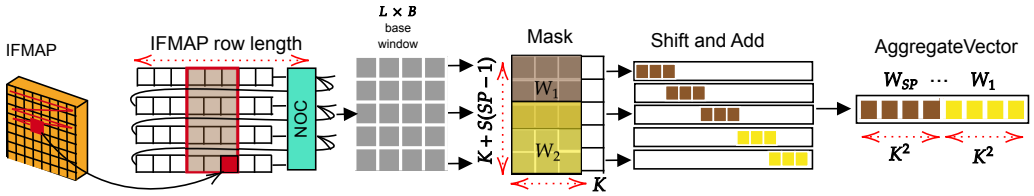


Figure 5.6: The line buffer pipeline inside our overlay generates stencil windows over a streaming input feature map.

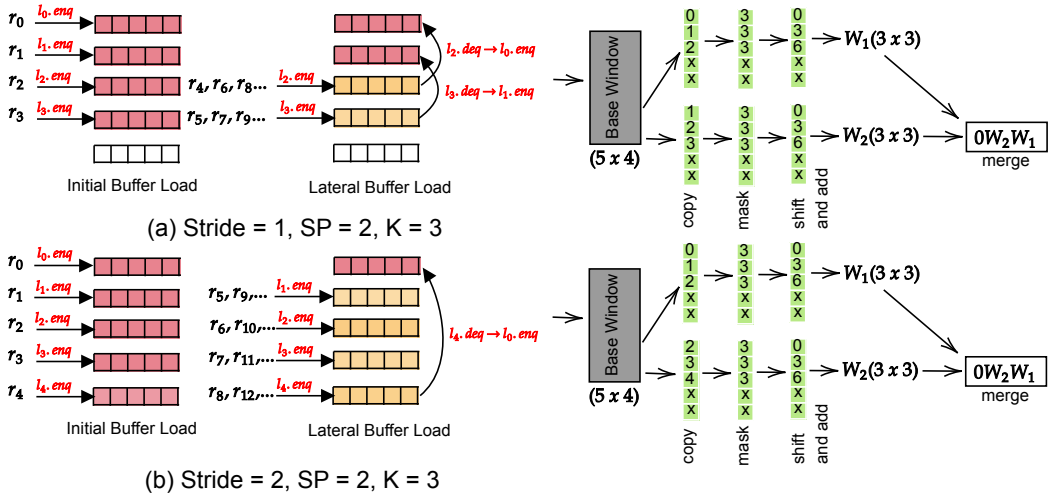


Figure 5.7: The line buffer is operating in two configurations generating $3x3$ windows with different stride values. The connection of the input to the FIFOs and connections between the FIFOs change with the stride factor.

5.4.1 Programmable Line Buffer

Exploiting kernel level parallelism is essential towards increasing the data-reuse per input fetched from the external memory. It is non-trivial to exploit full kernel level parallelism, amounting from a single K^2 sized window, in overlay architectures due to the variation in kernel sizes across different CNN layers.

Previous approaches provide line buffers that are either limited to a single window per cycle [20], [29], or support a specific window and/or image size. The work in [86], proposed a scalable line buffer design that can generate multiple windows of arbitrary size. The design was purely register based and had no support for windows moving with a stride greater than 1. We propose a novel programmable line buffer architecture for our accelerator, that can generate multiple variable sized windows over a streaming input source moving with stride values greater than 1. Our novel hybrid design optimally uses both BRAM and registers to improve throughput as shown in Figure 5.6. The generation of square windows by the line buffer facilitates full exploitation of kernel level parallelism.

5.4.1.1 Buffer Architecture

Every cycle, the line buffer generates SP convolutional windows from the IFMAP each of dimension $K \times K$. The input feature map is streamed to the line buffer in a row-major fashion. The SP windows are generated along the vertical direction over the IFMAP. We extract the surface windows in only one direction as it simplifies the flexible line buffer design. This also relaxes constraint (5.4) from section 4.2. In a steady-state, our line buffer reads in $S \times SP$ new inputs along $S \times SP$ rows of the IFMAP. The rows are buffered using an array of FIFOs. We use a hybrid FIFO implementation. In each FIFO, the first B values are stored in registers and rest of the row is stored in a BRAM block. Each FIFO is read from and written to independently. The connections between the FIFOs is programmable through a flexible interconnection network. With L FIFOs, $l_0 \cdots l_{L-1}$, the line buffer can hold a maximum of L IFMAP rows at any time.

The line buffer needs to buffer a total of Z rows and K columns ($Z = K + S \times (SP - 1)$) to generate SP, vertical surface windows, moving with stride S. There is an overlap of $K - S$ rows between two vertically adjacent windows. The IFMAP is loaded in two phases. In the initial loading phase, the rows are streamed sequentially until the first Z FIFOs of the line buffer are filled. After this, the lateral loading phase starts, wherein the remaining rows are streamed. In this phase, every cycle, $S \times SP$ values (one value from a different but adjacent row) are fed to the FIFOs l_{K-S} though l_Z. The FIFOs support parallel read/writes. With every en-queue operation, the FIFO is also de-queued. As the new data is en-queued to FIFO L_i, the old data from L_i is en-queued to FIFO $L_{i-(S \times SP)}$. This pattern ensures that the overlapping rows between the vertically adjacent convolutional windows are preserved inside the line buffer. These connections vary with the value of S and SP, and are enabled by setting the control words configuring the programmable all-to-all network between the FIFOs. For example, in part (a) of Figure 5.7, the new data is en-queued to FIFOs 2 and 3. The old data from FIFO 2 and 3 is

en-queued to FIFO 0 and 1, respectively. This replaces rows 0 and 1 in the line buffer with rows 2 and 3.

In the lateral loading phase, with data in Z FIFOs, the line buffer outputs a $Z \times B$ window every cycle. This is done by shifting out a column and shifting in a new column from across the Z FIFOs. A total of B values can be read per FIFO because of the register storage. From the $Z \times B$ base window, the sub-windows required to perform the convolutions are extracted. A set of copy registers, selects the rows required per window. For example, in part (a) of Figure 5.7, since two, $3x3$ convolution with stride 1 needs to be executed, the first two copy registers are used. The first one selects rows 0, 1 and 2 from the base window and the second selected rows 2, 3 and 4. The selected rows are copied into a vector register which is then passed through a dedicated masking register, that zeros out the extra columns. The masked out rows within the vector register are appropriately shifted and later added to construct a $1 \times K^2$ dimensional vector W_l. We call W_l a *window vector* to mean that it is logically a window but is physically stored in a vector register. All the input vectors required to perform SP convolutions are aggregated/merged into a single vector $W = [W_{SP}, \cdots, W_1]$ with $SP \times K^2$ elements in it. The vector thus constructed is passed to the processing engine.

The copy/mask/shift values and the FIFO interconnections represent the dynamic configuration of the line buffer. Knowing the kernel size and the SP value for a given computation batch, the host logic generates these values. It passes them to the overlay as control words during the configuration phase.

5.4.2 Processing Engine Architecture

The processing engine (PE) forms the core of our overlay, where the convolutions are processed. The PE receives SP windows from the line buffer which are convolved in parallel with FP pre-fetched filter windows of the same dimension. Depending on the computation batch, the PE can be configured to process any mix of FP and SP values. Recall, that the windows emanating out of the line buffer are aggregated into a vector $W = [W_{SP}, \cdots, W_1]$ and forwarded as input to the PE.

The PE is a fully pipelined structure. It consists of a distribution tree, a multiplier array, and a reduction tree. The vector W first enters the distribution tree, which creates FP replicas of the same and distributes them over the multiplier array. The size of the multiplier array characterizes the size of the PE. The PE design is entirely scalable depending on the DSP resources of the underlying FPGA. A multiply unit multiplies an IFMAP value with a filter weight. Unlike the IFMAP values, the filter weights are fed directly to the multipliers, bypassing the distribution tree. Using shift logic, the PE loads filter weights from external memory into a register array. Each register in this array is connected to an individual multiplier. The PE starts operating after loading the first FP filters. As the PE convolves the IFMAPs with the current filter set, it pre-fetches the next set in the same fashion, ensuring a compute-to-memory overlap. With the arrival of the IFMAP values, the multiplications are performed. The products are passed down to an additive binary reduction tree, where they are combined to produce the convolution output.

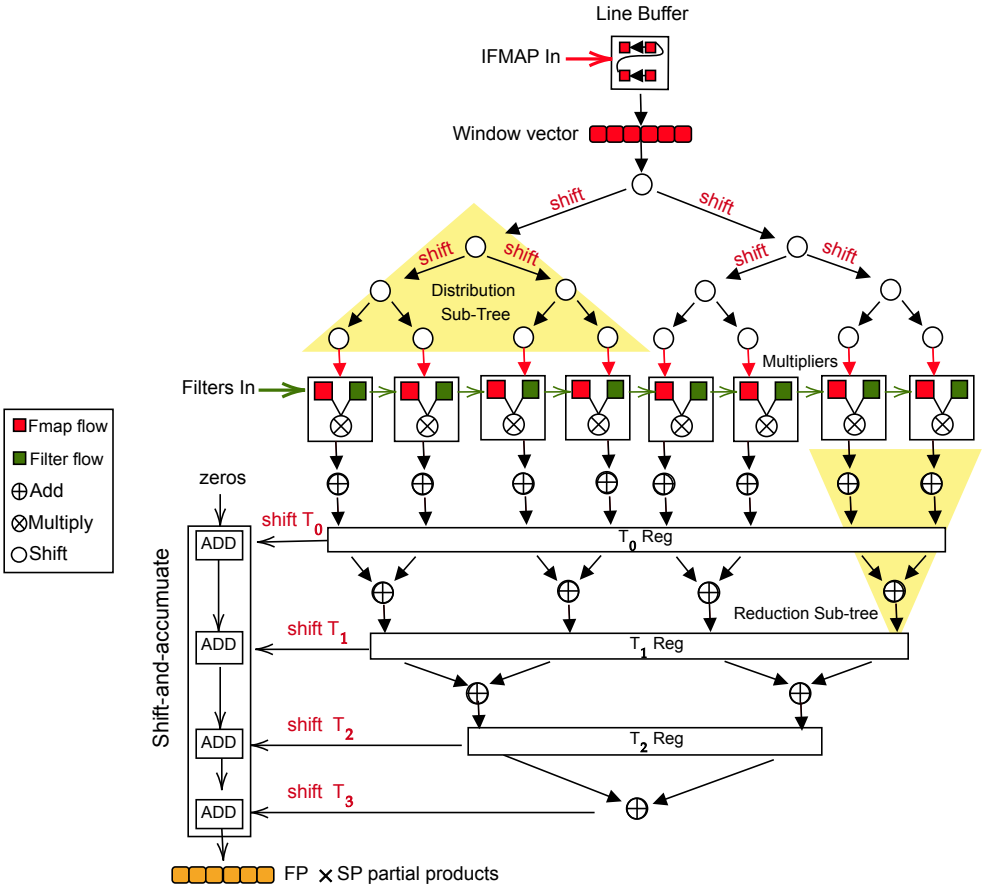


Figure 5.8: The micro-architecture of the Processing engine. An example is shown on the right of how the distribution tree distributes a window vector over 8 multipliers.

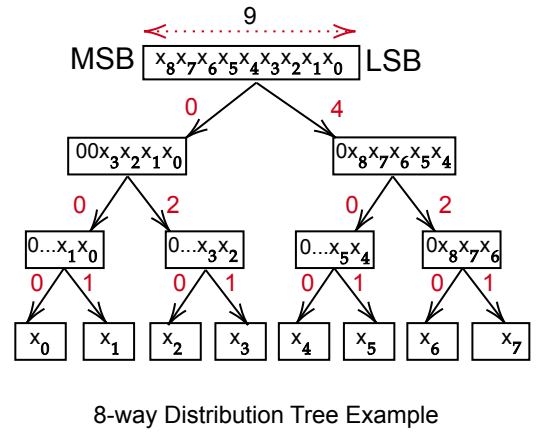


Figure 5.9: The micro-architecture of the Processing engine. An example is shown on the right of how the distribution tree distributes a window vector over 8 multipliers.

A binary reduction tree can only reduce window vectors whose size is a power of 2. But such window sizes rarely occur in CNNs. The problem can be addressed by zero padding the window vectors. This approach leads to sub-optimal utilization of PE compute resources. In our case, we achieve optimal PE throughput by orchestrating the distribution of the products (multiplier outputs), corresponding to the windows vectors, over a complete binary tree in a clever fashion without any padding, thus maximizing the compute utilization. We explain this data orchestration next.

Data Orchestration:- The aggregate vector W is replicated FP times to facilitate convolution with FP filters in parallel. This is logically equivalent to replicating each window vector W_i, $1 \leq i \leq SP$, into FP copies, W_i^j, $1 \leq j \leq FP$. Thus, overall, a total of $FP \times SP$ window vectors are created. Each window vector is further broken into smaller vectors whose size is a power of 2. For example, a 3x3 vector is broken into two smaller vectors of size 8 and 1 each. In general, a K^2 sized window is broken into at most w partitions, P_0 through P_w, where $w = \lfloor \log_2(K^2) \rfloor - 1$. The partition $P_{i,j}^m$ is related to the j^{th} copy of the i^{th} window vector and is of size m. This partition constitutes a window vector if the $\log m^{th}$ bit in the binary representation of the vector size is set. Note that, there will be a total of $FP \times SP$ partitions of a given size. In the proposed orchestration strategy, the same-sized partitions from all the windows are packed adjacent to each other. The partition groups are arranged in the decreasing order of their sizes from left to right. The filter windows are pre-arranged by the host in the same data-layout mentioned here and fed to the multipliers directly during runtime. When $SP > 1$, SP copies of a partition from the filter window is made using a Shift-and- Or logic and propagated internally between the multipliers. Once the copies are propagated, the next partition belonging to the same filter window is loaded from the host. A partition P^m of size m is mapped to m multipliers. The multiplication of the filter weights and IFMAP values corresponding to this partition creates a partial product vector of the same size. Recall that the multiplier array forms the base of a complete binary tree, used for reduction. This partial product vector is reduced to a single value using an appropriate subtree of the reduction tree. Using this computational layout, there is no necessity for zero padding while computing convolutions, thus optimal DSP utilization is achieved. For example, in a 3x3 convolution window, the partitions of size 8 and 1 are reduced by subtrees, that are complete binary trees, with 15 and 1 nodes, respectively. Since each multiplier is connected to a leaf node of the reduction tree, the subtree mapping is done automatically based on where the partition lies within the layout.

5.4.2.1 Distribution Tree

The above data orchestration is realized in hardware, inside the PE, with a distribution tree. The structure is organized as a complete binary tree. The number of leaves is greater than or equal to the number of multipliers. The IFMAP values inside the aggregate vector W, are routed through the distribution tree nodes to the multipliers through a sequence of shift operations. Every tree node has two shift units and is configured with a specific shift value. Each node performs a copy-and-shift operation wherein it makes two copies of the received vector, right shifts each copy by its corresponding shift amount, and forwards the modified vectors across the edges to both the children nodes. Note that a shift

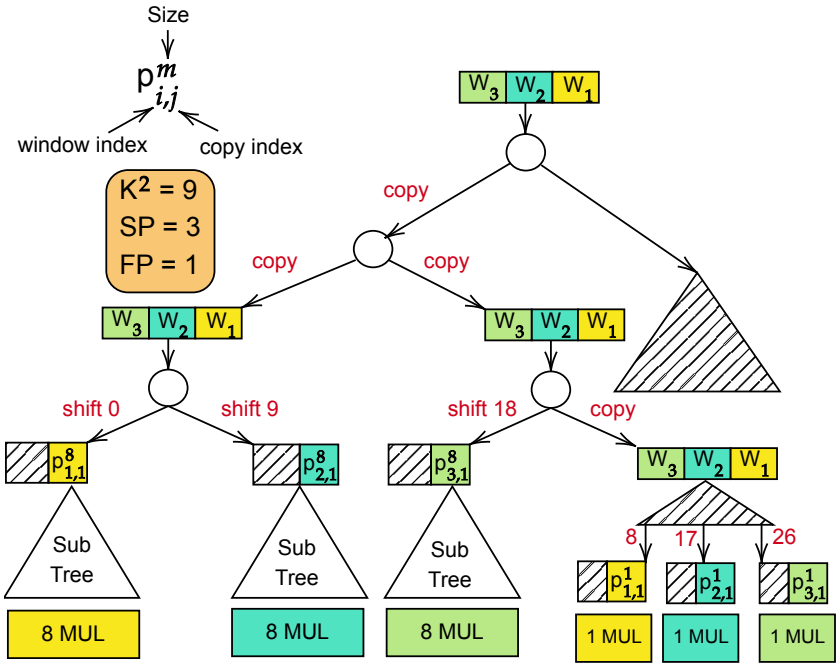


Figure 5.10: The distribution tree configuration for processing 3 convolutions with $SP = 3$, $K^2 = 9$, and $FP = 1$.

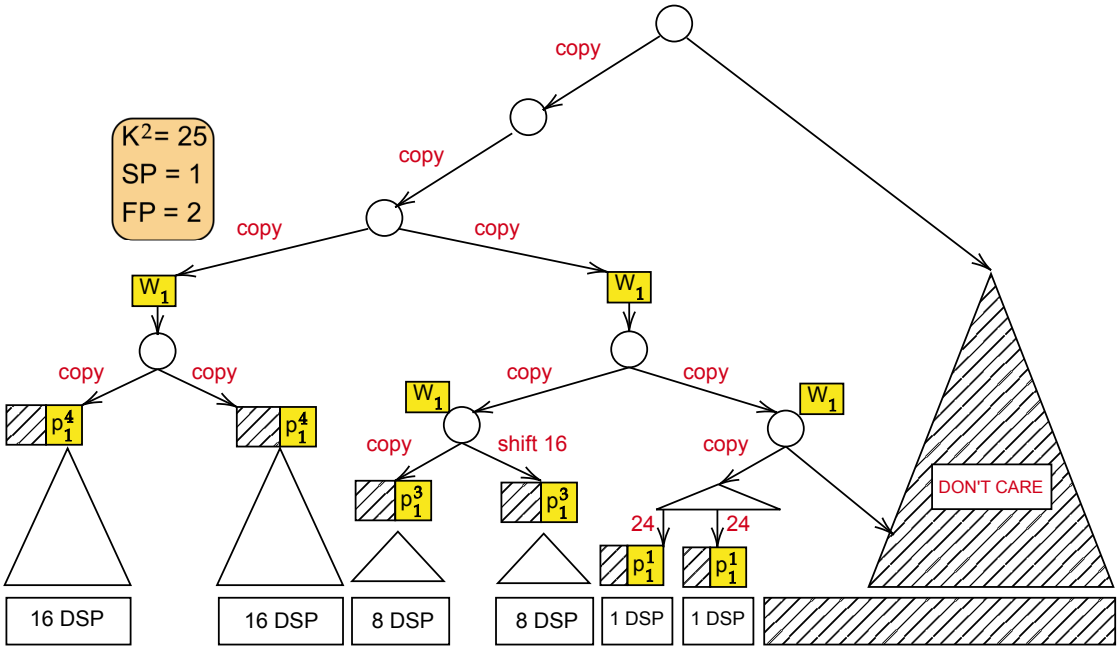


Figure 5.11: The distribution tree configuration for processing one convolution with $SP = 1$, $K^2 = 25$, and $FP = 2$.

value of 0 on an edge results in copying the input vector without shift. This is the default configuration of the distribution tree. Overall, there can be a maximum of H vector transformations on a root to a leaf path. Here H is the height of the distribution tree. Finally, each multiplier reads the first entry of the leaf-node vectors as input.

We further explain the details of the distribution tree logic using a running example, see Figure 5.10, where $K = 3$, $SP = 3$, and $FP = 1$. The aggregate vector contains 27 IFMAP values distributed across three window vectors W_1, W_2 and W_3, each of size 9. Each window vector is broken into 2 partitions of size 8 and 1. The data-orchestration rearranges the window vector partitions, $W = P^1_{3,1}P^8_{3,1}P^1_{2,1}P^8_{2,1}P^1_{1,1}P^8_{1,1}$, and puts them in the decreasing order of their sizes ($P^8_{1,1}$ occupies the least significant bits of W) resulting in a transformed aggregate vector $W^{'}$, where $W^{'} = P^1_{3,1}P^1_{2,1}P^1_{1,1}P^8_{3,1}P^8_{2,1}P^8_{1,1}$. Each partition in W', from right to left, is mapped to a group of multipliers/DSP. For example, the partition $P^8_{1,1}$ is mapped to the first 8 multipliers and so on. This mapping/data-orchestration is achieved by using sub trees from the distribution tree. A group of m multipliers is assigned a subtree, with its root located at level $H - \log m$ within the distribution tree of height H. The root-node of the subtree receives a vector with partition $P^m_{i,j}$ at the lowest significant position, from the tree nodes above. The subtree is configured to route the elements/values of the partition to the appropriate multipliers at the leaf level. Figure 5.11, another configuration where the same distribution tree is used to process a 2, 5x5 convolutions.

Figure 5.12, shows a subtree distributing a 8 sized vector partition over 8 multipliers. At every node, the input vector is copied across the left edge and right-shifted across the right edge. The shift amounts are adjusted such that desired value appears at index 0 of the vector in the leaf node. The aggregate vector W is routed to the parent node of the subtree through the default copy operation. It is right shifted, across the edge connecting the subtree, such that the partition mapped to the subtree appears at the lowest significant position of the resultant vector. In Figure 5.10 part (a), the partition $P^8_{2,1}$ is mapped to the second subtree from the left. This partition appears after the partitions $P^8_{1,1}$ and $P^1_{1,1}$ inside W at the parent node of the subtree. Thus, W is right-shifted by $|P^8_{1,1}| + |P^1_{1,1}| = 9$ units so that $P^8_{2,1}$ appears at the lowermost 8 positions of the resultant vector at the root of the subtree. Figure 5.13, show the multiplier configuration for the convolution defined by the distribution tree in Figure 5.11.

Setting up the shift values corresponding to the subtrees and their parent nodes completes the configuration of the distribution tree. The other nodes are left in the default copy propagation mode. Using the K, SP, and FP values, the host sets up the shift values using control words, thereby configuring the distribution tree for a new compute batch.

5.4.2.2 Reduction Tree

The partitions corresponding to a window are reduced to single values inside a binary reduction tree. The reduced value $r^m_{i,j}$ corresponding to a partition $P^m_{i,j}$ is generated at the $\log m^{th}$ level. For example in Figure 5.12, the computations from the 8 sized partitions $P^8_{1,1}$,$P^8_{2,1}$,$P^8_{3,1}$ are reduced to the single values $r^8_{1,1}$,$r^8_{2,1}$,$r^8_{3,1}$ at the third ($\log 8 = 3$) level of the reduction tree. All the reduced values corresponding

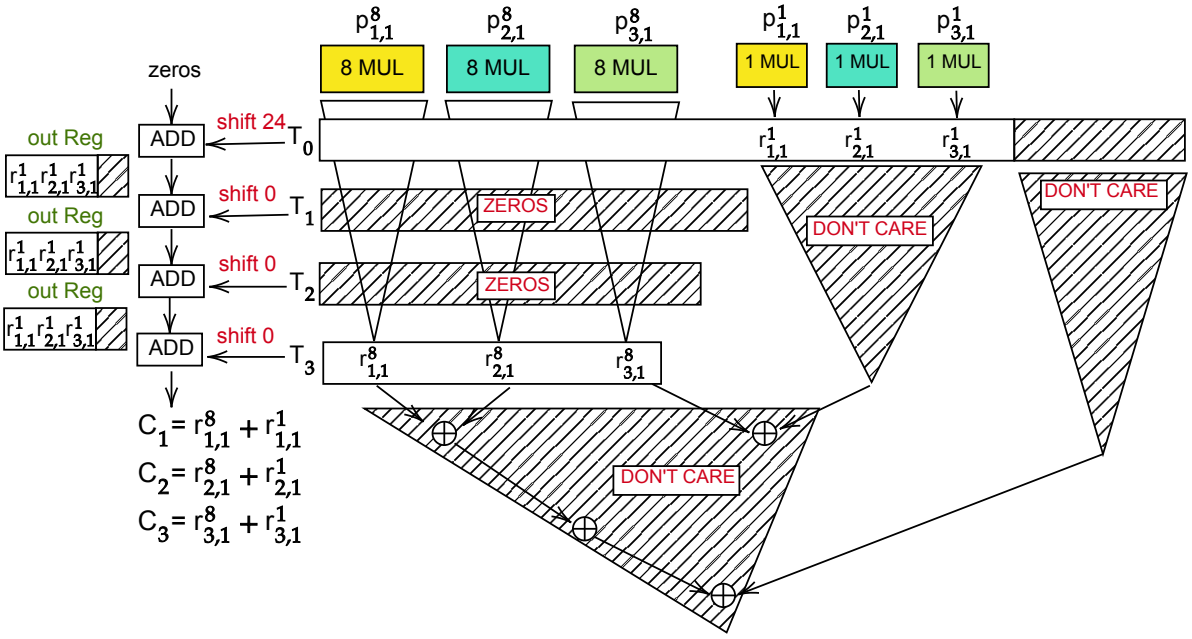


Figure 5.12: The reduction tree configuration for processing 3 convolutions with $SP = 3$, $K^2 = 9$, and $FP = 1$. The copy index is 1 throughout since $FP = 1$.

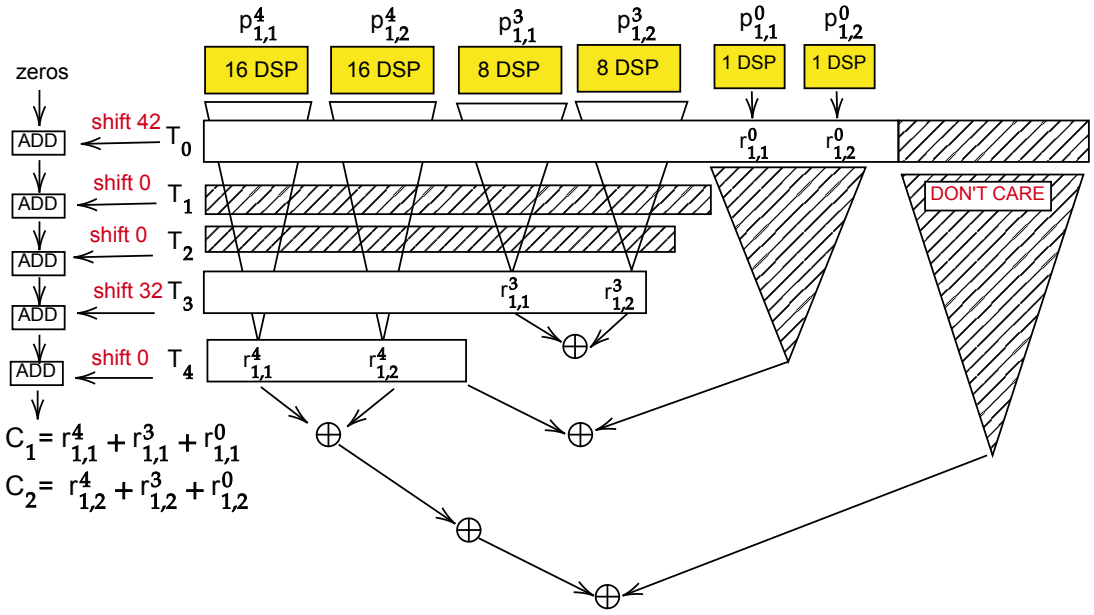


Figure 5.13: The reduction tree configuration for processing 1 convolutions with $SP = 1$, $K^2 = 25$, and $FP = 2$.

to a window vector must be aggregated to produce the final convolution output. This cannot be done inside the reduction tree since the reduced values lie at different levels, and no adder path exists between them. The reduced values from different levels are latched onto registers and passed through a shift and accumulate hardware that performs the final aggregation.

5.4.2.3 Shift and Accumulate

The per-partition reduced values at various levels of the reduction tree are latched on vector registers. The size of these vectors is set to the maximum number of reduced values produced at any given level. These vectors are pairwise added, using a sequence of shift-and-accumulate units, to create the final convolution outputs.

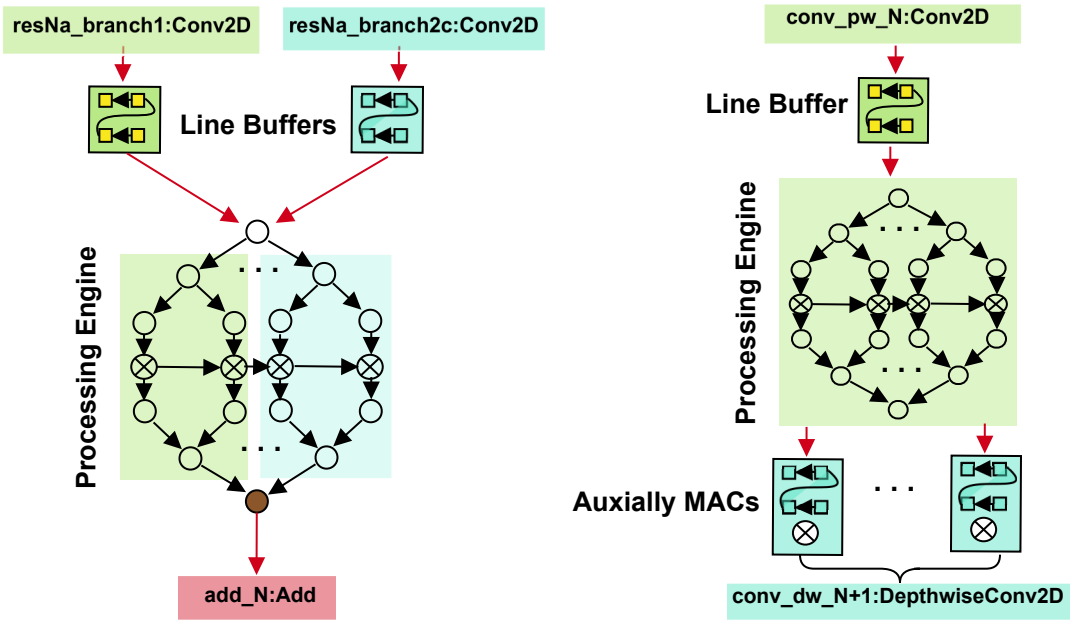
The shift-and-accumulate units are connected in a pipeline. A unit i adds the output of the previous unit with the vector T_i. Here T_i stores the reduced values from the i^{th} level of the reduction tree. The location of the reduced values inside the vector depends on the tree level. For example, in Figure 5.12, the reduced values in the first level vector T_0 lie at positions 24, 25, and 26 respectively. These values need to be aligned before the pairwise addition operation with the vector T_3, containing the reduced values corresponding to the 8 sized partitions at positions 0, 1, and 2, respectively. Consequently, T_0 is right-shifted by 24 units. The shift-and-accumulate sequence adds both T_0 and T_3 to produce the final convolution outputs. For vectors that do not contain any reduced values (T_1 and T_2 in the example) the corresponding shift-and-accumulate, performs a zero addition, thereby forwarding the input unmodified to the next unit. The adder units are connected in a pipeline through registers. In general, for the pairwise addition to happen correctly, with respect to two vectors T_i and T_j, $j > i$, T_i needs to be right-shifted by $p(i) - p(j)$ units. Here $p(i)$ and $p(j)$ are the starting location of the reduced values within T_i and T_j, respectively.

5.4.2.4 PE Configuration

The shift values of the distribution tree and the shift-and-accumulate stages represent the dynamic configuration of the PE. The host writes these values to control words dedicated to the PE inside the control memory of the overlay. Therefore, configuring the PE at runtime for processing a compute batch with a specific value of FP, SP, and K.

5.5 Handling Special Convolutions

We optimize the processing of different types of convolutions by altering the basic processing flow of our accelerator.



(a) Simulaneous Execution in ResNet **(b) Zig-Zag Execution in Mobile-Net**

Figure 5.14: The overlay is optimized to process depthwise convolutions in MobileNet and elementwise addition layer in ResNet.

5.5.1 Pointwise Convolution Layer

Point-wise convolutions occur in MobileNet, ResNet, YOLO. Since the kernel size is 1, every convolution results in a single MAC operation. To achieve a high compute utilization for such layers, the FP/SP value has to be increased. Recall that the output vector generated by the PE has $FP \times SP$ elements, and each value is written in parallel to the PE memory. Therefore, an increment in either FP or SP increases the compute utilization and the required memory bandwidth of the PE linearly. This is different for layers with $K > 1$, where a single unit increment in FP/SP results in quadratic (K^2) improvement in the compute utilization but increases the memory bandwidth requirement linearly. To achieve a similar utilization pattern, we exploit depth parallelism, CP, instead of surface parallelism, for point-wise layers. With every unit increase in FP, the compute utilization increases by a factor of CP and the bandwidth requirement increases by 1 (the values from CP channels adds up to a single output).

We exploit channel parallelism by feeding the PE with an input vector of size CP. The vector contains data from CP adjacent IFMAPs of the input volume. Every cycle, the multiplier array inside the PE produces $FP \times CP$ values. The CP values produced per convolution are added inside the merge network of the PE to generate only one value that needs to be written to the PE output memory. Therefore, the overlay increases the computations by a factor of CP with every increase in FP.

5.5.2 Depthwise Convolution Layer

Depthwise convolutions are found in sparse networks like MobileNet. In such a layer, each depth of the input volume is convolved with a separate filter. Refer to Figure 5.14.

In such a scenario, the FP value is upper bounded by 1, as this is the maximum number of filters that can be applied over a single surface, severely impacting the data reuse factor and bringing down the achievable throughput. Our overlay overcomes this limitation by executing parallel filter convolutions over different IFMAPs of the input volume. But the basic processing flow of our overlay is geared towards processing one IFMAP at a time. To achieve channel parallelism, we exploit the fact that, in general, a depthwise layer occurs after a regular convolution layer. Our overlay executes both the layers in a pipelined fashion.

In this flow, the overlay retains the output surfaces rendered after the execution of FP filters from the convolution layer, instead of flushing them to DRAM. The retained surfaces are passed through a set of auxiliary MAC units. Each such unit is made up of a small line buffer and a multiplier array. The filters from the next depthwise layer are applied over these surfaces, inside the auxiliary MAC units, whose output is later flushed to the DRAM. After this, the accelerator resumes the execution of the convolution layer with the leftover filter batches, followed by the depthwise layer, continuing the zig-zag pattern of execution.

5.5.3 Elementwise Addition Layer

The Elementwise layer occurs in networks like ResNet, Inception, etc. In such a layer, the outputs from two or more previous layers are combined using simple elementwise operations like addition. These layers are mostly memory bound since computationally they are simple but consumes at-least twice the input compared to a normal convolutional layer. Also, in most settings, the participating convolution layers are similar in structure concerning their kernel sizes. This is true for ResNet, which has a branching factor of 2 and the elementwise addition layer receives input from two pointwise convolution layers, each at a different branch. To increase the overall throughput and reduce external memory traffic, both the input convolution layers are executed simultaneously over the PE. Their outputs are later added inside the reduction tree. The simultaneous processing is enabled in our overlay with an extra line buffer for streaming input to the other convolution branch. The parallelism factors, FP, SP, and CP, are equally distributed between the branches. Refer to Figure 5.14.

5.6 Experimental Evaluation

We evaluate our overlay by synthesizing it for two Xilinx FPGAs, Virtex7-690t and Ultrascale+ VU9P. We use 16-bit fixed-point precision for the accelerator. We employ the synthesized hardware to process five networks namely, AlexNet [49], VGG16 [84], YOLO [78] MobileNet [39] and ResNet-50 [33]. We have reported throughput (in GOps/Sec) for two cases: with and without fully connected

layers. When considering FC layers for overall throughput calculation, VGG-16 and AlexNet undergo a reduction in throughput (only these two networks have FC layers amongst other networks mentioned above). This happens because of the overlay's restriction to process one batch (B = 1).

5.6.1 Experimental set-up:

Our overlay is synthesized at 166 MHz and operates with a data bit rate of 16 bytes/cycle in the read and write direction. The combined bit rate of our overlay is 32 bytes/cycle. Therefore the overlay operates at a bandwidth of 10.6 GB/s (5.3 GB/s in each direction). We run our overlay on two FPGAs, a Virtex7-690t and an Ultrascale+VU9P FPGA that vary with respect to the number of resources and the operational framework.

The Virtex7-690t FPGA is a standalone FPGA with 3600 DSP blocks and 6.4 MB of on-chip BRAM. It is connected to an Intel Core-i5 processor through a PCIe-8x link. The code running on the CPU streams input using PCIe to our overlay using the Xillybus PCIe core (http://xillybus.com/doc/revision-b-xl). The ideal data bandwidth of the core operating on the Virtex-7 device with 8x Gen3 lanes, utilizing the Gen3 Integrated Block for PCI Express v3.0, can be expected to reach 6.4 GB/s each direction (read from host and write to host), see http://xillybus.com/doc/xillybus-bandwidth.

The Ultrascale+ VU9P FPGA is present on the Amazon Web Services (AWS) EC2 F1 instance and has 6840 DSP blocks and 75.9 MB of BRAM. The overlay is wrapped in the standard AWS F1 Verilog wrapper that interacts with the AWS F1 shell to retrieve data from DDR4 memory. The host code uses OpenCL to send data to the FPGA DDR memory and from there it is read over the memory interface which provides a combined bandwidth of 16 GB/s.

We use Bluespec System Verilog (BSV) [70] to design our hardware. All the reported hardware characteristics of the design are obtained Post Place and Route (PPnR) in the Vivado Design Suite.

We compare the performance of our accelerator with other recent works. As listed in Table 5.3, the accelerator frameworks chosen for comparison come under two categories. In the **Specific** category, the accelerators are tailor-made for processing a specific network or a class of networks. In the **Generic** category, the accelerators are designed as overlays and are network agnostic.

5.7 Experiments

We evaluate our overlay by synthesizing it for two Xilinx FPGAs, Virtex7-690t and Ultrascale+ VU9P. We use 16-bit fixed-point precision for the accelerator. We employ the synthesized hardware to process five networks namely, AlexNet [49], VGG16 [84], YOLO [78] MobileNet [39] and ResNet-50 [33]. We have reported throughput (in GOps/Sec) for two cases: with and without fully connected layers. When considering FC layers for overall throughput calculation, VGG-16 and AlexNet undergo

Network	Reference	#Parameters	Description
AlexNet	[49]	60.97 M	A relatively small CNN with varying kernel sizes.
VGG-16	[84]	138.36 M	A uniform CNN with fixed kernel size and a large parameter space.
MobileNet-V1	[39]	4.2 M	A sparse CNN with point-wise and depth-wise convolution layers.
YOLO-V2	[78]	271.7 M	A dense CNN with alternating point-wise and canonical convolution layers.
ResNet-50	[33]	25.56 M	A residual CNN with point-wise and element-wise addition layer.

Table 5.1: Networks processed by our overlay.

	% KLUTs					Total KLUTs	Kilo FF	BRAM (36 Kb)	DSP
	Distribution Tree	Reduction Tee	Input Logic	Output Logic	Misc				
Virtex7-690t	29.5%	40.1%	5.2%	10.1%	14.8%	223/693	405/866	911/1470	3072/3600
Ultrascale+ VU9P	30%	40.3%	4.8%	10.5%	14.2%	448/1182	1112/2364	1836/2210	4096/6840

Table 5.2: FPGA resource consumption of our accelerator on both the FPGAs. The percentage utilization of the FPGA LUTs, out of the total LUTs consumed, across different hardware modules of our overlay is shown.

a reduction in throughput (only these two networks have FC layers amongst other networks mentioned above). This happens because of the overlay's restriction to process one batch (B = 1).

Experimental set-up: Our overlay is synthesized at 166 MHz and operates with a data bit rate of 16 bytes/cycle in the read and write direction. The combined bit rate of our overlay is 32 bytes/cycle. Therefore the overlay operates at a bandwidth of 10.6 GB/s (5.3 GB/s in each direction). We run our overlay on two FPGAs, a Virtex7-690t and an Ultrascale+VU9P FPGA that vary with respect to the number of resources and the operational framework.

The Virtex7-690t FPGA is a standalone FPGA with 3600 DSP blocks and 6.4 MB of on-chip BRAM. It is connected to an Intel Core-i5 processor through a PCIe-8x link. The code running on the CPU streams input using PCIe to our overlay using the Xillybus PCIe core (http://xillybus.com/doc/revision-b-xl). The ideal data bandwidth of the core operating on the Virtex-7 device with 8x Gen3 lanes, utilizing the Gen3 Integrated Block for PCI Express v3.0, can be expected to reach 6.4 GB/s each direction (read from host and write to host), see http://xillybus.com/doc/xillybus-bandwidth.

The Ultrascale+ VU9P FPGA is present on the Amazon Web Services (AWS) EC2 F1 instance and has 6840 DSP blocks and 75.9 MB of BRAM. The overlay is wrapped in the standard AWS F1 Verilog wrapper that interacts with the AWS F1 shell to retrieve data from DDR4 memory. The host code uses

OpenCL to send data to the FPGA DDR memory and from there it is read over the memory interface which provides a combined bandwidth of 16 GB/s.

We use Bluespec System Verilog (BSV) [70] to design our hardware. All the reported hardware characteristics of the design are obtained Post Place and Route (PPnR) in the Vivado Design Suite.

5.7.1 Performance Comparison with State-of-the-Art

We compare the performance of our accelerator with other recent works. As listed in Table 5.3, the accelerator frameworks chosen for comparison come under two categories. In the **Specific** category, the accelerators are tailor-made for processing a specific network or a class of networks. In the **Generic** category, the accelerators are designed as overlays and are network agnostic.

As can be seen in Tables 5.4 to 5.8, our overlay outperforms most of the frameworks across both the categories. With respect to the throughput metric in GOps/Sec, we outperform generic/overlay accelerators by a factor of approximately 1.2x to 5x on both the FPGAs. The highest speed up is observed for AlexNet and VGG-16 since these are regular CNNs with canonical convolution layers. FPGA 2020 [99] achieves higher throughput, approximately 1.2x higher. This can be explained by the fact that it uses a low precision (8-bit) floating-point (LPFP) quantization method to quantize both weights and activations. Our overlay, on the other hand, uses a 16-bit fixed-point representation. We perform at-par compared to the work in [93], which uses inter-layer parallelism on an FPGA cluster to do the processing. Our overlay operates at a slightly reduced throughput for networks with special convolutions, i.e., ResNet, YOLO, and MobileNet. We outperform the RTL-based overlay processor (OPU) in [101], which utilizes a fine-grained instruction control mechanism to process the individual layers using different parallelism mixes.

OPU is an 8-bit accelerator. All networks are quantized to 8-bit precision for both kernel weights and feature maps. We can perform two 8-bit MAC operations using a single DSP against a single 16-bit MAC operation. The effective memory bandwidth also increases by two times, keeping the DSPs busy in computations. Further, the capacity of other resources such as BRAMs, flip-flops, etc., effectively doubles, and the placement/routing complexity reduces non-linearly. All these factors contribute to higher throughput and a GOPs/DSP ratio. As can be seen in Table 5 and Table 6, the GOPs/KLUT of OPU (refer to the Table 3 of the OPU paper) is 4.1 and 3.8, which is comparable to our 16-bit design. This highlights that our accelerator consumes less FPGA resources, which can be attributed to the simple PE design and the control logic. OPU is more of an instruction set based accelerator, while we use a simple set of control words that are driven by the host to process the computations in different configurations. The Light-OPU [102] paper is a variant of the original OPU paper that handles only light convolutional networks like MobileNet. In our case, we can handle both dense nets like VGG-16 and light CNNs like MobileNet using the same micro-architecture. LightOPU on MobileNet has a GOPS/DSP value of 0.14 while we achieve a value of 0.27 (refer to Table 7).

Overlay designs warrant high FPGA resources. Performance density with respect to the area of an accelerator is measured in GOps per KLUTs. Our accelerator achieves better performance density by

almost a factor of 1.3x to 4x. A similar trend can be seen for performance density values with respect to the DSP resources. This shows that compared to other overlay designs, our design achieves higher throughput at lower resource consumption. Frameworks, like [99], employing 8-bit inferencing has higher performance density values because of the smaller data type. Compared to the Virtex7 FPGA, the performance density of our hardware is less on the VU9P board. This is due to the inefficient PCIe bandwidth utilization using the Xillybus protocol leading to the under-utilization of DSP blocks.

In the network-specific category, our overlay performs at par or better, compared to the other accelerators. On the VU9P FPGA, we slightly outperform the TGPA heterogeneous accelerator [97], running with batch size four, both in the throughput and performance density metric. We achieve approximately 1.4x higher throughput compared to the work in [27], which proposed a pipelined heterogeneous accelerator with different layers of a CNN mapped to different chip units within the FPGA. The work in FPL 2018 [107] outperforms our overlay for VGG-16 as it uses an optimized model for the VGG-16 network along with different hardware architectures for convolution layer and FC layer. In the case of ResNet-50, our overlay has comparable performance with others.

Amongst all the CNNs, our accelerator reports the lowest throughput for MobileNet on both the FPGAs. This is because of the presence of the depth-separable convolution layers. Although we optimize the processing of MobileNet using the Zig-Zag processing flow, the alteration of the pointwise and depth-wise layers lowers the overall throughput. All the accelerators processing MobileNet, considered in our comparison, fall under the network-specific category. These employ techniques to optimize the sparsity factor of MobileNet, especially the depth-separable layers. Our overlay achieves a throughput of 830 and 948 GOps/Sec on Virtex7 and VU9P FPGA, respectively. As can be seen from table 5.8, our overlay operates at par with the 16-bit accelerators but is outperformed by the 8-bit accelerators. When comparing the results of both FPGAs, the number of multipliers increases by 1.3x, resulting in 1.46x and 1.53x performance increase for VGG-16 and YOLO, respectively. Recall from Section 5, that at the end of processing a batch, all the computational pipelines will be flushed, and at the beginning, the pipelines will again be filled before a steady state is achieved. These overheads will be reduced as the number of batches decrease. Hence more speed up compared to the increase in the multipliers is observed.

The results presented in this section are for a 16-bit version of our accelerator. Our hardware design is parameterized to handle different fixed-point precision. For an 8-bit version, from a theoretical perspective, our accelerator reports 1.5x higher throughput than the 16-bit version. For example, for AlexNet, the theoretical peak throughput on the Virtex7 FPGA is 1853 GOPs.

5.7.2 Architecture Analysis

In this section, we do an architectural analysis of our overlay. First, we observe how our overlay reacts to changing External Memory Bandwidths (EMBs). Figure 5.15 through Figure 5.18, plots the percentage DSP utilization for each convolutional layer of AlexNet, VGG-16, YOLO, and MobileNet CNNs for different EMBs. We calculate the dynamic DSP utilization of the overlay. It is calculated by

	FPGA 2020 [99]	FGPA 2020 [102]	ICET 2020 [106]	VLSI 2020 [101]	IEEE Trans 2020 [93]	FCCM 2019 [57]	FPL 2019 [100]	VLSI 2019 [68]	VLSI 2019 [54]	IPSJ 2019 [85]	FPL 2018 [107]	FPGA 2018 [67]	VLSI 2018 [59]	IEEE Trans 2018 [9]	IEEE Trans 2018 [89]	ICCAD 2018 [97]
Network Specific		✓	✓				✓	✓		✓	✓	✓		✓	✓	✓
Generic/ Overlay	✓			✓	✓	✓			✓				✓			

Table 5.3: Classification of the reference works into Network specific and Generic processing architectures.

	FPGA 2020	FCCM 2019	ICCAD 2018	IEEE Trans. 2020	IEEE Trans. 2018	IEEE Trans. 2018	**Ours**	**Ours**
Architecture	Common, B = 1	Without FC	Common, B = 4	Common, runs on 15 FPGAs	Specialized P.E. for different layers	Without FC	**Common, B = 1**	**Common, B = 1**
FPGA	XC7K325T	Zynq ZCU102	Xilinx VU9P	Xilinx XC7VX690T	Virtex 7 690t	Zynq 7045	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/Sec (GOPS)	1066.4	223.4 (C)	1432	1157 (Per FPGA)	910.2	197.4(C)	**1030 / 1200 (C)**	**1356.57 / 1581 (C)**
Precision	8b floating	16b fixed	16b fixed	single floating point	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	6.89	0.405 (C)	3.06	-	2.98	-	**4.62 / 5.38 (C)**	**3.03 / 3.53 (C)**
Performance Density GOPS/DSP	1.388	0.195 (C)	0.317	-	0.305	0.220 (C)	**0.33 / 0.39 (C)**	**0.29 / 0.34 (C)**
Frequency MHz	200	200	200	-	150	125	**166**	**166**

Table 5.4: Performance comparison of our overlay for AlexNet with other recent works. Common signifies same architecture used for convolution and fully connected layer. B is an acronym for batch size. C represents throughput for only convolutional layers.

Table 5.5: Performance comparison of our overlay for VGG-16 with other recent works. Different signifies different architecture used for convolution and fully connected layer.

	FPGA 2020	IPSJ Trans. 2019	VLSI 2020	FCCM 2019	VLSI 2019	ICCAD 2018	FPL 2018	**Ours**	**Ours**
Architecture	Common, B = 1	Without FC	Common, B = 1	Without FC	Different	Common, B = 4	Different	**Common, B = 1**	**Common, B = 1**
FPGA	XC7K325T	Arria10 GX1150	XC7K325T	Zynq ZCU102	XC7VX690T	Xilinx VU9P	Stratix-V 5SGSD8	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/Sec (GOPS)	1086.8	960(C)	354 / 397 (C)	309(C)	760.83	1510	1928	**834.6 / 1025 (C)**	**1223 / 1503 (C)**
Precision	8b floating	16b fixed	8b fixed	16b fixed	8b floating	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	7.03	3.99 (C)	3.73/4.1 (C)	0.860 (C)	-	3.06	-	**3.74 / 4.60(C)**	**2.73 / 3.35(C)**
Performance Density GOPS/DSP	1.415	1.533 (C)	0.69 / 0.77 (C)	0.270 (C)	0.741	0.369	1.1	**0.27 / 0.33(C)**	**0.26 / 0.33(C)**
Frequency MHz	200	200	200	200	200	210	200	**166**	**166**

Table 5.6: Performance comparison of our overlay for ResNet-50 with other recent works.

	FPGA 2020	FCCM 2019 Resnet	FPL 2018	VLSI 2018	VLSI 2018	**Ours**	**Ours**
FPGA	XC7K325T	Zynq ZCU102	Stratix-V 5SGSD8	Intel Stratix-V GXA7	Intel Arria 10 GX	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/Sec (GOPS)	1101.9	291.4	973.2	243.3	611.4	**902**	**1003**
Precision	8b floating	16b fixed	16b fixed	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	7.13	0.53	-	1.38	2.76	**4.04**	**2.24**
Performance Density GOPS/DSP	1.435	0.255	0.579	0.950	0.408	**0.29**	**0.22**
Frequency MHz	200	200	200	150	200	**166**	**166**

	FPGA 2020 Tiny YOLO-v2	VLSI 2020	IEEE Trans. 2018	VLSI 2020 Tiny YOLO	FPGA 2018 Lightweight YOLO-v2	VLSI 2019 Tiny YOLO-v2	VLSI 2019 Slim YOLO-v2	**Ours**	**Ours**
FPGA	XC7K325T	XC7K325T	XC7Z020	XC7K325T	Ultrascale +	Virtex-707	Virtex-707	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/Sec (GOPS)	1095.4	391	62.9	366	610.9	464.7	1877	**1075**	**1649**
Precision	8b floating	8b fixed	8b fixed	8b fixed	(1-32,1-32) fixed	(1,6) fixed	(1,6) fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	7.08	4.1	2.11	3.8	4.52	5.40	12.11	**4.82**	**3.68**
Performance Density GOPS/DSP	1.426	0.758	0.331	0.709	1.620	2.766	6.901	**0.35**	**0.36**
Frequency MHz	200	200	214	200	300	200	200	**166**	**166**

Table 5.7: Performance comparison of our overlay for YOLO-v2 with other recent works.

	FPGA 2020	ArXiv 2020	FPL 2019	FPL 2019	FPL 2018	IEEE Trans. 2018	**Ours**	**Ours**
FPGA	XC7K325T	Stratix-10 2800	ZU2EG	Xilinx ZU9EG	Stratix-V 5SGSD8	Arria 10 SoC	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput FPS GOps/Sec	325.7 FPS -	4539 FPS -	205.3 FPS -	809.8 FPS -	- 592 GOPS	266.2 FPS 170.6 GOPS	**272.87 FPS 830 GOPS**	**275.15 FPS 948 GOPS**
Precision	8b fixed	16b fixed	8b fixed	8b fixed	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOps/KLUTs	-	-	-	-	-	2.08	**3.72**	**2.12**
Performance Density GOps/DSP	0.14	-	-	-	0.319	0.133	**0.27**	**0.21**
Frequency MHz	200	390	430	333	200	133	**166**	**166**

Table 5.8: Performance comparison of our overlay for MobileNet v2 with other recent works.

taking a weighted average across the DSP utilization of all the compute batches of a layer. Note that, our overlay employs greater number of DSP units, compared to the state-of-the-art. Thus, maximizing DSP utilization over other research works. In total, 360 DSPs are reserved for depthwise convolutions appearing in MobileNet. These DSPs will be idle for regular convolutions and contribute to 10% of the total number of DSPs used in the overlay. The DSP utilization plots provides a detailed DSP utilization plot of the 3072 DSPs, barring the 360 DSPs.

Based on the utilization plots, it can be seen that a CNN layer is either sensitive or insensitive towards bandwidth change. Bandwidth insensitive layers are those where our overlay manages to maintain a comparable DSP utilization across different EMBs. Layers `C6` from VGG-16 and layers `C4`, `C6` from YOLO CNNs are examples of such layers. The insensitivity of these layers can be attributed to the similarity in their compute schedules generated. Table 5.9 enumerates the compute schedules for a few representative CNN layers across different EMBs. Notice, the bandwidth insensitive layers highlighted in the table have similar compute schedules. They have a high degree of filter parallelism for the starting batches. For the final batch, the degree of surface parallelism varies depending on the bandwidth. For example, the schedule generated for layer `C6` of VGG, varies only in the final iteration, wherein the surface parallelism is maximized.

Our scheduling logic compensates for the bandwidth deficiency by increasing the degree of filter parallelism. But this can only be leveraged for layers with a high compute to memory overlap. As can be seen from the compute-to-overlap constraint in Equation 5.6 from Section 4, the higher degree of filter parallelism is balanced by a higher output dimension, OL^2 value, in the inequality. Therefore, bandwidth insensitive layers occur at the front of a CNN where the surface dimensions are comparatively larger. As we move towards the end layers, where the input dimensions are small, the compute to memory ratio decreases, and the layers become sensitive towards bandwidth change. This can be clearly observed for all the networks in Figure 6. Also, as can be seen from Table 5.9, the sensitive layers, for example, `C14`, have very different compute schedules for different memory bandwidths. Kernel size also affects the compute to memory ratio, thereby affecting the utilization. For example, all the pointwise layers in the YOLO CNN are bandwidth sensitive. The channel parallelism factor, see layer `C3` of YOLO CNN in Table 5.9, varies with the bandwidth in the compute schedules of the pointwise layers.

Recall from the earlier discussions that the compute schedules for our overlay are generated such that the pipeline stalls are minimized. If the average DSP utilization of the design is A operations per cycle, then the pipeline efficiency is defined as A/N^{DSP} where N^{DSP} is the number of DSPs on the FPGA. N^{DSP} operations per cycle is the theoretical maximum operations throughput achievable. The average DSP utilization A is computed by dividing the total number of operations in the CNN by the total execution cycles (compute and memory). As can be seen from Table 5.10, the overlay sustains a pipeline efficiency of approximately 80% for AlexNet, YOLO and VGG-16 CNNs. For the smaller networks, ResNet-50 and Mobile-Net, we see a dip in the efficiency, which can be attributed to the presence of depthwise sparse convolution and element wise convolution layers in these networks. With

	2 Bytes/Cycle				8 Bytes/Cycle				16 Bytes/Cycle			
	Schedule = iter x (FP, SP/CP*)				Schedule=iter x (FP, SP/CP*)				Schedule=iter x (FP, SP/CP*)			
	Alex	**VGG**	**YOLO**	**Mobile**	**Alex**	**VGG**	**YOLO**	**Mobile**	**Alex**	**VGG**	**YOLO**	**Mobile**
C1	16x(6,1)	10x(6,1) 1x(4,1)	1x(62,1) 2x(1,1)	5x(6,1) 1x(2,1)	4x(24,1)	2x(24,1) 1x(12,2) 1x(4,6)	2x(24,1) 1x(12,2) 1x(4,6)	1x(24,1) 1x(8,3)	3x(25,1) 1x(12,2) 1x(8,3) 1x(1,15)	1x(48,1) 1x(16,3)	1x(48,1) 1x(16,3)	1x(24,2) 1x(8,6)
C3	13x(28,1) 1x(20,1)	1x(128,1)	1x(128,1*)	1x(64,1)	1x(199,1) 1x(185,1)	1x(113,3) 1x(15,7)	1x(128,7*)	1x(56,6) 1x(8,7)	1x(341,1) 1x(43,6)	1x(85,4) 1x(31,11) 1x(12,15)	1x(128,15*)	1x(34,10) 1x(26,13) 1x(4,15)
C4	13x(28,1) 1x(20,1)	1x(128,1)	1x(256,1)	1x(128,1)	1x(199,1) 1x(185,1)	1x(113,3) 1x(15,7)	1x(170,2) 1x(85,4) 1x(1,7)	1x(113,3) 1x(15,7)	1x(341,1) 1x(43,6)	1x(85,4) 1x(31,11) 1x(12,15)	1x(170,2) 1x(85,4) 1x(1,15)	1x(31,11) 1x(85,4) 1x(12,15)
C5	9x(28,1) 1x(4,1)	1x(256,1)	1x(256,1*)	1x(128,1)	1x(199,1) 1x(47,3) 1x(9,6) 1x(1,7)	1x(170,2) 1x(85,4) 1x(1,7)	1x(256,7*)	1x(113,3) 1x(15,7)	1x(170,2) 1x(85,4) 1x(1,15)	1x(170,2) 1x(85,4) 1x(1,15)	1x(256,15*)	1x(31,11) 1x(85,4) 1x(12,15)
C6		1x(256,1)	1x(341,1) 1x(171,1)	2x(87,1) 1x(82,1)		1x(170,2) 1x(85,4) 1x(1,7)	1x(341,1) 1x(170,2) 1x(1,7)	1x(170,2) 1x(85,4) 1x(1,7)		1x(170,2) 1x(85,4) 1x(1,15)	1x(341,1) 1x(170,2) 1x(1,15)	1x(170,2) 1x(85,4) 1x(1,15)
C7		1x(341,1) 1x(171,1)	1x(256,1*)	2x(93,1) 1x(70,1)		1x(341,1) 1x(170,2) 1x(1,7)	1x(256,7*)	1x(170,2) 1x(85,4) 1x(1,7)		1x(341,1) 1x(170,2) 1x(1,15)	1x(192,15*) 1x(64,15*)	1x(170,2) 1x(85,4) 1x(1,7)
C13		18x(28,1) 1x(8,1)	1x(256,1*)	10x(50,1) 1x(12,1)		2x(199,1) 1x(85,2) 1x(28,4) 1x(1,7)	1x(256,7*)	1x(175,1) 1x(162,1)		1x(341,1) 1x(170,2) 1x(1,15)	1x(192,15*) 1x(64,15*)	1x(341,1) 1x(170,2) 1x(1,15)
C14			5x(100,1) 1x(12,1)	2x(81,1) 4x(13,1)			1x(341,1) 1x(170,2) 1x(1,7)	2x(81,1) 1x(38,1) 1x(14,2)			1x(341,1) 1x(170,2) 1x(1,15)	12x(81,1) 1x(38,2) 1x(14,4)

Table 5.9: Enumerating compute schedules of our overlay against different convolutional layers of AlexNet, VGG-16, YOLO, and MobileNet CNNs. The * mark in a schedule signifies the channel parallelism factor.

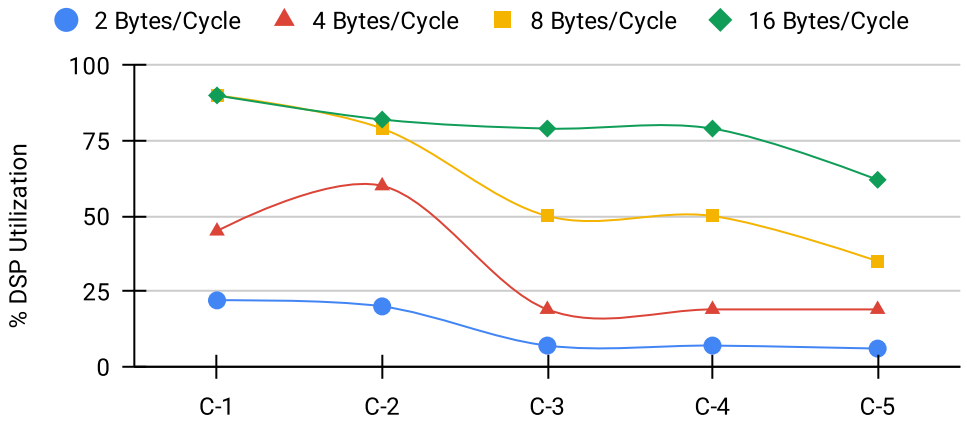


Figure 5.15: Variation in DSP utilization with changing external memory bandwidth for the convolution layers in AlexNet. The bandwidth is reported as Bytes/Cycle fetched from external DRAM.

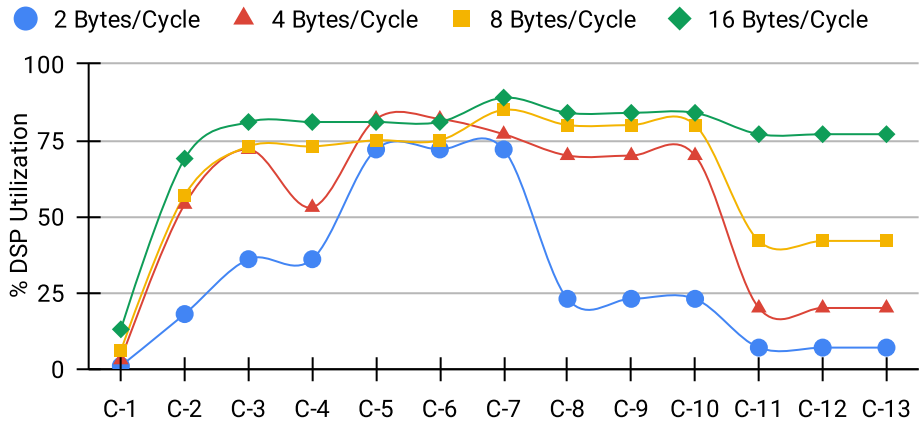


Figure 5.16: Variation in DSP utilization with changing external memory bandwidth for the convolution layers in VGG network. The bandwidth is reported as Bytes/Cycle fetched from external DRAM.

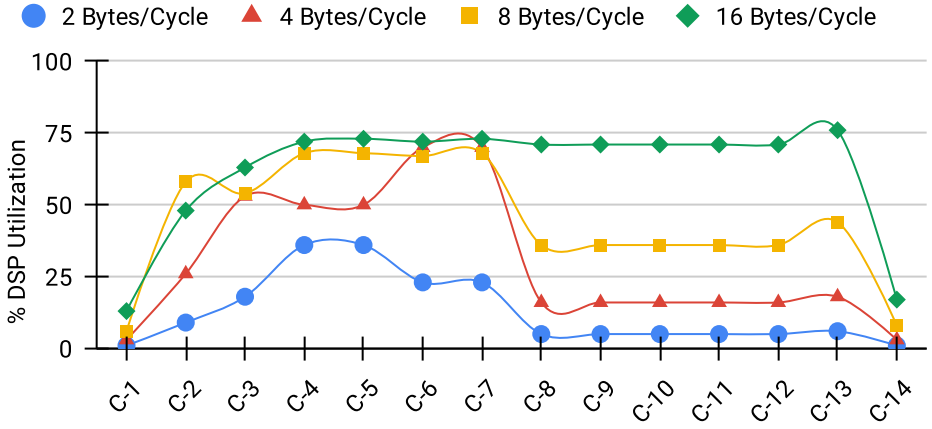


Figure 5.17: Variation in DSP utilization with changing external memory bandwidth for the convolution layers in MobileNet. The bandwidth is reported as Bytes/Cycle fetched from external DRAM. The DSP utilization for MobileNet combines a depthwise and canonical convolution layer under a single layer since they are processed in a pipelined fashion.

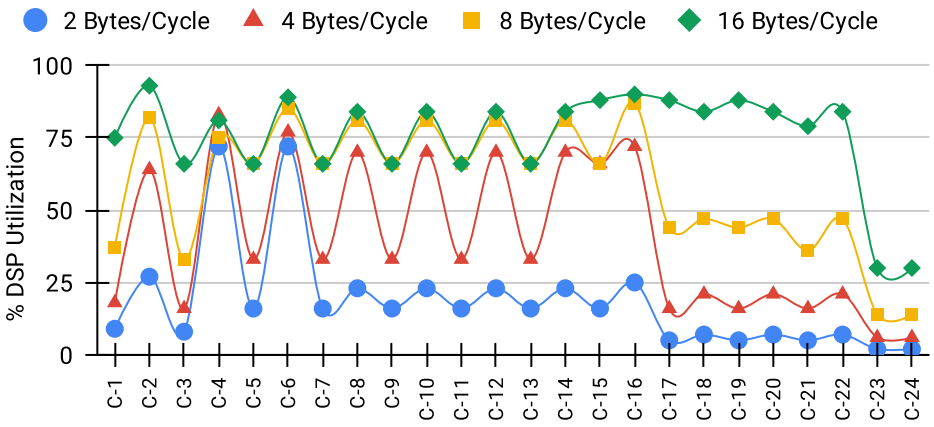


Figure 5.18: Variation in DSP utilization with changing external memory bandwidth for the convolution layers in YOLO-V2. The bandwidth is reported as Bytes/Cycle fetched from external DRAM.

Virtex7 690-t				
	CC ($\times 10^3$)	MC ($\times 10^3$)	PE ($\times 10^3$)	Latency (ms)
AlexNet	1068.4	210	82	6.5
VGG-16	6698.7	976.2	78	40.2
ResNet-50	3157.8	744.8	55	19.1
MobileNet	2892.5	1034.8	60	17.2
YOLO-V2	10133.1	2077.3	79	60.8

Table 5.10: The table lists the compute and memory cycles for processing various CNNs.

	Static Scheduling			**Dynamic Scheduling**	
Network	**Config (FP, FS)**	**Throughput (GOPS)**	**Calls**	**Throughput (GOPS)**	**Calls**
AlexNet	(77,4)	1098	27	1200	18
VGG-16	(32,10)	995	141	1020	30
YOLO-V2	(256,1)	820	111	1075	72
Resnet-50	(128,2)	710	264	902	85
Mobile-Net	(86,3)	720	114	830	61

Table 5.11: Comparison of Static and Dynamic Scheduling in our accelerator.

respect to the FPGA, the pipeline efficiency is slightly lower for the VU9P board. This is due to the inefficient PCIe bandwidth utilization leading to the under-utilization of DSP blocks.

We next contrast the effectiveness of a static schedule, prevalent in most accelerator designs, against a dynamic schedule (proposed in this work) in processing CNNs. We run our accelerator in static mode for this experiment. For a given CNN, a fixed schedule is used to process all the layers in the static version. The fixed schedule is generated by first choosing the $SP/CP/FP$ value that maximizes the throughput of a single layer. Then, from amongst all the CNN layers, the schedule that maximizes the total throughput of the entire network, i.e., all the layers combined, is chosen. As can be seen in Table 5.11, there is a throughput drop of 100 to 200 GOps/Sec between both the schedules. The lowest throughput drop is seen for VGG-16, as it is a relatively uniform CNN, and the same schedule suffices to extract fairly good throughput for all the layers. The throughput drop is more significant for non-uniform networks like ResNet/Mobile-Net that vary in kernel dimensions and convolution types. Another essential comparison metric is the number of accelerator invocations/calls that the host makes to process a complete CNN. As can be seen, compared to the dynamic schedule, the static schedule results in 1.3 to 3x higher accelerator invocations. Higher invocations result in more processing latency due to the long communication setup time between the host and the FPGA. Compared to the static schedules, the dynamic schedule can better adapt to varying computation shapes across the layers, which enables it to process more computations per batch, resulting in fewer batches.

5.8 Chapter Conclusion

In this chapter, we presented an FPGA overlay for CNN inference. Apart from regular convolutional layers, our overlay can handle point and depth separable layers. The synthesized architecture is dependent on the available FPGA resources alone and is agnostic to any specific CNN. The host machine configures the overlay through control words on a per layer basis so as to maximize the throughput based on the layer characteristics. These configuration parameters, surface and filter parallelism degrees, are determined through a constraint satisfaction problem. The performance of the overlay for various CNNs are thoroughly analyzed and their performance is compared against the theoretical limits obtained using hardware performance counters in the design.

PART III

Miscellaneous Topics

Chapter 6

A New Line Buffer Overlay Design

In this chapter, we present a new line buffer design, an enhancement over the one discussed in Chapter 4. The newly proposed design offers improved scalability, allowing for the generation of more windows per cycle while demanding fewer FPGA resources. Additionally, this updated design significantly reduces Block RAM (BRAM) utilization compared to its predecessor. The proposed line buffer design is particularly well-suited for Convolutional Neural Network (CNN) workloads involving the processing of Input Feature Maps (IFMAPs) using filter windows. Two key aspects merit attention:

- **Quantity:** When transforming IFMAPs into Output Feature Maps (OFMAPs), a large number of convolutions occur across various dimensions of the IFMAP, as detailed in the "Parallelism and Data-Reuse" section of Chapter 5. Concerning filter parallelism, substantial data replication takes place as multiple filters are applied to the same IFMAP. One straightforward approach to handle this parallelism is to create multiple copies of the same window from the line buffer and distribute them across all the filters. However, this approach is hindered by the increased window size due to hardware replication. The new line buffer design, as an alternative approach, uses the line buffer itself to create replicated columns and subsequently assembles these to create replicated windows. In terms of surface parallelism, the new line buffer design eliminates the need for copy-shift-add operations on the output window, directly producing parallel surface windows as output.
- **Size:** CNNs often employ window sizes such as 3x3, 5x5, or 7x7, which play a pivotal role in the CNN overlay configuration. The previous line buffer design handled this by sliding a larger 10x10 window, referred to as the base window, over the IFMAP. The desired window was then extracted from this larger base window through a series of mask and shift operations, a method used in various other works as well. However, a significant drawback of this approach is the initiation latency. The line buffer had to fill ten rows (assuming a base window size of 10) of data from the IFMAP before it could start generating output. In the context of CNNs, where numerous IFMAPs need to be processed in succession, this initiation latency substantially increased the overall processing time. In the new line buffer design, the line buffer only buffers the necessary number of rows to produce an output. For example, to process a 3x3 filter, the line buffer only needs to buffer three IFMAP rows.

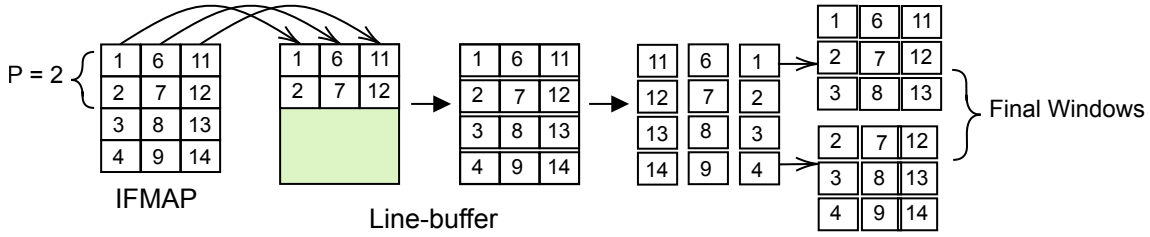


Figure 6.1: The central idea behind the design of the new line buffer.

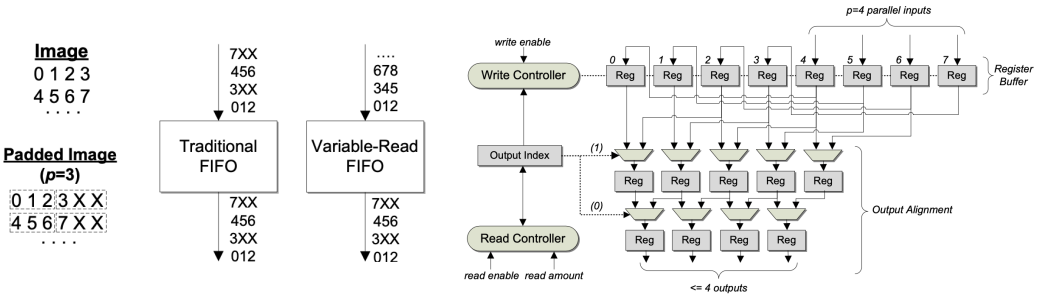


Figure 6.2: The Variable-read FIFO from the paper we refer here.

Central Idea: The central idea, depicted in Figure 6.1, behind the new line buffer design is as follows. Unlike existing designs, the IFMAP data is streamed in a hybrid column-row major order. Assuming an external memory bandwidth of P (where the host-FPGA interface can handle P data items per cycle), we stream P contiguous rows, which are part of a column, in a row-major fashion. The columns being streamed in are in an aggregated form, meaning they can generate constituent columns for output windows. For example, as illustrated in the figure, an aggregated column of size 4 can produce 2 disaggregated columns, each of size 3. These columns are constituents of their respective 3x3 output windows. The line buffer receives the aggregated columns as input and produces disaggregated columns based on the filter size. These disaggregated columns are then assembled to generate the final output windows.

6.1 Previous Work

In this section we describe in details an existing line buffer [86] that inspired the work in this chapter. In this work, the window generator consists of a variable-read FIFO, a window buffer, and a window coalescer. Initially, the user passes a stream of pixels (P each cycle), P is the external memory bandwith, to the variable-read FIFO, which provides a variable number of pixels from that stream to the window buffer to obviate input padding. The window buffer assembles the pixels into columns of window data that the coalescer combines into approximately P complete windows each cycle.

6.1.1 Variable Read FIFO

The variable-read FIFO (VRF), see Figure 6.2, streams pixels into the window buffer in a way that enables the window buffer to generate P windows in parallel, while eliminating the need for input padding. Although padding the image to have columns that are a multiple of P can eliminate the need for the VRF, such padding has a software preprocessing and PCIe overhead. Although such padding is conceptually easy to implement on the FPGA, creating a circuit that adds variable padding based on runtime parameters with no performance overhead is not trivial. The VRF provides this functionality. Figure 6.2, compares the difference between input streams when using a traditional FIFO with padding and the VRF. For an image with four columns and a system that can provide three pixels (P = 3) each cycle, the first read from the FIFO provides elements 0 to 2. To complete the processing of the first row, the application only needs element 3. However, if the FIFO provides three elements instead of one, the application will either be corrupted with invalid data for the first row, or must buffer the extra data somewhere internally. Since the FIFO already provides buffering, it makes more sense to read a variable amount of data from the FIFO instead of adding additional buffering elsewhere, which would also need to support variable amounts. With the VRF, the application reads three pixels to get pixels 0 to 2, then one pixel (3) to complete the first row, then pixels 4 to 6, then pixel 7, etc. This approach eliminates up to $P - 1$ padded pixels from the right edge of each image row, which is critically important for large P values where the padding overhead could be prohibitive. The VRF structure is conceptually similar to a FIFO, but has been modified to improve timing scalability for larger P. Both approaches write P pixels into a fixed set of registers within a buffer of $2P$ registers. In this buffer, the output index varies depending on previous reads. Initially, the output index starts at 0 and then increases after each read by the number of elements read from the FIFO. When the output index exceeds $P - 1$, the write controller shifts the register buffer left by P positions to ensure the index is always between 0 and $P - 1$.

Because the output index changes, the read controller must align the outputs with the appropriate P registers. The previous approach implemented output alignment using p separate P:1 muxes to select the appropriate register for each output, which is a significant timing-closure bottleneck. Although those muxes could potentially be pipelined, muxes in general are an expensive FPGA resource. To avoid this problem, and to ensure better scalability, our new approach ensures that the critical-path propagation delay (ignoring routing delays) is independent of P. In our approach, the VRF aligns outputs using a pipelined barrel shifter that shifts by the amount in the output index. Although this approach creates a several-cycle output delay, the user can still read every cycle. With this strategy, there is never more than a 2:1 mux in between registers for any value of P, which potentially enables the architecture to scale indefinitely up to any resource constraint without experiencing a timing-closure bottleneck. In addition to the logic in the figure, the VRF outputs a count of the words in the FIFO, in addition to bits that specify the validity of each output. For example, when the user requests one output, the VRF will provide P outputs, but will mark $P - 1$ outputs as invalid. Although the propagation delay of the count logic increases with larger P, that increase is logarithmic. For any realistic value of P, the count logic

is not a timing-closure bottleneck. Even for $P = 1024$, the count logic only requires a 10-bit adder and subtractor.

6.1.2 Window Buffer and Coalescer

The window buffer is responsible for buffering the input stream of pixels into separate rows, and then passing P columns from each row into the window coalescer each cycle. The basic structure of the window buffer is a chained sequence of wr FIFOs, which each buffer an entire row of the image. The window buffer reads P pixels at a time from the VRF into the bottom row FIFO, where each word consists of P pixels. When there are fewer than P pixels left in a row of the image, the window buffer requests the remaining number of pixels from the VRF. In this case, the window buffer marks any extra pixels as invalid to avoid including them in windows. Always the top FIFO is taken as as the top row of a window, the bottom FIFO as the bottom row, etc. To enable this functionality, the window buffer initially writes all incoming pixels into the bottom FIFO, storing P pixels per word. When the bottom FIFO contains an entire image row, every new write into the bottom FIFO also triggers a read from that FIFO. For each FIFO read, the buffer writes the read data into the next higher FIFO, which enables the pixels to gradually shift to the top FIFO. At this point, a controller starts reading from all FIFOs, which passes P columns each cycle into the coalescer. The coalescer assembles P windows by dividing each window into wc register columns that get shifted by P positions for every new set of columns from the window buffer. After some shifts, the coalescer contains P complete windows, where the index of the first window starts at the first column, the second window at the second column, etc.

To support any window size, the controller determines how far to slide the maximum-sized window across the image. For example, if the FPGA provides a 10×10 window, but the user requests a 3×3 window, the controller would slide the 10×10 window seven pixels past the right edge and bottom edge of the image. The controller pads all unused window elements with 0, which is done automatically on reset. With this strategy, although many window elements are unused, generation times for 3×3 windows are similar regardless of the maximum window size, with the only overhead being the initial time to fill up the extra row FIFOs. To support arbitrary image sizes, the architecture sets the FIFO depth to the maximum image width divided by P, and then simply starts reading from the FIFOs when the requested image columns are buffered in each FIFO.

6.1.3 IFMAPs and Line Bufferring

When it comes to supporting dynamic window sizes at runtime, the current design faces a significant flaw in processing a sequence of IFMAPs. The approach to accommodate dynamic window sizes appears rather simplistic. A window of size $K \times K$, where K is the window size, is extracted from a larger $K_{max} \times K_{max}$ window, assuming a maximum window size of 10 ($K_{max} = 10$). The line buffer can only produce the initial window once 10 IFMAP rows are buffered. As the disparity between K and K_{max} grows, the unnecessary initial buffering also increases. This issue is exacerbated by the fact

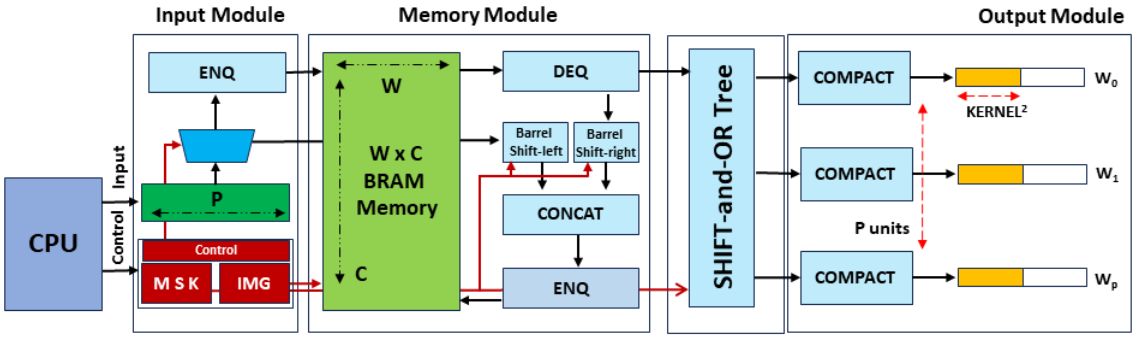


Figure 6.3: The block diagram of the newly proposed line buffer design.

that input IFMAPs are three-dimensional, with depth values ranging from 512 to 1024. The additional buffering latency is incurred at each depth, resulting in a substantial overall increase in latency during the initial buffering phase for processing the entire IFMAP volume. Our proposed line buffer design eliminates the need for buffering extra rows. In other words, there is no requirement to extract a smaller window from within a larger window to support dynamic window sizes within our line buffer.

6.2 New Line-buffer Architecture

Sliding-window applications fall within the domain of digital signal processing, involving the execution of application-specific computations on "windows" (subimages) within an image. Typically, these applications slide windows across an image horizontally from left to right at the top, progressing row by row until all windows have been processed. Our focus is on specific sliding behaviors, particularly single strides where each window moves column by column, and fully immersed windows that do not extend beyond the image borders. Although our approach is adaptable to various sliding behaviors, it may entail less reuse for larger strides. This section introduces the line buffer architecture designed for independently generating all necessary windows during application-specific computation. The input to our line buffer is an IFMAP (Input Feature Map) denoted as I, characterized by a depth d, with each feature having a width W and height H. The top-left pixel of the IFMAP is defined as $I[0, 0, 0]$. We use P to represent the number of pixels provided each cycle, the number of windows generated each cycle, and the amount of pipeline replication, all of which are equivalent in our architecture.

The line buffer comprises four interconnected modules: an input module, a memory module, a shifter module, and an output module. The line buffer's operation is configured by the host CPU through the initialization of an 8-bit control register (`KR`) and a 16-bit control register (`WD`). The value m in the `KR` register determines the window size, which is proportional to m^2. Simultaneously, the `WD` register sets the width of the input frame processed by the line buffer. Crucially, our buffer remains independent of the input image's height, relying solely on its width for functionality. This aspect will become clearer as we delve into the architecture in more detail. Following the initialization phase, the CPU streams

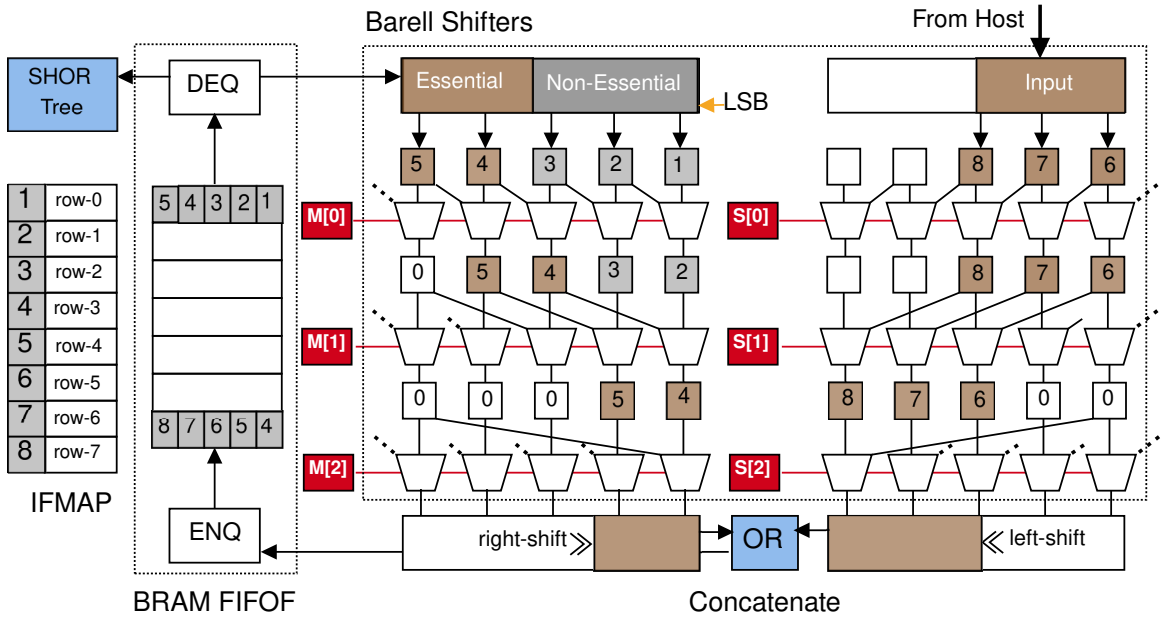


Figure 6.4: The detailed view of the memory module inside the new line buffer design.

aggregated IFMAP columns, each with a length of P, in row-major order. These columns are stored in a memory module constructed using BRAM memory, organized as a circular buffer with dual ports for simultaneous enqueue and dequeue operations. Subsequently, the aggregated columns are read from the memory module and directed to the Shift-Or (SHOR) tree, where the disaggregation process occurs. The disaggregated columns are then routed to the output module, where they are combined to generate an output window of size m^2.

6.2.1 Input Module

This module functions as an intermediary between the CPU and other line buffer modules, receiving input from the CPU and forwarding it accordingly. The input, presented as a column vector with a length of P, consolidates data from P consecutive rows. Within this module, control registers govern dynamic kernel size, parallelism value, and image width. The CPU updates these control registers as needed to configure the line buffer. Line buffer modules directly access and read these control registers for their configuration. Moreover, the input module closely collaborates with the memory module, tasked with storing the received input data. The input module enqueues data into the BRAM memory of the memory module or directs it to a series of barrel shifter modules within the memory. Here, the input undergoes shifting, concatenation, and subsequent enqueueing inside the BRAM memory block.

6.2.2 Memory Module

The memory module comprises two primary components: a section composed of a BRAM block and operational modules that alter the content of the adjoining BRAM memory block. Overall the memory subsystem is designed as a circular queue or ring buffer. A ring buffer is a First-In-First-Out (FIFO) data structure that maintains the order in which elements were added to the buffer and retrieved in the same order. It functions similarly to a circular queue, where data is enqueued at one end and dequeued from the other, ensuring that the oldest data is dequeued first.

The line buffer memory is a circular queue with W_{max} memory words, each consisting of C bits, as depicted in Figure 6.3. Here, W_{max} equals the maximum IFMAP width the line buffer can handle. The value of C is determined as $K_{max} + P$, where K_{max} represents the maximum filter size that the line buffer can accommodate, and P is the parallelism factor, influenced by the host-to-FPGA bandwidth. In the operation of this memory module, the enqueue operation inserts a column vector received from the input module, while the dequeue operation destructively reads a word from the memory. The dequeued column vectors undergo modification through a sequence of shift and concatenate operations before being enqueued back into the memory. When the enqueue operation reaches the last memory word, it wraps around the first memory word. In other words, this operation is performed modulo the memory size W. This logic also applies to the dequeue operation. We utilize dual-ported BRAM blocks with independent read and write ports to enable simultaneous enqueue and dequeue operations on the memory. Each block RAM in Xilinx-7 series FPGAs can store up to 36 Kbits of data and be configured as two independent 18 Kb RAMs or a single 36 Kb RAM. These 36 Kb block RAMs can further be configured as various sizes, including 64K x 1 (when cascaded with an adjacent 36 Kb block RAM), 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36, or 512 x 72 in simple dual-port mode. Similarly, the 18 Kb block RAMs can be configured as 16K x 1, 8K x 2, 4K x 4, 2K x 9, 1K x 18, or 512 x 36 in simple dual-port mode.

Assume that the data unit is of size 8-bits. There are two ways to design a $W \times C$, memory block, where C is a multiple of 8. The first way is to use an array of H BRAM blocks where $H = \frac{C}{8}$ each with W words. The other way is to use a single BRAM memory block with word size C and having W words. The first approach offers the advantage of parallelism, as operations can be performed concurrently on multiple BRAM blocks, potentially improving overall throughput. It also has a modular design, which makes it easier to scale and maintain by adding or removing BRAM blocks as needed. However, it can introduce complexity in terms of managing multiple blocks and can incur overhead due to logic resources and routing. The second approach is to use a single BRAM memory block with a word size of C and containing W words. This approach simplifies the design, as it requires less complex addressing and control logic, and it incurs potentially lower overhead since there's no need for routing and control logic for multiple blocks. However, it offers limited parallelism for memory operations, which can impact throughput, and it might be less flexible when it comes to accommodating changes in data unit size or dynamic reconfiguration. The choice between these approaches should consider specific performance requirements, resource constraints, flexibility needs, and complexity tolerance, as it often

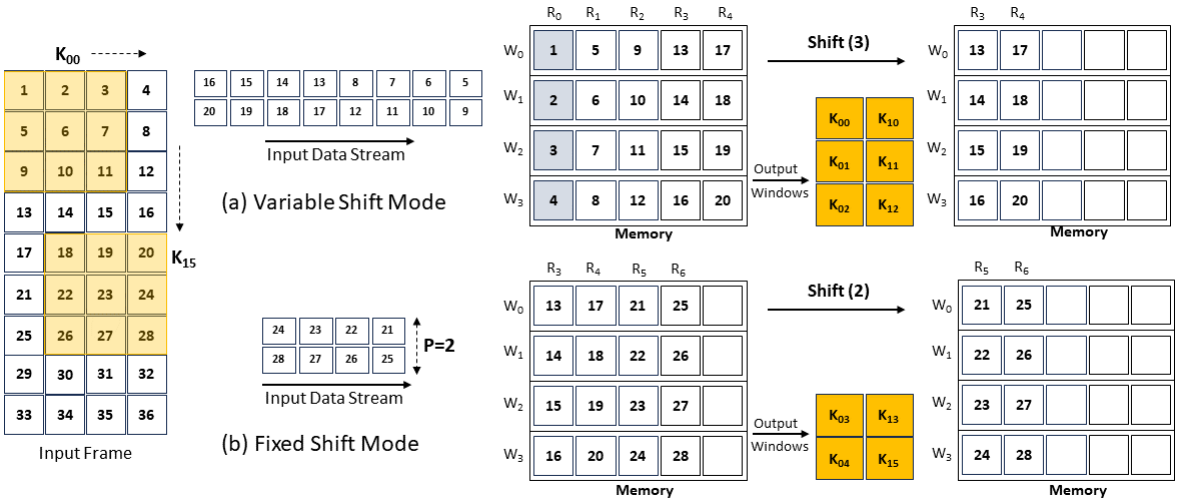


Figure 6.5: The Figure shows the concept of dynamic parallelism.

involves trade-offs between throughput, resource utilization, and design simplicity. In our design, we go with the second design approach.

Figure 6.4 provides a detailed illustration of the memory module. On the left, a single-depth IFMAP with 8 rows is depicted, each row labelled with its first pixel. Assuming $P = 3$, indicating that the CPU can stream 3 pixels of data per cycle, the input module sends the initial column vector [1,2,3,4,5] to the BRAM memory block in the first cycle. Notably, this vector comprises the first 5 values from the initial five rows of the IFMAP. Subsequently, the remaining pixels from these 5 rows are streamed in a similar manner, filling the BRAM memory block. At this juncture, the dequeue module activates and begins extracting data from the BRAM memory block. Each dequeued vector undergoes two processes. The first involves the shift-and-or tree, where the column vector is disassembled to produce constituent columns, as detailed in Figure 6.1. The second path encompasses a set of parallel shifters. These shifters, coupled with the concatenation module, manage the removal of obsolete data from the memory block and the appending of new input from the input module. This process unfolds as follows: assuming the line buffer generates 3x3 windows with a parallelism factor of 3 (capable of producing three 3x3 windows per cycle), the column vector [1,2,3,4,5] is not entirely needed for processing future windows. Specifically, pixels 1, 2, and 3 become unnecessary. After the initial 5 rows are completely buffered in the line buffer and the first word is dequeued from the memory block, space opens up for a new column vector. The CPU then streams the column vector [6,7,8], which is concatenated with the remaining values 4 and 5 from the previous vector. The resulting new column vector [4,5,6,7,8] is then enqueued back into the memory block. The parallel shifters, configured dynamically using the M and S registers, play a role in this process. The first shifter shifts the older input vector to the right, creating zeros in the vacated positions, while the second shifter shifts the new input vector to the left in a similar manner. The concatenation module performs a bitwise operation on the vector emerging from the parallel shifters,

sending the resultant data to the enqueue module, which appends the vector to the end of the memory block.

The variability of the parallelism factor during IFMAP processing is illustrated in Figure 6.5. It may occur that the number of rows in an IFMAP is not perfectly divisible by the parallelism factor P, denoted as $H\%P \neq 0$. In such cases, the last batch of P rows may contain fewer than P rows for the line buffer, resulting in underutilized bandwidth and empty slots in the input column vector. To address this inefficiency, we propose filling the vacant row slots with the initial rows of the next IFMAP depth in sequence, as depicted in Figure 6.5. For instance, in the illustrated IFMAP with dimensions $H = 9$ and $W = 4$, the BRAM memory block has 4 words, given $W = 4$ and a word length of $K_{max} + P = 3 + 2 = 5$. The top half of the figure illustrates the initial filling of the BRAM memory block, with Row 1 pre-filled from the previous IFMAP depth stream, followed by rows 2 through 5. With $P = 2$, two rows are filled simultaneously with column vectors of length 2. After the initial buffer, words are read sequentially, each containing 5 values, allowing three 3x3 windows to be generated per cycle. The initial filling creates six windows in 2 cycles, equating to 3 windows per cycle. The barrel shifter is configured to shift out 3 values from the read word as they are no longer part of future windows. The bottom part of the figure shows the subsequent state of the line buffer, where rows 6 and 7 are streamed from the CPU. These values are combined with rows 4 and 5 stored in the memory block after shifting, resulting in each word read having 4 valid values. This enables the generation of two 3x3 windows per cycle, changing the parallelism factor from 3 to 2. The barrel shifter is adjusted to shift out 2 values, maintaining a parallelism factor of 2 for processing the remaining rows of the IFMAP. Consequently, the parallelism factor adapts based on the valid values within the word read from the memory block, and this adjustment is made inside the line buffer by configuring the shift values of the barrel shifter units.

6.2.3 Shift and OR Tree

In this section, our attention transitions from the memory module to delve into the shift-and-or (SHOR) module, which is responsible for generating disaggregated columns from an aggregate column received from the memory module. The hardware configuration of the SHOR module within the line buffer is depicted in Figure 6.6. This module features a deeply pipelined structure with multiple levels, corresponding to a maximum value denoted as K_{max}. The structure resembles a tree, with each level comprising multiple SHOR units. At level j, a single unit at index i receives two inputs from level $j-1$, originating from units i and $i+1$. The number of SHOR units at each level decreases by one until it reaches P_{max}, representing the maximum parallelism supported by the line buffer at runtime. Each SHOR unit encompasses a shift module, an OR module, and a 2:1 multiplexer. Functionally, a unit at index i and level j either concatenates the two inputs received or forwards the left input unchanged to the next unit in the subsequent level. This design ensures efficient processing and utilization of the line buffer's parallelism capabilities. The concatenation operation within a SHOR unit unfolds as follows: the left input is OR-ed with a shifted version of the right input, where the right input undergoes a fixed

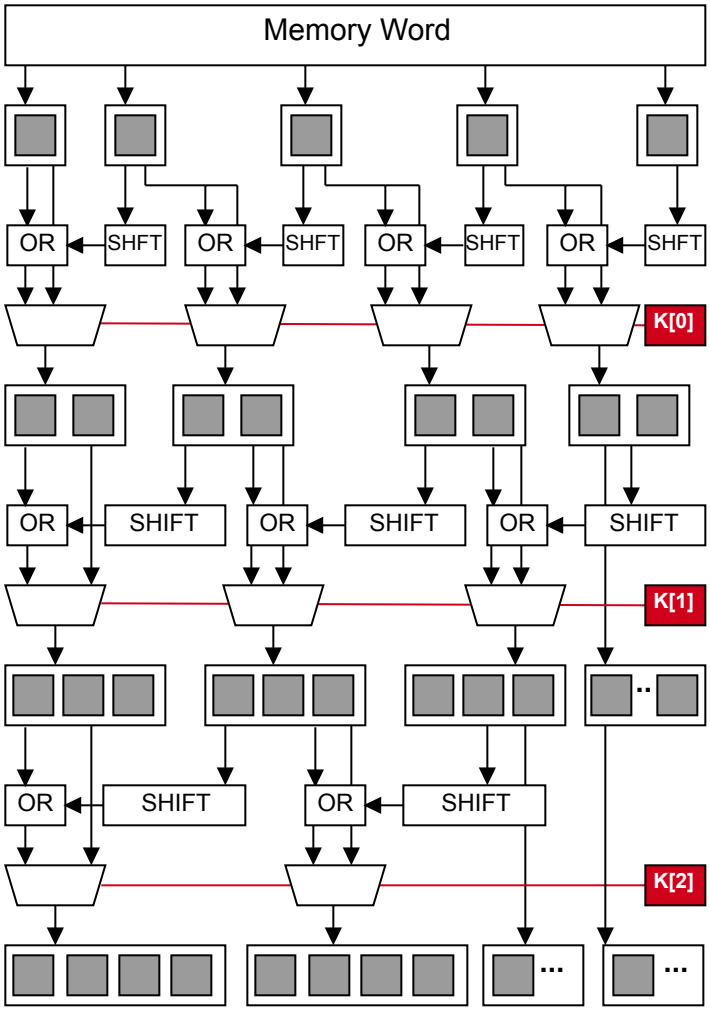


Figure 6.6: The figure shows the shift and or tree module inside the line buffer.

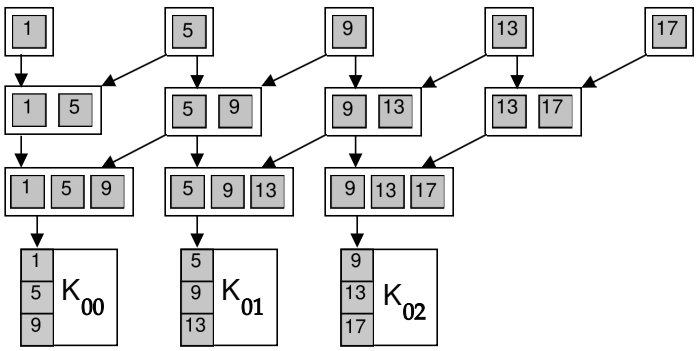


Figure 6.7: Figure shows the shift-and-or tree operating on a vector input of 5 elements to generate columns for a 3x3 window.

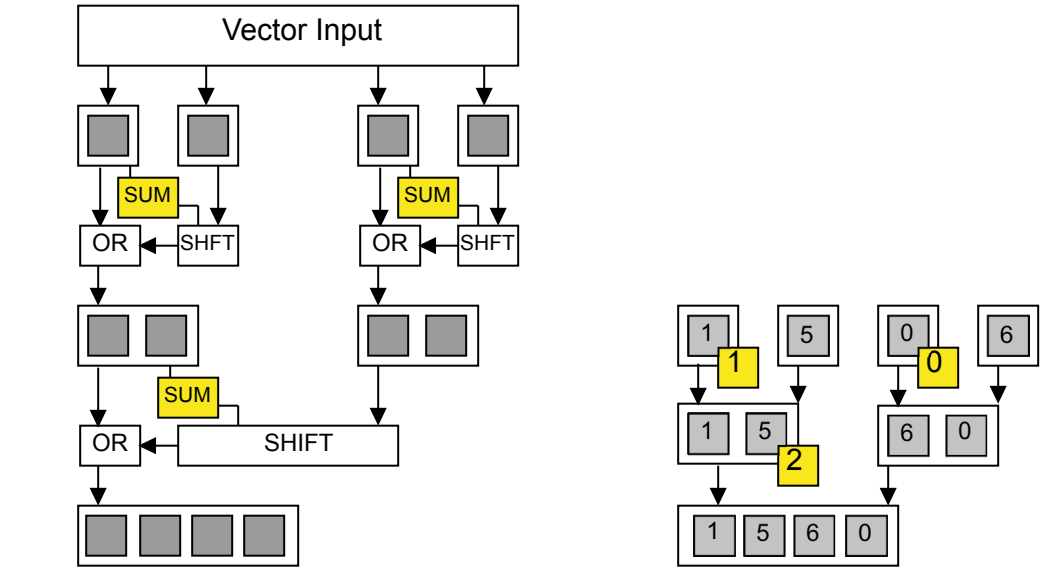


Figure 6.8: The Figure shows the compactor unit with an example on the right.

shift of d positions, with d representing the data type of the line buffer (e.g., $d = 8$ for 8-bit data units). It's crucial to note that the inputs to SHOR units are vectors.

Figure 6.7 provides an illustrative example wherein an aggregated vector of 5 elements is processed by the SHOR tree to generate three disaggregated columns, each of size 3. These disaggregated columns are then utilized to form 3x3 windows, as depicted in the figure. The processing involves three levels of the SHOR tree. In the lower levels, the SHOR units simply forward the received input values through the left input port. In the first levels, each SHOR unit concatenates two neighbouring values in the input vector $[1, 5, 9, 13, 17]$. For instance, the first unit outputs the vector $[1, 5]$, the second unit produces $[5, 9]$, and so forth. In the second level, each SHOR unit concatenates two vectors of length 2 to create a vector of 3 elements. For example, $[1, 5]$ is concatenated with $[5, 9]$, resulting in the output vector $[1, 5, 9]$. This process involves copying the left vector $[1, 5]$ into a larger vector X of length 3, i.e., $X = [1, 5, 0]$. The same is done for the right vector with an additional shift by 8 bits (assuming $d = 8$), creating the vector $Y = [0, 5, 9]$. The OR operation between X and Y vectors retains overlapping values (5 in this case), while non-overlapping values are OR-ed with 0. This yields the resultant vector $[1, 5, 9]$. This process is repeated in other SHOR units with their respective inputs to create the other two vectors of length 3. To ensure the line buffer generates only 3x3 windows, the shift and OR operation is confined to the second level. In the remaining levels, if any, the SHOR units simply copy the vectors through a forwarding operation into larger vectors with zeros appended at the end until they appear at the output.

6.2.4 Output Module

The windows generated from the line buffer contain padded zeros in every column. For instance, in a 3x3 window, zeros are padded in every row of the output window. This padding arises because the vectors from the SHOR tree have a length of K_{max}. In a scenario where $K_{max} = 5$ and $K = 3$, two additional zeros are present in the first three rows of the $K_{max} \times K_{max}$ output window. These extra zeros can cause complications for downstream MAC modules, which receive inputs from the line buffer. The line buffer window is flattened into a vector in row-major order and is read by the MAC units. Additionally, the MAC units will have interleaved zeros in the input vectors. To address this issue, we enhance the line buffer with additional compactor units. These units are responsible for extracting the extra zeros and consolidating the non-zero values of the window. Each line buffer has P_{max} compactor units, one for each output window.

Figure 6.8 depicts the hardware configuration of a compactor unit designed in the form of a binary tree. In this structure, each node is responsible for executing shift-and-OR operations on the input data. At every node, the right input undergoes a shift operation by a value stored in a sum register and is then OR-ed with the left input. Notably, the sum registers keep track of the count of non-zero values in the left input. In the second part, denoted as (b) in Figure 6.8, the process of compacting a vector of length 4 is illustrated, highlighting the elimination of the zero at the 3rd index. The sum registers at the initial level store a binary value indicating the presence of non-zero values in the corresponding positions of the vector. As the process advances to the second level, these sum registers now store the count of non-zero values in a vector of length two. The values in the sum registers are updated incrementally during the merging process. For instance, when combining the first and second indices with values 1 and 5, the sum value of both 1 and 5 is added to derive the resulting value in the second level.

6.3 Host Streaming

In this section, we elucidate the process through which the CPU host compacts the IFMAPs and streams the data to the line buffer. Figure 6.9 provides a detailed illustration of the streaming logic. Let's assume we have an IFMAP with dimensions $W = 5, H = 5$, and $d = 3$, consisting of three IFMAP surfaces labelled A, B, and C. These surfaces are interleaved and packed onto a single array, as depicted in the top right of the figure. Notably, two surfaces are packed together to accommodate potential spillage into the second surface during streaming, especially when the division of H by P is not exact. The interleaving process involves placing the i^{th} rows from adjacent surfaces next to each other inside the array. Once surfaces A and B are packed, the host initiates the streaming process. During each read operation, P values are exchanged between the host and the FPGA. Simultaneously, the CPU begins packing the next surface, C in this instance. The rows of C are positioned in the array's areas previously occupied by the rows of surface A. Additionally, C is packed at a row offset of 1, and this offset increases with the surface index. For instance, the first row of C is placed in the array slot

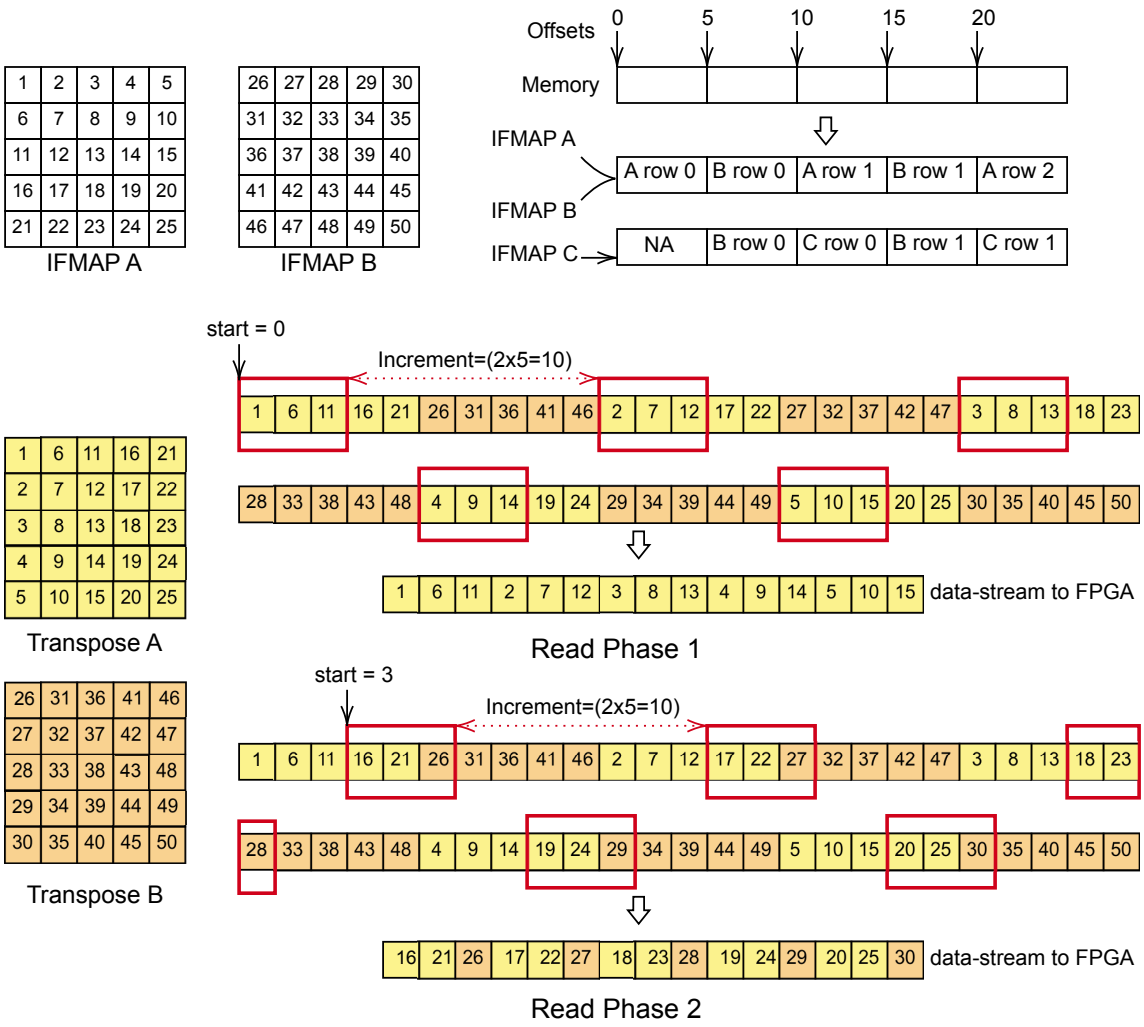


Figure 6.9: The Figure shows how the host logic packs three IFMAP surfaces A,B and C into a single data stream read by the FPGA.

originally holding the second row of A. Subsequent surfaces follow a similar pattern, utilizing the space occupied by the rows of the preceding surface (e.g., B), and so forth.

In the lower section of Figure 6.9, a more detailed representation of the packing process and FPGA's data reading is illustrated. The initial step involves transposing two IFMAP surfaces, a necessary operation due to the mixed traversal of row and column-major orders over an IFMAP surface. The packing process begins with the first read phase (depicted as read phase 1 in the figure). During this phase, the FPGA reads the first three rows from surface A, starting at index 0 and extracting the values [1, 6, 11], which represent the initial values of the first three rows of IFMAP surface A. Following this, the start offset increases by 10 positions, equivalent to twice the size of a single row (with a size of 5). This offset adjustment is necessary because the rows of surface A are streamed with the rows of surface B interspersed between them. At the new offset of 10, the second set of values from the first three rows—[2, 7, 12]—is sent to the line buffer. The first read phase concludes, and the second read phase commences, streaming the remaining rows of surface A. With only two rows remaining and the stream bandwidth supporting three rows, the first row of surface B is simultaneously streamed with rows 4 and 5 of surface A. The start offset is reset to 3 at the beginning of the second read phase. In this phase, the first read streams values [16, 21, 26]. Values 16 and 21 correspond to the initial values of rows 4 and 5, respectively, while 26 is the first value of the initial row of surface B. Similar to the first read phase, the start offset increases by 10. By the end of the second read phase, surface A is completely streamed to the line buffer, and the first row of surface B is included in the stream. The line buffer stores the first row of surface B in the memory block, awaiting the subsequent read phase to access the next set of rows from surface B for window production over IFMAP surface B.

6.4 Experimental Results

In this section, we first evaluate the scalability of the line buffer. We then provide performance comparisons between our design and [86], in addition to performance projections for convolutional neural nets namely AlexNet and VGG-16 networks.

Figure 6.10 illustrates a comparison between the maximum clock frequency of our proposed method and a prior study [86]. We utilized Vivado version 2023.1 to ascertain the maximum clock frequency post-synthesis, placement, and routing on a Vitex-7-690t FPGA. The presented results focus on 3×3 windows and image sizes of 2048×2048, with similar frequencies observed for other window and image dimensions.

The figure highlights the scalability of our line buffer design in contrast to the earlier approach. Notably, at lower replication factors (64 and below), our design exhibits a lower synthesis frequency compared to the competition, remaining nearly constant at 250 MHz—approximately 20% lower. Beyond a replication factor of 64, our design matches the frequency of the competition, converging at the same frequency for a replication factor of 512. Interestingly, as replication increases, both designs experience a decrease in frequency, with our design demonstrating a slower decline, maintaining fre-

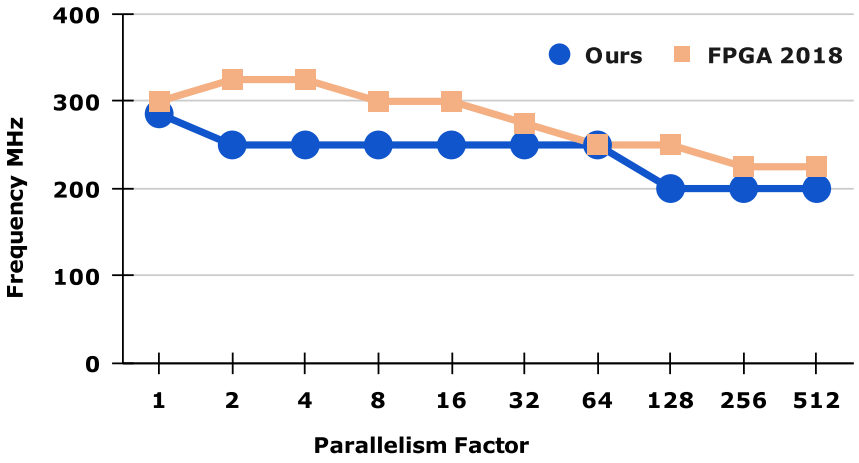


Figure 6.10: Comparing the Frequency attained by our line buffer design at various replication factors.

quencies above 200 MHz even with 512 pipelines. The maximum achievable bandwidth in our line buffer reaches 336 GB/s. These findings suggest that the proposed approach facilitates FPGAs in fully leveraging increased memory bandwidth for the foreseeable future.

Table 6.1 provides a comparison of the utilization of lookup tables (LUTs), flip-flops (FFs), and block RAMs (RAMs) between the proposed approach and prior work across various window sizes and replication amounts. In Part (a) of the table, we present the absolute counts of LUTs, flip-flops, and block RAMs consumed by the line buffer when synthesized for different window sizes. This decision to synthesize the line buffer for specific window sizes was made to facilitate a more accurate comparison with competing approaches. In the majority of cases, our line buffer demonstrates a more efficient use of resources, specifically LUTs, Flip-Flops, and BRAMs, when compared to previous methodologies. All reported results are based on a maximum image size of 2048×2048, which might lead to a more conservative estimate of RAM usage for common scenarios involving smaller images. Table 6.1 and Figure 6.12 and Figure 6.11 presents a comprehensive comparison of resource consumption between our design and the previous approach. Examining the LUT utilization, notable improvements are evident, particularly as we increase parallelism factors. In the context of 3x3 windows, the LUT savings are 2.5x for a parallelism value of 1, and 2.7x for a parallelism value of 512. A consistent trend is observed where the savings become more pronounced with larger window sizes. For instance, the improvement reaches 3.3x for 9x9 window sizes with a parallelism factor of 512. Similar patterns emerge when considering Flip-Flop consumption. Moving on to BRAM, our line buffer design consistently exhibits significantly lower BRAM usage compared to the earlier approach. The BRAM savings reach a factor of 2.4x for the 9x9 configuration with a parallelism factor of 512. These savings in BRAM usage become more substantial across all window sizes as we progress to higher parallelism factors. Table 6.3 presents the absolute resource consumption metrics for the generic line buffer design concerning LUTs, Flip-

(a) This Work

Replications	3 x 3			5 x 5			7 x 7			9 x 9		
	LUT	FF	RAM	LUT	FF	RAM	LUT	FF	RAM	LUT	FF	RAM
1	167	299	3	270	410	4.5	327	446	5	459	637	7
2	195	371	4	324	522	5.5	375	592	6	551	887	8
4	268	514	5.5	417	697	7.5	454	785	7	676	1272	10
8	414	798	9	610	1044	11	653	1204	11	843	1771	14
16	681	1363	16	981	1737	18	1017	2023	18	1209	2717	21
32	1220	2489	30.5	1708	3127	32	1734	3659	32	2740	6851	35
64	2133	4757	59	3154	5897	60.5	3425	8064	61	4485	11681	64
128	4392	9344	116	6226	12100	117.5	6839	16037	118	8458	22210	120
256	8964	18823	229.5	12407	24716	231.5	13722	32998	231	17200	45615	234
512	15526	36308	457	21916	46210	459	24938	65388	459	21816	60139	462

(b) FPGA 2018

Replications	3 x 3			5 x 5			7 x 7			9 x 9		
	LUT	FF	RAM	LUTs	FF	RAM	LUT	FF	RAM	LUT	FF	RAM
1	425	631	4	602	917	6	805	1278	8	1040	1695	10
2	461	704	5	638	1032	7	859	1428	9	1107	1891	11
4	560	990	5	763	1366	7	996	1952	9	1264	2377	11
8	823	1632	10	1158	2252	14	1528	2856	18	1911	3484	22
16	1379	3017	20	1925	4121	28	2489	5231	36	3059	6354	44
32	2444	5919	35	3216	8070	49	3997	10176	63	4742	12284	77
64	4826	11967	65	6265	16119	91	7711	20290	117	9218	24465	143
128	10001	24661	130	12747	32879	182	15494	41194	234	18306	49486	286
256	20704	51117	260	26262	67636	364	31603	83955	468	37062	100310	572
512	43360	106385	515	54097	100000	721	64928	200000	927	75862	204114	1133

Table 6.1: The absolute numbers for the LUT, Flip Flops and the BRAM resource consumption of our line buffer. The presented data is for four different window sizes.

	3 x 3			5 x 5			7 x 7			9 x 9		
	LUT	FF	RAM	LUT	FF	RAM	LUT	FF	RAM	LUT	FF	RAM
1	2.5x	2.1x	1.3x	2.2x	2.2x	1.3x	2.4x	2.8x	1.6x	2.2x	2.6x	1.4x
2	2.3x	1.9x	1.2x	1.9x	1.9x	1.2x	2.2x	2.4x	1.5x	2.0x	2.1x	1.3x
4	2.0x	1.9x	0.9x	1.8x	1.9x	0.9x	2.1x	2.4x	1.2x	1.8x	1.8x	1.1x
8	1.9x	2.0x	1.1x	1.9x	2.1x	1.2x	2.3x	2.3x	1.6x	2.2x	1.9x	1.5x
16	2.0x	2.2x	1.2x	1.9x	2.3x	1.5x	2.4x	2.5x	2.0x	2.5x	2.3x	2.1x
32	2.0x	2.3x	1.1x	1.8x	2.5x	1.5x	2.3x	2.7x	1.9x	1.7x	1.7x	2.2x
64	2.2x	2.5x	1.1x	1.9x	2.7x	1.5x	2.2x	2.5x	1.9x	2.0x	2.0x	2.2x
128	2.2x	2.6x	1.1x	2.0x	2.7x	1.5x	2.2x	2.5x	1.9x	2.1x	2.2x	2.3x
256	2.3x	2.7x	1.1x	2.1x	2.7x	1.5x	2.3x	2.5x	2.0x	2.1x	2.2x	2.4x
512	2.7x	2.9x	1.1x	2.4x	2.1x	1.5x	2.6x	3.0x	2.0x	3.4x	3.3x	2.4x

Table 6.2: The improvement of the LUTs, Flip-Flops and BRAMs compared to the previous work. The values in this table are calculated by dividing the value is part (b) of Table 6.1 with part (a) of the same table.

Replications	LUTs	FFs	BRAMs
1	818	2001	8
2	1033	2962	9
4	1464	4887	11
8	2326	8740	15
16	4032	16444	22
32	7983	31809	36
64	16636	62661	64
128	32539	124085	121
256	63500	247056	235
512	124849	493072	463

Table 6.3: Resource Consumption of the Line buffer overlay design. This design can generate all possible window sizes 3x3, 5x5, 7x7 and 9x9.

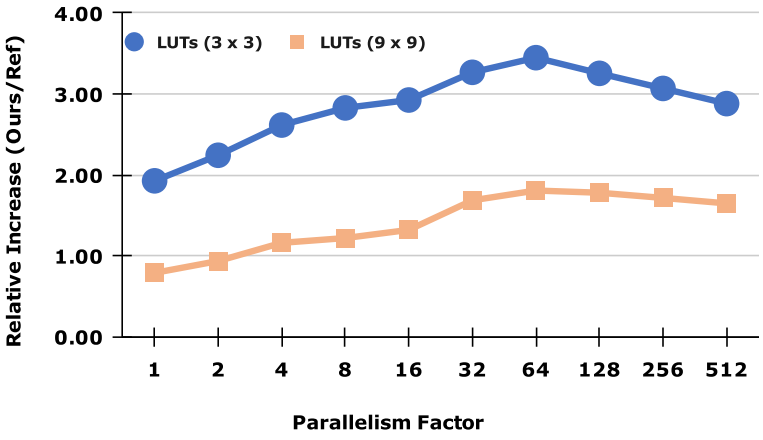


Figure 6.11: Comparing the relative increase in LUTs between the FPGA 2018 design and Ours.

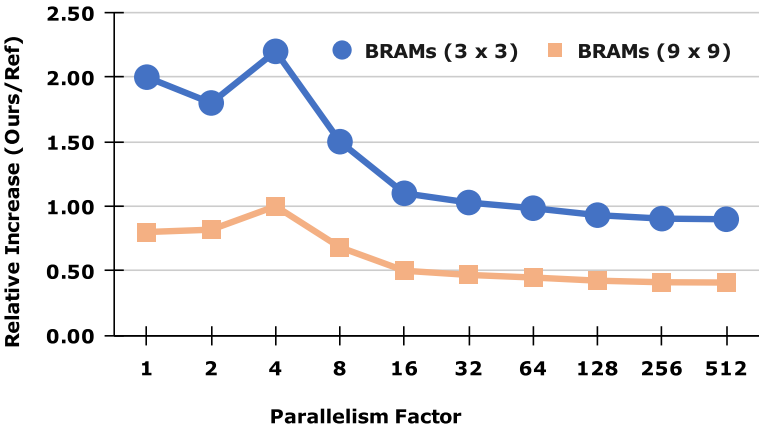


Figure 6.12: Comparing the relative increase in BRAM between the FPGA 2018 design and Ours.

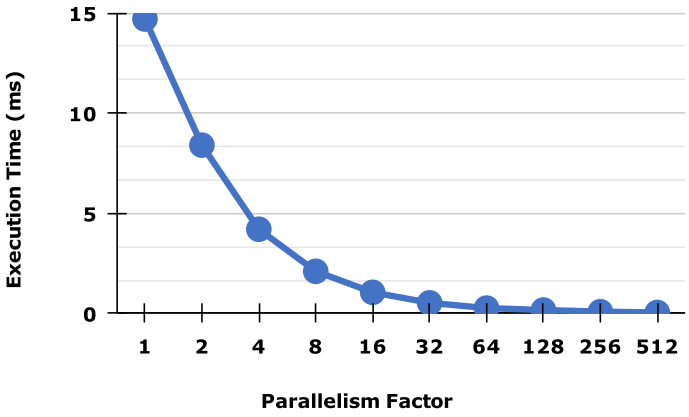


Figure 6.13: The execution time scaling with replication factors ranging from 1 through 512.

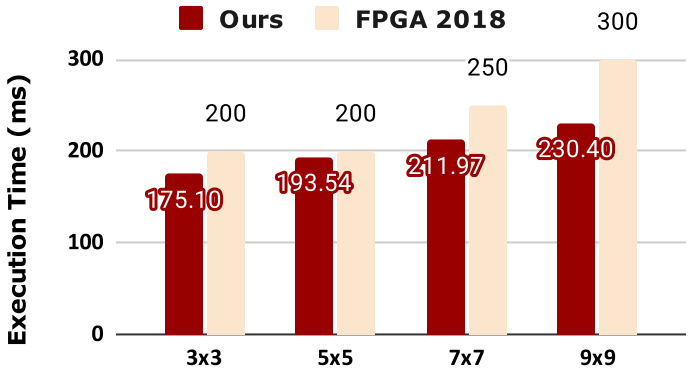


Figure 6.14: 2D convolution execution times for different kernel and for a 1024x1024 image.

Flops, and FPGA BRAM. This design is capable of generating all window sizes outlined in Table 6.1. Notably, the overall resource consumption of the generic design remains comparable to the 9x9 window size values of the previous methodology. In fact, the BRAM consumption of our generic line buffer design surpasses that of the earlier work when configured for 9x9 window sizes.

6.4.1 Performance Evaluation

Experimental Setup: We synthesize the generic line buffer design at 250 MHz and operate it with a data bit rate of 16 bytes/cycle in the read and write direction. The combined bit rate is 32 bytes/cycle. Therefore the overlay operates at a bandwidth of 10.6 GB/s (5.3 GB/s in each direction). We run our overlay a Virtex7-690t FPGA. The Virtex7-690t FPGA is a standalone FPGA with 3600 DSP blocks and 6.4 MB of on-chip BRAM. It is connected to an Intel Core-i5 processor through a PCIe-8x link. The code running on the CPU streams input using PCIe to our overlay using the Xillybus PCIe core (http://xillybus.com/doc/revision-b-xl). The ideal data bandwidth of the core operating on

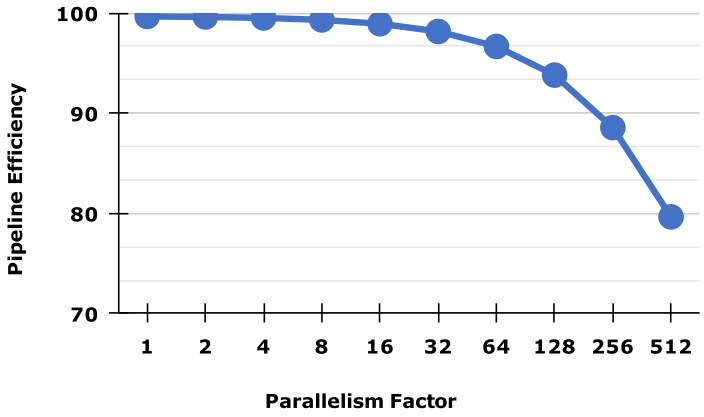


Figure 6.15: The Pipeline Efficiency of the line buffer with replication factors ranging from 1 through 512.

the Virtex-7 device with 8x Gen3 lanes, utilizing the Gen3 Integrated Block for PCI Express v3.0, can be expected to reach 6.4 GB/s each direction (read from host and write to host), see http://xillybus.com/doc/xillybus-bandwidth. We use Bluespec System Verilog (BSV) [70] to design our hardware. All the reported hardware characteristics of the design are obtained from Post Place and Route (PPnR) in the Vivado Design Suite version 2023.1.

This section explores the traditional use cases of employing one filter per image, utilizing kernel sizes ranging from 3×3 to 9×9, in conjunction with images spanning dimensions from 256×256 to 2048×2048. All instances incorporate inseparable kernels to illustrate worst-case performances, and the colour channels are standardized at 8 bits. To optimize FPGA performance, we replicated the 2D convolution pipelines by employing the provided window generator. The enhancements in FPGA execution time are showcased in Figure 6.13, illustrating different levels of pipeline replication for a 3×3 kernel on a 2048×2048 image with 8-bit colour channels. Similar trends were observed for various window and image sizes. The results indicate nearly perfect performance improvements initially, with each replication achieving a 1.99× speedup over the preceding replication level. However, with 32 replications, the improvement decreased to 1.18× due to the exhaustion of memory bandwidth. Notably, there was no improvement with 64 replications, and this trend emerged as the primary performance bottleneck for almost all the presented results.

Figure 6.14 contrasts execution times with previous work for different kernels on a 1024×1024 image. In these results, we assess the maximum achievable performance by excluding all PCIe transfers from the execution times. For 1024×1024 images, our implementation consistently delivered the best performance, exhibiting lower execution times for all kernel sizes, with FPGA speedup ranging from 1.0× to 1.4×.

Pipeline efficiency pertains to the effectiveness and performance of a pipeline system in processing and transmitting data or tasks. A pipeline consists of sequential stages, where each stage performs a specific operation on the data, and the output of one stage serves as the input for the next. The efficiency

	Throughput GOps/Sec	Performance Density GOps/KLUTs	Precision Fixed-Point	Frequency MHz
AlexNet	1200	5.38	16 bits=8,8	166
VGG-16	1025	4.60	16 bits=8,8	166

	Execution Kilo Cycles (EC)	Memory Kilo Cycles (MC)	Theoretical Kilo Cycles (TC)	Pipeline Rate (EC+MC)/TC
AlexNet	1096	112	856	1.45
VGG-16	7046	431	5149	1.44

Table 6.4: The table lists the Execution Cycles (EC), Memory Cycles (MC) and Theoretical Cycles (TC) required by our overlay to process various networks. The pipeline rate (PR) to process each network is also reported.

of a pipeline is often gauged by how well it utilizes its resources and minimizes idle time. The pipeline efficiency of our designs is depicted in Figure 6.15. We measure this as the ratio between the execution cycles and the expected number of cycles for a given parallelism factor. As seen, efficiency decreases at higher replication factors, attributed to the increased latency in obtaining data matching the parallelism factor. Notably, the decrease in efficiency is linear.

6.4.2 CNN Performance Projections

In this section, we evaluate the window generator for convolution parameters common to CNNs. Specifically, we use an image size of 256×256, filter sizes of 3×3 and 5×5, filters per image ranging from 32 to 512, common to VGG-16 and AlexNet. We test the adaptivity of the line buffer by using it to process the AlexNet and VGG-16 CNNs. The convolution operator is realized by using MAC units coupled with the line buffer. We were able to achieve a throughput of 1200 GOPs and 1025 GOPs,

respectively, at 166 MHz. Table 6.4 contains cycle-level details of the architecture. We achieved a Pipeline Rate (PR) of 1.4, close to the theoretical rate of 1, signifying minimum pipeline stalls.

6.5 Chapter Conclusion

In conclusion, this chapter has introduced a novel line buffer design that represents a significant enhancement over the previous iteration discussed in Chapter 4. The proposed design focuses on improving scalability and reducing resource utilization, particularly in the context of Convolutional Neural Network (CNN) workloads involving Input Feature Maps (IFMAPs) and filter windows. Two key aspects, quantity, and size, were addressed to enhance the efficiency of the line buffer design. In terms of quantity, the new line buffer design adopts an innovative approach by using the line buffer itself to create replicated columns, which are then assembled to generate replicated windows. This eliminates the need for hardware replication and directly produces parallel surface windows as output, addressing the challenges associated with filter and surface parallelism. Regarding size, the new design significantly reduces initiation latency by buffering only the necessary number of rows to produce an output. This approach is particularly advantageous for CNNs, where processing multiple IFMAPs in succession is common. The line buffer now buffers the exact number of rows needed to process a specific filter, reducing initiation latency and improving overall processing time. The central idea behind the new line buffer design is illustrated in Figure 6.1, showcasing the streaming of IFMAP data in a hybrid column-row major order. The aggregated columns received by the line buffer are then used to generate constituent columns for output windows, demonstrating the efficiency and adaptability of the proposed design. The experimental results presented in this chapter provide a comprehensive evaluation of the new line buffer design's scalability, performance, and resource utilization. The design outperforms previous approaches in terms of clock frequency, showcasing its potential scalability. Additionally, resource utilization comparisons, illustrated in Figure 6.10, Figure 6.12, and Figure 6.11, reveal significant improvements in terms of LUTs, flip-flops, and Block RAMs, reinforcing the efficiency gains achieved by the proposed design. The performance evaluation section further demonstrates the effectiveness of the new line buffer design in traditional use cases involving one filter per image and various kernel sizes. The comparisons with previous work, as depicted in Figure 6.13 and Figure 6.14, highlight the consistent improvements achieved by the proposed design, particularly in terms of execution times and FPGA speedup. Lastly, the section on CNN performance projections reveals the adaptability of the line buffer for common CNN parameters, such as image size, filter sizes, and filters per image. The achieved throughput for AlexNet and VGG-16 demonstrates the practical applicability of the proposed design in real-world CNN scenarios.

In summary, the new line buffer design presented in this chapter offers a compelling solution to the challenges posed by scalability, resource utilization, and performance in the context of CNN workloads. The experimental results provide strong evidence of its efficacy and potential for enhancing the efficiency of FPGA-based accelerators for convolutional operations.

Chapter 7

Composing the Overlay with Fixed-Function Hardware

In this chapter, we enhance the capabilities of our image-processing overlay by integrating it with fixed-function designs. The objective is to extend the overlay's capacity to accelerate a broader spectrum of algorithms within the domain, surpassing the limitations of the basic design presented in Chapter 4. To enable compatibility between any fixed-function logic and our overlay, we require a basic wrapper with a FIFO-IN and FIFO-OUT interface, illustrated in Figure 7.1. This interface is also applied to wrap the computing block of the overlay. Memory interfaces, separate from the compute pipeline, serve as input and output buffers for the overall design.

Integrating the overlay with fixed-function hardware yields several advantages for the image-processing system. The main motivation is the ability to accelerate a wider array of domain-specific algorithms. Fixed-function designs augment the overlay's capabilities, empowering it to efficiently handle more intricate and diverse computational tasks. Specialized for specific functions, fixed-function hardware provides dedicated processing tailored to certain algorithms, often resulting in superior performance compared to a generic, programmable overlay. Consequently, the overall image-processing system achieves heightened throughput and reduced latency. Designed for their intended tasks, fixed-function designs ensure efficient resource utilization. Offloading specific functions to dedicated hardware enables the overlay to leverage available resources more effectively, circumventing the overhead associated with generic programmable components.

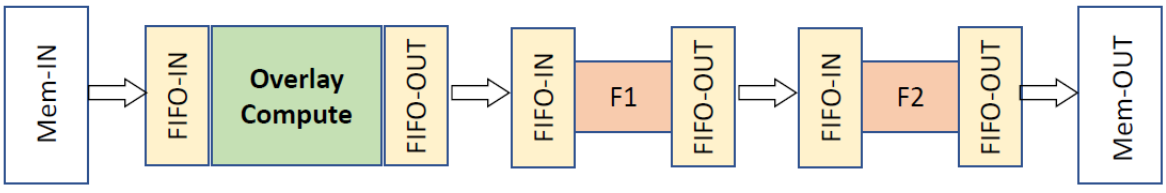


Figure 7.1: Figure shows a hybrid pipeline containing our overlay composed with fixed-function hardware F_1 and F_2.

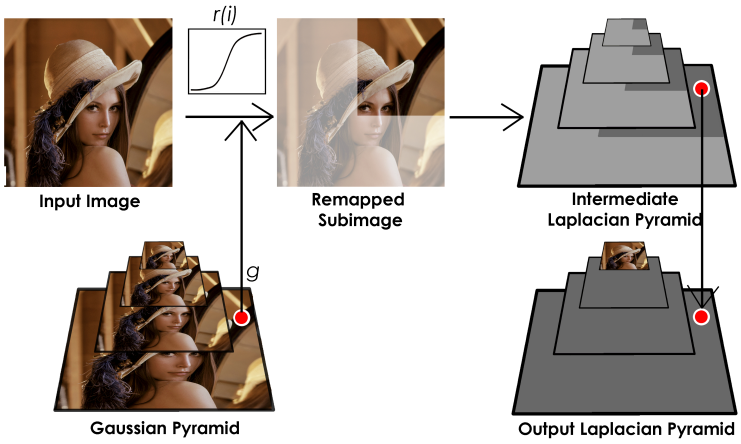


Figure 7.2: Overview of Local Laplacian Filtering. The figure has been taken from [45].

The adoption of a simple FIFO-IN and FIFO-OUT interface for fixed-function logic ensures a standardized and straightforward integration process, streamlining the overall system architecture. This approach facilitates the incorporation of various fixed-function designs into the overlay. Integration with fixed-function hardware fosters modularity in the image processing system. Each fixed-function block can be independently designed, enhancing reusability and adaptability across different applications. This modular approach enhances the system's flexibility and scalability.

We design two accelerator systems using this hybrid approach: a Local-Laplacian pipeline [74] and a stereo vision system [80, 38]. The computations in these algorithms are split across our overlay and a fixed-function logic block designed using a high-level synthesis language. Also, in both these designs, our overlay is configured to perform different operations.

7.1 Local Laplacian Filtering

This section explains the Local Laplacian Filtering algorithm as a solution to the limitations of Laplacian pyramids in edge-aware image processing tasks. Introduced by Burt and Adelson in 1983, Laplacian pyramids are multi-scale representations for images, widely used in image processing. However, they are criticized for lacking edge-awareness due to the use of spatially invariant Gaussian kernels. The spatially invariant nature of Laplacian pyramids causes problems like edge degradation and artifact introduction, such as halos. A technique that addresses these concerns, Local Laplacian Filtering, is introduced as an edge-aware approach for tasks like enhancing details, smoothing details, tone mapping, and inverse tone mapping. $G[I]$ and $L[I]$ denote the Gaussian and Laplacian pyramids for an image I. $G_l[I]$ and $L_l[I]$ represent level l of the Gaussian and Laplacian pyramids, respectively. Below are steps in the Local Laplacian Filtering algorithm.

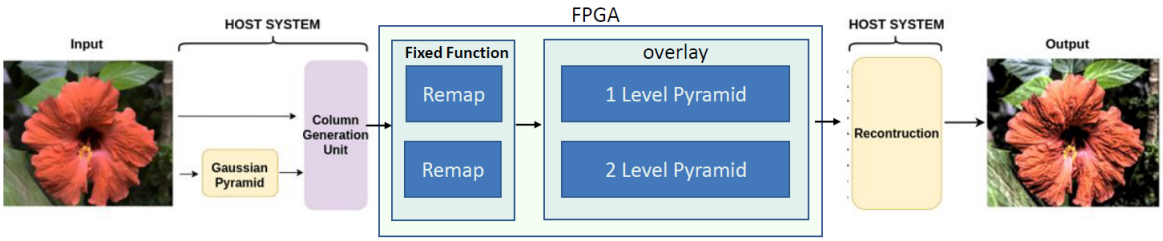


Figure 7.3: Figure shows the overall design of the local laplacian accelerator. The pyramid construction is done using the FlowPix overlay.

Initiate by constructing the Gaussian pyramid $G[I]$. For each pixel $g = G_l[I](x, y)$, identify a sub-image R in the input image. Use a remapping function to categorize each pixel i in R as either an edge or a detail based on the threshold θ. If $|i - g| \leq \theta$, treat the pixel as fine-scale detail and modify it using the detail remapping function r_f.

$$r_f(i, g, \theta) = g + \text{sign}(i - g)\theta f_f\left(\frac{|i - g|}{\theta}\right) \tag{7.1}$$

The function f_f can smooth ($\beta > 1$) or enhance ($0 < \beta < 1$) details, where β is a user-defined parameter. If $|i - g| > \theta$, categorize the pixel as an edge and modify it using the edge remapping function r_e.

$$r_e(i, g, \theta) = g + \text{sign}(i - g)\left(f_e(|i - g| - \theta) + \theta\right) \tag{7.2}$$

The edge-aware tone mapping function f_e performs tone mapping ($0 \leq \gamma < 1$) or inverse tone mapping ($\gamma > 1$), where γ is a user-defined parameter. Collect the modified pixels to form the remapped image $\tilde{R}$. Build Gaussian and Laplacian pyramids $G[\tilde{R}]$ and $L[\tilde{R}]$ for the remapped image. Replace Laplacian coefficients $L_l[I](x, y)$ with $L_l[\tilde{R}](x, y)$ to generate an edge-aware output Laplacian pyramid $\tilde{L}[I]$.

7.1.1 Accelerator Implementation

The host system establishes the Gaussian pyramid, denoted as $G[I]$, linked to the input image. Within the overlay section, illustrated in Figure 7.3, the computationally intensive task of constructing the Laplacian pyramid $\tilde{L}[I]$ takes place on the FPGA. To emphasize the importance of the Flowpix overlay, it is worth noting that this overlay, designed for benchmarks like the Laplacian Pyramid, is specifically responsible for the Laplacian pyramid construction. A crucial aspect of this process involves the creation of a fixed function logic, seamlessly integrated into the overlay. The subsequent responsibility of reconstructing the image from the Laplacian pyramid falls on the host system.

The hardware configuration for constructing Laplacian pyramids on the FPGA involves nine pyramid units, each accelerated by an instance of the overlay. Each instance is closely associated with a remap

	β = 1				γ = 1		
$\theta \backslash \gamma$	**0.25**	**0.5**	**2**	$\theta \backslash \beta$	**0**	**0.5**	**1**
0.1	45.5	48.03	48.33	**0.1**	40.56	45.42	49.98
0.2	40.55	44.88	46.12	**0.2**	43.13	47.37	49.98
0.4	34.75	40.19	40.88	**0.4**	49.94	49.11	49.98

Table 7.1: PSNR values between the CPU implementation with the accelerator implementation for the given flower image.

unit. Noteworthy is the simultaneous processing of all three RGB channels in an image. For each channel, three levels of output Laplacian pyramid ($\tilde{L}_1$, $\tilde{L}_2$, and $\tilde{L}_3$) are constructed in parallel. Despite the depiction of only two levels in Figure 7.3, it is crucial to understand that the overlay's pyramid unit encompasses a convolution stage, a downsampling stage, and an upsampling stage.

To highlight the role of the Flowpix overlay, it is essential to recognize that the host independently feeds data to the processing units through nine separate input streams. Subsequently, the output data from the nine processing units is efficiently transferred from the FPGA to the host using nine independent output streams.

Size (MP)	**Latency (ms)**				
	L1	**L2**	**L3**	**Sequential**	**Parallel**
0.25	133	64	49	246	133
0.5	267	129	99	495	267
0.75	400	194	148	772	400
1	534	259	198	991	534

Table 7.2: Latency of constructing laplacian pyramids while processing images of different sizes.

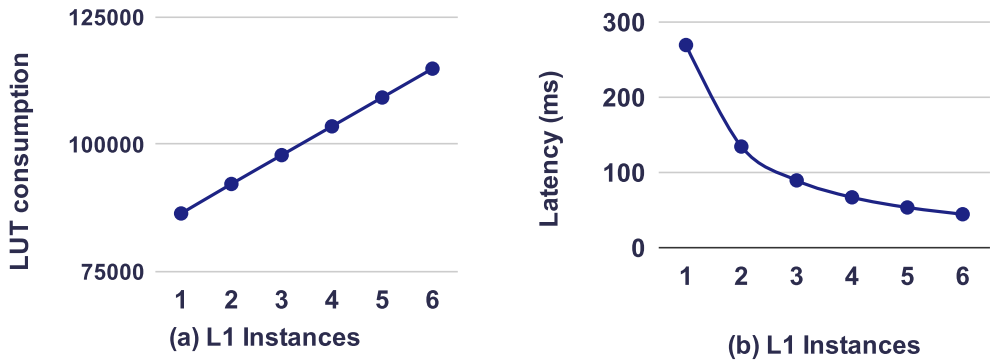


Figure 7.4: Graph (a) shows the latency of laplacian pyramid construction with the increase in the number of instances. Graph (b) shows the resource utilization of the hardware with the increase in the number of instances of the laplacian pyramid.

7.2 Results on Local-Laplacian

Experiments were conducted using a Virtex-7 FPGA connected to a 3 GHz Intel Core-i5 through a 3.5 GBPS PCIe-8x link. The host code, written in C++11, underwent compilation on a Ubuntu-16.4 system using GCC-5.4. The hardware synthesis functioned at a clock frequency of 100 MHz. The three Level Processing Units (LPUs) in the accelerator were synthesized as separate overlay IPs and seamlessly incorporated into the overall system design. The instantiation of LPUs was adaptable based on FPGA resources and memory bandwidth availability, allowing the creation of deeper LPUs capable of handling higher layers of Laplacian pyramids.

Table 7.2 illustrates that our accelerator accomplishes the processing of a 1-megapixel image in 534 milliseconds (ms), in contrast to the 4 seconds taken by the 8-core CPU implementation. This results in a 7.5x speedup over the baseline. The initial part of Table 7.1 provides PSNR values during detail enhancement and smoothing operations, with $\beta = 1$. As σ increases, there is a corresponding decrease in PSNR, attributed to the detail remapping function modifying more pixels, potentially including edge pixels. The latter part of Table 7.1 displays PSNR values during tone mapping and inverse tone mapping, with $\alpha = 1$. Here, a small σ results in the misclassification of pixels as edges, yielding low PSNR values. However, increasing σ reduces misclassification, enhancing overall PSNR values. Our approximation scheme for Local Laplacian Filtering ensures computational feasibility without compromising image quality, as assessed by PSNR or visual inspection.

With all three RGB channels, constituting nine processing units, operating on the Virtex-7 FPGA, the LUT resource consumption amounts to 19% of the total available. Given the independence of each pixel's computation in the output Laplacian pyramid, there is room for hardware scaling by replicating LPUs. Among the three levels, $\tilde{L}_1$ computes the maximum number of pixels but has the least resource consumption. Scaling $\tilde{L}_1$ by replicating it multiple times in the hardware significantly reduces overall

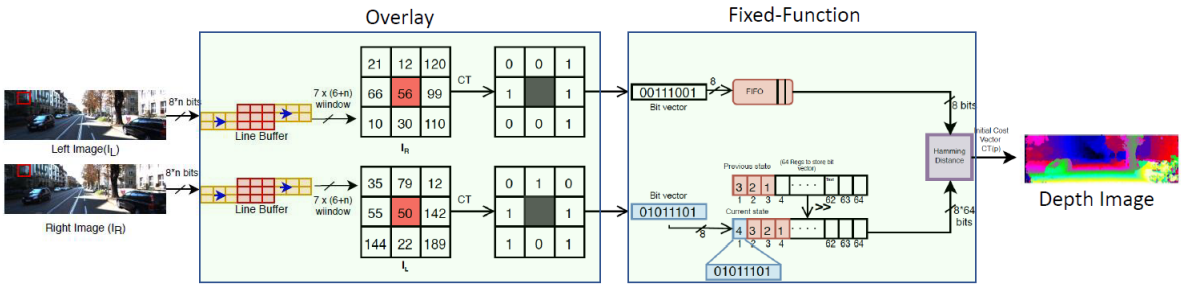


Figure 7.5: Figure shows the overall design of the Stereo Vision system using our overlay.

latency, as depicted in Figure 7.4 (b), with effective hardware scaling evidenced by the decrease in latency as the number of instances of $\tilde{L}_1$ increases. Figure 7.4 (a) illustrates the corresponding LUT usage.

7.3 Stereo Vision System

Stereo vision is a process that involves reconstructing the three-dimensional (3D) structure of a scene based on a pair of 2D images simultaneously captured from different viewpoints using cameras. The key steps include identifying corresponding pixels between the left and right images, rectifying the stereo pair to align these corresponding pixels, and computing the disparity, which represents the distance between a pixel in the right image and its corresponding pixel in the left image. The focal length of the stereo cameras is essential for computing the depth of a 3D point projected onto the corresponding pixel pair. The ultimate goal is to generate a disparity map, denoted as $D(x, y) \in \{0, \cdots, d_{max} - 1\}$ for a constant d_{max}, from a rectified stereo image pair. This disparity map is crucial for applications in fields such as robotics and driverless cars, where accurate 3D geometry estimation of the surroundings is essential.

Stereo correspondence algorithms, which play a central role in stereo vision, are categorized into three main types: local, global, and semi-global methods [80, 38]. Local methods determine pixel disparities by comparing neighbourhood windows in the right image with those of pixels on the same scan line in the left image, using metrics such as sum of squared intensity differences (SSD), the sum of absolute intensity differences (SAD), and census transform [10]. However, local methods may result in disparity maps with streaky artifacts. On the other hand, global methods aim to create smoother disparity maps by incorporating penalties for discontinuities in the image. These methods collectively contribute to the accurate reconstruction of 3D scenes, with applications in various fields.

7.3.1 Accelerator Implementation

The diagram in Figure 7.5 illustrates the comprehensive structure of our proposed accelerator for stereo vision. In this architecture, the CT module retrieves 5x5 windows from the line buffers of both

Table 7.3: Frames per Second for different image sizes with increasing accelerator PUs.

PU	**1226 x 3000**	**1226 x 960**	**1226 x 370**
1	25.8	80.7	209.1
2	51.6	161.3	417.8
3	77.4	241.9	626.7
4	103.2	322.1	834.3

Table 7.4: FPGA resource utilization and performance for 1226x960 resolution image.

PU	**MDE/s**	**KLUTs**	**MDE/KLUT**	**BRAM**
1	6279.53	56.008 (13%)	112.12	172 (12%)
2	12548.85	99.06 (23%)	126.68	328.5 (22%)
3	18822.55	163.388 (38%)	115.20	486 (33%)
4	25056.96	211.26 (49%)	118.61	641.5 (43%)

left and right images, performing the census transform on them. The census transform of a 5x5 window produces a 48-bit vector (excluding the central pixel). The bit vectors from the right image are collected in a FIFO with a capacity corresponding to the number of disparity levels ($d_{max} = 64$). Simultaneously, the bit vectors from the left image are stored in a register with a width of 64×48 (equivalent to 3072). As each 48-bit vector emerges from the left image, it undergoes a right shift into the corresponding register. Once the right FIFO is full, indicating the availability of all census transform vectors required for calculating the disparity of the first pixel in the FIFO, the Hamming Distance module removes the first element from the right FIFO. Subsequently, it reads the register to concurrently compute all 64 elements of the vector. The minimum value among these 64 is chosen as the disparity value for that pixel position. The computation of the Hamming distance and the minimum operation occurs within fixed-function hardware. Line buffering and census transform calculation takes place within a single instance of our overlay. As a recall from Chapter 4, the stencil section of the overlay comprises line buffer modules aiding in stencil computation, and the census transform is recognized as a stencil operation.

7.4 Results on Stereo-vision

Examining the augmentation of processing units (PUs) across various frame sizes with 64 disparity levels, as outlined in Table 7.3, reveals a consistent increase in throughput. The frames per second (FPS) exhibit a proportional rise corresponding to the integration of PUs into the hardware. For instance, in the case of a frame size of 1226x370, the FPS increases from 209 to 834 as the number of PUs scales from 1 to 4.

In the assessment of hardware architectures, resource utilization becomes a critical metric. Table 7.4 allows for a comparison of resource consumption, specifically in terms of KLUTs and BRAMs, as the number of PUs increases. The observed trend indicates a linear increment, with KLUTs and BRAMs scaling by 3.77 and 3.72 times, respectively, when transitioning from 1 PU to 4 PUs. Furthermore, this scaling is associated with a 3.99-fold increase in FPS.

7.5 Chapter Conclusion

In this chapter, we embarked on enhancing the capabilities of our image-processing overlay by integrating it with fixed-function designs, aiming to overcome the limitations of the basic design presented in Chapter 4. The integration involved the development of a basic wrapper with a FIFO-IN and FIFO-OUT interface, fostering compatibility with various fixed-function logic blocks. This hybrid approach showcased several advantages for the image-processing system, including the ability to accelerate a broader range of domain-specific algorithms and achieve heightened throughput with reduced latency.

The adoption of a standardized FIFO interface for fixed-function logic facilitated a streamlined integration process, promoting modularity in the overall system architecture. This modularity, exemplified through the design of two accelerator systems—a Local Laplacian pipeline and a Stereo Vision system—enhanced reusability and adaptability across different applications. The Local Laplacian Filtering algorithm, employed for detail enhancement, smoothing, tone mapping, and inverse tone mapping, showcased impressive results in terms of image quality and computational efficiency. The accelerator implementation demonstrated a significant speedup over a CPU baseline, underscoring the effectiveness of the proposed hybrid approach. Similarly, the Stereo Vision system, focusing on the reconstruction of 3D structures from a pair of 2D images, leveraged the overlay's capabilities for efficient processing. The accelerator implementation yielded notable improvements in throughput as the number of processing units scaled, demonstrating linear scaling in resource consumption with a corresponding increase in frames per second.

In conclusion, the integration of our image-processing overlay with fixed-function hardware proved to be a powerful strategy for extending the system's capabilities, enabling efficient acceleration of diverse algorithms. The modular and scalable nature of the hybrid approach positions the system for versatility across a range of applications, promising continued advancements in the realm of image processing and computer vision.

Chapter 8

Thesis Conclusion

The evolving landscape of hardware design in the wake of diminishing returns from traditional semiconductor scaling laws such as Moore's Law and Dennard scaling. The breakdown of these scaling predictions has led to a paradigm shift in the industry, with a growing emphasis on domain-specific accelerators to meet the computational demands of emerging applications like artificial intelligence and vision. The exploration of DSAs has highlighted the trade-offs between ASICs and FPGAs.

The thesis has delved into the evolving role of FPGAs, positioning them as a promising alternative to custom ASICs for designing DSAs due to their low power consumption and increased parallelism. Notably, the concept of overlay accelerators has been introduced as a flexible and adaptable approach, challenging the fixed-function hardware model tied to specific tasks. The proposed homogeneous overlay accelerator design methodology, as exemplified by FlowPix and FlexNN, addresses the limitations of heterogeneous overlays by optimizing throughput through parallel processing. FlowPix, tailored for image processing pipelines, and FlexNN, optimized for Convolutional Neural Networks (CNNs), showcase the potential of uniform, non-instruction-based overlays in achieving domain-specific acceleration. The experimental evaluations of FlowPix and FlexNN on FPGA platforms demonstrate their practical viability. FlowPix, benchmarked on a micro-benchmark image processing suite, exhibits competitive performance with minimal latency degradation. FlexNN, designed for CNN processing, achieves significant improvements in throughput and performance density compared to existing designs.

The thesis has also delved into the intricate design principles governing the data path of these accelerators. The data path, responsible for executing specific computations, plays a pivotal role in optimizing the performance-to-area ratio of DSAs. Several key principles have been established to guide the design of the data path, ensuring efficient processing and resource utilization. In the design DSAs, four fundamental principles guide the optimization of their data paths. Firstly, a common data path design is employed for shared processing units to minimize hardware redundancy and enhance FPGA real estate efficiency. Secondly, a parametric design approach is adopted, allowing for scalable adaptation by configuring parameters during synthesis, ensuring flexibility and efficient resource utilization across varying FPGA configurations. Thirdly, the imperative of maximizing data reuse is emphasized, reducing dependence on off-chip memory access to enhance overall performance and mitigate latency.

Lastly, the implementation of a distributed control memory, as demonstrated in the proposed overlays, optimizes dynamic configuration, balancing the trade-off between hardware cost and improved configuration latency. The practical implementations of FlowPix and FlexNN on FPGA platforms underscore the effectiveness of these principles. FlowPix, specialized for image processing pipelines, demonstrates competitive performance with minimal latency degradation. FlexNN, tailored for Convolutional Neural Networks (CNNs), substantially improves throughput and performance density over existing designs.

The findings presented in this thesis contribute valuable insights into the design and implementation of efficient overlays with a superior performance-to-area ratio. As hardware designers navigate the complexities of the post-Dennard, post-Moore world, the concepts and methodologies proposed herein pave the way for the development of adaptable and efficient solutions to address the ever-changing requirements of contemporary computing tasks.

8.1 Limitations and Future work

Both FlexNN and FlowPix overlays exhibit limited computing capabilities tailored to specific types of computations. FlexNN is designed for the efficient processing of CNNs on FPGAs, emphasizing CNN-specific operations and parallelism optimizations. But it is limited to Convolutions, Pooling and Activation functions only. New generation networks like Graph neural networks of LLMs can not be mapped easily. Similarly, FlowPix may not be as adaptable to a broad range of general computations containing image computations like histograms, temporal operations, etc.

Adopting DSL, in the context of the FlowPix overlay, poses challenges for users, primarily due to a steep learning curve as they acquaint themselves with the language's syntax, semantics, and conventions. Users may be hesitant due to concerns about the DSL's stability, and the transition costs, involving rewriting or adapting existing code, can be substantial. Additionally, the immaturity of the DSL's ecosystem, with limited libraries and tools, may restrict users' options and hinder widespread adoption.

Combining the FlexNN and FlowPix overlays presents a unique opportunity to leverage their strengths within a unified computational framework. This could be done since both the overlays have similar computational patterns. The flexibility of this combined approach allows for the seamless integration of other CNN structures using pointwise computational nodes. Similarly, the image processing pipelines with a huge budget can benefit from the big stencil array in the FlexNN overlay.

Bibliography

[1] *Fifth International Workshop on Computer Architectures for Machine Perception (CAMP 2000), September 11-13, 2000, Padova, Italy*. IEEE Computer Society, 2000.

[2] M. S. Abdelfattah and et.al. DLA: compiler and FPGA overlay for neural network inference acceleration. *CoRR*, abs/1807.06434, 2018.

[3] K. Abdelouahab, C. Bourrasset, M. Pelcat, F. Berry, J.-C. Quinton, and J. Serot. A holistic approach for optimizing dsp block utilization of a cnn implementation on fpga. In *Proceedings of the 10th International Conference on Distributed Smart Camera*, ICDSC '16, page 69–75, New York, NY, USA, 2016. Association for Computing Machinery.

[4] K. Abdelouahab, M. Pelcat, J. Sérot, F. Berry, C. Bourrasset, and J. Quinton. Hardware automated dataflow deployment of cnns. *CoRR*, abs/1705.04543, 2017.

[5] P. Amiri, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leißa, and S. Hack. Flower: A comprehensive dataflow compiler for high-level synthesis. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9. IEEE, 2021.

[6] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 271–276, New York, NY, USA, 2012. ACM.

[7] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. An Open CL deep learning accelerator on ARRIA 10, 2017.

[8] D. F. Bacon, R. Rabbah, and S. Shukla. Fpga programming for the masses. *Commun. ACM*, 56(4):56–63, Apr. 2013.

[9] 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*, 65(10):1415–1419, 2018.

[10] J. Banks, M. Bennamoun, and P. Corke. Non-parametric techniques for fast and robust stereo matching. In *TENCON'97 Brisbane-Australia. Proceedings of IEEE TENCON'97. IEEE Region 10 Annual Conference. Speech and Image Technologies for Computing and Telecommunications (Cat. No. 97CH36162)*, volume 1, pages 365–368. IEEE, 1997.

[11] M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[12] J. M. Cardoso and P. C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[13] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 247–257, New York, NY, USA, 2010. ACM.

[14] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[15] Z. Choudhury, S. Shrivastava, L. Ramapantulu, and S. Purini. An fpga overlay for cnn inference with fine-grained flexible parallelism. *ACM Trans. Archit. Code Optim.*, 19(3), may 2022.

[16] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. A dsl compiler for accelerating image processing pipelines on fpgas. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 327–338, New York, NY, USA, 2016. ACM.

[17] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. A DSL compiler for accelerating image processing pipelines on FPGAs. In *International Conference on Parallel Architectures and Compilation (PACT)*, pages 327–338, 2016.

[18] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang. Fpga hls today: Successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst.*, 15(4), aug 2022.

[19] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE TCAD*, page 491, 2011.

[20] Y. Dong, Y. Dou, and J. Zhou. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. M. P. Cardoso, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, 2007.

[21] D. Durst, M. Feldman, D. Huff, D. Akeley, R. Daly, G. L. Bernstein, M. Patrignani, K. Fatahalian, and P. Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–422, 2020.

[22] C. Elliott, V. Vijayakumar, W. Zink, and R. Hansen. National instruments labview: A programming environment for laboratory automation and measurement. *JALA: Journal of the Association for Laboratory Automation*, 12(1):17–24, 2007.

[23] W. et.al. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. DAC '17, New York, NY, USA. ACM.

[24] J. Fowers, G. Brown, P. Cooke, and G. Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. pages 47–56, 02 2012.

[25] J. Fowers and O. et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*.

[26] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017.

[27] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou. Maloc: A fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2601–2612, 2018.

[28] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[29] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *SIGPLAN Not.*, 39(7):249–256, June 2004.

[30] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient hardware code generation for FPGAs. *ACM Trans. Archit. Code Optim.*, 5(1):6:1–6:26, May 2008.

[31] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient hardware code generation for fpgas. *ACM Trans. Archit. Code Optim.*, 5(1):6:1–6:26, May 2008.

[32] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[33] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[34] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[35] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.*, 35(4):85:1–85:11, July 2016.

[36] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[37] The Heterogeneous Image Processing Acceleration Framework. http://hipacc-lang.org/.

[38] H. Hirschmuller. Stereo processing by semi-global matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):328–341, Feb 2008.

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[40] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[41] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9):50–59, 2018.

[42] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[43] N. Kapre. Custom fpga-based soft-processors for sparse graph acceleration. pages 9–16, 07 2015.

[44] V. Kathail. Xilinx vitis unified software platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–174, 2020.

[45] S. Khandelwal, Z. Choudhury, S. Shrivastava, and S. Purini. Accelerating local laplacian filters on fpgas. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 109–114. IEEE, 2020.

[46] R. Khemiri, S. Bouaafia, A. Bahba, M. Nasr, and F. E. Sayadi. Performance analysis of opencl and cuda programming models for the high efficiency video coding. In P. E. Ambrósio, editor, *Digital Image Processing Applications*, chapter 4. IntechOpen, Rijeka, 2021.

[47] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127, June 2016.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[50] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[51] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[52] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, 2016.

[53] J. Li, Y. Chi, and J. Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. pages 51–57, 02 2020.

[54] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji. High-performance fpga-based cnn accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1874–1885, 2019.

[55] Z. Liu, Y. Dou, J. Jiang, and J. Xu. Automatic code generation of convolutional neural networks in FPGA implementation. In Y. Song, S. Wang, B. Nelson, J. Li, and Y. Peng, editors, *2016 International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016*, pages 61–68. IEEE, 2016.

[56] Z. Liu, Y. Dou, J. Jiang, and J. Xu. Automatic code generation of convolutional neural networks in fpga implementation. *2016 International Conference on Field-Programmable Technology (FPT)*, pages 61–68, 2016.

[57] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25, 2019.

[58] S. Lym and M. Erez. Flexsa: Flexible systolic array architecture for efficient pruned DNN model training. *CoRR*, abs/2004.13027, 2020.

[59] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. Optimizing the convolution operation to accelerate deep neural networks on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1354–1367, 2018.

[60] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 45–54, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J. sun Seo. Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration, the VLSI Journal*, 62:14–23, June 2018. Funding Information: In this paper, ALAMO RTL compiler is proposed to accelerate CNNs on FPGA platforms, where the computing primitives could be easily compiled from the parametrized hardware library. Representative CNN algorithms of AlexNet and NiN have been demonstrated on an Altera Stratix-V FPGA board, which show an end-to-end throughput of 114.5 GOPS and 117.3 GOPS, resulting in 1.9X improvement compared to an optimized OpenCL design on the same FPGA board. Future work includes adopting techniques in Ref. [31 , 32] to increase the compiler's generality and efficiency of data and weight transfer for larger state-of-the-art CNN models [6 , 25 , 26]. A cknowledgment This work was in part supported by the National Science Foundation within the Directorate for Engineering under Grants 1230401 and 1237856 , the NSF I/UCRC Center for Embedded Systems through NSF grants 1361926 and 1432348 , NSF grant 1652866 , and Samsung Advanced Institute of Technology. Publisher Copyright: © 2017 Elsevier B.V.

[62] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[63] J. McCanny, J. McWhirter, and E. Swartzlander, Jr., editors. *Systolic Array Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[64] G. E. Moore. No exponential is forever: but" forever" can be delayed![semiconductor industry]. In *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, pages 20–23. IEEE, 2003.

[65] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, Mar. 2015.

[66] W. A. Najjar, A. P. W. Böhm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8):63–69, 2003.

[67] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato. A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 31–40, New York, NY, USA, 2018. Association for Computing Machinery.

[68] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. Lee. A high-throughput and power-efficient fpga implementation of yolo cnn for object detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1861–1873, 2019.

[69] R. S. Nikhil and Arvind. What is bluespec? *SIGDA Newsl.*, 39(1):1–1, Jan. 2009.

[70] R. S. Nikhil and Arvind. What is bluespec? *SIGDA Newsl.*, 39(1):1–1, Jan. 2009.

[71] M. A. Özkan, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leißa, S. Hack, J. Teich, and F. Hannig. Anyhls: High-level synthesis with partial evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3202–3214, 2020.

[72] P. R. Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[73] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu. Efficient compilation of cuda kernels for high-performance computing on fpgas. *ACM Trans. Embed. Comput. Syst.*, 13(2):25:1–25:26, Sept. 2013.

[74] S. Paris, S. W. Hasinoff, and J. Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. In *ACM Transactions on Graphics*, pages 68:1–68:12, 2011.

[75] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, Aug. 2017.

[76] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM.

[77] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, Dec. 2017.

[78] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[79] O. Reiche, M. A. Özkan, R. Membarth, J. Teich, and F. Hannig. Generating fpga-based image processing accelerators with hipacc. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1026–1033. IEEE, 2017.

[80] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1):7–42, 2002.

[81] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[82] Y. Shen, M. Ferdman, and P. A. Milder. Maximizing cnn accelerator efficiency through resource partitioning. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 535–547, 2017.

[83] F. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty. Ippro: Fpga based image processing processor. 10 2014.

[84] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[85] S. Sombatsiri, S. Shibata, Y. Kobayashi, H. Inoue, T. Takenaka, T. Hosomi, J. Yu, and Y. Takeuchi. Parallelism-flexible convolution core for sparse convolutional neural networks on fpga. *IPSJ Transactions on System LSI Design Methodology*, 12:22–37, 01 2019.

[86] G. Stitt, A. Gupta, M. N. Emas, D. Wilson, and A. Baylis. Scalable window generation for the intel broadwell+arria 10 and high-bandwidth fpga systems. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 173–182, New York, NY, USA, 2018. Association for Computing Machinery.

[87] T. J. Todman, G. A. Constantinides, S. J. Wilton, O. Mencer, W. Luk, and P. Y. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings-Computers and Digital Techniques*, 152(2):193–207, 2005.

[88] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[89] S. I. Venieris and C. Bouganis. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE Transactions on Neural Networks and Learning Systems*, 30(2):326–342, 2019.

[90] S. I. Venieris and C.-S. Bouganis. fpgaconvnet: Automated mapping of convolutional neural networks on fpgas (abstract only). In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 291–292, New York, NY, USA, 2017. ACM.

[91] S. I. Venieris, A. Kouris, and C.-S. Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Comput. Surv.*, 51(3):56:1–56:39, June 2018.

[92] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, et al. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 13–26, 2017.

[93] T. Wang, T. Geng, A. Li, X. Jin, and M. Herbordt. Fpdeep: Scalable acceleration of cnn training on deeply-pipelined fpga clusters. *IEEE Transactions on Computers*, 69(08):1143–1158, aug 2020.

[94] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. DAC '16.

[95] S. Wei and Y. Lu. The principle and progress of dynamically reconfigurable computing technologies. *Chinese Journal of Electronics*, 29(4):595–607, 2020.

[96] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong. Tgpa: Tile-grained pipeline architecture for low latency cnn inference. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, New York, NY, USA, 2018. Association for Computing Machinery.

[97] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong. Tgpa: Tile-grained pipeline architecture for low latency cnn inference. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[98] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[99] C. Wu, M. Wang, X. Chu, K. Wang, and L. He. Low precision floating point arithmetic for high performance fpga-based cnn acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 318, New York, NY, USA, 2020. Association for Computing Machinery.

[100] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan. A high-performance cnn processor based on fpga for mobilenets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 136–143, 2019.

[101] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He. Opu: An fpga-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020.

[102] Y. Yu, T. Zhao, K. Wang, and L. He. Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 122–132, 2020.

[103] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2016.

[104] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, 2016.

[105] J. Zhang and J. Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17. Association for Computing Machinery, 2017.

[106] S. Zhang, J. Cao, Q. Zhang, Q. Zhang, Y. Zhang, and Y. Wang. An fpga-based reconfigurable cnn accelerator for yolo. In *2020 IEEE 3rd International Conference on Electronics Technology (ICET)*, 2020.

[107] R. Zhao, H.-C. Ng, W. Luk, and X. Niu. Towards efficient convolutional neural network for domain-specific applications on fpga. pages 147–1477, 08 2018.

[108] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. Fpga-based accelerator design from a domain-specific language. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016.