Hardware-Software Co-Design of FPGA-based Neural Network Accelerators for Edge Inference

Jude Christudas Haris

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

2024

Abstract

The demand for efficient Deep Neural Network (DNN) accelerators has increased due to the growing popularity of DNNs in various applications, including image classification, speech recognition, and natural language processing. However, designing flexible, reconfigurable, and efficient DNN accelerators is challenging due to the computational intensity and memory requirements of DNN models. As such, Field-Programmable Gate Arrays (FPGAs) have become a popular choice for implementing DNN accelerators due to their ability to be reconfigured to suit the requirements of the workload and their energy efficiency compared to traditional general-purpose CPUs and GPUs. However, designing efficient accelerators for resource-constrained edge devices with FPGAs is challenging. As such, this thesis focuses on solving the difficulties of designing new efficient DNN accelerators for resource-constrained edge FPGAs.

First, this thesis presents the SECDA methodology (SystemC Enabled Co-design of DNN Accelerator), which enables hardware-software co-design of resource-constrained hardware accelerators for DNN inference on edge FPGAs. To expand upon the SECDA methodology, SECDA-TFLite and SECDA-LLM were developed to quickly adopt the design methodology within TensorFlow Lite and *llama.cpp*, two popular frameworks for DNN inference on edge devices.

Second, this thesis presents the design of the MM2IM architecture for accelerating Transposed Convolution (TCONV) operations within Generative Adversarial Networks (GANs) for resource-constrained edge devices. This architecture was developed utilising the SECDA methodology and the SECDA-TFLite toolkit. The MM2IM accelerator achieved an average speedup of 84× across 261 TFLite TCONV problem configurations compared to an ARM Neon-optimised CPU baseline.

Finally, this thesis presents AXI4MLIR, an extension to the MLIR compiler framework that enables efficient host-accelerator communication by automatically generating host driver code that is aware of the accelerator architecture and capable of performing efficient data transfers. Our experiments using specialised FPGA accelerators demonstrate AXI4MLIR's versatility across different types of accelerators and problems, showcasing significant CPU cache reference reductions (up to 56%) and up to a 1.65× speedup compared to manually optimised driver code implementations.

Acknowledgements

The depth of my gratitude cannot be expressed, but I will try. To Dr José Cano Reyes, I will be forever grateful. As my advisor during my postgraduate studies, he has gone far and beyond his responsibilities. His mentorship helped me understand the importance of scientific rigour and grow as a researcher. Thank you for believing in me even when I doubted myself and for pushing me to achieve my potential.

To my colleagues and friends, I thank you for suffering my unbearable rants. By simply listening, you helped me process many problems. Your time with me has helped me gain new and interesting perspectives in both research and life. I would like to express my appreciation towards Professor David Kaeli, who has consistently guided and supported me throughout my PhD journey. I am especially thankful to Dr. Perry Gibson and Nicolas Bohm Agostini, my seniors in studies. Your advice, expertise, and help were invaluable.

I would also like to thank Professor Wim Vanderbauwhede, along with my examiners, for their time and effort in reviewing and evaluating my thesis and (hopefully) getting excited about it. Additionally, I would like to thank those who have reviewed and provided feedback over the years. They have taught me the importance of constructive criticism in improving research.

To my guardian angel, you found the tiny little faith in me that I could not see and gave me hope. Finally, my dear family, who have been unwavering in their support, you have made me the person I am today. I am blessed beyond belief.

Thank you.

Declaration

I declare that this thesis was composed by myself and that the work contained herein is my own, except where explicitly stated otherwise in the text; and that this work has not been submitted for any other degree or professional qualification except as specified. The final part of this thesis, the AXI4MLR project, was conducted in collaboration with Nicolas Bohm Agostini. As such, we share co-first authorship, and it will form part of their PhD thesis. The start of the AXI4MLIR chapter will provide additional details of my exact contributions to the work. Some of the material used in this thesis has been published in the following papers:

- Jude Haris, Perry Gibson, José Cano, et al., 'SECDA: Efficient Hardware/-Software Co-Design of FPGA-based DNN Accelerators for Edge Inference', in IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct. 2021, pp. 33-43. DOI: 10.1109/SBAC-PAD53543.2021.00015. [Har+21]
- Jude Haris, Perry Gibson, José Cano, et al., 'SECDA-TFLite: A Toolkit for Efficient Development of FPGA-based DNN Accelerators for Edge Inference', in Journal of Parallel and Distributed Computing (JPDC), Vol. 173, Mar. 2023, pp. 140-151. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.11.005. [Har+23]
- Jude Haris, José Cano, et al., 'SECDA-LLM: Designing Efficient LLM Accelerators for Edge Devices', in ARC-LG Workshop at the 2024 International Symposium on Computer Architecture (ISCA). June 2024. [Har+24b]
- Nicolas Bohm Agostini, Jude Haris, et al., 'AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators', in 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Mar. 2024, pp. 143-157. DOI: 10.1109/CGO57630.2024.10444801. [Ago+24]

(*Jude Christudas Haris*)

To my parents, Philomina and Christudas, who without, I am nothing.

Table of Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

As technology advances, the availability and use of AI (Artificial Intelligence) based solutions for everyday tasks grow. The field of Machine Learning (ML) can contribute to this growth in AI-based solutions. ML is a branch of AI that uses algorithms to identify patterns and make sense of large data sets, in some sense imitating how humans learn by watching what others do. Deep Neural Networks (DNNs) are a class of machine learning (ML) models that enable improved accuracy on traditional classification (e.g., identify objects in a picture) and predictive tasks (e.g., guess the next number in a sequence). Due to their success in performing a wide range of tasks, e.g., image classification [KSH17], speech recognition [Zha+16b] and natural language processing (NLP) [SVL14], they have become increasingly popular and are being used to tackle real-world applications from the medical field [Ama+13; Fuk+18; Jum+21] to automatic plant counting [Fri+19].

The growth in DNN popularity and use cases has led researchers to understand that fundamentally DNN inference, i.e., the execution of DNN models, is computationally intensive, memory hungry [Cha+23] in terms of both capacity and bandwidth, and can incur high energy consumption. Hence, the increasing popularity of DNNs has become a major driving factor which has led to the development of specialised hardware accelerators that are used to efficiently process DNNs while minimising power, latency, and area. These accelerators include: GPUs [Don+21; Mar+18; Ott+20; Rad+19; RCO19], NPUs [Bou+20; Esm+12; Jan+21],CGRAs [Liu+19b; TK12], ASICs such as the TPU [Jou+17], NVDLA [NVIa] and many others [Che+19; KSK18], and FPGA-based designs [Mor+19; Pra+17].

Additionally, as the capabilities and accuracy of DNNs improved through the development of new model architectures and better training methods [HGC23; HHC24], edge computing applications for DNNs such as autonomous vehicles [Coc+21], IoT devices [Had+18], and mobile devices [GK22; LCO18] have become more prominent. This trend has led industry and academia to look towards efficient processing of DNNs in edge environments with resource-constrained devices. However, current solutions that attempt to deploy DNNs on low-power and resource-constrained edge devices

(e.g., smartphones, tablets, wearables, etc.) are inefficient [Had+19], as they are required to be low-latency and power-efficient to be a viable solution for edge DNN computing. This presents challenges that can span several levels of the hardware/-software stack to run efficiently [Gib+25; Tur+18]. Hence, new specialised hardware accelerators for DNNs are needed to meet the performance and energy targets within resource-constraint environments.

As the demand for specialised resource-constraint accelerators grows, the challenge of designing and deploying efficient DNN accelerators has become critical to enabling further progression and widespread adoption. An efficient accelerator design requires a careful co-design of the hardware and software components, particularly when the hardware resources are limited. Hence, to efficiently design resource-constrained hardware accelerators for DNNs, we need to consider the following aspects: (i) the target applications, including the variance in workloads and operations; (ii) the available hardware resources; (iii) the communication between the host CPU and accelerator.

This thesis focuses on the design process of specialised hardware accelerators for DNNs, especially focusing on these three key aspects. To end this, this thesis proposes that by reducing the time and effort required to iteratively design, integrate and evaluate new hardware accelerator architectures we save development time and resources. These improvements critically allow for more fine-grained exploration of the hardware-software co-design space, enabling more efficient end-to-end acceleration solutions with new hardware accelerators for DNN inference.

1.1 Designing FPGA-based DNN Accelerators

DNNs have become increasingly popular and are being used for a wide range of applications. DNN model architectures have also evolved, producing various types of DNNs such as Convolutional Neural Networks (CNNs) [Lec+98], Generative Adversarial Networks (GANs) [Goo+14], and Transformer-based models [Dev+19; Vas+17] including Large Language Models (LLMs) [Bro+20]. However, these models have been increasing in complexity and size, which has made it more and more challenging to deploy them, especially on resource-constrained edge devices. Additionally, the evolving nature of DNNs has made it challenging to execute these models efficiently on general-purpose hardware.

Hence, the need for efficient hardware accelerators to execute DNN workloads has grown. This has led researchers to utilise Field-Programmable Gate Arrays (FPGAs) to design, prototype and deploy specialised hardware accelerators for DNNs. FPGAs provide a reconfigurable fabric that can be programmed to implement new accelerator

architectures. However, designing specialised FPGA-based accelerators requires a lot of effort, time, and hardware-software expertise. The following section motivates the need for efficient hardware accelerators for DNNs and, in turn, the need to improve the design process of new FPGA-based hardware accelerators for DNNs. Additionally, the section presents the system model used throughout this thesis, highlighting key aspects that need to be considered when designing FPGA-based hardware accelerators for DNNs on resource-constrained edge devices.

1.1.1 Motivation

General-purpose computing has gained performance over the last couple of decades due to the increasing number of transistors that can fit into a microchip. However, this trend (i.e., Moore's Law) is slowing down. It has become more difficult to keep increasing the number of transistors without drastically increasing cost; hence, performance uplifts due to improved hardware are reduced compared to the demand for computational power. This is especially true with the advent of large computational workloads, which have become increasingly important and prevalent in machine learning. Hence, given the diminishing performance gains provided by today's general-purpose computing [JD18], there has been renewed interest in exploring specialised hardware accelerators.

Specialised accelerators can support architecture-level optimisations customised for a target application or workload, improving performance and efficiency [Hsi+23; KLL23; Muñ+23; Sha+23; Zha+22; Zhe+22]. Additionally, accelerators are more energy and power-efficient compared to general-purpose processors, which is critical for power-constrained edge devices. Hardware accelerators are also ideal for resource-constrained edge platforms, where we have to squeeze the highest performance out of the limited hardware resources available. The hardware logic allocated can be tailored to fit the target workloads, in our case, DNNs.

DNN inference is a compute-intensive workload, requiring different sets of operations such as matrix multiplications, covolutions, and activation functions. Generally, these operations contain a high degree of data-level parallelism, which hardware accelerators can exploit to achieve high performance. However, exploiting this parallelism requires a system-level hardware-software co-design approach for acceleration. Therefore, developing new specialised accelerators requires multiple design iterations with system-level integration and end-to-end evaluation to estimate realistic performance. FPGA-based design exploration allows full system integration and evaluation without the time-consuming efforts required to design and produce an Application-Specific Integrated Circuit (ASIC). Additionally, FPGAs allow for fine-grained control over the

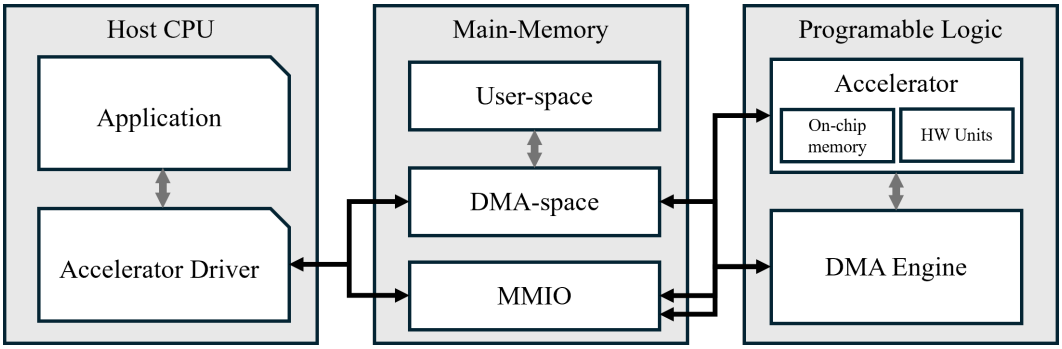


Figure 1.1: Typical FPGA-based host-accelerator system design.

hardware design compared to Coarse-Grained Reconfigurable Arrays (CGRAs) [TK12], enabling the designer to prototype intricately formed accelerator architectures; for example allowing bit-level data access for arbitrary arithmetic precision, which can be used to optimise designs for quantised DNNs.

1.1.2 Host-Accelerator System Model

This section provides a high-level overview of the system model that will represent the hardware platform we have used throughout the work contained within the thesis. The system model, shown in Figure 1.1, describes a typical resource-constrained FPGA-based edge system consisting of three main hardware components:

- **Host CPU:** The host processor (e.g., ARM-based), typically running a Linux-based operating system, runs target applications and controls the hardware accelerator.
- **Main Memory:** The main memory is used for data storage and transfer between the host processor and the hardware accelerator via Direct Memory Access (DMA) engines within the hardware platform.
- **FPGA:** The FPGA provides a reconfigurable logic consisting of Configurable Logic Blocks (CLBs), Look-Up Tables (LUTs), Flip-Flops (FFs), and Digital Signal Processors (DSPs). The FPGA is used to implement custom hardware accelerators for DNNs.

In additional to the hardware components, there are some system-level components that play a crucial role in the overall design of the acceleration pipeline, especially when designing for resource-constrained edge devices.

- **Application**: The target application running in the host CPU, which in our case consists of DNN inference. This application loads any required data into main memory, and executes the operation, i.e., DNN inference. The application calls the accelerator driver when the operation is suitable for hardware acceleration.
- **Accelerator Driver**: Responsible for managing both the DMA and the accelerator. The driver also implements the offloading algorithm for the data transfer between the main memory and the accelerator. This algorithm depends on the operation being executed, i.e., convolution, matrix multiplication, etc., and the accelerator architecture. The accelerator driver is a 'user-space' driver that interacts with the accelerator through memory-mapped data buffers and memory-mapped control registers for the DMA engine and the accelerator.
- **DMA Engine**: Responsible for transferring data between main memory and the accelerator. The DMA engine is controlled by the accelerator driver. Since the DMA engine sits outside the operating system, it can access main memory directly by using its physical address.
- **Accelerator**: Specialised hardware designed to accelerate the target operations within the application. It usually contains some control logic or a micro-ISA instruction decoder, data buffers, and multiple compute units.

In terms of memory management, the system model can be divided into four main memory spaces, which are used for different purposes:

- **User-space**: Area of main-memory where user programs can allocate and access data directly. The access to this area is controlled by the operating system and enabled through the virtual address space. DMA engines cannot access the virtual address space, so data needs to be moved to the DMA-space before being transferred to the accelerator.
- **DMA-space**: Area of main-memory allocated for the DMA memory pool during kernel boot-up. This allocation allows user-space drivers (the accelerator driver) to memory map (mmap) the physical address of the DMA-space memory pool. Moving user-space data into the memory mapped (mmapped) DMA-space allows the DMA engine to access data directly without requiring the CPU to be involved in the data transfer.
- **Memory Mapped Input/Output**: Area of virtual memory that is used to mmap the control/status register space for any connected device. In our system model, this can be used to control/monitor the DMA engine and the accelerator if required.

Table 1.1: Zynq 7020 Hardware Details.

Resource	Count	Resource	Count
CLBs	85,000	Processor	Dual Core Cortex-A9
LUTs	53,200	Architecture	ARMv7-A
FFs	106,400	Clock Speed	650 MHz
DSPs	220	SIMD	ARM NEON
BRAMs	140	Main Memory	512 MB DDR3
On-chip RAM (Mbits)	4.9	Memory Speed	525 MHz

- **On-chip Memory**: Area of block RAM (BRAM) on the FPGA that can be used as data buffers within the accelerator. This is the fastest and smallest memory space available in the system model and is used to store the data immediately required by the accelerator's compute units.

While this thesis emphasises the accelerator design and driver, the overall system model must be understood and considered when designing and evaluating new accelerator designs. The interactions between the system components and memory spaces are crucial to the overall system performance, especially when the main memory bandwidth can become a bottleneck for the accelerator performance. Managing efficient data caching and movement between user space, DMA space, and on-chip memory can significantly improve overall end-to-end performance.

Hardware Details

Throughout the thesis, we focus on the Xilinx Zynq 7000 SOC platform, specifically the Z-7020 device, which contains an ARM processor alongside an FPGA fabric, running an Ubuntu 18.04 operating system. Note all experiments conducted throughout the thesis runs the Zynq 7020 programming logic at 200 MHz and the ARM processor at 650 MHz, similarly we do not adjust for DVFS or other power management features when taking power measurements.

This platform will act as the reference for the resource-constrained edge system model as it contains limited hardware resources, which are listed in Table 1.1. Such resource-constrained edge platforms present an interesting set of challenges for designing FPGA-based hardware accelerators for DNNs, and this includes the acceleration paradigm possible by these devices, which is discussed in Section 1.1.3.

1.1.3 FPGA-based Acceleration Paradigms

Here, we discuss the possible acceleration paradigms with FPGA-based DNN inference accelerators. We can categorise the acceleration paradigms into two main categories: **DNN-to-FPGA-Dataflow** paradigm and **DNN-to-FPGA-Accelerator** paradigm.

The **DNN-to-FPGA-Dataflow**: paradigm is a high-level approach that allows the designer to take a DNN model and automatically generate a dataflow-based hardware design for the model or parts of the model. Works like HLS4ML [Fah+21] utilise this approach; these works take the DNN's low-level operations (multiplications, additions, etc.) and map them to the FPGA fabric directly using dataflow graphs. While this approach is easier to use, it has significant limitations in terms of capability since mapping a full model to an FPGA requires hardware resources to support it fully. For example, a model with 20 MB of weight data will require at least 20 MB of on-chip memory to store the weights, which is impossible with resource-constrained edge FPGAs such as the Zynq 7020. This makes the approach either infeasible or inefficient for large models to be mapped to the FPGA. Additionally, the generated hardware design is usually specific to the model, making it difficult to reuse it across multiple models.

The **DNN-to-FPGA-Accelerator**: paradigm is a low-level approach that requires the designer to custom hardware accelerators which can be used to accelerate parts of the DNN model, usually targeting the compute-intensive operations such as matrix multiplications or convolutions. While the hardware-software accelerator solution becomes more complex and requires more effort, resource-constrained edge FPGAs can be used more efficiently and accelerate larger models. Returning to our previous example, a DNN model with 20 MB of weight data across can be accelerated within the Zynq 7020 by offloading the key compute-intensive operations to the accelerator while the rest of the model can be executed on the host CPU. This allows the designer to use the FPGA's hardware resources more efficiently and allows the accelerator to be reused across multiple models.

1.2 Challenges and Objectives

As highlighted in Section 1.1.1, several challenges should be addressed to efficiently design, integrate, and deploy new custom hardware accelerators. This section outlines the three key challenges faced when designing DNN hardware accelerators within the system model described in Section 1.1.2. Tackling and providing solutions to these key challenges is the main goal of the thesis. Hence, this section highlights the three key objectives that the thesis addresses to overcome these challenges.

1.2.1 Development Time of New Accelerators

As new types of workloads emerge within the DNN domain, it is crucial to develop new hardware accelerators quickly and efficiently. For accelerator development, we need to consider the time to design, implement, and verify the hardware accelerator.

The traditional design process for a new specialised accelerator involves several steps, such as architectural exploration, RTL (Register Transfer Level) design, verification, and synthesis. This process is very time-consuming and requires a high level of expertise in hardware design. Development through RTL design requires a hardware expert to design the accelerator and a software expert to develop the host-accelerator communication code. In addition, handwritten RTL code is highly error-prone and can lead to difficult debugging bugs. Finally, the development process must be more scalable, as the design process needs to be repeated for each new accelerator design.

When developing new accelerators through multiple iterations, design space exploration (DSE) can provide valuable insights between iterations. Therefore, it is crucial to reach a good design point where all crucial performance targets for a given accelerator are met. Additionally, a DSE approach that supports hardware-software co-design enables further performance gains by tuning both the hardware and software aspects of the acceleration solution.

Hence, the **first key objective** of the thesis is to create a high-level design methodology that enables a fast and iterative design loop and allows the evaluation and profiling of both the accelerator and the overall end-to-end application.

1.2.2 Problem Specific Design and Optimisations

As mentioned within Section 1.1.1, general-purpose computing has performance limitations, and hence, specialised accelerators for a given workload can provide significant performance and energy efficiency improvements. One major challenge, especially in a resource-constrained hardware platform, is understanding the target workload and partitioning it between the host processor and the hardware accelerator. Once the target operation is identified, the designer must consider operation-specific optimisations to improve the accelerator's performance and efficiency. These optimisations can include operation fusion, data quantisation, data formatting, tiling, and dataflow strategies. Therefore, the challenge is to understand the target workload, identify the key operations that can be offloaded to the hardware accelerator, and apply the necessary optimisations to improve the accelerator's performance and energy efficiency.

Hence, the **second key objective** of this thesis is to develop problem specific accelerator designs by exploring the architectural design space that is unique to DNN-based workloads.

1.2.3 Efficient Host-Accelerator Communication

As DNN models become more and more complex and scaled, the amount of data that needs to be processed grows significantly. This especially impacts end-to-end performance for resource-constrained edge devices, where due to the limited accelerator on-chip memory, input data needs to be partitioned and transferred from the main memory to the accelerator on-chip memory. Since data transfer between main memory and the accelerator can take an order of magnitude more than the number of cycles required to perform a computational operation, it is crucial to minimise data transfers. Otherwise, the required data transfer can become the main bottleneck in the overall system model if not managed efficiently.

Therefore, to reduce the effect of data transfer overheads, we need to consider the following challenges: (i) how to reduce the amount of data that needs to be transferred between the host processor and hardware accelerator; (ii) how to hide the data transfer latency by overlapping the data transfer with computation; (iii) how to format the data so that the accelerator can efficiently process it.

Hence, the **third key objective** of the thesis is to develop methods and optimisations to improve the host-accelerator communication that solves these challenges, ensuring that we can achieve high performance and efficiency for DNN inference workloads.

1.3 Contributions

The key contributions of this thesis are as follows:

- **SECDA Methodology:** A new design methodology, SECDA (SystemC Enabled Co-design of DNN Accelerator), which enables hardware-software co-design of specialised hardware accelerators for DNN inference on resource-constrained edge devices with FPGAs. Chapter 4 presents the design methodology, which focuses on iterative design exploration, co-design of the accelerator (hardware) and host-side driver code (software), and continuous evaluation of new hardware designs using SystemC [Des23] simulation and FPGA-based hardware evaluation. The SECDA methodology is instantiated across multiple tools and hardware accelerators developed in the thesis. Overall, SECDA provides a structured approach to co-design of new hardware accelerators and their host-side

driver for a given application and target FPGA platform. Open-source link: `https://github.com/gicLAB/SECDA`.

- **Toolkits:** The thesis presents multiple toolkits that employ the SECDA methodology. The toolkits, presented in Chapters 5, 6 and 8, are designed around and integrated with existing application frameworks. We focused on machine learning and compiler frameworks such as TensorFlow Lite (TFLite) [Aba+16], *llama.cpp* [Ger24b] and MLIR [Lat+21]. The toolkits provide a rapid prototyping environment for the target application framework, enabling developers to design new hardware accelerators, build host-side driver code that is integrated with the application framework, and execute the end-to-end application while offloading the target operations to the hardware accelerator. Open-source links for SECDA-TFLite: `https://github.com/gicLAB/SECDA-TFLite`, AXI4MLIR: `https://github.com/AXI4MLIR/axi4mlir`, and SECDA-LLM: `https://github.com/gicLAB/SECDA-LLM`.

- **Hardware Accelerators:** Multiple hardware accelerators for various DNN models and operations. These accelerators were developed to tackle the problem of accelerating compute-intensive DNN operations on resource constrained edge devices with FPGAs. Chapter 4 presents the Vector MAC (VM) and Systolic Array (SA), both accelerators where designed to accelerate convolutional layers using two varying computational architecture. Chapter 5 presents the Fully Connected General Matrix Multiply (FC-GEMM) accelerator that was designed to accelerate fully connected layers which are common within Transformer-based models. Chapter 6 presents the block floating point quantised (BFP) accelerator that was designed to accelerate matrix multiplication operations Finally, Chapter 7 presents the design of the **M**at**M**ul to col**2IM** [Devc] (MM2IM) hardware accelerator for GANs. MM2IM was designed to efficiently execute the Transposed Convolution (TCONV) operation, which is a key operation in GANs.

1.4 Publications

The work presented in this thesis has been cultivated from ideas, explorations, and experiments which have been previously described in detail in several peer-reviewed publications. Additionally, some of the work is currently under review or in preparation for submission.

The following publications are associated with Chapters 4, 5 and 6.

1. Jude Haris, Perry Gibson, José Cano, et al., 'SECDA: Efficient Hardware/-Software Co-Design of FPGA-based DNN Accelerators for Edge Inference', in IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct. 2021, pp. 33-43. DOI: 10.1109/SBAC-PAD53543.2021.00015. [Har+21]

2. Jude Haris, Perry Gibson, José Cano, et al., 'SECDA-TFLite: A Toolkit for Efficient Development of FPGA-based DNN Accelerators for Edge Inference", in Journal of Parallel and Distributed Computing (JPDC), Vol. 173, Mar. 2023, pp. 140-151. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.11.005. [Har+23]

3. Jude Haris, José Cano, et al., 'SECDA-LLM: Designing Efficient LLM Accelerators for Edge Devices', in Addressing the Computing Requirements of LLMs and GNNs Workshop (ARC-LG) at the 2024 International Symposium on Computer Architecture (ISCA). June 2024. DOI: 10.48550/arXiv.2408.00462. [Har+24b]

Note that the SECDA-LLM work is an ongoing work, and we plan to expand the initial work presented in the publication.

The following publications are associated with Chapter 8:

1. Nicolas Bohm Agostini, Jude Haris, et al., 'AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators', in 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Mar. 2024, pp. 143-157. DOI: 10.1109/CGO57630.2024.10444801. [Ago+24]

2. Jude Haris, José Cano, et al., 'Data Transfer Optimizations for Host CPU and Accelerators in AXI4MLIR', in Compilers for Machine Learning Workshop (C4ML) at the 2024 International Symposium on Code Generation and Optimization (CGO), Mar. 2024. DOI: 10.48550/arXiv.2402.19184. [Har+24a]

Note that the AXI4MLIR, first paper for the AXI4MLIR project, was conducted in collaboration with Nicolas Bohm Agostini. We share co-first authorship, and it will form part of their PhD thesis.

The following work associated with Chapter 7 is a work in progress, and the initial paper is currently under review.

1. Jude Haris and José Cano, 'Accelerating Transposed Convolutions on FPGA-based Edge Devices', under review at International Conference on Field-Programmable Logic and Applications (FPL) 2025.

Finally, the following publication is complementary to the work presented in this thesis. It is not directly related to the thesis, but provides additional context and background to the work presented in this thesis.

1. Rappy Saha, Jude Haris and José Cano, 'Accelerating PoT Quantization on Edge Devices', in IEEE International Conference on Electronics Circuits and Systems (ICECS). 2024.

1.5 Thesis Structure

The rest of the thesis is structured as follows:

Chapter 2 provides an overview of the background concepts required to understand the thesis. The chapter covers the basics of Deep Neural Networks (DNNs), key DNN algorithms, popular DNN frameworks, hardware accelerators, optimisations, and co-design methodologies.

Chapter 3 provides a review of the related work in the field of hardware accelerators for DNN inference. The chapter covers the state-of-the-art in DNN accelerators, optimisations, and co-design methodologies.

Chapter 4 presents the SECDA design methodology, which enables hardware-software co-design of specialised hardware accelerators for DNN inference on resource-constrained edge devices with FPGAs. The chapter contains a case study to demonstrate the SECDA methodology, where we design and evaluate two hardware accelerators for convolutional layers: the Vector MAC (VM) and Systolic Array (SA) accelerators.

Chapter 5 presents the SECDA-TFLite toolkit, which employs the SECDA design methodology to design and evaluate hardware accelerators for TFLite models. In this chapter, we highlight the integration of the SECDA with the TFLite delegate system, which allows for SECDA design methodology to be followed easily by hardware designers while developing custom hardware accelerators for TFLite models. We expand the original case study presented in Chapter 4 to include the design of a new hardware accelerator for Fully Connected layers: the FC-GEMM accelerator, along with updates to the VM and SA accelerators to be compatible with the TFLite delegate system.

Chapter 6 presents the SECDA-LLM toolkit, which employs the SECDA design methodology to design and evaluate hardware accelerators for LLMs, using the *llama.cpp* framework. The chapter contains a case study to demonstrate the SECDA-LLM toolkit by designing and evaluating a hardware accelerator for LLMs, specifically

targeting Matrix Multiplication operations that use Block Floating Point (BFP) quantisation. The case study also highlights the importance of iterative design space exploration using the SECDA methodology.

Chapter 7 presents the design and evaluation of a hardware accelerator for Generative Adversarial Networks (GANs). The chapter focuses on the Transposed Convolution operation, the key operation in GANs. Our new accelerator design, dubbed 'MM2IM', provides an efficient hardware-software co-designed solution for the Transposed Convolution operation on resource-constrained edge FPGAs.

Chapter 8 presents AXI4MLIR, an extension to the MLIR compiler framework that enables efficient host-accelerator communication. The chapter focuses on automatic code generation of host driver code for specialised hardware accelerators and highlights the importance of efficient host driver code to fully exploit the capabilities of custom hardware accelerators.

Chapter 9 concludes the thesis by providing a summary of the key contributions, a discussion about future work and a reflection on the PhD journey.

1.6 Summary

The demand for high-performance computing continues to grow, and hardware accelerators have become increasingly popular, especially in the context of Deep Neural Networks (DNNs), which are computationally intensive and require high memory bandwidth. This chapter introduced the motivations and challenges of designing efficient hardware accelerators for DNN inference on resource-constrained edge devices.

The chapter also presented the thesis's contributions, including a new design methodology for hardware-software co-design (SECDA), multiple toolkits that employ the design methodology, and hardware accelerators for various DNN models and operations.

2 Background

This chapter provides the general overview of all on key concepts of this thesis in terms of designing DNN accelerators for resources-constrained edge devices on FPGAs. The chapter first introduces the general concepts of DNNs and some key algorithms. Then it presents a broad view of the hardware aspect of DNN acceleration, expanding on the system model 1.1, and focusing on the key concepts of hardware development. Finally, it discusses key hardware-software co-design techniques and optimisations required for efficient DNN acceleration.

2.1 Deep Neural Networks

A neural network is a computational model that is inspired by the human brain's neural structure. The most basic form contains three layers: input, hidden, and output. The input layer receives the input data, the hidden layers process the input using some learned bias and weights, and the output layer produces the final output. Deep Neural Networks (DNNs) is a class of machine learning (ML) models that depend on increased depth of neural network hidden layers to improve accuracy on traditional classification and predictive tasks. DNNs are exploding in terms of research activity and adoption in consumer and industrial processes. DNN models are being created and engineered to solve a wide variety of tasks over many varied problem domains; these models can range from solutions for autonomous driving [Coc+21] to protein folding [Jum+21], and simpler tasks such as image classification [How+17], speech recognition [Abd+14; Zha+16b], mobile malware detection [MMM17] and computer vision in robotics[PG16].

2.1.1 DNN Fundamentals

Model Architecture

DNNs are composed of 'layers'. There are different types of layers, such as Convolutional layers (CONV), Fully Connected layers (FC), Pooling layers, and Non-linearity

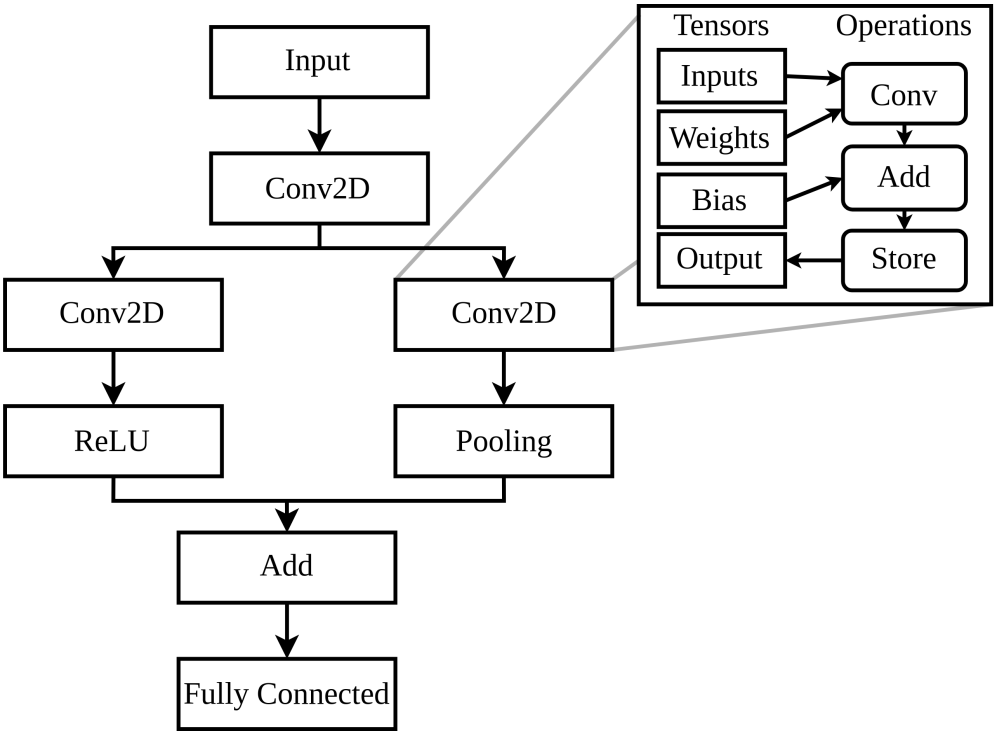


Figure 2.1: Example of a DNN model architecture.

layers. Within these layers, operations are performed on the input data. Additionally, the layers can contain a set of weights, which alter the inputs to produce the outputs. The set of outputs from the previous layer is used as inputs for the next layer, and this process continues until the final layer produces the model's output. Within this thesis, for a given DNN, we refer to the layers and how they are connected as the 'model architecture'. The wide variety of model architectures enables DNNs to be used in various applications. Figure 2.1 shows an example of a DNN model architecture, where one of *CONV2D* is expanded to show the components of a typical 'layer'. For example, within this *CONV2D* layer, there are 4 tensors: (i) the input activations, which is normally the output tensor from the previous layer; (ii) the weights, which contain the learned parameters; (iii) the bias, which adds a constant value to the output of the convolution operation; and (iv) the output activations, which is where the output of the *CONV2D* layer is stored. The *CONV2D* layer also consists of operations: (i) the convolution operation, which is performed on the input activations using the weights; (ii) the bias addition operation, which adds the bias to the output of the convolution operation; (iii) and the store operation, which usually contains post-processing operations and stores the output activations to the output tensor.

Training and Inference

There are two fundamental processes that apply to DNN models: training, which is the learning phase; and inference using the trained model, which is the prediction phase. Once a DNN model architecture is defined, usually through the use of a DNN development framework such as TensorFlow[Aba+16] or PyTorch [Pas+17], the developer can 'train' the DNN model. The training usually involves running the model on a large training data set and adjusting the model weights to minimise the error between the model's output and the expected output, i.e., the ground truth. Adjusting the weights enables models to learn key features from the training dataset. The training phase is very computationally expensive and memory-demanding; due to the large data set, higher memory capacity is needed to ensure that data does not need to be fetched from devices further away and that computational device(s) do not stall waiting for data transfers. For this reason, state-of-the-art solutions currently use GPUs to perform the process, as they have high memory capacity and bandwidth and can perform up to trillions of floating-point operations per second (TFLOPs). Then, the end user can perform inference with the trained models, which is usually a prediction/classification/generation-based task; the tasks vary depending on the user's needs. Training dictates the accuracy of the model, i.e., how well the model performs the target task on new unseen data. A simple measure of accuracy for classification tasks is the percentage of correctly classified samples, usually measured as Top-1 or Top-5 accuracy, which is how often a model picks the correct class within the Top-1 or Top-5 of its choices.

Inference varies depending on the end-user problem domain and task they want to tackle; for example, a CNN model can perform image classification inference to classify different species of birds, or a generative model can synthesise a piece of abstract art from textual description. Nevertheless, the inference process is much less computationally demanding than the DNN training phase since significantly less data needs to be processed to make a prediction or generate a new output. Due to inference's less resource-demanding nature, unlike training, resource-constrained edge devices can used for inference tasks without a significant performance penalty. This enables DNN inference to be performed using general-purpose CPUs and GPUs but can come with the drawback of high latency or higher power consumption per inference. Therefore, hardware acceleration of DNN inference via specialised hardware, such as the TPU [Jou+17] ASIC and FPGA-based accelerators, is being explored to achieve optimal performance based on the variance of the DNN workloads and devices available. While inference is less resource-demanding than training, it can still be computationally expensive and power-hungry relative to the capabilities of resource-constrained devices. As such,

many factors need to be considered, e.g., the model size, batch processing, quantisation, sparsity, memory bandwidth, and many more.

This thesis focuses on the inference phase, specifically on accelerating DNN inference using resource-constrained edge devices with specialised hardware accelerators.

2.1.2 Types of DNNs

There are varying types of DNNs and even different ranges of shapes and sizes within the same type of DNN. Examples of the range of DNNs include the large convolutional layers of InceptionV3 [Sze+16], as compared to the small depth-wise separable convolutions of MobileNets [How+17]; or the 'Residual Blocks' of ResNets [He+16], as compared to the 'Attention Head' of the transformer-based architecture [Vas+17] in models such as BERT [Dev+19]. This thesis focuses on the following types of DNNs: Convolutional Neural Networks (CNNs), Transformer models, and Generative models. While these models differ in architecture, they share common computational primitives/operations, such as convolution. The key computational primitives/operations will be further discussed in Section 2.4.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [KSH17] are primarily used for image classification tasks. CNNs are mainly composed of Convolutional layers (CONV), which are used to extract features from the input data. The CONV layer performs the convolution operation on the input data using a set of filters (kernels) to produce a feature map. Since CONV layers are the most computationally intensive layers in CNNs, it becomes essential to maximise the convolution operation's efficiency to improve a CNN's overall performance. Convolutions are sometimes referred to as downsampling since the feature map produced is smaller than the input data.

CNNs usually contain one or more fully connected (FC), pooling, and non-linearity layers. Fully Connected layers (FC) use all the output activations from the previous layer when processing each production of the current layer. In image classification, FC layers are only used for the last few layers to obtain the class label with the highest percentage. The pooling layers are used to reduce the dimensions of the feature maps using operators such as max or average operators. Finally, non-linearity layers (e.g., ReLU) are added to add non-linearity to the DNN model. In Chapter 4 and Chapter 5, we accelerate CNNs through the design of a specialised hardware accelerator for various quantised CNN models.

Transformer Models

The Transformer architecture [Vas+17], and its successors such as BERT (Bidirectional Encoder Representations from Transformers) [Dev+19], have achieved state-of-the-art performance in a range of Natural Language Processing (NLP) tasks. These models primarily consist of transformer encoder layers. The encoder layer consists of an attention layer and an FC layer followed by some normalisation layer. While Transformer and BERT-based models can still be considered new to the field of DNNs, research and development on hardware accelerators have already begun [Pat+22]. Considering the scale of these models in terms of memory and computational demands, they have greater room for optimisations and performance gains through the use of well-defined hardware accelerators. In Chapter 5 and Chapter 6, we focus on the acceleration of transformer-based models through the design of specialised hardware accelerators for BERT models and LLMs, respectively.

Generative Models

Generative models[XLZ15] can be described as a family of DNN models where the task of the models is to generate new data. This ranges from generating images of faces similar to celebrities[Kar+18b] to generating a new piece of music [Dha+20]. Generative models comprise of Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), Diffusion models and many more. While these models may contain traditional CONV and FC layers similar to CNNs and Transformers, they can also contain new types of layers to upscale the input data. For example, new layers such as Transposed Convolution layers present different properties requiring new custom hardware accelerator designs. In Chapter 7, we focus on the acceleration of GANs through the design of a specialised hardware accelerator capable of efficiently performing the transposed convolution operation.

2.2 Software Libraries

Throughout this thesis, we look at different software tools and frameworks as the frontend to define and execute our target workload, mainly DNN inference. In this section, we focus mostly on machine learning (ML) frameworks, which are software libraries that provide a set of APIs to enable the development and deployment of DNN models.

Additionally, we look at MLIR (Multi-Level Intermediate Representation) [Lat+21]

which is a compiler infrastructure framework that enables the development of domain-specific compilers.

2.2.1 TensorFlow & TensorFlow Lite

TensorFlow [Aba+16] is an open-source ML framework developed by Google. Using TensorFlow, developers can define, train, and deploy machine learning models. TensorFlow provides a set of APIs for defining the model architecture, the training process (including the loss function and optimiser), and the deployment of the model for inference.

TensorFlow Lite (TFLite) is a lightweight version of TensorFlow designed for mobile and edge devices. TFLite is deployed on devices with limited computational resources, such as smartphones, tablets, wearables, and microcontrollers. For example, TFLite supports a range of quantisation formats, including 8-bit integer quantisation and 16-bit floating-point quantisation, which allows the model size to be reduced to ensure they fit in the memory of the target resource-constrained device. TFLite consists of several smaller tools, such as the 'TFLite converter', which enables the conversion of TF models to TFLite models, and the 'TFLite benchmark model' tool, which can be used to evaluate TFLite models.

Within this thesis, we use TensorFlow to acquire DNN models and generate test workloads. As for TFLite, we use the TFLite tools to convert, run and evaluate the models. Additionally, we take advantage of the TFLite delegate system (see section 5.2.1) to easily integrate new hardware within TFLite; see Chapter 5 for more information on how we use TFLite to not only deploy models on custom FPGA-based accelerators but also to design new FPGA-based accelerators for TFLite-based DNN inference.

2.2.2 *llama.cpp*

llama.cpp [Ger24b] is an ML framework for LLM inference. Its main goal is to enable LLM inference with minimal effort and setup while providing the best performance on various hardware devices. This project has increased the usability of edge-based LLM inference by providing simple utilities to load, run, evaluate and interact (for example, via chatbot) with pre-trained LLMs. Note that the simplicity of *llama.cpp* lies in its pure C/C++ library, with minimal external dependencies, enabling LLMs inference on a wide range of hardware that supports standard C/C++ compilers.

Currently, *llama.cpp* supports a wide range of LLMs, including some multi-modal and custom-defined models. *llama.cpp* [Ger24b] employs the GPT-Generated Unified Format model format (GGUF) to represent LLMs. In this format, using BFP quantisation,

it is possible to represent the weights of an LLM with as few as 1.5 bits. These quantised weights enable users to run the model on resource-constrained devices such as the Raspberry Pi and Pixel phone [Sim]. In *llama.cpp* implementation, BFP quantisation is leveraged to quantise the weights of the LLMs, and it includes a few quantisation variations, such as 1.5-bit, 2-bit, 3-bit, 4-bit, 5-bit, 6-bit, and 8-bit BFP quantisation. These variations are typically denoted as Qx_y, where x represents the number of bits per weight and y denotes the type of quantisation, see Section 2.4.4 for more information on BFP quantisation.

Enabling LLM inference to work has become more straight forward on resource-constrained CPUs or GPUs due to the optimised support provided by *llama.cpp* for AVX, AVX2 and AVX-512 on x86 architectures as well as custom CUDA kernels for running on NVIDIA GPUs. However, LLM inference on FPGAs is not straightforward, as the design process for new FPGA-based accelerators has yet to be integrated with inference platforms like *llama.cpp*.

Within this thesis, we use *llama.cpp* to design and evaluate new FPGA-based accelerators for LLM inference across different LLMs; see Chapter 6 for further details.

2.2.3 MLIR

In addition to the ML frameworks, in this thesis we utilise MLIR [Lat+21] to enable and explore hardware-software co-design when designing new FPGA-based accelerators, specifically for the AXI4MLIR work presented in Chapter 8. MLIR is a compiler infrastructure that facilitates the creation of domain-specific compilers by providing code generators, translators, optimisers, and the infrastructure to define subsets of operations that expose well-defined language abstractions. MLIR is designed to easily allow progressive lowering between current and new operations, promoting the reuse of abstraction levels and compiler passes already implemented in the framework.

MLIR is composed of a set of dialects, each built up by a set of operations, types, and attributes. Dialects are used to capture the semantics of a specific layer of abstraction. This can be at a low level, such as the 'amdgpu' dialect which contains operations specific to AMD GPUs, or at a higher level, such as the 'tf' dialect which contains mapping MLIR operations to TensorFlow operations. While users can create new dialects, some 'core' dialects are used to define common operations, such as the 'arith' dialect, which contains basic arithmetic operations.

```
#matmul_trait = {
  indexing_maps = [
    affine_map<(m, n, k) -> (m, k)>, // A
    affine_map<(m, n, k) -> (k, n)>, // B
    affine_map<(m, n, k) -> (m, n)>  // C
  ]
  iterator_types = [
    "parallel", "parallel", "reduction"
  ],
}
func.func @matmul_call(...) {
  linalg.generic #matmul_trait
    ins (%A, %B : memref<60x80xf32>, memref<80x72xf32>)
    outs(%C : memref<60x72xf32>) {
    ^bb0(%a: f32, %b: f32, %c: f32):
      %0 = arith.mulf %a, %b : f32
      %1 = arith.addf %c, %0 : f32
      linalg.yield %1 : f32 }
  return }
```

(a) Linalg Abstraction with generic operation.

```
func.func @matmul_call(...) {
  // Declare constants %c0 %c1 %c4 %c60 %c72 %c80 ...
  scf.for %m = %c0 to %c60 step %c4 { // Tiling by 4,4,4
    scf.for %n = %c0 to %c72 step %c4 {
      scf.for %k = %c0 to %c80 step %c4 {
        // Grab handle for the sub-tiles:
        %sA = memref.subview %A[%m, %k] [4, 4] [1, 1] : ...
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1] : ...
        %sC = memref.subview %C[%m, %n] [4, 4] [1, 1] : ...
        // Matmul computation of a 4x4x4 tile:
        scf.for %mm = %c0 to %c4 step %c1 {
          scf.for %nn = %c0 to %c4 step %c1 {
            scf.for %kk = %c0 to %c4 step %c1 {
              %3 = memref.load %sA[%mm, %kk] : !mr4x4_0
              %4 = memref.load %sB[%kk, %nn] : !mr4x4_1
              %5 = memref.load %sC[%mm, %nn] : !mr4x4_1
              %6 = arith.mulf %3, %4 : f32
              %7 = arith.addf %5, %6 : f32
              memref.store %7, %sC[%mm, %nn] : !mr4x4_1
  } } } } } }
  return }
```

(b) Structured Control Flow (SCF) abstraction with tiling.

Figure 2.2: MLIR representations of a Matrix-Matrix Multiplication Operation in different abstractions.

```
1 typedef struct {
2 float *allocated; // For deallocation
3 float *aligned;   // Base address
4 size_t offset;    // Offset in # of elements
5 size_t size[N];   // One size per dim
6 size_t stride[N]; // One stride per dim
7 }
```

Figure 2.3: The underlying data structure of a rank==N MLIR `memref` buffer.

Linalg Dialect

In support of the underlying algorithms and kernels used by many machine learning frameworks (e.g., TensorFlow and PyTorch), MLIR offers a linear algebra dialect called `linalg` that exposes (named) operations such as convolutions, matrix multiplications, and others. Operations expressed in higher-level dialects can target `linalg` operations and leverage all subsequent transformations supported by `linalg` and lower-level abstractions. As an example, Figure 2.2 presents an MLIR matrix-multiplication (MatMul) implementation in different abstractions.[1] The operation is initially represented using a `linalg.matmul` and subsequently undergoes conversion, transformation, and lowering by the compiler. In Figure 2.2a-L11 and L17, the `linalg.matmul` is converted into a `linalg.generic`. The `linalg.generic` is a core MLIR operation that can represent most of the `linalg` *named ops*, by careful selection of its *operation trait*[2] `indexing_maps` (L2), `iterator_types` (L7), and kernel (L15 to L18). Finally, the generic operation can be converted into a tiled (4×4×4) implementation of the MatMul (Figure 2.2b) using the structured control flow (`scf`) dialect. When supporting an accelerator that can process a $\text{MatMul}_{4\times4\times4}$ operation. Note a 2D MatMul operation is MatMul_{MxNxK}: C(M,N) = A(M,K) x B(K,N). The code in Figure 2.2b-L11 to L19 has to be replaced by the runtime library calls that drive the accelerator.

MLIR Memory References

Within MLIR, memory buffers exist as N-dimensional (rank=N) memory references, or `memref`s. Accessing the elements of an MLIR `memref` requires accessing the values in the equivalent C struct of Figure 2.3. The AXI4MLIR DMA runtime library, presented in Section 8.2.1, supports bidirectional data movements between `memrefs` and memory-mapped buffers (raw pointers) while respecting strides, sizes, and dimensions.

[1] Some MLIR code is omitted for the sake of brevity.
[2] See *linalg.generic* in https://mlir.llvm.org/docs/Dialects/Linalg

2.3 Hardware

While DNN inference is less demanding than training, it is still deemed to be expensive to perform DNN inference on low-power and resource-constrained edge devices such as smartphones, tablets, etc. To address these issues, hardware-based optimisations to reduce the compute and power requirements of inference are being developed via ISA-level extensions for CPUs and GPUs, along with custom ASIC, CGRA and FPGA architectures. The inefficiency of inference on edge devices is only exasperated by the development of novel DNN model architectures with a greater number of layers and complexity, which increases the computational and memory bandwidth demands and is particularly critical for resource-constrained devices. Sze et al., [Sze+17] explore the different problems and solutions for efficient processing of DNNs, and also outlines the current trend in hardware accelerators for DNN inference. The work examines the ASIC and FPGA-based designs to highlight the importance of spatial architecture and different types of dataflows to enable efficient processing of DNN models. Designing spatial accelerators for edge DNN inference has many possible optimisations to be taken into consideration, as they have certain limitations not found in GPUs, such as low on-chip memory, limited off-chip memory access bandwidth, and low computational power. Power consumption and energy efficiency also need to be taken into consideration to allow custom solution to perform in energy-constrained environments. As this thesis focuses on hardware-software co-design of FPGA-based accelerators for DNN inference on edge devices, within this section we discuss FPGAs, accelerators and the hardware development process for designing FPGA-based accelerators.

2.3.1 FPGAs

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices that can be programmed to perform a wide range of tasks. FPGAs are used in a variety of applications, including digital signal processing, networking, and machine learning. FPGAs represent a middle ground between general-purpose processors like CPUs, GPUs and NPUs, and Application-Specific Integrated Circuits (ASICs) like the TPU and NVDLA, while providing finer grain control compared to Coarse-Grained Reconfigurable Arrays (CGRAs).

While general-purpose CPUs are flexible and easy to program, they are inefficient for parallel processing of highly parallel tasks, such as those found in machine learning workloads. This inefficiency is because CPUs are designed to execute a wide range of operations/instructions, which means that the hardware resources are optimally utilised for a specific task. Meanwhile, GPUs/NPUs can perform well on parallel task

processing, but they are highly energy inefficient and can often be underutilised because the workload does not suitably match the GPU/NPU capabilities. Overall, the flexible nature of general-purpose CPUs/GPUs/NPus usually leads to lower performance in terms of throughput, power efficiency, and latency compared to an ASIC or FPGA-based accelerator.

ASICs are custom-designed chips that are optimised for a specific task. Since ASICs are higly specialised for a specific task, they can outperform general-purpose CPUs/GPUs and FPGAs in terms of throughput, power efficiency, and latency. However, ASICs are very time-consuming and costly to design, and once they are manufactured cannot be reconfigured. This means that first, the design process becomes more complex and laborious as the designer has to ensure that the design is correct before it is manufactured. Secondly, if the task changes, the ASIC can become less efficient or even obsolete.

Meanwhile CGRAs provide some flexibility and performance as they typically contain functional units interconnected by a mesh style network with some configurability. Due to their coarse-grain nature and lack of gate-level reconfigurability, CGRAs limit the problem specific optimisations, and gate-level/circuit level optimisation which are supported by FPGAs.

Hence, FPGAs are a good middle ground between general-purpose processors and ASICs. FPGA-based accelerators enable high performance, low latency, and power efficiency compared to general-purpose CPUs/GPUs due to the parallel processing capabilities of FPGAs. But, due to the reconfigurability of FPGAs, they are less efficient than ASICs. However, the flexibility of FPGAs makes them an attractive choice for developing hardware accelerators for machine learning workloads, as the workloads are constantly evolving. Additionally, reconfigurability of FPGAs allows for rapid prototyping and testing of accelerator designs alongside the target software stack.

2.3.2 FPGA Architecture

FPGAs are composed of different primitive resources such as Configurable Logic Blocks (CLBs), Block RAMs (BRAMs), Digital Signal Processing (DSP) blocks, Input/Output (I/O) blocks and programmable interconnects.

- **Configurable Logic Blocks (CLBs)**: CLBs are the basic building blocks of FPGAs, they consist of Look-Up Tables (LUTs), flip-flops (FFs), and multiplexers (MUXes). Using CLBs, the designer can implement any combinational or sequential logic function.

- **Block RAMs (BRAMs)**: BRAMs are used to store data and implement memory structures such as caches, buffers and FIFOs. BRAMs are used to store DNN weights, activations, and intermediate results during the computation. Note that in the case of FPGA-based accelerators BRAMs are referred to as on-chip memory, as they are used to store data on the FPGA itself, which is faster to access than off-chip main memory.
- **Digital Signal Processing (DSP) blocks**: DSP blocks are used to implement complex arithmetic operations such as multiplication and accumulation. Within Zynq-7000 SoC, the DSP48E supports 18×25-bit multipliers and 48-bit adders.
- **Input/Output (I/O) blocks**: I/O blocks are used to interface the FPGA with external devices. I/O blocks are used to transfer data between the FPGA and the host system. For example, the High Performance AXI [Deva] (HP AXI) ports on the Zynq-7000 SoC are used to transfer data between the FPGA and the ARM processor.
- **Programmable Interconnects**: Programmable interconnects are fundamental to the FPGA architecture, as they enable combining the different resources on the FPGA together to implement the desired functionality.

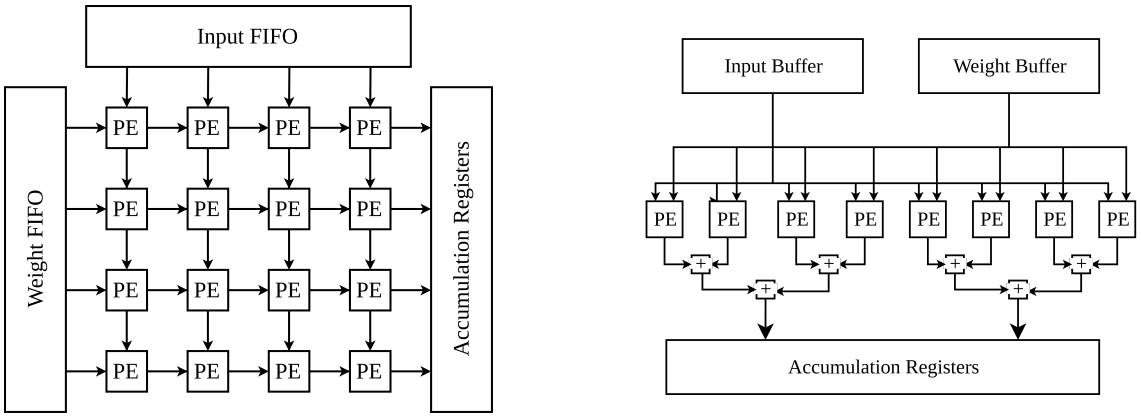
Depending on the specific FPGA device, the number of these resources available on the device vary. For the purpose of this thesis, we will focus on Xilinx FPGAs, specifically the FPGA devices available on the Zynq-7000 SoC (System on Chip) [AMDb], see Table 1.1 for the specifications.

2.3.3 Accelerators

Accelerators are hardware devices designed to execute specific computational tasks. Accelerators can be used to offload computationally intensive tasks from the CPU to improve performance in terms of throughput, latency, and power efficiency. These tasks can range from simple image processing to cryptography and machine learning workloads. Accelerators can be implemented in a variety of ways, including custom ASICs, FPGAs, CGRAs and GPUs. In the context of this thesis, we focus on FPGA-based accelerators for DNN inference, but the concepts discussed here can be applied to other types of accelerators as well.

Accelerator Architecture

The accelerator architecture can be described as the combination of three main concepts: *processing units* (PUs), *dataflow* and *memory hierarchy*.



(a) 4-by-4 Systolic Array Processing Unit. (b) A 8x Vectorised Processing Unit.

Figure 2.4: Example of different dataflows architectures for processing data.

Processing Units: The PU defines what computation is performed on the data, including PE (processing element) design within the PU. Note that within the thesis, we distinguish between PUs and PEs. PUs are larger, more complex, and contain several PEs, whereas PEs are more primitive computational units that can perform simple arithmetical operations such as multiplying and adding. The PU abstracts away the computation within the accelerator and allows the designer to focus on the mathematical operations that need to be performed on the data. Ideally, the PU should be designed such that an accelerator can contain multiple instances of the PU to increase the level of parallelism and improve the performance of the accelerator. Additionally, different PUs can perform different operations; for example, a PU can be designed to perform matrix multiplication, while another PU can post-process the matrix multiplication results.

Dataflow: The dataflow defines the computational paradigm of the accelerator, and how the computation is scheduled and executed within the accelerator. Figure 2.4 contains two different PUs highlighting two common dataflow paradigms used in DNN accelerators: the systolic array and the vectorised design, these dataflow paradigms form the basis of the accelerator designs presented in Section 4.4.3. In the systolic array design, the computation is pipelined, where the data is passed from one processing element to the next. In the vectorised design, the computation is performed in a SIMD (Single Instruction Multiple Data) manner, where the same operation is performed on multiple data elements in parallel across the processing elements.

Memory Hierarchy: The overall memory hierarchy defines how the data is stored and accessed across the entire system (see Figure 1.1), which includes how data is stored in main memory and on-chip memory within the accelerator. Within the context of

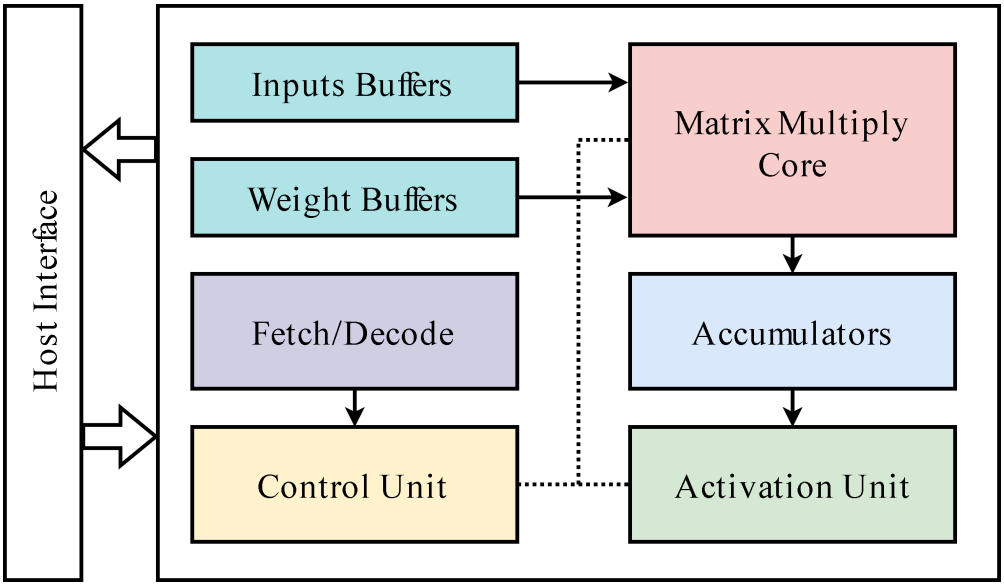


Figure 2.5: Generic DNN accelerator architecture.

accelerator design, we focus on the on-chip memory hierarchy within the accelerator. This includes details on how the on-chip memory is allocated for different data (e.g., weights, activations, intermediate results) and how it is distributed to the PUs within an accelerator. The design of the local memory hierarchy of the PUs is also important, as it can dictate the performance of the PEs within the PUs. For example, in the systolic array architecture, data needs to be allocated to fill the input and weight FIFOs every cycle to maximise throughput. Similarly, in the vectorised architecture, memory needs to be arranged so that data can be broadcast to all PEs in parallel.

DNN Accelerators

Due to the highly parallel nature of DNN workloads, general purpose GPUs are often used to accelerate DNN inference, as they can provide significant speedup over general-purpose CPUs. However, specialised ASIC designs such as the TPUs [Jou+17], MAERI [KSK18], and Eyeriss [Che+19] have shown that specialised DNN accelerators can provide significant speedup over general-purpose CPUs and GPUs for DNN inference.

DNN inference accelerators are designed to target computationally intensive layers in DNN models, such as convolutional layers. For this reason, DNN accelerators are often designed to perform either matrix multiplication or vector-matrix multiplication, as these operations are prevalent in DNNs. Figure 2.5 shows a generic DNN accelerator architecture, which usually consists of: (i) on-chip memory buffers to store weights, activations, and intermediate results during the computation to help hide the latency of off-chip memory accesses; (ii) custom activation function units, which are used to apply

activation functions to the intermediate results; (iii) high-speed interface to transfer data between the CPU and the accelerator; (iv) control logic to manage the dataflow and the computation; (v) fetch/decode unit to process the instructions from the host CPU.

2.3.4 Hardware Development

Developing new accelerators for FPGAs is a complex process that requires a deep understanding of hardware design methodologies. To develop a hardware accelerator, the hardware designer has to be familiar with hardware description-based development flow. Hardware Description Language (HDL) based design flows use highly detailed hardware descriptions at the Register Transfer Level (RTL), for example in languages such as Verilog [Des06] and VHDL [Des19] to define the desired behaviour of the accelerator.

While this approach allows for fine-grained hardware designs, it comes with high development time, and high codebase complexity and size required to define designs [Pel+16], when compared to High-Level Synthesis (HLS) based solutions. HLS tools allow the designer to write less descriptive algorithmic code in C/C++/SystemC, which can be translated into low-level HDLs such as Verilog and VHDL. These HDLs can then be used to realise the designs on ASICs or FPGAs.

Listing 2.6 shows an example of Verilog code for a simple adder, and listing 2.7 shows the same adder using SystemC code that can be used to generate the same Verilog code using Vivado HLS. SystemC code is written at a higher level and is easier to understand than the Verilog code, which makes it easier to develop and maintain. Additionally, some trade-offs exist between HLS and purely HDL-based design processes; we highlight the simulation capabilities in Section 2.5.

OpenCL [SGS10] is another design methodology that uses a host-device programming model where the host code, i.e., the driver, prepares and transfers data to be executed by the device, i.e., the accelerator. Hence, the OpenCL approach allows for the co-design of the driver. The device code is written in high-level OpenCL code, which defines computation kernels that process the target workload. This high-level OpenCL code is translated into a synthesisable hardware design. OpenCL follows a rigid programming model, where the designer defines the computation kernels that need to be accelerated. The designer can configure the number of hardware instances allocated to each computation kernel. The higher the number of instances, the greater the number of instructions executed in parallel for the given kernel.

```
module adder (
    input wire [31:0] a,
    input wire [31:0] b,
    output wire [31:0] c
);
assign c = a + b;
endmodule

```

Figure 2.6: Verilog code for a simple adder.

```
SC_MODULE(adder) {
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> c;

    void add() {
        c.write(a.read() + b.read());
    }

    SC_CTOR(adder) {
        SC_METHOD(add);
        sensitive << a << b;
    }
};

```

Figure 2.7: SystemC code for a simple adder.

High-Level Synthesis (HLS)

High-Level Synthesis (HLS) is a design methodology that allows to write algorithmic code in high-level languages such as C/C++/SystemC, which can then be synthesised into low-level HDLs such as Verilog and VHDL. There are many different HLS compilers/tools which are able to perform HLS on high-level descriptions of hardware designs. HLS compilers take the target hardware platform into account when generating the hardware design, which can allow the compiler to optimise the design both in terms of performance and resource utilisation. One drawback of HLS is that the designer has less control over the hardware design, as the HLS compiler is responsible for generating the hardware design from the high-level code. This can lead to the generation of sub-optimal hardware circuits. To mitigate this issue, HLS tools often provide the designer with the ability to provide hints to the compiler to guide the hardware design generation through HLS 'pragmas' or 'directives'. These pragmas can drastically change the design generated by the HLS compiler and lead to more efficient hardware designs.

In this thesis we utilise the AMD's Vivado HLS [Xilb] tool with SystemC as the input language to generate the Verilog/VHDL code. Once HLS has generated the Verilog/VHDL code, Vivado HLS exports IP (Intellectual Property) blocks, which package the generated IP into a reusable block that can be imported for Logic Synthesis.

Logic Synthesis

Logic synthesis is the process of mapping an RTL hardware design to a target FPGA device, which involves translating the RTL code into a netlist of logic gates. The netlist is then mapped to the target FPGA device, which involves placing the logic gates on the FPGA and routing the connections between them. Logic synthesis of hardware designs to an FPGA is a time-consuming process that can take minutes to hours, depending on the complexity of the design.

From the user's perspective, the critical part of the logic synthesis process is creating a block design that integrates their desired IP blocks and the generated IP block, in our case, the accelerator. Figure 2.8 contains a screen capture of the Vivado Design Suite, which shows the block design of an accelerator integrated with the Zynq-7000 SoC. Once the block design is complete, the designer can run the Vivado Design Suite to generate the bitstream, which consists of main steps, including synthesis, implementation, and bitstream generation. The generated bitstream can then be programmed onto the target FPGA device, configuring the FPGA to implement the desired hardware design.

2.4 Key Algorithms

Across the different types of DNN layers, there are some key operations that are repeatedly used in the computational graph. These operations can be thought as the algorithmic primitives that are used to build the computational graph of a DNN. Here we will discuss and highlight the primitive operation that will be referred across this thesis. Note that for a type of layer, there can be multiple ways to implement the operation.

2.4.1 Convolution

Looking at convolutional layers provides a good example of many computationally different implementation strategies for the same convolution operation. The simplest

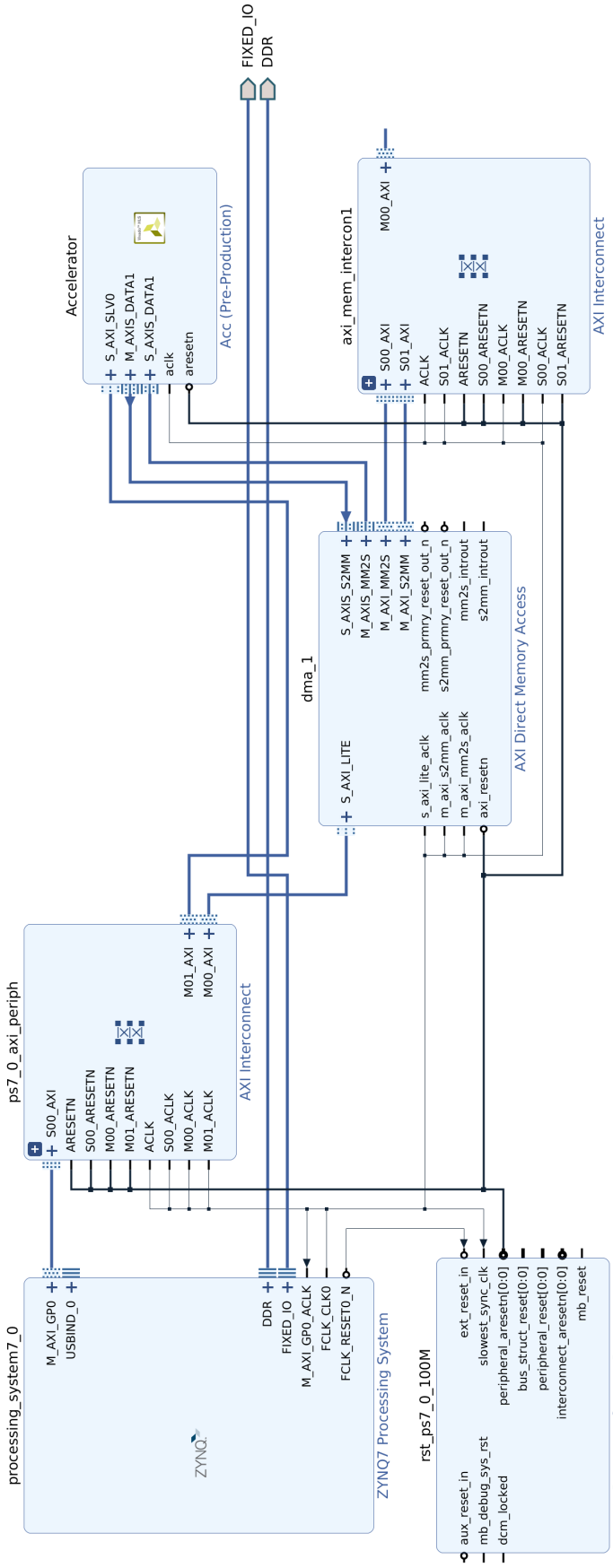


Figure 2.8: Block design of an accelerator integrated with the Zynq-7000 SoC in Vivado Design Suite.

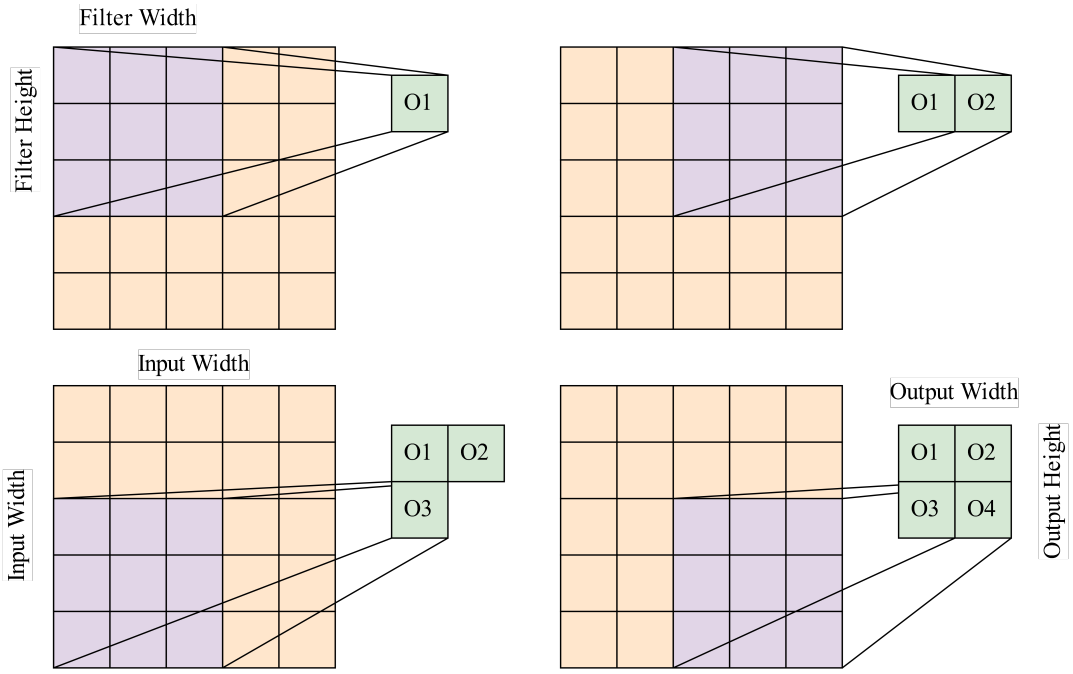


Figure 2.9: Direct Convolution Example. The filer kernel is slid across the input activations to produce the output activations.

implementation is referred to as direct convolution which simply performs the convolution operation according to its definition by sliding the filter kernels across the input activations.

Figure 2.9 shows an example of direct convolution, where a 3×3 filter kernel is slid across the 5×5 input activations to produce the 2×2 output activations, with a stride of 2 and no padding.

Direct convolution requires irregular memory access patterns which can be inefficient on hardware such as GPUs. Hence, many optimised implementations of convolution operations have been developed to improve the performance of convolutional layers. A commonly used implementation referred to as GEMM convolutions is performed by rearranging the convolution operation to matrix multiplication by flattening the filter weights and output activations and through the replication of input activations using the IM2COL algorithm [CPS06]. This strategy poses many advantages as matrix multiplication-based convolutions allow for a high degree of parallel computing while employing varying tiling strategies suited to the available hardware. We utilise this method in Chapter 4 to implement and accelerate the convolutional layers of the CNN models.

Transformation-based convolutions such as Fast Fourier Transform [CJM20] and Winograd [LG16] are sometimes used to reduce the number of expensive multiply operations usually by increasing memory demands, introducing shift-based operations or increas-

Algorithm 1: Matrix Multiplication (MM)

Input: A = $shape(m, k)$, B = $shape(k, n)$
Output: C = $shape(m, n)$
for $i = 1$ *to* m **do**
  for $j = 1$ *to* n **do**
    $C_{ij} = 0$
    for $l = 1$ *to* k **do**
      $C_{ij} = C_{ij} + A_{il} \times B_{lj}$

ing the number of add operations. Different methods pose varying trade-offs which need to be accounted for when pursuing optimal performance.

2.4.2 Matrix Multiplication

Matrix Multiplication (MM) is a fundamental operation in DNNs, used in many layers such as fully connected layers and convolutional layers.

Algorithm 1 shows the general matrix multiplication in its most basic form. Commonly, to improve performance of MM operations, depending on the hardware architecture, the algorithm can be optimised by using techniques such as loop unrolling, tiling, and vectorisation. As previously mentioned, GEMM convolutions which essentially are MM, are used to perform convolution operations by rearranging the inputs using the IM2COL algorithm, flattening the weights and then performing the GEMM operation, thus yielding the same result as direct convolution but with improved performance. Figure 2.10 shows an example of the IM2COL operation. Here the input activations are transformed into a matrix, where each row corresponds to a kernel size patch of the input activations, the patches are determined by the kernel size and stride of the convolution operation. The filter kernels are also flattened into a matrix, where each row corresponds to a filter kernel. Note that while transforming the input activations into the input matrix yields data replication and increased memory usage, the GEMM operation can be highly optimised for parallel computations due to its regular memory access patterns. We use matrix multiplication as a target operation for accelerator in Chapter 5, Chapter 6 and in Chapter 8.

2.4.3 Transposed Convolution

Transposed Convolution (TCONV), sometimes referred to as deconvolution within the field of ML, is used in upsampling layers. Upsampling layers are used to increase the

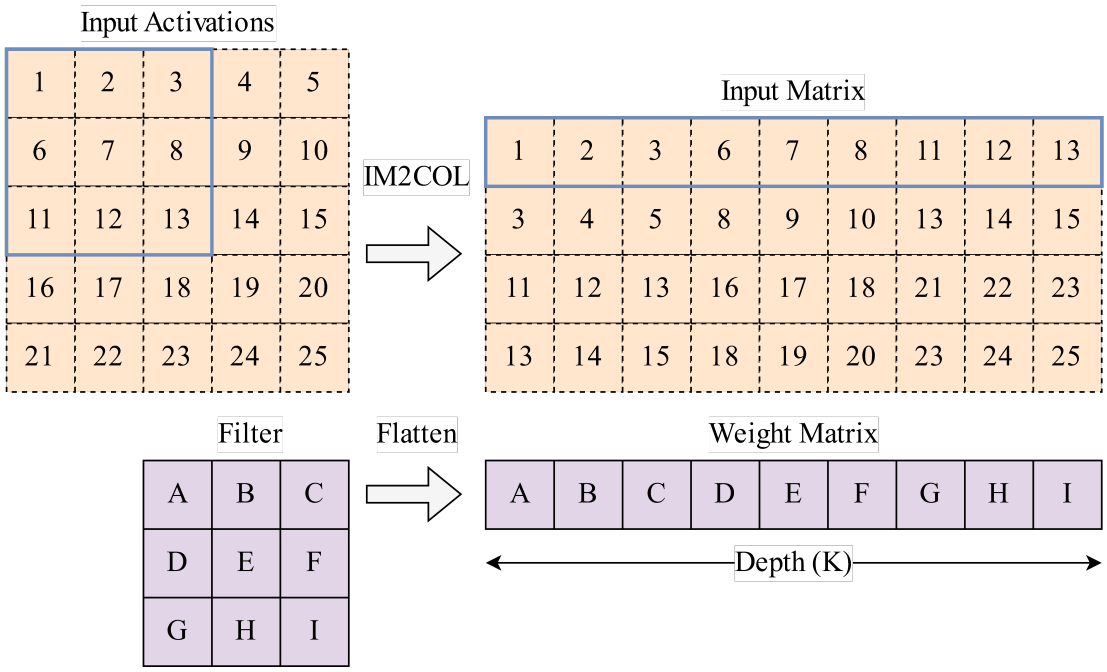


Figure 2.10: IM2COL. The input activations are transformed into the input matrix, and the filter kernels are flattened into the weight matrix. Note the input matrix and the weight matrix now share the common 'Depth (K)' dimension.

spatial resolution of the input activations. These are used in many DNN architectures, such as U-Net [RFB15] and SegNet [BKC17] for image segmentation tasks. The TCONV operation be defined by the following eight dimensions, with the output being defined as:

$$out(O_h, O_w, O_c) = tconv(I_h, I_w, I_c, Ks, O_c, S) \tag{2.1}$$

where I_h, I_w, I_c are the input height, width and channels, respectively; with kernel size K_s, output channels O_c and stride S. The output dimensions, output height O_h and width O_w are defined as: $O_{hw}{=}S \times I_{hw}$. When $K_s > S$, executing the direct TCONV operation requires the coalescing of intermediate outputs into the same final output; this coalescing is known as the overlapping sum problem. A simple $tconv(2, 2, 1, 2, 1, 1)$ example of direct TCONV is shown in Figure 2.11, highlighting the coalescing of intermediate outputs.

Since the mapping of intermediate outputs to final outputs is not a fixed ratio, the complexity of the output mapping increases.

There are three optimised methods for implementing TCONV: (i) Zero-Insertion; (ii) Transforming Deconvolution to Convolution (TDC); (iii) and Input-Oriented Mapping (IOM).

Zero-Insertion resolves the overlapping sum problem by padding the input zeros, albeit with added compute, memory, and bandwidth overhead, approximately 75% [Xu+18].

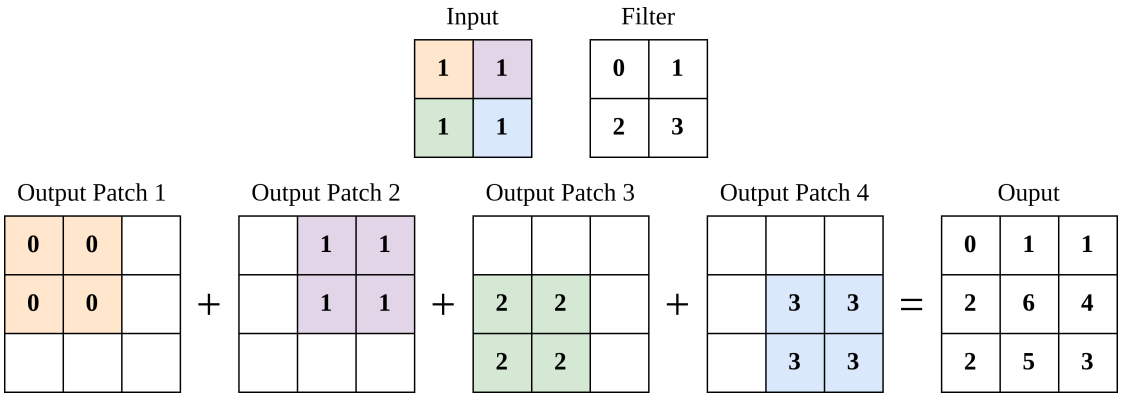


Figure 2.11: Direct TCONV Example. The filter kernel is slid across the inputs to produce the intermediate output patches. The four intermediate output patches are summed to produce the final output.

The TDC method transforms TCONV operations into Convolution operations by generating sub-filter kernels to avoid the overlapping problem, but this method requires additional hardware to process the sparse sub-filter efficiently [CKK20]. The TDC method expands the input kernels to the same size as the sub-filter (F_s) kernels, and the filter data is replicated across F_s^2 sub-filters.

For the IOM method, introduced within GNA [Yan+18], each activation is multiplied by the filters and the partially overlapped intermediate results are summed to produce the final output. We can express the IOM method as:

$$out(O_h, O_w, O_c) = col2im(gemm(I, W_T), O_h, O_w, O_c) \tag{2.2}$$

where $I(I_h, I_w, I_c)$ is the input data, $W(Ks, Ks, O_c, I_c)$ is the filter data, $gemm$ is the matrix multiplication operation, and $col2im$ is the operation to convert the output of the matrix multiplication to the final output. Note an example of the TCONV operation using the IOM method is shown in Figure 7.1. The IOM method reduces the number of operations required to perform TCONV, as it does not require additional padding or transformation of inputs or weights. The drawback of the IOM method is that it requires an efficient architecture to overcome overlapping sums, and contains a significant amount (up to 28%) of ineffectual computation due to cropped outputs which are calculated with the standard MM-based implementations of IOM. In Chapter 7, we design and implement a specialised TCONV accelerator using the IOM method that tackles the current limitations of the IOM method.

2.4.4 Quantisation

Within the field of Machine Learning and DNNs, quantisation is a technique used to reduce the precision of the weights and activations in a DNN model. This technique is used to reduce the memory footprint and computational complexity of the model, thus making it more suitable for deployment on resource-constrained devices. Quantisation can be applied to weights, activations, or both, and can be done in a variety of ways, such as floating-point quantisation, integer quantisation, and block floating point quantisation. Here we focus on integer quantisation and block floating point quantisation, which have been used across this thesis.

Integer Quantisation

Integer quantisation, also known as fixed-point quantisation, is a type of quantisation which involves mapping the floating-point values to a smaller set of integer values, which can be represented using fewer bits.

The most common form of integer quantisation is 8-bit signed integer quantisation (INT8), which maps the floating-point values to 8-bit signed integers, more specifically to the range $[-128, 127]$. Unsigned 8-bit integer quantisation (UINT8) is also used, which maps the floating-point values to the range $[0, 255]$, but this is less conventional due to the mid-point not being '0', which is not ideal for symmetric quantisation.

For frameworks like TFLite, quantisation is a key optimisation technique used to reduce the memory footprint and computational complexity of the model, making it more suitable for deployment on resource-constrained devices [TFL]. Within TFLite the inputs are mapped to INT8 quantisation, but they are asymmetrical by nature, hence the zero-point is not '0'. Fortunately, the weights can be forced to become symmetrical with zero-point being '0', and this allows for more precision to be retained in the weights, and potentially reduced number of operations to adjust the quantised weights [TFL]. All models used in Chapter 4, Chapter 5, and Chapter 7 utilise INT8 quantisation for the weights and activations.

Block Floating Point Quantisation

Block Floating Point (BFP) quantisation is a technique used to quantise the weights and activations of a DNN models. BFP quantisation involves mapping the floating-point values to a smaller set of integer values, which can be represented using fewer bits. The key idea behind BFP quantisation is to reduce bit-width of the values but then keep track of two scaling factors. Depending on the BFP format, the first scaling

factor is for a small block of values, and the second scaling factor is shared across a 'super-block' of values. In Chapter 6, we design a MatMul accelerator around a specific BFP format described in Section 6.3.1.

2.5 Hardware-Software Co-Design

Hardware-Software (HW-SW) co-design is a critical aspect when designing and deploying DNNs on edge FPGAs. Since resources are more limited on edge FPGAs, and DNNs workloads are large in terms of their memory footprint and computational demands, a given DNN is unlikely to fully fit on an edge accelerator. Thus, for inference, the accelerator must operate in close communication with the CPU, which requires careful co-design with the host CPU code to ensure that data is managed efficiently.

2.5.1 SystemC

SystemC [Des23] is a TLM (Transaction Level Modelling) language based on C++ that is used for hardware-software co-design. SystemC is used to model hardware components and their interactions, and can be used to model the hardware-software interface. TLM is a modelling abstraction that allows the designer to model the communication between hardware components at a higher level of abstraction. This allows the designer to focus on the functionality of the components rather than the details of the communication between them. Within this thesis, we use SystemC to design, simulate and generate synthesisable hardware accelerators for DNNs. Specifically, the SECDA design methodology described in Chapter 4 uses SystemC as the core language for hardware design to enable simulation and synthesis of the hardware accelerators.

SystemC Hardware Definition

SystemC allows the hardware designer to define modular hardware design using specialised SystemC classes. Each SystemC class is referred to as a hardware module. A hardware module can be as simple as implementing an 'and' gate or as complex as a full DNN accelerator. Within each hardware module, the designer can initiate a clock signal, I/O ports, buffers, internal signals, and hardware 'threads', which contain combinatorial or sequential logic that implements the functionalities of the hardware module. Each SystemC 'thread' can be thought of as a hardware circuit that will run independently of the other threads, unless the synchronising logic is implemented with another thread using internal signals.

```
SC_MODULE(FIFO) {
  sc_in<bool> clk;
  sc_fifo<int> data_fifo;

  void write_thread() {
    int write_counter;
    while (true) {
      data_fifo.write(write_counter);
      write_counter++;
      wait(1, SC_NS);
    }
  }

  void read_thread() {
    int read_counter;
    while (true) {
      int data = data_fifo.read();
      read_counter++;
      wait(1, SC_NS);
    }
  }
  SC_CTOR(FIFO) : data_fifo("data_fifo") {
    SC_THREAD(write_thread, clk.pos());
    SC_THREAD(read_thread, clk.pos());
  }
};

```

Figure 2.12: SystemC code for a simple read/write FIFO module.

Figure 2.12 shows an example of a SystemC module, which defines a simple data read/write from/to a data FIFO using two threads. One thread writes to the FIFO, and the other reads from it. Note that the FIFO initialised inside the module is accessible to both threads, but the '*read_counter*' and the '*write_counter*' are only accessible to the respective threads.

While this simple design contains only one module, more complex designs require multiple modules connecting via I/O ports. SystemC modules can also contain submodules, enabling the abstraction of physical resources between different submodules. For example, a 'processing unit' submodule can have internal buffers that the parent accelerator module cannot access. Additionally, submodules can help design accelerators that can support multiple types of 'processing units' or even have a scalable number of submodules.

SystemC Simulation

SystemC provides an hardware-software event-driven co-simulation environment that allows the designer to simulate the hardware design before synthesising it. Event-driven simulation depends on events, such as when a signal changes its value or a FIFO is written and filled. The simulation environment allows the designer to test the functionality of the hardware design and debug any issues that may arise. Additionally, due to co-simulation, the designer creates and uses software testbenches that imitate the target workload. Co-simulation helps provide data inputs to the hardware simulation and verify the output data.

Since SystemC TLM, the modules communicate with each other through transactions, which occur across communication channels. This type of model can allow the designer to simulate the hardware design at different levels of abstraction. For example, the designer can model the design at a finer granularity, achieving cycle-accurate simulation, but this will lead to a slow simulation time similar to RTL simulation. Alternatively, SystemC is typically used to perform cycle-approximate simulation using loosely-timed models. This approach implies that not all cycles contain events, i.e., clock edges rise and fall. Hence, the simulation only occurs when accelerated defined events occur, for example, when a FIFO is written to or read from. We describe how SystemC simulation is used across this thesis in Section 4.3.3.

2.5.2 Host-Accelerator Communication

This thesis focuses on the hardware-software co-design from the perspective of the low-level host-driver interaction with the accelerator. This interaction is a vital component of the DNN acceleration pipeline, as the efficiency of data movement between the software-managed main memory and hardware-managed on-chip memory can dictate the overall performance, especially in resource-constrained devices with limited on-chip memory. Within our system model, we refer to the 'accelerator driver' as the component that handles the algorithm which implements the data transfer required for the target operation. The driver can perform data transfer as it controls both the accelerator and the DMA engines, which move data between the main memory and accelerator memory. As our system model uses an ARM-based SoC with an FPGA, the standard protocol for data movement within the SoC is the AXI (Advanced eXtensible Interface) protocol [Deva][3] and MMIO (Memory-Mapped Input/Output).

[3]Specifically the AXI4 version.

Advanced eXtensible Interface (AXI) Protocol

There are three main types of AXI protocols: AXI-MM (Memory-Mapped), AXI-Stream (AXI-S) and AXI-Lite.

AXI-MM: enables memory mapping of a region of main memory, such that the accelerator can treat this area of memory as addressable. While this approach eases the hardware implementation, it can induce latency overhead due to the mapping and random address access.

AXI-S: enables a stream-based data interface which allows for data to be streamed in via FIFO from main memory to the accelerator (MM2S) and accelerator to main memory (S2MM). This approach requires careful management of the data stream, as the data must be sent in the correct order, and the correct amount of data must be sent; but it enables lower latency and higher throughput. This approach is suitable for transferring large amounts of data, such as the input and output tensors of a DNN operation.

AXI-Lite: is a simplified version of AXI-MM, which is used for control registers and status registers. An accelerator's I/O ports, such as the start signal, the done signal, and the configuration registers, can be accessed using the AXI-Lite protocol. From the driver's perspective, the AXI-Lite protocol is accessed using MMIO. The driver memory maps the control registers and status registers to the virtual memory space of the host CPU and can read and write to these registers to control the accelerator. Similarly, the driver uses the AXI-Lite combined with MMIO to access and control the AXI DMA engine [Xila], initiating data transfers as required.

These all three types of AXI protocols are used throughout the accelerators described within this thesis, with the AXI-MM being used for the FC-GEMM accelerator presented in Section 5.4.2, whereas all the other accelerators use AXI-S and AXI-Lite.

2.5.3 Hardware Specific Optimisations

There are several techniques that can be used to transform a DNN operation into a form that can be efficiently executed on hardware accelerators. These techniques can improve the performance of the DNN operations by exploiting the parallelism and memory hierarchy of the hardware accelerators.

Tile-based Processing

Tile-based processing, or tiling, is a technique for partitioning one or more of the data tensors for a given problem into smaller 'tiles'. Tiling enables the processing of large

data tensors in smaller chunks, which can be beneficial for the hardware. For example, a tile can be loaded entirely on the L1 cache, enabling faster processing, or multiple tiles can be sent to on-chip memory within an accelerator. Tiling can also be used to exploit parallelism within the hardware. For example, different processing elements/units can process multiple tiles in parallel.

Algorithm 2: Tiled Matrix Multiplication

```
1  Input: A = shape(m, k), B = shape(k, n), Tile = shape(t, t)
2  Output: C = shape(m, n)
3  Parameters: t = Tile size
4  for i = 0 to m step t do
5      for j = 0 to n step t do
6          for l = 0 to k step t do
7              for ii = 0 to t do
8                  for jj = 0 to t do
9                      for ll = 0 to t do
10                         C_{i+ii,j+jj} = C_{i+ii,j+jj} + A_{i+ii,l+ll} × B_{l+ll,j+jj}
```

Tiling is ubiquitous in executing tensor operations on hardware accelerators. For example, the basic matrix multiplication operation can transform into a tiled multiplication operation. Algorithm 1 can be transformed into a tiled matrix multiplication operation as shown in Algorithm 2. The tile size t is a parameter that can be tuned to achieve optimal performance depending on the hardware architecture. Algorithm 2 introduces three new loops in L5-L9, which iterate over the tile size t. We can replace these loops with a call to a hardware accelerator to process the tile in parallel. Algorithm 3 shows the tiled matrix multiplication operation with the tile processing offloaded to a hardware accelerator. Additionally, Algorithm 3 shows an example of three common dataflow patterns: output stationary, input stationary and weight stationary; these dataflow patterns can be used to optimise data movement between the accelerator and main memory. Stationary in this context refers to the data tiles; by keeping them 'stationary', in other words, cached in the local accelerator buffers, the stationary data tiles can be reused as many times as required without requiring additional off-chip memory access. For example, in the output stationary dataflow, each output tile is cached in the accelerator buffers, meaning it only needs to be sent once, whereas the input and weight tiles are sent m/t and n/t times, respectively. Input/weight stationary dataflows follow the same trend by caching the input/weight tiles.

To elaborate further on Algorithm 3, the function **Send_Tile_To_Accelerator** sends the tile to the accelerator, **Execute_Accelerator** executes the tile processing on the accelerator, and **Receive_Tile_From_Accelerator** receives the processed tile from the accelerator. These accelerator-specific functions and the related algorithms are

Algorithm 3: Output/Input/Weight Stationary Tiled Matrix Multiplication with Accelerator

```
1  Input: A = shape(m, k), B = shape(k, n), Tile = shape(t, t)
2  Output: C = shape(m, n)
3  Parameters: t = Tile size
   // Output Stationary Dataflow
4  for i = 0 to m step t do
5      for j = 0 to n step t do
6          Send_Tile_To_Accelerator(C, i, j, t)
7          for l = 0 to k step t do
8              Send_Tile_To_Accelerator(A, i, l, t)
9              Send_Tile_To_Accelerator(B, l, j, t)
10             Execute_Accelerator()
           // Receive the accumulated output tile
11         Receive_Tile_From_Accelerator(C, i, j, t)
   // Input Stationary Dataflow
12 for i = 0 to m step t do
13     for l = 0 to k step t do
14         Send_Tile_To_Accelerator(A, i, l, t)
15         for j = 0 to n step t do
16             Send_Tile_To_Accelerator(B, l, j, t)
17             Send_Tile_To_Accelerator(C, i, j, t)
18             Execute_Accelerator()
               // Receive the partial output tile and accumulate on the host
19             Receive_Tile_From_Accelerator(C, i, j, t)
   // Weight Stationary Dataflow
20 for l = 0 to k step t do
21     for j = 0 to n step t do
22         Send_Tile_To_Accelerator(B, l, j, t)
23         for i = 0 to m step t do
24             Send_Tile_To_Accelerator(A, i, l, t)
25             Send_Tile_To_Accelerator(C, i, j, t)
26             Execute_Accelerator()
               // Receive the partial output tile and accumulate on the host
27             Receive_Tile_From_Accelerator(C, i, j, t)
```

implemented in the host-driver code that manages the accelerator. A hardware designer must develop and implement these functions for their accelerator.

Pipelining

Pipelining is a technique where an operation is broken down into multiple stages such that parallel hardware units can execute different stages of multiple operations at the same time. Typically, different hardware units are used within accelerators to pipeline the execution of an operation, improving the accelerator's throughput. For example, in the tiled matrix multiplication operation, the accelerator can be pipelined to process multiple tiles in parallel. Algorithm 3 can be pipelined by sending new tiles to the

accelerator while the accelerator is executing the previous tiles as shown in Algorithm 4. This example shows a pipelined execution of output stationary dataflow for MM. Note that the first weight and input tiles are preloaded (L7-L8) before the inner loop, which calls for the accelerator to start processing.

Algorithm 4: Pipelined Tiled Matrix Multiplication with Accelerator

```
1  Input: A = shape(m, k), B = shape(k, n), Tile = shape(t, t)
2  Output: C = shape(m, n)
3  Parameters: t = Tile size
   // Output Stationary Dataflow
4  for i = 0 to m step t do
5      for j = 0 to n step t do
6          Send_Tile_To_Accelerator(C, i, j, t)
           // Preload the first input and weight tiles
7          Send_Tile_To_Accelerator(A, i, 0, t)
8          Send_Tile_To_Accelerator(B, 0, j, t)
9          for l = t to k step t do
10             Pipelined_Execute_Accelerator()
11             Send_Tile_To_Accelerator(A, i, l, t)
12             Send_Tile_To_Accelerator(B, l, j, t)
           // Execute the last tile
13         Pipelined_Execute_Accelerator()
14         Receive_Tile_From_Accelerator(C, i, j, t)
```

Additional pipeline optimisation is possible between the accelerator and the host CPU. For example, in a scenario where the host CPU is required to pre-process data tiles before sending them to accelerator, the host CPU and accelerator can be pipelined such that the first tile's execution is concurrent with the second's pre-processing. This type of pipelining ensures that the host CPU is not idle while the accelerator executes the required operations.

2.6 Summary

This chapter provided an overview of the background concepts required to understand the thesis. The topics discussed are tailored to the design of efficient hardware accelerators for DNN inference on resource-constrained FPGA-based edge devices.

We first introduce the key concepts of DNNs in Section 2.1, including their architecture, training, and inference processes, along with the different types of DNNs which are used throughout the thesis. We then discuss the fundamental software libraries used throughout the thesis and their roles across the different chapters in Section 2.2. The overview of the hardware concepts is explained in Section 2.3, including details of FPGAs, accelerators and hardware development processes. Then, we introduce the key

algorithms and techniques used throughout the thesis in Section 2.4, and finally, we elaborate on the fundamental concepts of hardware-software co-design in Section 2.5. The following chapter will present the landscape of related works and the state-of-the-art works relevant to this thesis.

3 | Related Work

This chapter provides a survey of the related work in the area of Deep Neural Networks (DNNs) acceleration, focusing mainly on the works relevant to the research conducted in this thesis. Section 3.1 provides an overview of the key DNN models used in this thesis and some of the most popular DNNs used in the literature. Section 3.2 provides an overview of the key DNN accelerators and a detailed look at GAN and LLM accelerators. Finally, Section 3.3 looks at relevant hardware-software co-design methodologies, frameworks and tools that enable the development of efficient DNN accelerators.

3.1 Deep Neural Networks

As mentioned in Section 2.1, DNNs have been adopted to tackle a wide range of problems in various domains. Additionally, different types of DNNs have been developed to address different types of problems and data. Throughout the thesis, we will focus on three types of DNNs: CNNs, Transformers (including LLMs), and GANs, as these provide interesting and challenging workloads for accelerating on resource-constrained edge devices. This section provides an overview of the key DNN models within these three types of DNNs, some of which form part of the experiments and evaluations conducted in this thesis.

3.1.1 Convolutional Neural Networks

CNNs models, typically used for image recognition tasks, are one of the most well-studied and popular in machine learning, hence throughout this thesis we chose to use them as part of target workloads for hardware evaluation. The following models are some of the CNNs that are used within Chapters 4 and Chapter 5.

Inception [Sze+15]:, also known as GoogLeNet, is a family of CNN models that introduced the concept of 'inception modules', which consist of multiple parallel convolutional layers with different kernel sizes and strides. The inception modules consist

of convolution operations with different kernel sizes, along with a max-pooling operation. The inception modules enabled increased parameter efficiency (i.e., maintain high accuracy with fewer parameters).

ResNets [He+16]: is a family of CNN models that introduced the concept of residual learning, which allows for the training of very deep networks by using skip connections between convolution and identity 'blocks' to solve the vanishing gradient problem [BSF94].

MobileNets [How+17]: is a family of lightweight CNN models designed for mobile and embedded devices. The MobileNet architecture is based on standard 2D convolutional layers, but its speciality comes from the use of depthwise separable convolutions, which consist of a depthwise convolution followed by a pointwise convolution, to reduce the computation complexity of the model. The depthwise convolution applies a single filter to each input channel instead of all filters to each input channel; while the pointwise convolution applies a 1×1 convolution to combine the output of the depthwise convolution.

EfficientNets [TL19]: is a family CNN models based on the idea that properly scaling the model depth, width, and resolution can improve the model's performance. To do so, EfficientNets use a compound scaling method to scale the model's depth, width, and resolution uniformly.

3.1.2 Transformer Models

Transformers models are a type of DNN architecture based on the attention mechanism [Vas+17] that has been widely adopted in natural language processing (NLP) tasks. The attention mechanism allows the model to weigh the importance of different parts of the input sequence and dynamically update their influence on the output. This is important to keep track of the context of the information within the input sequence when generating the output sequence. Computationally, the core of the attention mechanism can be broken down into matrix-matrix multiplication operations or matrix-vector multiplication operations. Transformer models have become increasingly popular through the notion of Large Language Models (LLMs), which are large models trained on vast amounts of text data to learn the underlying structure of the language. In this thesis, we investigated smaller transformer models, alongside LLMs, to design new hardware for them in a resource-constrained environment.

BERT [Dev+19]: is a transformer model that introduced the concept of bidirectional encoders. This enables the model to learn the context of a word based on the entire input sequence, rather than just looking at the text from left to right or right to left.

MobileBERT [Sun+20]: is a lightweight version of BERT designed for mobile and embedded devices. It is model learned from 'BERTLARGE', the largest version of BERT, using a knowledge distillation technique [HVD15] to distil the knowledge from the large 'teacher' model to the smaller 'student' model.

TinyBERT [Jia+20]: is a lightweight version of BERT designed for mobile and embedded devices. Similar to MobileBERT, TinyBERT is based on a novel knowledge distillation technique that contains two stages of distillation: one for pre-training and one for fine-tuning for the target task.

ALBERT [Lan+19]: is a transformer model that introduced the concept of factorised embedding parameterisation which decomposes the embedding matrix into two smaller matrices. This reduces the number of parameters in the model, which can help to reduce the memory footprint and improve the model's performance.

GPT (Generative Pre-trained Transformer) [Bro+20]: is family of transformer models, which are larger as the version number increases. It uses autoregressive language models to generate text by predicting the next word in a sequence based on the previous words.

We utilise MobileBERT and Albert within the experiments in Chapter 5 to evaluate performance of our Fully-Connected layer accelerator.

3.1.3 Generative Adversarial Networks

GANs [Goo+14] are a type of DNN architecture that consists of two networks, a generator and a discriminator, that are trained simultaneously. The dual model aspect of GANs makes them particularly interesting for hardware acceleration, as both models can be optimised to run efficiently on hardware accelerators. We focus on the generative aspect of GANs, as they have different characteristics compared to other DNN models, which make them difficult to accelerate on hardware accelerators. See Chapter 7 for more details on the challenges of accelerating GANs.

DCGAN [RMC16]: is a GAN model that improves upon the original GAN model by using strided convolutions for discriminator and transposed convolutions for the generator.

StyleGAN [KLA21]: is a GAN model that introduces the concept of style-based generator, which generates images at starting from low-resolution to high-resolution.

We utilise DCGAN within the experiments in Chapter 7 to evaluate performance of our Transposed Convolution accelerator.

3.2 DNN Acceleration

In this section, we provide an overview of specialised DNN inference accelerators. First, we provide an overview of DNN accelerators, followed by a discussion of hardware acceleration of Generative Adversarial Networks (GANs) and Large Language Models (LLMs). Finally, we discuss the role of quantisation in DNN acceleration, along with the notable works in this area.

3.2.1 Overview of DNN Accelerators

Due to the popular nature of DNNs, there has been a significant amount of research into developing specialised hardware accelerators to accelerate DNN inference. Many of these accelerators are designed to exploit the parallelism and regularity of convolutional layers within the models. Here we give an overview of the relevant works in the overall field of DNN accelerators.

Early Accelerators

DianNao [Che+14a], introduced in 2014, was one of the first ASIC-based DNN accelerators to achieve high throughput and energy efficiency. DianNao consisted of NFUs (Neural Functional Units), capable of performing arithmetic operations required for convolutions and activation functions, supporting 16-bit fixed-point arithmetic operations. Overall, this design was able to accelerate CNNs, achieving up to 452 GOPS.

Early works, such as Eyeriss [Che+17], FlexFlow [Lu+17] and the fused-layer CNN accelerator [Alw+16], focused on the dataflow aspect of CNN accelerators. Eyeriss [Che+17] optimised for energy efficiency by mainly exploiting dataflow. It utilises row-stationary dataflow, which is reconfigurable for the computation shape of a given convolutional problem. This reconfigurability allows Eyeriss to increase data reuse and reduce the expensive data access to DRAM. FlexFlow [Lu+17] explored and supported three different dataflows: systolic array, 2D-mapping, and tiled, through the addition of local buffers within the PE microarchitecture. This flexibility allows FlexFlow to adapt to different convolutional layer shapes and sizes.

While Eyeriss and FlexFlow focused on dataflow within each convolutional layer, the fused-layer CNN accelerator [Alw+16] focused on dataflow across layers. Alwani et al. designed a CNN accelerator that removes the need for off-chip memory access between layers by re-ordering how the input data is sent to the accelerator in order to fuse the processing of multiple consecutive CNN layers.

Reconfigurable Architectures

FPGA-based accelerators are often designed with popular DNN models in mind that are usually experimented on. However, ML experts develop new DNN models with different types of layers and configurations as they progress their research. Previous accelerators could be adapted to run these new layers. However, they will not be as computationally efficient as an accelerator built with the latest types of layers and configurations in mind. For just CNNs, many possible configurations and optimisations can be made; looking at different types of DNNs such as Recurrent/LSTM neural networks[She20], it becomes easy to understand the high demand in flexibility that is required to design accelerators for DNNs.

FPGA-based reconfigurable accelerators include the VTA [Mor+19] accelerator, which creates a programmable DNN accelerator architecture that is interfaced through a micro-ISA. Changes in architecture parameters, such as buffer sizes and data bus widths, can tune the VTA architecture to a specific DNN model. The VTA accelerator is supported by the state-of-the-art compilation framework TVM [Che+18b]. The integration with TVM enables DNN models to be compiled for inference through TVM specifically for the VTA accelerator, encoding VTA-specific instructions within the compiled binary. VTA can optionally leverage the AutoTVM tuning tool [Che+18a] for additional design space exploration. Later on the thesis in Chapter 5, we will compare the performance of our accelerator with the VTA accelerator. VTA is a good candidate for comparison as it is an FPGA-based accelerator following the **DNN-To-FPGA-Accelerator** acceleration paradigm (see Section 1.1.3), similar to the accelerator we propose in this thesis. VTA also fits the resource-constrained edge device criteria as it uses the same Zynq-7020 SoC as the one used in our experiments.

Another approach to reconfigurability is through partitioning the available FPGA resources into multiple processing elements, each element suited for different layers within a given DNN model; the partitioning is performed via an automated design tool which can take into account the main layers of a given model [SFM17].

Works such as Plasticine [Pra+17] and MAERI [KSK18] also look into providing a reconfigurable architecture. Plasticine was designed with parallelism in mind to ensure that reconfigurability does not come at the cost of performance. Meanwhile, MAERI is a reconfigurable DNN accelerator that exploits data reuse during DNN execution. MAERI's reprogrammable architecture, which depends on small switches, allows a diverse set of mapping strategies to be applied to the accelerator, which overall increases resource utilisation.

Application-Specific Integrated Circuits

Due to the nature of AI workloads, the demand for high-throughput and low-latency inference has led for industry-based solutions such as the NVIDIA Deep Learning Accelerator (NVDLA) [NVIa]. NVDLA is a configurable DNN accelerator that is designed to be scalable and flexible, supporting a wide range of DNN models and applications. One of the main components of NVDLA is the convolution core, which is designed to support a wide range of convolutional layer shapes and sizes.

Google has also developed the Tensor Processing Unit (TPU) [Jou+17], a custom ASIC designed to accelerate DNN inference. The original TPU core consisted of 65,536 8-bit MACs within its matrix multiply unit, which is capable of performing up to 23 TOPS. Since the original TPU core, Google has developed many versions of the TPU, one of the latest, TPUv5 is capable of 393 TOPs, with 8-bit integer operations. Similarly, Google has developed the Coral Edge TPU [Ses+22], a custom ASIC designed for edge devices; the chip can be accessible as a solder-able module or as different products such as a development board. It can perform inference on TFLite models, which are compiled with their custom compiler [Cor].

Many industry-based NPUs [Bou+20; Esm+12; Jan+21] have been proposed; these are typically integrated with the latest CPU cores. Intel's NPUs [Int24], which are integrated into the Intel Core Ultra processors, consists of two 'Neural Compute Engines' which contain MAC array alongside some processing units for activation functions; the latest Intel NPU design is capable of performing up to 48 TOPS. Similarly, the AMD's NPUs [AMD24] are integrated within the new XDNA architecture and consist of AI Engine (AIE) tiles, where each AI engine tile contains a memory module and an AI engine, which is essentially a vector processor; the latest AMD NPU design within the XDNA2 architecture is capable of performing up to 50 TOPS. Additionally, AMD's Versal AI Cores [AMDa] also contain the AI Engine (AIE) tiles alongside programmable logic, DSP engines and two sets of CPU cores (application processor and real-time processor).

3.2.2 Accelerating Generative Adversarial Networks

GANs consists of a generator and a discriminator, that are trained simultaneously. Discriminative component of GANs consists mostly of convolutional layers which can be accelerated using traditional DNN accelerators. Here we focus on accelerator design developed to accelerate the transposed convolution (TCONV) operation, which are used in the generative component of GANs to 'upscale' input data.

Methods for Implementing TCONV

As mentioned in Section 2.4.3, there are several methods for implementing TCONV; each has advantages and disadvantages. For example, the Zero-Insertion [Yu+20] and Transforming Deconvolution to Convolution (TDC) methods [CKK20] have computational and transformation overheads. Thus, researchers have been focusing on the Input-Orientated-Mapping (IOM) method [Yan+18] for TCONV. IOM reduces the number of operations required to perform TCONV without requiring additional padding or transformation to inputs or weights. However, implementing IOM on resource-constrained edge devices requires a hardware-software co-designed solution to ensure optimised tiling and offloading of the TCONV operation to the accelerator, while tackling three key interlinked problems efficiently: (i) storing intermediate/partial results; (ii) processing overlapping sums; (iii) handling cropped outputs. First, partial results should be stored to reduce the latency of sending data back to the main memory. This means that on a device with limited memory space, storage of results should be optimised so that it takes up minimal space. Second, the overlapping sum problem occurs when partial results produced by spatially separate dot products must be coalesced to create a single output value. Since the spatial locality of the partial results corresponding to a single final output varies for each output and also between the TCONV problem dimension, creating a specialised accelerator to handle the complex output mapping efficiently becomes challenging. Finally, the standard IOM approach creates additional output data that needs to be cropped from the final results to maintain consistent dimensions across the model execution. This cropping process not only leads to additional overhead but also ineffective computations since the cropped output values are computed simply to be dropped later.

However, existing solutions [Ma+22; SPS23; Xu+18; Zha+17] do not tackle these three problems efficiently for resource-constrained edge devices. They especially neglect the issue of the cropped outputs, including the ineffectual computations (up to 28%), to calculate the outputs before being dropped.

TCONV Accelerators

Given their significance in GAN and overall generative models, there has been a growing interest in developing accelerators that efficiently perform the TCONV operation. Different accelerator architectures have been proposed, employing methods such as TDC [CKK20], Winograd-Transformed Transposed Convolution [Cha+20; Di+20], and the Zero-Insert TCONV method [Yu+20]. Although these approaches have shown some effectiveness, they still have transformation overheads due to algorithmic limitations inherent to their respective methods.

Some works have proposed implementing the TCONV operation using the IOM method on FPGAs. For example, Ma et al., [Ma+22] exploit the intermediate-centric dataflow, a variation on the IOM method, but their accelerator only supports fixed dimensions for given problems. The HLS template-based approach proposed by Sestito et al. [SPS23] suffers from the same constraint. Although they can adjust their accelerator for different problem sizes, this requires re-synthesis and re-mapping of the accelerator. Additionally, these implementations target large FPGAs with MBs of on-chip memory and do not consider constrained edge devices with limited on-chip memory. Zhang et al. [Zha+17] proposed a design suitable for edge devices. However, their output-oriented approach solves the overlapping sum problem but introduces hardware complexity, degrading the accelerator's performance.

Other works such as GNA [Yan+18] and FCN-Engine [Xu+18] exploit the IOM method with ASIC designs, but similar to all previously mentioned IOM-based approaches they do not consider the cropped outputs. Therefore, they perform ineffectual computations of output pixels that are not required. Thus, they need an additional cropping operation to produce the final output.

3.2.3 Accelerating Large Language Models

Large Language Models (LLMs) have become popular in recent years due to their ability to generate human-like text. However, as the size of LLMs increases, the computational demand for LLM inference also increases. Hence, a myriad of research studies have developed accelerators for LLM inference. As LLMs are based on the transformer architecture, the accelerators developed for LLMs focus on accelerating the attention operation, the key component of the transformer architecture.

Some works have focused on accelerating LLMs/transformers by improving the algorithmic efficiency of the attention operation. The A^3 accelerator [Ham+20] focuses on accelerating the attention operation using algorithmic approximation. A^3 approximates the attention operation by pre-processing the key matrix to obtain the likely set of rows to score a high value during the key-query matrix-vector multiplication. With their proposed algorithmic approximation scheme, the A^3 accelerator modules enable partial skipping of the dot product and softmax operations at the start of the attention mechanism. Meanwhile, the FTRANS [Li+20] work looks at reducing the weight footprint within the transformer architecture by proposing an enhanced block-circulant matrix representation for the weight compression which maintains the accuracy of the model while achieving a $16x$ compression ratio. To support their new approach, they propose a hardware accelerator consisting of computational units for the multi-head attention mechanism, the feed-forward computations, add, and norm operations.

Other works have also focused on the multi-head attention (MHA) mechanism. For example, Lu et al. [Lu+20] proposed a hardware solution that accelerates the MHA mechanism. They propose a customised hardware accelerator for two of the largest components, in terms of the number of trainable parameters, within transformer networks: the BHA ResBlock and the FNN (feed-forward network) ResBlock. By partitioning the matrices in the MHA and FNN ResBlocks, they reuse the systolic array to perform the matrix multiplication for both.

3.2.4 Quantisation-based Accelerators

Quantisation is a technique based on reducing the number of bits used to represent weights and input data. DNN models are normally trained at 32-bit float precision, which means that, theoretically, quantisation can reduce the model size by 32x if binary quantisation is applied to the target DNN model. Hence, there is considerable potential for performance gains through quantisation. Therefore, quantisation has become a key approach to reducing not only the on-chip storage requirements of custom hardware solutions but also computational demands and alleviating bottlenecks caused by limited bandwidth between on-chip and off-chip data transfers. Reducing the precision of both the trained weights and the input activations can lead to a significant reduction in data footprint, up to 92%, while only degrading accuracy by 1% [Gup+15; Jud+16a].

Studies look at applying quantisation at different levels of DNN models. Initial works looked at applying model wide quantisation [Gon+14; HMD16], but more recent works show that re-evaluating the precision required for each layer of a model can lead to a greater reduction in size while maintaining small accuracy loss [Jud+16a]. Later works investigated region-based quantisation schemes which define groups within the same layer of weights and apply different quantisation schemes per group [Zho+17].

The hardware side of DNN inference is adapting to keep up with these model-based optimisations. GPUs [Gup+15] and specialised accelerator [Che+14b] solutions have started to support lower precision from the standard 32-bit float to 16-bit fixed-point representation. Hardware-based solutions such as Stripes [Jud+16b] enable per-layer choice of precision through the use of serial multiplication to reduce the precision of layers independently of the precision requirements of other layers. Bit-Fusion [Sha+18] allows for the same level of control by implementing a composable accelerator that is able to dynamically combine bit-level processing nodes to create higher precision processing nodes. This idea has been further developed through DRQ [Son+20], which allows for region quantisation to be accelerated by switching between a higher and lower precision processing for different regions of weights and activation for any given layer.

With the advent of Transformer/BERT-based models, quantisation is becoming more prevalent due to the large memory footprint of these models. Quantisation not only helps alleviate the memory space requirements to ensure that smaller edge devices can support these new types of DNNs, but it also enables low-precision computing which decreases the computational demand. Furthermore, this leads to bespoke accelerator designs targeting quantised BERT models [LLC21] to fully take advantage of low-precision computation.

3.3 Hardware-Software Co-Design of Accelerators

Hardware-software co-design is fundamental to the performance of hardware-based DNN acceleration. In this section, we review a wide range of hardware-software co-design related works, including design methodologies, frameworks and tools that enable the co-design of hardware accelerators in general, and also specifically for DNNs.

3.3.1 Co-Design Methodologies

The focus within the machine learning community has been on developing more complex models which give a higher QoR (Quality of Results), hence DNNs are designed such that the desired QoR is achieved but without considering the target hardware platform where they will run on. Cong et al., [Hao+19] propose that co-designing the accelerator and DNN will provide a greater opportunity to optimise and get a more favourable result. This idea was realised by their four-component system which was able to produce good results with low power consumption and high energy efficiency. While their proposed methodology allows the user to create accelerators with the model in mind, they limit the potential optimisations and improvements of the accelerator by using a fixed architecture template.

While templated-based approaches enable automated design space exploration (DSE), they can limit potential architectural exploration, for example, the memory hierarchy, the dataflow, and the number of processing elements. Hence, it is important to have a design methodology that allows for the co-design of the accelerator and the DNN model, while also enabling designing fundamentally new architectures. Additionally, templated-based approaches can be adopted to provide further exploration, once the initial architecture has been defined. The following design methodologies provide a more flexible approach to designing the initial accelerator architecture.

OpenCL [SGS10] uses a host-device programming model, where the host code (i.e., the driver) prepares and transfers data to be executed by the device (i.e., the accelerator).

Hence, the approach allows for the co-design of the driver. The device code is written in high-level OpenCL code which defines computation kernels that perform the processing of the target workload. This high-level code is translated into a synthesisable hardware design. The designer defines the computation kernels to be accelerated, being able to configure the number of hardware instances each kernel is allocated. The higher the number of instances, the greater number of instructions executed in parallel. The level of *design control* offered by OpenCL's programming model can be restrictive, since the designer cannot easily define the low-level behaviour of the accelerator, such as at the transaction level, which can enable control of each subcomponent and interfaces between them. The Intel FPGA SDK for OpenCL[Alt11] allows for emulation of accelerator designs on x86 machines, which allows for verification of the behaviour of the design. To gather dynamic performance of the hardware accelerator, the designer has to perform slow cycle-accurate simulation, or hardware profiling on the target FPGA, which is very time consuming.

Hardware Description Language (HDL) based design flows use highly detailed hardware descriptions in languages such as Verilog [Des06] and VHDL [Des19], to define the desired behaviour of the accelerator. While this approach allows for fine-grained hardware designs, it comes with high development time, resulting in high code-base complexity and strict size requirements to define a design [Pel+16], as compared to HLS or OpenCL-based solutions. Additionally, although HDL solutions can use RTL simulator to provide cycle-accurate simulation, the level of simulation detail makes the process much slower than non-RTL based simulations. An HDL-based approach to designing accelerators does not lend itself well to co-design the host driver, or end-to-end evaluation, since RTL simulators are testbench based and inherently slow.

SMAUG [Xi+20] provides a simulation-based design methodology that uses gem5-Aladdin [Sha+16a] to perform full system simulation of the host system, the off-chip memory accesses and the accelerator design itself. While this approach provides high fidelity in terms of design performance insights, the simulation speed is very slow due to simulation of the entire system (e.g., several hours for ResNet50). Rather than integrating with an existing DNN framework, models must be redefined using SMAUG's Python API. In addition, SMAUG does not offer an approach where a design can be directly synthesised to a target FPGA and integrated with a DNN framework of choice.

SYCL [Rey+20] is similar to OpenCL, but provides a higher level of abstraction, allowing for programming of heterogeneous devices. SYCL is a single-source programming model that allows for the development of host-code that can be executed on the CPU and also the kernel code that is executed on the target hardware. An advantage of SYCL is that it allows for the development of code that can be executed on a

range of devices, including CPUs, GPUs, and FPGAs. For FPGAs, triSYCL [Goz+20] is an open-source implementation of SYCL for Xilinx FPGAs that allows HLS-based development of FPGA accelerators, where the kernel code is written in SYCL and then HLS tools are used to generate the hardware design. Unfortunately, the support from triSYCL is limited to Xilinx FPGAs, and the level of design control is limited to the kernel code, thus the designer cannot easily define the low-level behaviour of the accelerator as possible with HDL-based designs.

SystemC-HLS [acc16] is a high-level synthesis (HLS) approach that uses a subset of SystemC [**davisSystemCIEEEStandard2005**] to define the hardware designs and utilise HLS tools to generate RTL designs. The benefit of using SystemC is that it allows for the development of hardware designs that can be simulated at a higher level of abstraction, enabling faster simulation and design exploration. However, the SystemC-HLS is a combination of tools that can be used together to design accelerators, but there are no guidelines or defined methodologies for designing hardware accelerators, especially DNN accelerators. Chapter 4 presents the SECDA design methodology, which will supersede SystemC-HLS and provide a complete design flow for DNN accelerators.

3.3.2 Co-Design Frameworks

Here we discuss the frameworks used to enable hardware-software co-design of DNN accelerators. This includes frameworks which are used for: designing and generating FPGA-based accelerators, enabling simulation of specialised hardware accelerators, as well as frameworks for developing SoC integrated accelerators.

Template-based Generation of FPGA-based Accelerator

Due to performance and power constraints, FPGA-based solutions for DNN inference are often sought for edge devices. Hence, various tools enable the development and deployment of FPGA-based accelerators for DNNs. These tools often use a template-based approach to create FPGA-based accelerators to reduce the DSE and development time.

DeepBurning [Wan+16] and DNNWeaver [Sha+16b] are two of the earlier works in this area. DeepBurning contains a library of basic neural components which are used by the hardware generator to create the accelerator. DNNWeaver [Sha+16b] translates the DNN specification into an ISA (Instruction Set Architecture) based on the operations of the DNN model. Using this ISA, their own tiling and scheduling algorithms

combined with handwritten hardware designs, they can generate the accelerator from Caffe [Jia+14] defined models.

Similarly, FP-DNN [Gua+17] provides a framework that allows for end-to-end automation of the accelerator creation process. The user provides a symbolic description of a DNN via a TensorFlow model, and the tool will output an FPGA accelerator optimised according to that description. The accelerator is created using the author's RTL-HLS hybrid templates, where RTL is used to design the computation engine, and HLS is used to implement the control logic.

DNNBuilder [Zha+18] is another framework for generating FPGA-based accelerators using a template-based approach but with pre-built RTL components, allowing them to perform a DSE to find the best performance according to their estimator. Additionally, approaches like FINN [Umu+17] are more focused on the model-based co-design of the accelerator. FINN is an ML dataflow-based compiler framework that provides an end-to-end flow for exploring and implementing low-bit ($<= 8bits$) quantised DNN inference on FPGAs. It uses a templated-based approach for generating hardware accelerators, where an existing HLS and RTL-based library of components are used to implement each individual layer of the DNN model. Similarly, HLS [Fah+21] is a workflow that follows the **DNN-to-FPGA-Dataflow** acceleration paradigm, where the DNN model is first converted to a compressed model, and then the compressed model is used to generate a hardware design using HLS tools. The approach is not entirely template-based but uses pre-defined optimisation to convert the model into HLS-ready code. While the approach provides a high level of abstraction, allowing non-hardware experts to develop DNN accelerators rapidly, it has inherent limitations for resource-constrained devices. The generated hardware design is monolithic, and it needs enough resources to map the entire DNN fully, limiting the mapping of bigger models onto resource-constrained FPGA fabric.

More recently, approaches like DSAGEN [Wen+20] have been developed to provide a more flexible approach to generating FPGA-based accelerators. DSAGEN uses an architecture description graph combined with modular spatial architecture components (e.g., PEs, Switches, Memory) to enable greater architectural flexibility while still enabling automated design space exploration. The modular components enable DSAGEN to express designs similar to pre-existing accelerator designs such as MAERI [KSK18] and SoftBrain [Now+17].

While these approaches provide a quick and easy way to generate accelerators, they are limited by the templates and pre-defined components that the specific framework provides. This can limit the potential optimisations that can be achieved, especially when considering that resource-constrained devices can gain significant performance

improvements by fine-grain workload-specific architectural optimisations. Additionally, template-based approaches are desirable alongside co-design methodologies, as the co-design methodologies can be used to define the initial architecture, and then template-based approaches can be used to explore the design space of that architecture.

Exclusively Simulation-based Approaches

There are a range of exclusively simulation-based tools that enable the development and evaluation of hardware accelerators, these approaches rely purely on simulations to design, profile and evaluate hardware accelerators. For example, SystemC simulation has been used as a part of co-design methodologies in other domains such as cryptographic SoCs [KH08] and image processing [CHZ11], which demonstrate the streamlined development time advantages of leveraging SystemC.

In terms of DNN accelerators, TFLITE-SOC [Ago+20] features the use of SystemC to perform full end-to-end simulation of the DNN model and the accelerator design. Using TFLite-based DNN models, the framework can provide insights into the performance of the accelerator design on a per-layer basis.

Other exclusively simulation-based approaches have also been proposed for DNN accelerator design: STONNE [Muñ+21] provides cycle-accurate simulation for deep learning accelerator designs such as MAERI [KSK18] and SIGMA [Qin+20]. However, STONNE does not integrate full system simulation as SMAUG [Xi+20]. The drawbacks of these exclusively simulation-based approaches are that they often do not have a direct path to mapping candidate designs to real hardware and running hardware evaluation on target FPGA devices with the chosen DNN framework. Hence, these design approaches, while fruitful, can be inaccurate and time-consuming compared to other hybrid approaches, due to lack of real hardware evaluation.

Accelerator Development for SoC

As SoCs enable tight integration of hardware accelerators with the host system, there is a need for tools that facilitate the design of specialised accelerators for SoCs. Open source frameworks such as Chipyard [Ami+20] enable the development of Chisel-based [Bac+12] SoCs. Chisel is a hardware construction language that allows for generating hardware accelerators from DNN models. Chipyard provides integration with different RISC-V processor cores, such as Rocket Chip [Asa+16], BOOM [Zha+20b] and CVA6 [ZB19], and provides access to accelerators such as Gemmini [Gen+21] and NVDLA [NVIa]. Chipyard supports multiple development flows, including RTL and FPGA-based simulation using FireSim [Kar+18a].

Similarly, the ESP project [Man+20] provides an open-source platform for heterogeneous SoC development. It allows for the integration of custom accelerators in tiled-based SoC configurations. Similar to Chipyard, ESP provides integration with RISC-V cores and accelerators such as the CVA6 [ZB19] and NVDLA [NVIa]. Additionally, ESP supports various accelerator design and integration flows, such as SystemC with Stratus HLS and Chisel. Like Chipyard, ESP supports RTL and FPGA-based simulation for prototyping hardware accelerator-enabled SoCs.

3.3.3 Code-Generation for specialised Accelerators

CPUs and GPUs which have compilers and scheduling tools [Che+18b; GC20; GC23] to enable efficient code-generation for a target DNN problem, but these tools do not directly support code-generation for specialised accelerators. Hence, one of the key challenges in developing specialised accelerators is to create efficient software drivers that can interface with the accelerator.

Due to wide range of possibilities in accelerator design, hence code generation could enable rapid development of specialised accelerators and deployment of DNN models on these accelerators. Alas, there are few tools that provide code generation for specialised accelerators, as most tools focus on generating code for existing accelerators, or subset of accelerator design that can be captured within their custom hardware description language. Here we discuss the tools existing in the literature that provide code generation for specialised accelerators.

HeteroFlow [Xia+22], which extends the work of HeteroCL [Lai+19], provides a programming model that allows the decoupling of implementation of the target algorithm from the data placement/movement to the custom memory hierarchy of the accelerator. By using the '*.to()*' primitive, it is able to specify data placement at different levels of the memory hierarchy. HeteroFlow is limited to accelerator co-design within HeteroCL, and does not necessarily provide code generation for fully custom accelerators.

Other tools such as Interstellar [Yan+20], DMazeRunner [Dav+19] and Bifrost [SGC22] delve into the challenge of mapping algorithms into specialised accelerators. For example, Interstellar modifies the Halide [RBA] compiler to show that dataflow and micro-architecture of existing accelerators can be expressed as schedules within Halide. Additionally, it then uses the Halide compiler to generate the hardware designs for different DNN accelerators, meanwhile optimising the memory hierarchy to improve the energy efficiency. Moreover, DMazeRunner provides a tool for exploring the design space of mapping nested loops onto dataflow accelerators to accurately estimate the performance of the accelerator design. Finally, Bifrost connects the TVM [Che+18b]

compiler with the STONNE [Muñ+21] accelerator simulator, enabling more straightforward exploration of compiler-hardware co-design exploration using STONNE.

3.4 Summary

In this chapter, we provided an overview of the relevant related works, first by discussing the state-of-the-art DNN models, including CNNs, Transformers, and GANs in Section 3.1. We highlight the importance of these models in the context of DNN accelerators, as they are the primary workloads we aim to accelerate throughout this thesis.

Then, the state-of-the-art in DNN acceleration was addressed in Section 3.2, which consisted of an overview of DNN accelerators, including GANs and LLMs accelerators and quantisation-based acceleration. We highlighted the variety of DNN accelerators, including early work in the DNN accelerator domain, reconfigurable architectures and ASIC accelerators. Additionally, we discussed the key related accelerator designs that are relevant to the work presented in the later chapters of this thesis, including the design of accelerators for GANs and LLMs.

Finally, we discussed the related works in terms of the accelerator co-design process, discussing the design methodologies, frameworks, and tools used in the design and deployment of DNN accelerators in Section 3.3. We elaborate on the drawbacks of the current design methodologies, which often lack simulation speed, design control, or a mixture of necessary design features. This highlights the need for a better design methodology to address the challenges in the design and deployment of DNN accelerators. The discussion on the co-design frameworks summarises related works that present specialised solutions to the challenges in the design and deployment of DNN accelerators, some of which could be combined with the work presented in this thesis.

In the next chapter, we introduce the SECDA methodology, one of this thesis's foundational contributions. This methodology provides a systematic approach to designing FPGA-based DNN accelerators.

4 SECDA

Within Section 1.2.1, we highlight one of the challenges of designing DNN hardware accelerators, which is the high effort required for the development of new accelerators; resolving this challenge is the first key objective of this thesis. Hence, this chapter introduces one of the core contributions of this thesis, the SystemC Enabled Co-design of DNN Accelerators (SECDA) methodology.

The chapter is structured as follows: Sections 4.1 and 4.2 introduce and motivate the need for the new design methodology. Section 4.3 presents the SECDA methodology in detail. Section 4.4 presents a case study using the SECDA methodology, which is evaluated in Section 4.5. Finally, Section 4.6 summarises the chapter.

4.1 Introduction

Due to the increasing popularity of Deep Neural Networks (DNNs) in a wide range of applications and the advent of new DNN architectures that require more computational resources, the task of deploying DNNs on edge devices has become more demanding. To meet this demand, hardware-based optimisations to reduce DNN inference compute and power requirements is an active area of research and include ISA-level extensions to CPUs and GPUs [Mar+18; Ott+20], as well as TPUs [Jou+17] and other custom hardware solutions for FPGAs and ASICs [Che+19; KSK18].

As discussed in Section 1.1, FPGA-based accelerators are ideal for the ever-changing DNN workloads, as they provide a reconfigurable fabric that can be programmed to support new specialised hardware accelerators. Hence, we dedicate our efforts to developing FPGA-based DNN accelerators for resource-constrained edge devices.

After establishing the system model (discussed in Section 1.1.2) to set the scope of this work, we tried to design FPGA-based DNN accelerators for edge devices. Our initial attempts to design novel and efficient DNN accelerators were challenging, time-consuming, and unclear. The design process of new FPGA-based hardware accelerators

for DNNs on resource-constrained edge devices required great engineering effort, time, and hardware-software expertise.

As discussed in Section 3.3, there are a handful of design methodologies for developing DNN accelerators, along with some frameworks and tools that can automate the generation of hardware accelerators from pre-defined hardware templates. However, the process of developing FPGA-based DNN accelerators for resource-constrained devices using these methodologies and tools is under-documented, and prior work focuses on new accelerator architectural features and results, rather than the design methodologies and the process of developing the accelerators. Since resources are more limited on edge FPGAs, and DNN workloads are large in terms of their memory footprint and computational demands, a given DNN model is unlikely to fit fully on an accelerator. Thus, for inference, the accelerator must operate in close communication with the CPU, which requires careful co-design with the host CPU code to ensure that data is managed efficiently. Therefore, an effective design methodology for a DNN accelerator design should, for a given set of hardware resource constraints, produce performant accelerators that effectively leverage available resources and can respond quickly to changing workload requirements (e.g., introduction of new types of layers or operations).

Additionally, logic synthesis, the process that is used to map candidate hardware designs to an FPGA (see Section 2.3.4), is a time-consuming process that can take from tens of minutes to hours, depending on the complexity of the design, and the target FPGA device. Compounding the long synthesis times with the number of iterations in a typical design process, synthesis can create a clear bottleneck in the hardware development process. Existing solutions either accept the synthesis time overhead [Liu+11], surrender low-level design fidelity [SGS10], or develop accelerators using a purely simulation-based approach [Muñ+21; Xi+20] that frequently results in non-synthesisable hardware solutions.

The difficulties of designing our initial FPGA-based DNN accelerator and the lack of effective design methodologies to guide the process motivated us to propose SECDA (*SystemC Enabled Co-design of DNN Accelerators*), a new hardware-software co-design methodology to efficiently produce optimised DNN inference accelerators for edge devices using FPGAs.

SECDA uses SystemC [Des23] as an accelerator simulation framework, allowing candidate designs to be iterated upon quickly. Using SystemC High-Level Synthesis (HLS), we can produce a synthesisable design from the same accelerator definition used for simulation. We leverage SystemC's modularity to reduce the time required to make design changes by reusing and adapting existing components.

Co-design of a software accelerator driver and a hardware accelerator is achieved by integrating SystemC's simulation features within the target edge-based DNN framework (e.g. TensorFlow Lite). This integration allows designers to quickly test potential optimisations such as varying data transfer and tiling strategies. Embedding the simulation environment and the hardware accelerator into the same software environment reduces the effort of exploring hardware-software co-design trade-offs via simulation relative to synthesising the design on an FPGA with every change.

Overall, SECDA satisfies the following five key features required for a suitable design methodology for FPGA-based DNN accelerators: *Design Control*; *End-to-end Evaluation*; *Driver Co-Design*; *System Integration*; and *Simulation Speed*. We expand on these features in Section 4.2.

We demonstrate the utility of SECDA with a case study that targets the acceleration of General Matrix Multiplication (GEMM), heavily used in convolutional layers, the most computationally expensive portion of many DNNs [Don+18; Gib+20]. For this case study, we develop two accelerators, the Vector MAC (VM) and Systolic Array (SA) designs. The DNN framework used is TensorFlow Lite (TFLite), a mobile-friendly version of TensorFlow [Aba+16]. The target device is the PNYQ Z1 board [Dig], a platform with a dual-core CPU and an edge FPGA.

The contributions of this chapter are as follows:

- We motivate the need for a better design methodology for DNN accelerators and define five key features that an improved new methodology should consider, which are essential for fast design exploration and integration of hardware accelerators for DNNs using FPGAs.
- We introduce SECDA, a new design methodology to efficiently explore the design space of DNN accelerators for edge devices with FPGAs, and quickly arrive at optimised solutions. We show how SECDA meets the five features and reduces the time to obtain efficient designs.
- We demonstrate the capabilities of SECDA via a case study, where we design two GEMM-based accelerator designs for DNN inference.
- We evaluate our accelerator designs and show that they outperform the CPU baseline.

4.2 Motivation

Here, we further motivate the need for a new design methodology for developing FPGA-based DNN accelerators. To do so, we first clearly distinguish between two stages of developing hardware accelerators and then discuss the key features that a design methodology should include to efficiently produce optimised FPGA-based DNN accelerators. Finally, we compare standard FPGA design methodologies viewed through the lens of the key features presented in Section 4.2.2 and discuss how SECDA addresses the limitations of existing methodologies.

4.2.1 Stages of Hardware Accelerator Development

We separate the development of hardware accelerators for DNNs into two stages. First is the process of designing and developing the accelerator architecture and the software stack to support the accelerator design; this includes defining the hardware primitives that are used to compose accelerator architecture. The first stage requires a hardware-software co-design methodology, such as the ones discussed in Section 3.3.1; this is the focus of this chapter.

The second stage is optional but recommended. This is when the designer exposes aspects of the design as a set of templates with tunable parameters (e.g., buffer sizes and number of processing elements) and allows automated design space exploration tools to find more performant designs. The second stage requires design-space exploration tools/frameworks such as the ones discussed in Section 3.3.2.

We distinguish the two stages to highlight that although there is a rich and growing literature on the second stage, there is a lack of discussion of the design methodologies for the first stage and how they can better accommodate the features of DNNs to address inefficiencies with the current process of creating specialised accelerator solutions, especially for resource-constrained devices.

4.2.2 Key Features of DNN Accelerator Design Methodologies

When designing hardware accelerators for DNNs using constrained FPGAs, two key workload characteristics must be acknowledged. First, although DNNs are getting more efficient [HB20], they are still large programs with ever-increasing memory and computing demands. DNNs typically contain a large number of weights and operations, making them difficult to fit on a resource-constrained FPGA without partitioning them into stages. Secondly, DNNs feature a variety of operations (in terms of layers) with

varying frequencies and computational demands. It may be preferable to use the CPU for less frequent operations and focus accelerator resources on the most expensive layer types. These characteristics mean that the accelerator architecture must be closely designed with the CPU host-side software to ensure efficient workload balancing.

To effectively tackle these two characteristics, we define five key features that accelerator design methodologies used to efficiently produce optimised FPGA-based DNN accelerators should consider:

- *Design Control*: The degree of control given to the designer for both high-level and low-level features, such as the overall dataflow at a high-level and behaviour and interconnection of individual components at a low-level, balancing model depth against overall simplicity.
- *End-to-end Evaluation*: Inference evaluation (either in simulation or in hardware) of full DNN models is vital. This process should be as fast as possible to keep design iterations short. Benchmarking only single layers may cause the designer to miss the bottlenecks that only emerge with realistic workloads.
- *Driver Co-Design*: The interface between the accelerator and the target DNN framework can play a pivotal role in the efficiency and performance of the design [Wan+21; Xi+20]. A good design methodology should enable the designer to co-design the software driver and the hardware accelerator, thus allowing the designer to explore different degrees of workload offloading and create an effective workload balance between the CPU and the accelerator.
- *System Integration*: We need both ease and speed in the process of mapping a proposed accelerator design to an FPGA. This includes the integration of the accelerator with the DNN framework software. The goal is minimal overhead in realising a design on real hardware.
- *Simulation Speed*: Leveraging cycle-based simulation can reduce the time taken for logic synthesis within the design process. Hence, it is crucial that simulation is fast, accurate, and does not become the bottleneck in the design loop.

4.2.3 Comparison of Methodologies

Table 4.1 compares different state-of-the-art design methodologies discussed in Section 3.3.1 based on the previous five key features against SECDA. As shown, SECDA provides a high degree of design control, the ability to perform end-to-end evaluation, driver co-design support and ease of system integration while maintaining fast simulation times.

Table 4.1: Comparison of Design Methodologies with Five Key Features.

Feature \ Methodology	OpenCL	HDL	SMAUG	triSYCL	SECDA
Design Control	Low	Very High	Medium	Low	High
End-to-End Evaluation	✔	✖	✔	✖	✔
Driver Co-Design	✔	✖	✔	✔	✔
System Integration	Simple	Difficult	Simulation Only	Difficult	Simple
Cycle-based Simulation	Slow	Very Slow	Slow	NA	Fast

To begin with, SECDA uses the SystemC programming model to define the behaviour of an accelerator at a transaction level and HLS to produce synthesisable designs. This provides a high degree of design control while mitigating the issues of cumbersome HDLs. SECDA integrates a SystemC simulation environment within the target DNN framework, allowing co-design of the accelerator driver and simulation of end-to-end inference. Unlike SMAUG, SECDA does not simulate the full host system, avoiding the large overheads and keeping simulation times in the order of minutes, as full host system information is not relevant for most design iterations since most design choices are related to the accelerator's performance. Once the accelerator designs are refined through simulation, we can identify issues related to the full system, such as off-chip memory accesses, by leveraging SECDA's HLS capabilities to test on real FPGA hardware. Thus, we avoid the high simulation time seen in methodologies such as SMAUG and expensive hardware synthesis with each design iteration. For the rapid design of DNN accelerators for edge FPGAs, SECDA achieves a good trade-off in terms of simulation fidelity, design granularity, and ease of deployment on real hardware.

Development Time Formulation

We now compare the development time of these methodologies with illustrative estimates of the "idle" time spent waiting to evaluate candidate designs. For SECDA, we compute this time (E_t) with the following equation:

$$E_\mathrm{t} = \#\mathrm{Sim} * (C_\mathrm{t} + IS_\mathrm{t}) + \#\mathrm{Synth} * (S_\mathrm{t} + I_\mathrm{t}) \tag{4.1}$$

Where $\#Sim$ is the number of simulated design iterations we perform; C_t and IS_t are the times to compile and run an end-to-end inference in simulation, respectively; $\#Synth$ is the number of hardware synthesis passes we perform; S_t is the time to perform logic synthesis of the accelerator design; and I_t is the time to perform inference on the FPGA, where typically $I_t \approx C_t < IS_t \ll S_t$. Since the time for S_t dominates the overall time, minimising the number of logic synthesis performed is desirable. Additionally, design methodologies such as OpenCL and HDL would incur significantly higher IS_t than SECDA, since they use cycle-accurate simulations.

We could also follow a design methodology that eliminates simulation and relies only on iterations using logic synthesis. From our understanding, triSYCL does not provide a simulation environment, and the designer must rely on logic synthesis for each design iteration. The equivalent time spent waiting for evaluation results is given by:

$$E_{\mathrm{t}} = (\#\mathrm{Sim} + \#\mathrm{Synth}) * (S_{\mathrm{t}} + I_{\mathrm{t}}) \tag{4.2}$$

Finally, a design methodology using full system simulation to perform all design iterations (e.g., SMAUG) would have a similar idle evaluation time estimate as in Equation 4.2, but with the simulation time replacing the synthesis time.

$$E_{\mathrm{t}} = (\#\mathrm{Sim} + \#\mathrm{Synth}) * (C_{\mathrm{t}} + IS_{\mathrm{t}}) \tag{4.3}$$

However, the simulation time I_{t} would be significantly higher than SECDA's simpler SystemC simulation due to a more complex simulation, which we argue is unnecessary in SECDA due to our two-stage approach. OpenCL, triSYCL and HDL-based methodologies can use either, or a mix of, the approaches characterised by Equations 4.2 and 4.3. However, simulation and synthesis are expensive in terms of development time for these methodologies. In contrast, in SECDA, we take advantage of fast simulation, which is sufficient for most design iterations and only occasional synthesis.

4.3 SECDA Methodology

This section presents the *SECDA* (*SystemC Enabled Co-design of DNN Accelerators*) methodology in detail. SECDA provides fast accelerator design space exploration (DSE), integrates software and hardware design choices, and reduces barriers when evaluating designs in real hardware.

As discussed in Section 4.2.2, SECDA targets DNN inference at the edge, specifically on resource-constrained devices. Based on the characteristics of DNN workloads, SECDA focuses heavily on efficient host-accelerator communication, which requires careful co-design.

Figure 4.1 shows a high-level overview of the proposed methodology. The following sections provide details on the key components of the methodology, including how components are interconnected to form the SECDA design loop.

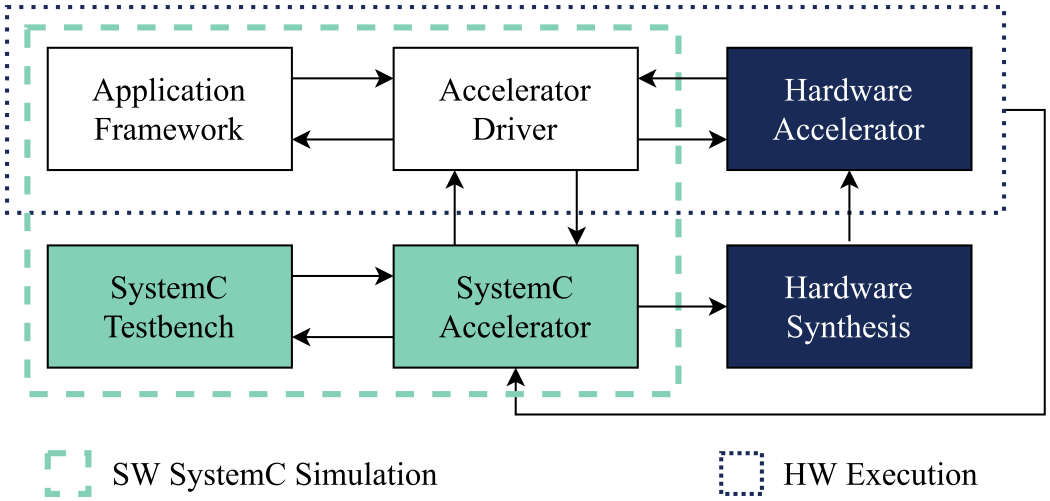


Figure 4.1: Overview of the SECDA methodology. Components in the dashed lines correspond to the design in simulation, and components in the dotted lines correspond to the design running on real hardware. *Application Framework* and *Accelerator Driver* software are common to both.

4.3.1 Application Framework

The *Application Framework* is the DNN software framework which runs the target DNN models, from which we offload work to the accelerator. We characterise it as software that could run the full workload independently of the accelerator, for example, edge inference-specific versions of popular deep learning frameworks such as TensorFlow's [Aba+16] TFLite and PyTorch [Pas+17] Mobile. These frameworks reduce the feature set of the original frameworks (i.e., TensorFlow, PyTorch) to run inference more efficiently using fewer resources.

In SECDA, we ensure that the *Application Framework* is integrated early in the design cycle so that real workloads inform the accelerator development. That co-verification is improved by avoiding software compatibility issues, such as misaligned data or conflicting data types. The SECDA methodology is instantiated with support for running full DNN workloads from the start to ensure that designers have a realistic understanding of the bottlenecks in their designs, and so that they can focus on the most relevant aspects of their design for the target workloads.

4.3.2 Accelerator Driver

The *Accelerator Driver* is the software component in the co-design methodology, the bridge between the *Application Framework* and the hardware accelerator. It is responsible for managing aspects such as data preparation, output data unpacking, control flow and memory management for DMAs, and thread synchronisation. The efficiency of the

Accelerator Driver can be very impactful on the overall runtime performance [Wan+21; Xi+20], hence why driver co-design is a key feature of the SECDA methodology. For example, the design of the input data preparation stage is crucial because the data format of the *Application Framework* may not be suited for a given accelerator design. Non-accelerated CPU code may reshape data to leverage vector instructions, but we may prefer to reshape data differently to leverage the design of a given accelerator's architecture. Thus, we may face co-design trade-offs where we must choose a data format that balances the efficiency of processing it on our hardware design and the efficiency of our CPU-side driver in rearranging the data to and from this format.

With both input preparation and output unpacking stages, the driver should ensure that data transfers between the accelerator and main memory are performed efficiently, since they can dominate both inference time and energy consumption for DNN accelerators [Sze+17]. The driver is also responsible for balancing the workload between the accelerator and CPU and should ensure that the aforementioned stages are pipelined so that the CPU is not idle while the accelerator is working.

We co-design the *Accelerator Driver*, along with the accelerator, in an end-to-end *SystemC Simulation* environment integrated with the *Application Framework*. As observed in Figure 4.1, the *Accelerator Driver* is reused in both simulation and hardware evaluation, the latter giving performance analysis on system components not modelled in detail by the simulation such as off-chip memory accesses.

4.3.3 SystemC Simulation

SystemC [Des23] is a C++ library that models and simulates the behaviour of hardware designs. We use SystemC for Transaction-Level Modelling (TLM) [Ghe05], which simulates complex designs without the overhead of exact register-level details while still ensuring bit-level accuracy. SystemC Simulation is the cornerstone of the SECDA methodology. Using simulation combined with HLS, we can gain insight into candidate designs. SECDA is over an order of magnitude faster than using logic synthesis alone to configure the FPGA in our case study. In SECDA, we use two levels of SystemC Simulation (testbench and end-to-end) to further refine our co-designed steps, one for designing low-level components (such as the processing element design) and the other for evaluating the full accelerator design.

SystemC Testbench simulation is based on unit testing the accelerator design and its components on various input datasets, enabling developers to iteratively design accelerator components without running a full workload. Using SystemC HLS, we feed performance estimates such as clock cycle counts and overall resource utilisation for

each component into the design simulation model. The testbench environment allows for quick design development without the need for compatible drivers to interface with a full-scale DNN framework.

End-to-end SystemC Simulation runs entire DNN models using our candidate accelerator designs, with the integration of the *Application Framework* via the *Accelerator Driver*. This higher level of abstraction tests the correctness of the full system and leverages the accelerator's per-component performance estimates to show metrics for full workloads. Using end-to-end simulation, we capture the accelerator's behavioural and performance information when simulated with the input data produced by any given model.

The metrics captured from these simulations can include the number of total clock cycles spent within the accelerator, BRAM utilisation, processing element utilisation and various other metrics. These metrics can motivate further design iterations and highlight components representing bottlenecks. For example, we can identify inefficient processing elements or provide guidance on whether to explore the design space more broadly, investigate different data-flow strategies, or increase resource utilisation. In our case study, the accuracy of the clock cycle count is over 99% compared to the same designs synthesised on hardware.

The simulation accuracy level achieved within the case study should extrapolate to more complex designs, as the SystemC timing model is refined by HLS-reported timings of low-level components, which are composable and hierarchical.

4.3.4 Hardware Synthesis

A key step of SECDA is mapping a candidate accelerator design to real hardware in order to collect data which the designer uses to improve the overall system performance (e.g., in terms of inference time and/or energy consumption). FPGAs are an ideal platform for testing hardware designs, as well as running workloads. When an accelerator design meets the performance targets in the simulation, we can map our SystemC Accelerator onto the FPGA using HLS, followed by logic synthesis. Then we can perform *Hardware Evaluation* running the *Application Framework* with the *Hardware Accelerator* using the *Accelerator Driver*, as shown in Figure 4.1. This involves running a full end-to-end evaluation of target DNN models using the synthesised accelerator design.

Logic synthesis is one of the most time-consuming stages of any FPGA-based design process. Hence, we opt to perform most of our accelerator design space exploration using SystemC Simulation. Compared to hardware synthesis, compiling the same design to run in SystemC Simulation is much faster, around 25× faster for the Vector MAC

design in our case study (Section 4.4). The advantage of running the application on the FPGA synthesised accelerator is that we collect actual performance values rather than the estimates generated through simulation. Following this methodology can highlight bottlenecks created by the host system, such as data transfer overheads, which are not modelled in our fast simulations.

4.3.5 SECDA Design Loop

SECDA relies on two different iterative design loops to explore the accelerator design space for DNNs. The most frequently used design loop iterates through inexpensive SystemC simulations, and the second loop involves hardware benchmarking on FPGAs. Hardware benchmarking requires logic synthesis, which is very time-consuming. Thus, SECDA aims to minimise the number of times this occurs.

SECDA enables the designer to choose between the two iterative design loops. The SystemC simulation design loop is chosen when profiling the performance of the accelerator's individual components or the overall performance of data processing within the accelerator.

The performance profile of a given DNN model within the accelerator can also be evaluated in simulation, which, with a diversity of models, can highlight weaknesses in the hardware design. The hardware benchmarking design loop is chosen when the designer is interested in accurate performance data of DNN models, particularly the data transfer latencies between off-chip and on-chip memory, which are not modelled by simulation to limit the duration of simulation.

SECDA achieves increased productivity by giving the designer this choice. Through the SystemC simulation, the designer can effectively avoid expensive hardware synthesis until an efficient accelerator design is fully developed. At this point, the design can be mapped to the target FPGA to gather actual performance metrics.

Full-system simulation, as used in SMAUG [Xi+20], would avoid the need to synthesise. However, this simulation often takes longer than synthesis. Thus, we utilise a less expensive simulation combined with minimal hardware evaluation as our approach within SECDA. Hardware synthesised designs are evaluated and used to inform further iterations in simulation until the final design is chosen to meet the expected performance targets, such as reducing inference time or energy consumption.

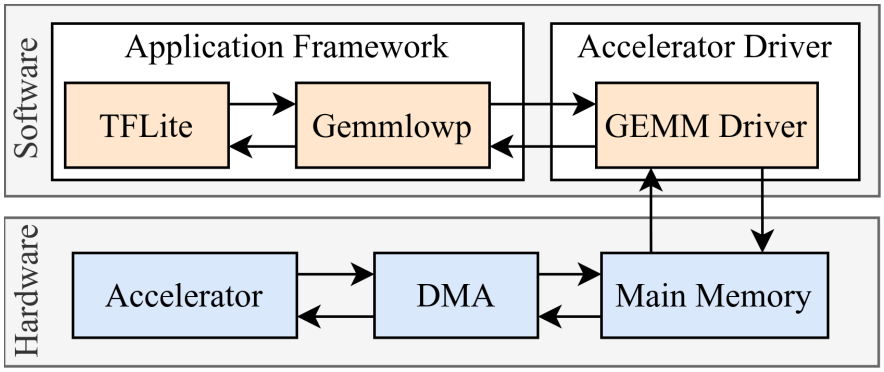


Figure 4.2: Runtime Model of our TFLite GEMM Convolution Acceleration.

4.4 Case Study

To demonstrate the value of the SECDA methodology, we designed and implemented two different FPGA-based accelerators for DNN inference, a Vector MAC (VM) based design and a Systolic Array (SA) based design. The *Application Framework* chosen was TFLite, a popular DNN inference framework for resource-constrained edge devices such as our target device, the PYNQ-Z1 board. We accelerate the convolutional layers, which in TFLite are implemented using the *GEMM convolution* algorithm. Thus, we develop the custom accelerators and their respective drivers to reduce the inference time of the model. Our accelerators use 8-bit quantised DNN models, a popular machine learning optimisation that can reduce the inference time with a low accuracy penalty [Zho+17]. Figure 4.2 shows the execution flow when performing DNN inference using a GEMM accelerator. Our accelerator offloading is integrated inside the TFLite runtime through TFLite source code modifications.

We describe the design workflow used throughout the case study and provide details of the designs in the following sections.

4.4.1 SECDA Instantiation

Initialisation

The initialisation step is essential to identify the target workload and to achieve integration within the *Application Framework*. This step varies depending on the goal of the designer (i.e., the target application framework and workload). Note that during the first case study using the SECDA methodology, we initialised the methodology through manual ad hoc integration of the SystemC environment with the TFLite framework. Later, we remove the initialisation time overhead, which is one of the main contribu-

tions of Chapter 5. Keep in mind that fully automating this initialising step to be framework-agnostic would be significantly challenging, as it could limit the capabilities of SECDA as a design methodology to work with new application frameworks and workloads.

As mentioned before, for the initial case study, SECDA was initialised by integrating end-to-end simulation with TFLite, thus establishing the foundation of our co-design/co-verification environment. The first step of the initialisation stage was to identify where in TFLite to intercept GEMM calls to offload expensive computations to the target accelerator. In our case, the *Gemmlowp* backend library was the suitable point of connection between TFLite and the SECDA design environment. Then, we used a native C++ implementation of the GEMM function to emulate the functional behaviour of our target operation. With the addition of SystemC hardware definitions, we evolved our initial software implementation into a SystemC-based model of a simple GEMM accelerator.

Throughout the initialisation, we integrated 'development hooks' to quickly switch between the default TFLite implementation and our custom simulation environment, which included SystemC-defined hardware components. With the initialisation complete, we could simulate the execution of the TFLite framework with our custom accelerator design. These 'development hooks' were function calls and C++ classes to enable the SECDA design environment with the TFLite framework. After this initial integration, we reused the SECDA-integrated TFLite codebase to develop the VM accelerator design and then the SA accelerator design, enabling a much faster development process.

SystemC Simulation co-design/co-verification

Once we had our initial SystemC-based GEMM accelerator, we created a co-design/co-verification testbench environment as discussed in Section 4.3.3. This environment allowed us to prototype new hardware components (e.g., buffers, controller logic, computation units) and test them in isolation before integrating them into the full accelerator design.

Using the testbench and the end-to-end simulation environment, we go through several iterations where we fine-tune our accelerator components. For example, reducing the number of clock cycles or changing the behaviour of the *Accelerator Driver* which improves data reshaping.

Design Loop

When we had an accelerator design with no major bottlenecks in simulation in terms of clock cycles, which estimates efficient resource utilisation of the target device, we used *Hardware Synthesis* to map it onto the PYNQ-Z1's FPGA. This enabled hardware evaluation of the design, i.e., a comprehensive full system evaluation, to identify further areas of improvement. A key strength of SECDA is that benchmarking on real hardware uses the same *Application Framework* (TFLite) and *Application Driver* as the simulated version.

The following sections describe the two GEMM accelerator designs, the VM and SA designs, and the GEMM accelerator driver, which connects the accelerator to the TFLite framework, used for both designs.

4.4.2 GEMM Accelerator Driver

The software *GEMM Driver* is co-designed with the hardware accelerator and connects to the *Application Framework* TFLite. It intercepts GEMM calls within the *Gemmlowp* library, as shown in Figure 4.2. The GEMM driver is responsible for handling the execution of convolutional layers utilising the accelerator.

It receives both weight and input data from TFLite, and reshapes them to our chosen accelerator data format.

This data format was co-designed with the accelerator, such that:

i) CPU-side data preparation leverages vectorised loads to reduce transformation overheads;

ii) data is partitioned across multiple memory-mapped buffers, hence can be sent concurrently over the DMA interface, as shown in Figure 4.2;

iii) data in each partition is organised such that it can be distributed efficiently inside the accelerator.

Once the data is reshaped, the driver is responsible for sending it to the accelerator, and for collecting and storing the output data. We pipelined the execution of the operations within the GEMM driver across multiple batches of GEMM operations within each layer to ensure that the CPU is not idle while the accelerator is processing inputs.

Algorithm 5 shows the pipelined execution flow of the GEMM driver. Note that 'Recv' is the critical function that, depending on the boolean parameter, either blocks until

Algorithm 5: Pipelined GEMM Driver Execution

```
// Prepares & Transfers post-processing data
PrepPostProcessing()
TransferPostProcessing()
// Partitions Input & Weight data
Wblocks ← WeightBlocks(LayerDims,AccDims)
Iblocks ← InputBlocks(LayerDims,AccDims)
foreach wb in Wblocks do
    // Prepares & Transfers weight data
    PrepWeights(wb)
    TransferWeights()
    firstbatch ← true
    pID, tID, sID ← 0
    foreach ib in Iblocks do
        // Prepares & Transfers input data
        if (firstbatch) then
            PrepInputs(ib,pID++)
            TransferInputs(tID++)
            firstbatch ← false
        else
            done ← Recv(false)
            PrepInputs(ib,pID++)
            if (done) then
                TransferInputs(tID++)
                StoreResults(sID++)
    while sID!=pID do
        // Transfers & Process remaining GEMM
        Recv(true)
        if (tID!=pID) then TransferInputs(tID++)
        StoreResults(sID++)
```

the accelerator has finished processing the current batch of GEMM operations or only checks if the accelerator has finished processing the current batch of GEMM operations. Additionally, the 'pID', 'tID', and 'sID' variables are used to track the progress of the GEMM operations within the driver and ensure that the output data is stored in the correct order.

In later design iterations, we found that the bottleneck was no longer the GEMM operations. Hence, we moved software-side post-processing steps (see Section 4.4.4) to the accelerator, with the GEMM driver managing the new functionality. Note that the critical difference between the drivers for the VM and SA designs is handling output data, as the output data layouts differ.

4.4.3 GEMM Accelerator Designs

Both VM and SA designs follow an output-stationary dataflow approach [Kwo+19], which was chosen to remove the need to store many intermediate results on valuable on-chip memory, or incur time and power costs associated with storing them off-chip.

Vector Mac Design (VM)

Figure 4.3 shows an overview of the VM accelerator design consisting of four SIMD-style compute units, which we call GEMM units. We are limited to four GEMM units by the resource constraints of the target device. Each GEMM unit broadcasts sets of weights and inputs to its internal MAC units to produce 4×4 output result tiles. Each output value is calculated using a set of four MAC units, with the intermediate results reduced to the final output value through an adder tree.

Systolic Array Design (SA)

Figure 4.4 shows an overview of the SA accelerator design. The design contains a single computation unit constructed as a 16×16 MAC-based systolic array, where each MAC unit accumulates towards a single output value. MAC units work by reading and storing the input and weight values of the neighbouring MAC units in their own registers. Hence, the systolic array moves weight and input values vertically and horizontally, respectively, once at the start of each step.

The inputs and weights for the starting row and column of the MAC units are read from a set of data queues which are filled by the scheduler.

4.4.4 Accelerator Components

Our designs are constructed with basic components, developed and tested both individually in the SystemC testbench and together in end-to-end simulation. Both designs contain similar components, although their behaviour and connections vary. Adapting, reusing, and recomposing these components for new designs is a valuable feature of any hardware design methodology, especially in DNNs, where a given design may quickly lose relevance due to novel DNN workloads emerging. Below is a brief description of the major hardware components.

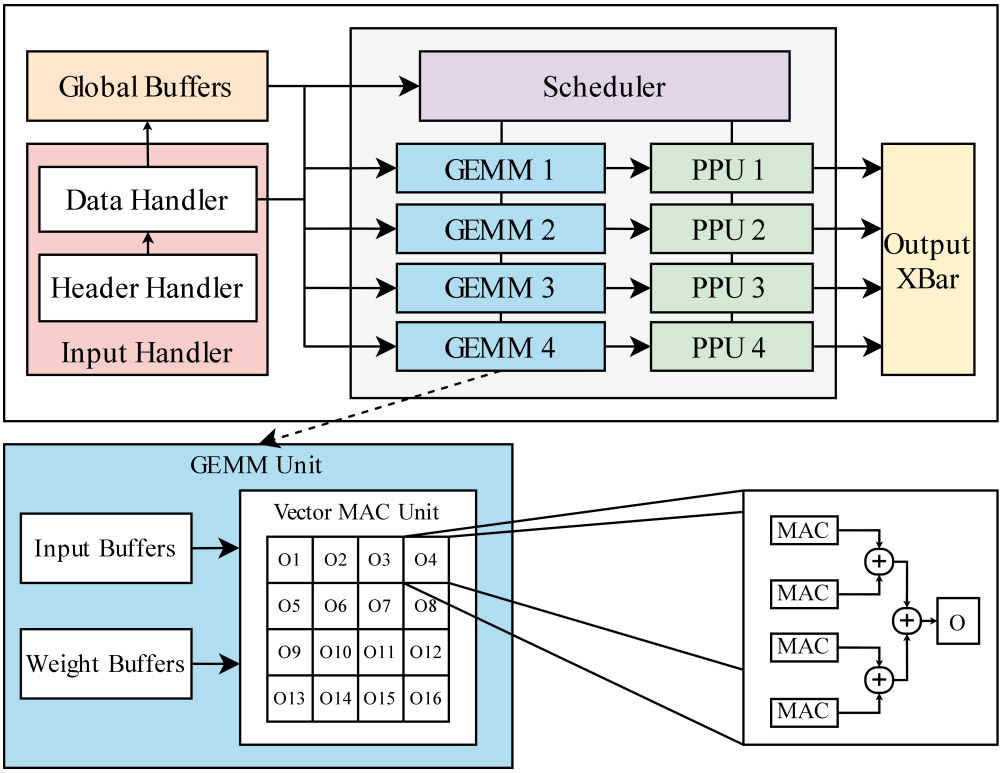


Figure 4.3: Vector MAC Accelerator design, featuring four GEMM Units.

The Input Handler

The Input Handler receives all data sent by the GEMM Driver from main memory via DMA, as shown in Figure 4.2. Metadata added by the driver is used to direct the incoming data to the appropriate accelerator buffers. The arrangement of the buffers varies between both designs. The VM design uses local buffers within each GEMM unit to store all input values and the active tile of weight data, where the global buffers are used for storing all weight tiles; the SA design only uses global buffers for both input and weight data.

The Scheduler

The Scheduler orchestrates computations which occur within the processing units of each design. For the VM design, the *Scheduler* assigns work to each GEMM unit, broadcasting weight data tiles to all GEMM units and ensuring maximum weight data tile reuse to minimise redundant loads. For the SA design, the *Scheduler* feeds input and weight data to the corresponding data queues, which feed the outer MAC units within the array.

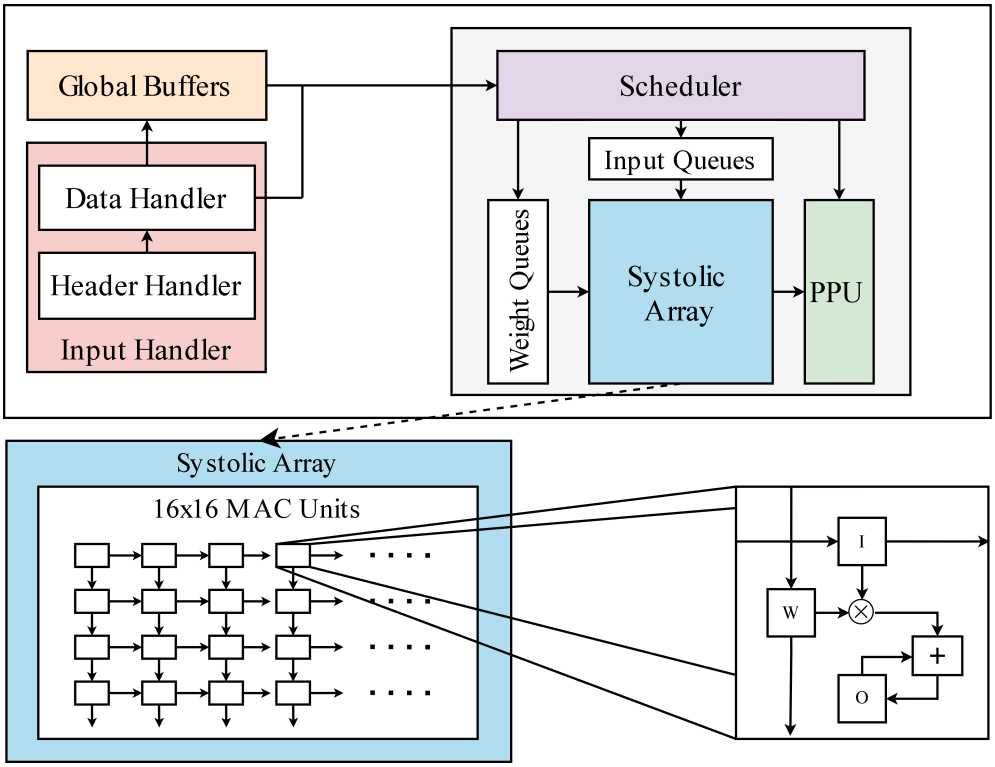


Figure 4.4: Systolic Array Accelerator design, featuring a 16×16 Systolic Array.

Post Processing Unit

The Post Processing Unit (PPU) receives `uint32` output tiles from their adjacent processing unit and applies the post-processing pipeline to obtain the quantised `uint8` result tiles. Originally performed on the CPU side, this data size reduction enabled us to reduce output data transfer time by $4\times$ at the cost of additional resource usage. Additionally, the PPU performs all other functionality provided by *Gemmlowp*'s "unpacking" function, including bias addition, scaling, and applying the activation function. For the VM design, multiple smaller PPUs process the output from each GEMM Unit. The PPU outputs were combined later by the *Output Crossbar*. In comparison, the SA design contains a single PPU that processes all the 16×16 output tiles and sends them back to main memory.

Output Crossbar

The Output Crossbar is used to collect the output tiles from all PPUs (only VM design). It rearranges the tiles so that the results are returned to the main memory in the desired order.

4.4.5 Accelerator Design Improvements

Our SECDA methodology enables fast and iterative development of DNN accelerator designs. Here, we discuss the major design improvements we made throughout the case study to optimise the end-to-end performance for both designs.

Improved Data Distribution & Bandwidth Utilisation

During the design process of the VM accelerator, in simulation we observed a lower BRAM bandwidth utilisation than expected. To address the low BRAM utilisation, we added extra functionality to the *Input Handler* to distribute the incoming input and weight data across multiple BRAMs, increasing the number of data accesses possible per cycle.

The synthesis of our first VM design consisted of four GEMM units. It highlighted a data transfer bottleneck between off-chip and on-chip memory that was not modelled within the simulation. We alleviated this bottleneck by ensuring that we leveraged all the high-performance AXI data links available on the PYNQ-Z1 board. From this change, we used end-to-end simulation to quickly redesign the accelerator and the accelerator driver to leverage the improved data links, significantly reducing data transfer times.

For the SA design, allocating 32 data queues to feed the outer MAC units of the systolic array and enabling the *Scheduler* to fill the data queues in parallel with the processing of the systolic array minimised the MAC unit idle time within the SA accelerator due unavailability of data.

Scheduling & Post Processing

For the VM design, the simulation highlighted a slowdown that occurred within each GEMM unit when reading the weight tiles into the local buffers. To address this slowdown, we added the *Scheduler Unit*, which improved the ordering of computations, reducing the number of reads from global weight buffers by 4×.

Through *Hardware Execution*, we obtained a breakdown of the inference time, which indicated that post-processing performed within the *Gemmlowp* library was the new bottleneck. Hence, we enhanced the capabilities of the accelerators by implementing post-processing within them. By adding the PPU, we obtained 1.5× and 1.3× speedup on single and dual-thread inference, respectively, when compared to previous VM designs without it. To move more functionality to the accelerator, we adapted the GEMM driver to receive quantised 8-bit results produced by the post-processing, as opposed to

the 32-bit results generated by the GEMM operations, reducing output data transfer time by 4×.

Varying Systolic Array Sizes

The SA design was prototyped, varying the dimensions of the array. We explored 4×4, 8×8 and 16×16 designs, evaluating trade-offs obtained by varying the output tile sizes and resource utilisation. In simulation, we found that the $4times4$ design lacked the compute power for the accelerator to improve against the CPU-based GEMM. The 8×8 design outperformed the CPU baseline, but it left much of the PYNQ Z1 FPGA fabric unused. The 16×16 design improved performance by 1.7× across the various models for single thread inference compared to the 8×8 design, at the cost of higher resource utilisation of the board.

DNN Specific Design Optimisations

With SECDA, we were able to make model specific changes easily to accelerator designs, either in the host driver code or the accelerator design configurations, to improve the performance for a given model. Due to device constraints, neither SA nor VM designs can be allocated enough global weight buffer space to fit some larger layers of InceptionV1 and ResNet18 entirely on the accelerator. With SECDA's ability to quickly simulate the performance and correctness of new designs, we co-designed a weight tiling scheme that was fast to produce on the CPU side and process in the accelerators. This sped up the average inference time for InceptionV1 and Resnet18 by 2× and 2.2×, respectively, compared to the previous accelerator designs.

Note that some convolutional layers of ResNet18 were still too large to fit into the local buffers within the GEMM units of the VM design. We were able to reconfigure, validate, and synthesise a modified VM design for ResNet18. This design trades off global buffer space for local buffer space, enabling native execution of all layers within the accelerator and reducing the inference time by 1.6× over the previous design.

4.5 Evaluation

4.5.1 Experimental Setup

In our case study, we evaluated the two accelerator designs on the PYNQ-Z1 board, which includes an edge FPGA and a dual-core ARM Cortex-A9 CPU. We bench-

mark four widely-used DNN models quantised to 8 bits: MobileNetV1 [How+17], MobileNetV2 [San+18], InceptionV1 [Sze+15] and ResNet18[He+16]; all defined on the ImageNet dataset [Rus+15]. For each DNN model, we evaluate CPU-only inference times in TFLite (TensorFlow version 2) using 1 and 2 CPU threads, taking the median (to not let outlier runs affect the results) of 100 runs to compare against our two accelerator designs. We gather energy metrics using a COOWOO [COO] digital USB power meter. All reported run have a standard deviation of less than 0.7%. For more details on the experimental hardware setup, refer back to Section 1.1.2.

4.5.2 Case Study Results

Table 4.2 shows the breakdown of inference time and energy consumption for the four DNN models under study for a single image using the CPU (1 and 2 threads) and the two accelerator designs (VM and SA). The time is split between convolutional (CONV) layers, which our accelerators target, and all other (Non-CONV) layers, which run on the CPU. Figure 4.5 visualises the inference latency performance results.

Note that both accelerator designs could have been further optimised to improve performance, and we tackle this aspect in Chapter 5. However, the purpose of this initial case study is to highlight that, using SECDA, we were able to quickly develop and iterate upon viable accelerator designs that significantly improved inference time performance and energy consumption compared to the CPU-only case.

Overall Performance

For the VM accelerator, we observe an average speedup across models of 3× and 2× and an average energy saving of 2.7× and 1.8× for one and two threads, respectively, in each case when compared to CPU-only inference. Similarly, for the SA accelerator, we observe an average speedup across models of 3.5× and 2.2× and an average energy saving of 2.9× and 1.9× for one and two threads, respectively, in each case when compared to CPU-only inference.

We observe less speedup and energy consumption with dual-thread execution, as expected since the CPU's compute capacity doubles while both accelerator designs remain the same. However, our accelerated runtime using two threads improves inference time since the CPU-side *Accelerator Driver* can leverage threads.

Table 4.2: Inference time (ms) and energy consumption (J) results for the four DNN models under study when using different numbers of CPU threads and accelerator designs.

DNN	Hardware setup	CONV	Non-CONV	Overall	Energy
MobileNetV1	CPU (1 thr)	635 ms	141 ms	776 ms	1.84 J
	CPU (1 thr) + VM	123 ms	141 ms	264 ms	0.68 J
	CPU (1 thr) + SA	90 ms	141 ms	231 ms	0.65 J
	CPU (2 thr)	329 ms	73 ms	402 ms	1.04 J
	CPU (2 thr) + VM	105 ms	73 ms	178 ms	**0.43 J**
	CPU (2 thr) + SA	86 ms	73 ms	**159 ms**	0.54 J
MobileNetV2	CPU (1 thr)	526 ms	176 ms	702 ms	1.66 J
	CPU (1 thr) + VM	156 ms	176 ms	332 ms	0.79 J
	CPU (1 thr) + SA	103 ms	176 ms	279 ms	0.83 J
	CPU (2 thr)	277 ms	95 ms	372 ms	1.01 J
	CPU (2 thr) + VM	128 ms	95 ms	223 ms	**0.61 J**
	CPU (2 thr) + SA	97 ms	95 ms	**191 ms**	**0.61 J**
InceptionV1	CPU (1 thr)	1416 ms	117 ms	1533 ms	3.60 J
	CPU (1 thr) + VM	263 ms	117 ms	380 ms	0.97 J
	CPU (1 thr) + SA	225 ms	117 ms	**342 ms**	1.12 J
	CPU (2 thr)	736 ms	117 ms	853 ms	2.20 J
	CPU (2 thr) + VM	249 ms	117 ms	366 ms	**0.97 J**
	CPU (2 thr) + SA	225 ms	117 ms	**342 ms**	1.12 J
ResNet18	CPU (1 thr)	1762 ms	132 ms	1894 ms	5.4 J
	CPU (1 thr) + VM	555 ms	132 ms	687 ms	2.12 J
	CPU (1 thr) + SA	405 ms	132 ms	**537 ms**	1.76 J
	CPU (2 thr)	919 ms	132 ms	1051 ms	3.24 J
	CPU (2 thr) + VM	550 ms	132 ms	682 ms	2.12 J
	CPU (2 thr) + SA	405 ms	132 ms	**537 ms**	1.76 J
	CPU (2 thr) + VTA	–	–	737 ms	**1.51 J**

Bottleneck Analysis

While analysing our designs, we observe that we hit a threshold for performance gains achieved by our hardware designs, with the bottleneck for inference performance shifting to two other areas. Namely, (i) CPU-side CONV data preparation and results unpacking; (ii) and non-accelerated layers.

For (i), breaking down single-threaded CONV time for VM, we observed that only 31% of the time is spent performing off-chip data transfers and the accelerator computations. The CPU-side data preparation and resulting unpacking represent the majority of the CONV time, 69%, which highlights the importance of hardware-software co-design to ensure that additional hardware changes cannot further reduce this time.

For (ii), in single thread CPU-only inference, Non-CONV layers only represent 14% of the inference time on average. However, by accelerating the CONV layers, the relative

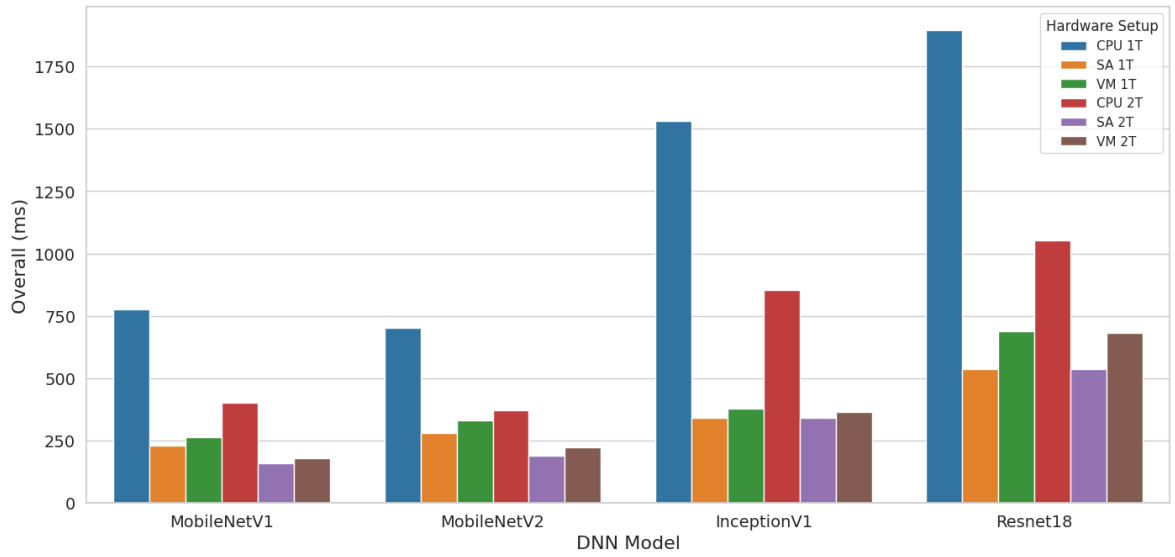


Figure 4.5: Comparison of inference time for the four DNN models under study across different hardware setups.

importance of Non-CONV layers increases, representing 39% and 46% of single thread inference time for VM and SA, respectively.

VM vs SA Performance Comparison

Comparing our two designs, SA achieves slightly better performance, 16% on average in latency and up to 4% in energy savings. From these observations, we conclude that while the core compute units of VM and SA use different strategies to perform GEMM, we achieve similar end-to-end performance from both designs due to the shift in the inference performance bottlenecks to the CPU side.

Model Performance Analysis

We also observe that InceptionV1 achieves the best speedup relative to the CPU-only version, with 4× and 2.3× speedup for one and two threads, respectively, for VM, and 4.5× and 2.5×, respectively, for SA. Comparing to MobileNetV1 and MobileNetV2, which feature depthwise separable convolutions (meaning that each convolutional layer performs fewer MACs per input), InceptionV1's standard convolutions have greater potential for GEMM acceleration, since the relative time-cost of its data preparation stage is smaller. Additionally, for InceptionV1 and ResNet18 we observe negligible speedup for multithreaded execution, relative to the other models due to the larger GEMM operations coupled with our pipelined execution. This means that the CPU-side latency is "hidden" by the accelerator's computation, resulting in minimal benefits from two threads.

Development Time Discussion

Finally, in terms of development time, by replacing synthesis iterations with simulations, as estimated by Equation 4.1, we observed a 25× difference between S_t and C_t. This suggests that we spent on average 16× less time evaluating end-to-end inference of a given design in simulation for our GEMM accelerators, compared to developing with all evaluation performed on the FPGA.

4.5.3 Comparison with state-of-the-art DNN accelerators

We now validate that our designs are competitive with another state-of-the-art DNN accelerator in terms of inference time, our main design goal. We compare our designs against VTA, which is supported through the state-of-the-art DNN compiler framework TVM [Che+18b]. We chose it over other accelerator frameworks due to its recent release, support from an active open-source community, and its use of 8-bit quantisation similar to our designs. The final row of Table 4.2 shows the performance of VTA for ResNet18, taking the median of 100 runs on the PNYQ Z1 board. ResNet18 was the only publicly available model compatible with both VTA and TFLite at the time of the case study. We refer the reader to the TVM VTA documentation[1] for details on synthesis and execution — note that VTA leverages both threads of the CPU. The results show that the designs developed using the SECDA methodology are competitive with VTA, with our VM design outperforming VTA by 8% in terms of latency, while VTA reports 29% less energy consumption per inference. Our SA design outperforms VTA by 37% in terms of latency, while VTA has 14% lower energy consumption. VTA runs more of its layers on the accelerator, resulting in fewer off-chip data transfers and achieving greater energy efficiency than our design. In terms of our target performance metric, inference time, we have demonstrated that designs produced via SECDA can be competitive with a state-of-the-art accelerator.

4.6 Summary

Within this chapter, we first motivated the need for an efficient design methodology for FPGA-based DNN accelerators, highlighting the key challenges faced by developers and the key features required for a suitable design methodology. We then introduced the SECDA methodology, detailing its main components and usage. We presented a case study using the SECDA methodology, where we first initialised SECDA within

[1] `https://tvm.apache.org/docs/topic/vta/tutorials/index.html`

the TFLite framework through simple ad-hoc integration. Using this integration, we developed two GEMM-based accelerator designs for convolutional layers. Finally, we evaluated the accelerators' performance against the CPU baseline and demonstrated that they outperform the CPU in all cases. We also made comparisons to VTA, a state-of-the-art FPGA-based DNN accelerator for resource-constrained devices. Overall, we demonstrated that SECDA is a suitable design methodology for developing FPGA-based DNN accelerators. Hence, the rest of the thesis will build up the SECDA methodology and utilise it for FPGA-based DNN accelerators.

5 SECDA-TFLite

This chapter introduces the SECDA-TFLite toolkit, which extends the work that was initially presented within the case study for the SECDA methodology, as described in Section 4.4.

This chapter is structured as follows: Section 5.1 introduces the SECDA-TFLite toolkit, and the motivation behind its development. Section 5.2 presents the SECDA-TFLite toolkit in detail. Section 5.3 discusses the SECDA-TFLite benchmarking suite and the automation of hardware design synthesis. Section 5.4 presents a case study using the SECDA-TFLite toolkit, which is evaluated in Section 5.5. Finally, Section 5.6 summarises the chapter.

5.1 Introduction

To tackle the challenges of designing DNN hardware accelerators, we developed the SECDA methodology [Har+21] as a guideline for efficient hardware-software co-design of FPGA-based DNN accelerators for edge inference. A key part of SECDA is that hardware design iterations are performed with the simulation to guide the design process. After that, the design can be easily synthesised on real hardware (e.g., an FPGA) for more robust testing. As discussed in Chapter 4, the approach uses a unified codebase and encourages tight integration of the target Application Framework (i.e., the DNN framework such as TensorFlow's [Aba+16] TFLite or PyTorch [Pas+17] Mobile) with the accelerator's software driver and hardware design. The hardware-software co-design of the accelerator driver alongside the accelerator reduces the time for deployment to the target hardware once the accelerator design is ready for real hardware evaluation. The unified codebase is achieved by leveraging SystemC [Des23], which enables fast simulation and High-Level Synthesis (HLS) to an FPGA.

The first step in any instantiation of a SECDA-based workflow is the initial integration with the target *Application Framework*. This instantiation includes sub-steps such as setting up the simulation environment and providing a path to offload host-side

computations to the accelerator designer. From there, developers can begin to define their first accelerator designs and follow the iterative design loops of SECDA to produce optimised designs.

However, our observation is that the number of strong candidate application frameworks for DNN accelerators is limited. TVM [Che+18b], TFLite, TensorRT [NVIb], PyTorch [Pas+17] Mobile, and ONNXRuntime [dev11] constitute the most relevant DNN inference frameworks used today. Thus, various developers creating their own integration for these frameworks may lead to redundant work, as they will all to perform the same initialisation steps. Hence, we decided to develop SECDA toolkits that would streamline the integration process for candidate DNN frameworks.

Due to TFLite's inherent connection with the popular TensorFlow ML Framework and its relatively mature support for features such as quantisation, sparsity, and custom operations, we decided to develop our initial SECDA toolkit for development and deployment using TFLite. Therefore, we proposed SECDA-TFLite, an open-source toolkit that extends the TFLite DNN framework so that it can be more easily used to develop new DNN hardware accelerators using the SECDA design methodology.

The SECDA-TFLite toolkit leverages the TFLite delegate system to provide a robust and extensible set of utilities for integrating DNN accelerators for any TFLite-supported DNN operation. Ultimately, this increases the productivity of hardware accelerator developers, as they can begin developing and refining their design more quickly. To aid in this, the toolkit consists of four key components: SystemC Integration, Simulation Profiling, Data Communication, and Multi-threading libraries. These provide the essential utilities that enable developers to develop their designs. Additionally, we provided a set of extra tooling to automate the process of hardware design synthesis, model benchmarking across different accelerators, and the generation of reports and visualisations of profiled simulation data.

We demonstrate the utility of SECDA-TFLite with a case study which provides accelerators for both CNN and transformer-based DNN architectures. We port and improve the two CNN accelerator designs from the original SECDA case study 4.4, integrating and exploiting the FPGA resources with the more robust tooling provided by SECDA-TFLite. In addition, we bring a new accelerator design targeting the BERT [Dev+19] family of models, which uses a transformer neural architecture. The target device is the PYNQ-Z1 board [Dig], a platform with a dual-core CPU and an edge FPGA. The contributions of this chapter are as follows:

- SECDA-TFLite, an open-source toolkit to enable the efficient development of custom DNN hardware accelerators for TFLite, focused on edge devices, leveraging the SECDA design methodology.

- We describe the key features of the SECDA-TFLite toolkit (simulation, profiling, and data communication utilities) and how they integrate with the upstream TFLite framework.
- We present additional tooling to automate and ease the process of evaluating accelerators developed using SECDA-TFLite.
- We present a SECDA-TFLite case study, where we ported and updated two existing CNN accelerators previously developed within the SECDA case study with our toolkit for a newer version of TFLite. In addition, we developed a new accelerator design targeting transformer models.
- We evaluate the performance of the accelerators developed for our case study by benchmarking them against several state-of-the-art DNN models. Our CNN accelerators are comparable to or outperform their original counterparts, demonstrating that SECDA-TFLite does not introduce significant overheads compared to a more ad-hoc integration. For our new BERT accelerator, we outperform the CPU-only inference by an average of 2.1× and 2× in terms of inference time and energy efficiency, respectively.

5.2 SECDA-TFLite Toolkit

SECDA is a generic design methodology that can be applied to various application frameworks (e.g., DNN inference frameworks). The first step is to the SystemC environment integrate with the target framework. However, this initial step is time-consuming and could hinder the adoption of the methodology. Additionally, once initialised for a given framework, the same environment can be re-used between accelerator designs defined for that framework, further reducing the barriers to developing new accelerators.

SECDA-TFLite is a TFLite-specific toolkit that provides the initial development environment when using the SECDA methodology within TFLite and a set of utilities to aid development. This enables the developer to begin prototyping and integrating their new design with significantly reduced initial setup overhead. While the original SECDA case study was embedded within TFLite, the integration was ad hoc because it focused on being a proof of concept for the methodology rather than a generic integration for future developers. SECDA-TFLite aims to be a robust open-source toolkit for anyone who wants to develop new DNN accelerators within TFLite.

The rest of the section expands on the key aspects of SECDA-TFLite and how it streamlines FPGA-based DNN accelerator development for TFLite using the SECDA

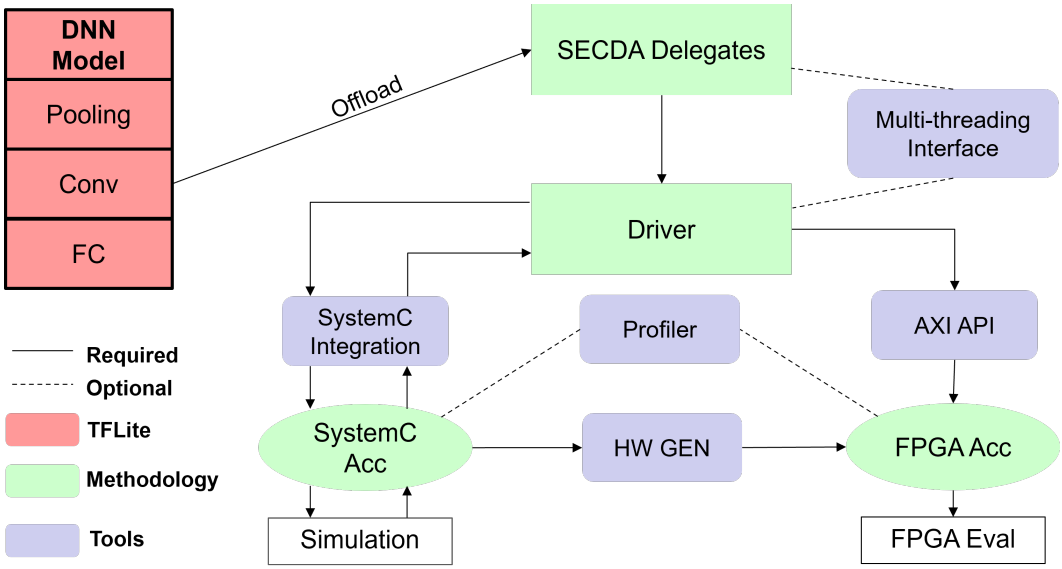


Figure 5.1: Overview of the SECDA-TFLite toolkit and how it is used within the SECDA design methodology for TFLite.

methodology. Section 5.2.1 gives context on where the integration with TFLite occurs. Section 5.2.2 describes the four main components of the toolkit, and Section 5.2.3 discusses the toy accelerator design that we provide as a starting point for developers, which leverages all features of the SECDA-TFLite toolkit.

5.2.1 SECDA-TFLite Delegates

The SECDA-TFLite toolkit provides the integration of the SECDA environment with the TFLite DNN inference framework, giving the accelerator developer a starting point to produce new DNN hardware accelerators. This integration exploits TFLite's so-called 'delegate' system, where operations from DNNs can be efficiently offloaded to the target accelerator while still providing the original CPU inference for non-accelerator layers. Figure 5.1 shows an overview of the key aspects SECDA-TFLite's integration in TFLite. For future SECDA toolkits targeting other frameworks, we will exploit similar subsystems such as TVM's *BYOC*[1], or ONNX Runtime's *Execution Provider*. Section 5.2.1 gives a brief overview of the TFLite delegate system, while Sections 5.2.1 and 5.2.1 discuss the SECDA delegates that SECDA-TFLite enables for developing their custom offloading mechanism for accelerator designs. Note that we discuss the SECDA delegates in terms of simulation and FPGA delegates for ease of explanation. However in practice, a single delegate can be used for both simulation and FPGA deployment with a simple change of compilation flags.

[1]'Bring Your Own Codegen' `https://tvm.apache.org/docs/dev/how_to/relay_bring_your_own_codegen.html`

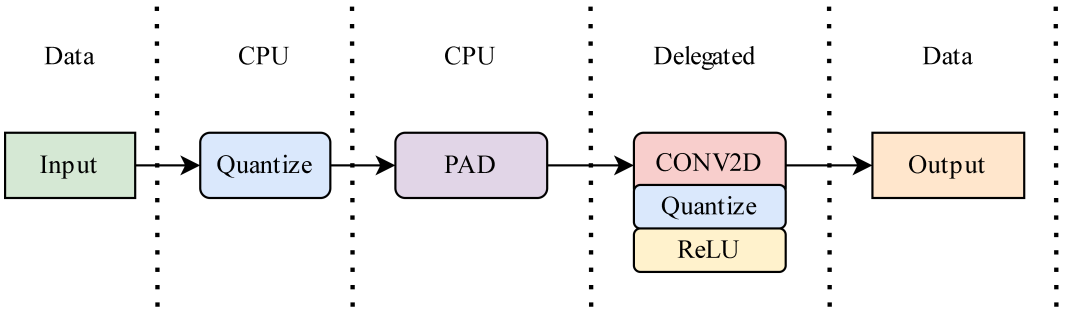


Figure 5.2: Simplified example a DNN running on TFLite, with some nodes running on the CPU, and a group of three running on a delegate.

TFLite Delegate System

The TFLite delegate system [Ten] is available in later versions (post v2.7) of TFLite with the purpose of providing simplified support for different hardware and software backends for DNN operations. While TFLite provides some delegates for Android and iOS devices, creating custom delegates is required to deploy new custom hardware accelerators. For DNN inference, TFLite delegates can be used to offload individual (or groups of) TFLite operations within a DNN model to other backends, including hardware accelerators. Figure 5.2 shows an example of three operations in a DNN being executed on a delegate, with the rest of the operations being run using the default CPU runtime. In the example, the three nodes are grouped together, enabling the potential for further optimisations through the delegate.

In addition, creating custom delegates in TFLite can be cumbersome and requires expertise to connect TFLite with low-level hardware drivers. Hence, the SECDA-TFLite toolkit provides the bulk of the initial delegate integration required for developers to implement new DNN hardware accelerator designs following the SECDA methodology. Within the SECDA-TFLite design flow, the developer defines a new delegate for their accelerator design, which can be used for both simulation and FPGA deployment.

SECDA-TFLite Simulation Delegate

The purpose of a simulation delegate is to allow end-to-end simulation as defined within the SECDA methodology within TFLite. The simulation delegate connects TFLite to simulated DNN accelerator hardware designs. Under SECDA, the SystemC simulation is required to provide a fast evaluation time for changes to the accelerator design, enabling verification of correctness and resource efficiency. Additionally, the simulation delegate integrates with the *Profiler*, more specifically the *Simulation Profiler*, which is described in Section 5.2.2. These profiling tools can be extended to meet the developers' needs while providing the essential features required.

The simulation delegate is also used to co-design the accelerator driver, which is used to communicate with the accelerator. The accelerator driver is responsible for managing aspects such as tiling strategies, data preparation, output data unpacking, control flow and generating accelerator instructions. Overall, the efficiency of the accelerator driver can be very impactful on the overall runtime performance [Wan+21; Xi+20], hence why the co-designing of the accelerator and the accelerator driver is prioritised within the SECDA methodology.

SECDA-TFLite FPGA Delegate

The purpose of the FPGA delegate is to provide the interface to versions of the accelerator running on real hardware, namely an FPGA. The simulation delegate connects TFLite to the DNN accelerator hardware designs on an FPGA. Under SECDA, the purpose of running on real hardware is to identify bottlenecks that are not revealed through simulation, for instance, the impact of off-chip memory accesses. To this end, SECDA-TFLite provides the *Data Communication library*, discussed in Section 5.2.2. This provides approaches for the designer to convert simulation constructs defined for data communication into real hardware AXI data transfer implementation. Similarly, accelerator driver code can be threaded to improve the CPU-side performance. Thus, SECDA-TFLite provides a *Multi-Threading API* (see Section 5.2.2). Finally, the FPGA delegate integrates with the *Profiler*, more specifically the *Hardware Profiler*, to provide the developer with the necessary tools to profile the accelerator and driver code running on the FPGA.

5.2.2 Toolkit

Along with the two TFLite delegates, which provide starting points for the integration and development of DNN hardware accelerator designs with TFLite, the SECDA-TFLite toolkit also provides four core components: SystemC integration, Profiler, Data Communication, and Multi-threading API; which helps the designer to develop their hardware designs. We describe the four components below.

SystemC Integration

SystemC end-to-end simulation is key to the SECDA methodology; however, TFLite does not natively support the SystemC simulation environment. Thus, with SECDA-TFLite, we define a software library within the TensorFlow Bazel [Devb] workspace, which can be included as a dependency when compiling any TensorFlow binaries and

delegates. This software library provides the user with the necessary components for running SystemC simulation. It enables the SystemC API to initialise and bind hardware modules defined in SystemC to the simulation environment via delegate calls.

As well as reducing the work developers must do to integrate SystemC, we also provide the *SystemC integration API* that allows developers to define data transfer between the simulated accelerator and the TFLite allocated data tensors. Thus, a key benefit of the SystemC integration library is that we can ensure SystemC simulation constructs are easily accessible throughout the delegate code, with standard boilerplate code predefined for TFLite.

```
// SystemC Accelecrator.h
// Define profiling for hardware accelerator
ClockCycles *per_batch_cycles = new ClockCycles("total_cycles", true);
ClockCycles *read_cycles = new ClockCycles("read_cycles", true);
ClockCycles *compute_cycles = new ClockCycles("compute_cycles", true);
ClockCycles *send_cycles = new ClockCycles("send_cycles", true);
std::vector<Metric *> profiling_vars = {total_cycles, read_cycles,
                                        compute_cycles, send_cycles};
// ---------------------------------------------------------------------
// C\texttt{++} SimulationDelegate.cc
// Save profile of the target hardware module after simulation
profile.saveProfile(accelerator.profiling_vars);
...
...
// Save all profiled data to csv
profile.saveCSVRecords(profile_output_file_name);
```

Listing 5.1: Example of using the Simulation Profiler to define, profile and export specific clock cycle metrics across simulation.

Profiler

Simulation Profiler: End-to-end SystemC simulation can be used to quickly evaluate the potential performance impact of changes to the hardware and software components of the accelerator design, as well as verifying the correctness of the implementation. In order to profile the end-to-end simulation, the developer needs to add additional code to keep track of hardware and software metrics (such as simulated clock cycles spent), throughout the end-to-end DNN inference. SECDA-TFLite provides a system called the *simulation profiler*, which provides a method to define the different types of metrics to capture from the accelerator and software driver, along with common metrics developers will be interested in, such as the number of clock cycles.

Listing 5.1 shows an example of how a developer can use the simulation profiler to

capture various clock cycle metrics for their hardware accelerator design. This saves the developer time by not having to define, capture, and process common profiling metrics manually. However, the simulation profiler is extensible to bespoke metrics not defined by SECDA-TFLite, such as accelerator instruction count. In addition, the simulation profiler provides an export function to a CSV file for analysis.

Hardware Profiler: The SECDA-TFLite's hardware profiling system is used to profile the performance of the accelerator and the driver code running on the FPGA. The simplest form of hardware profiling is to measure points of execution within the driver code, such as the time taken to send an opcode to the accelerator.

```
// Driver code
prf_start(0); // Start profiling
int *in0 = drv.mdma->dmas[0].dma_get_inbuffer();
int in10 = 0;
int opcode = 16;
in0[in10++] = opcode;
drv.mdma->dmas[0].dma_start_send(in10); // Send opcode
drv.mdma->multi_dma_wait_send();      // Wait for send to complete
data_transfered += in10;
prf_end(0, drv.p_t.p_start_sched);  // End profiling
```

Listing 5.2: Example of using the hardware profiling to capture the time taken to send a single opcode to the accelerator.

Data Communication

Edge FPGA-based DNN accelerators require a high degree of data transfer between on-chip and off-chip memory, as the on-chip memory will have insufficient capacity to store all the weights and input data required through inference. Hence, during DNN inference, new sets of data need to be sent to the accelerator to be processed, and the resultant data needs to be transferred back to main memory. Since the Advanced eXtensible Interface (AXI) is the standard data interface for data movement between the FPGA and ARM CPU cores, within SECDA-TFLite we provide a simple AXI API to allow the designer to quickly implement data transfers between CPU and accelerator through the three main types of AXI data transfers: AXI-MM, AXI-Lite, and AXI-Stream. As with the other components of the SECDA-TFLite toolkit, this avoids the need for the hardware developer to define their own data communication methods while still allowing them to adapt the system if necessary.

Multi-threading API

The multi-threading API consists of simple classes that help the delegate developer define CPU-side tasks that need to be performed in a multithreaded fashion. Most commonly, this will be in the *accelerator driver*, which needs to transfer data between the hardware accelerator and TFLite efficiently. The API allocates tasks to worker threads to execute. While threading is not required to improve the accelerator's performance, it can improve the performance of any computationally expensive calls, such as data packing and unpacking, to reduce the bottleneck in data preparation and storage. In addition, the API does not exclude the use of other common threading libraries; however, it is provided such that developers can benefit from multi-threading in the context of SECDA-TFLite without having to produce a custom solution.

5.2.3 Template Delegate and SystemC DMA-Engine

Along with the SECDA-TFLite toolkit, we provide a set of simulation and FPGA delegates for a toy accelerator. This simple design can serve as a quick starting point for developers to showcase the usage and main features of SECDA-TFLite. The simulation and FPGA delegate also highlight the differences between the simulation driver and the actual AXI-based FPGA driver. We also provide a hardware definition of a simple DMA engine that can be used to simulate any data communication using the AXI-Stream interface.

5.3 Automation

Due to the complexity of the SECDA-TFLite toolkit, we have developed additional tooling to automate the process of hardware synthesis, benchmarking DNN models across different accelerators, and generating reports and visualisations of profiled simulation data.

5.3.1 Hardware Design Synthesis Automation

Automated hardware synthesis allows developers to quickly translate their accelerator designs into FPGA bitstream with minimal effort. The tooling depends on the Xilinx Vivado Design Suite [Xilb] for the synthesis and implementation of the hardware design. Hence, the developer must have Vivado installed on their system. To generate the hardware design, the developer must provide a JSON-based configuration file that describes the accelerator metadata, such as the accelerator name and version, the

target FPGA device, and the clock frequency. Additional features, e.g., copying the generated bitstream to the target FPGA board, are also provided but are optional.

Once the configuration file is provided, the developer can invoke the hardware synthesis automation tool. During invocation, the user can specify whether to perform full logic synthesis to generate the bitstream or HLS only to generate the accelerator IP, the estimated resource utilisation, and the timing schedule for the design.

5.3.2 Benchmarking Suite

The benchmarking suite automates the process of running experiments on the target FPGA board and collecting the results. Figure 5.3 shows the high-level architecture of the benchmarking suite. The suite is designed to be extensible, allowing developers to add new benchmarks (DNN models) and accelerators with minimal effort. Additionally, the suite provides a versioning system to track the performance of the accelerators across different versions of the accelerator design.

An 'experiment' configuration within the benchmarking suite consists of the DNN models to execute, the metrics to measure (power, latency or accuracy), and the accelerators to use. Once the experiment is configured, the user can invoke the benchmarking suite, which will run the experiment on the target FPGA board and collect the results. Before running the experiment, the hardware synthesis automation tool must be executed to ensure the accelerator bitstreams are available for the benchmarking suite.

5.3.3 Profile Visualisation

Figure 5.4 shows an example of the simulation visualisation tool, which graphs the simulation-generated profiling data. It shows the accelerator's performance per hardware sub-modules in terms of cycles spent in different states (e.g., idle, busy, or stalled). Each bar represents the breakdown of the cycles spent in each state for a specific hardware sub-module. The total number of cycles is shown in the title of each bar. Note that the total number of cycles is equal for each of the bars in this figure. This is because all three of these sub-modules were active for the entire simulation runtime. The accelerator developer can define the states of each sub-module within the SystemC accelerator definitions (see Listing 5.1). This tool is useful for identifying bottlenecks in the accelerator design. For example, the top bar chart shows the post-processing module (PPU) spending most of its time in 'S_1', which is the idle state. This indicates that the post-processing module is underutilised and could be optimised further.

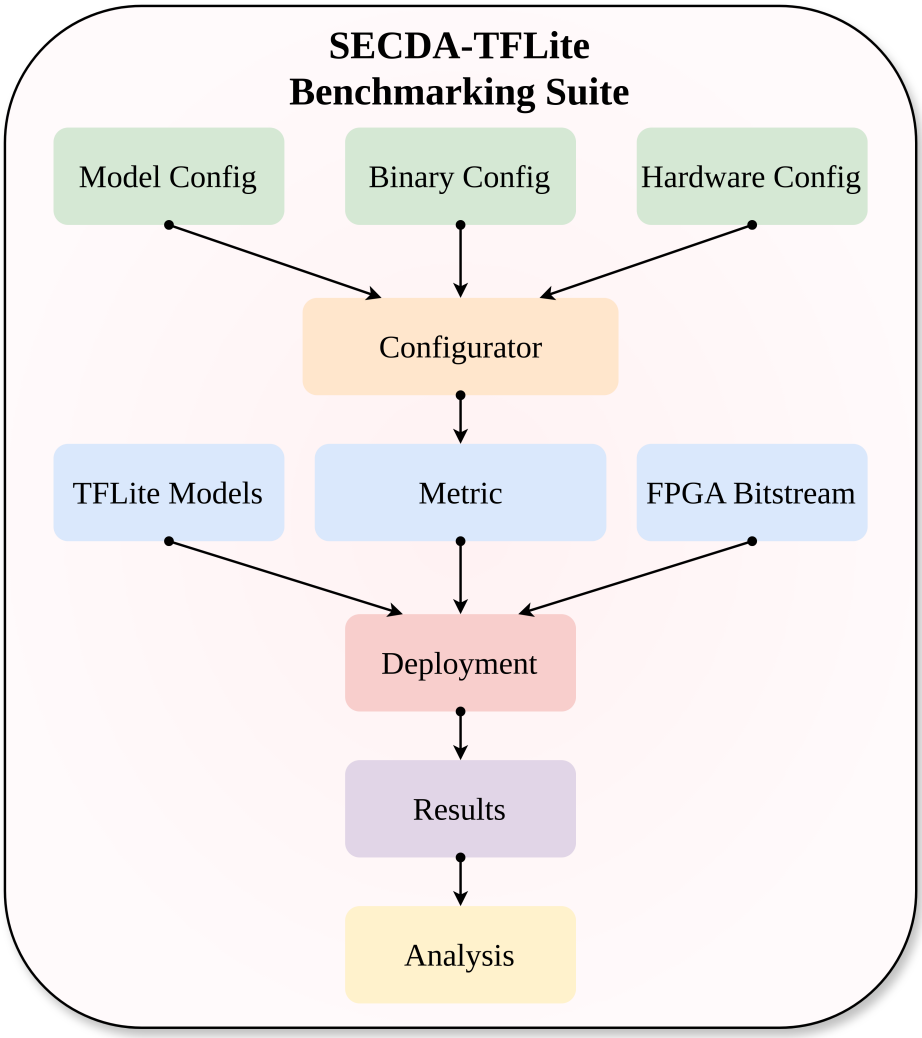


Figure 5.3: High-level architecture of the benchmarking suite.

5.4 Case Study

To demonstrate the value of the SECDA-TFLite toolkit and how it provides a foundation to efficiently develop DNN accelerators within TFLite using the SECDA methodology, we develop three different FPGA-based DNN accelerator designs targeting Convolutional and BERT-based DNN models.

We develop the designs for resource-constrained edge devices such as our target device, the PYNQ-Z1 board. For Convolutional Neural Network (CNN) models, we port, improve, and integrate the Vector MAC (VM) and Systolic Array (SA) based-designs, which were previously defined within the original SECDA case study, using the SECDA-TFLite toolkit. Since the original case study, TFLite has changed significantly (from v2.2 to v2.7+). Thus, the newly defined accelerators now target signed 8-bit inference rather than the unsigned 8-bit inference used in earlier versions of TFLite, which has

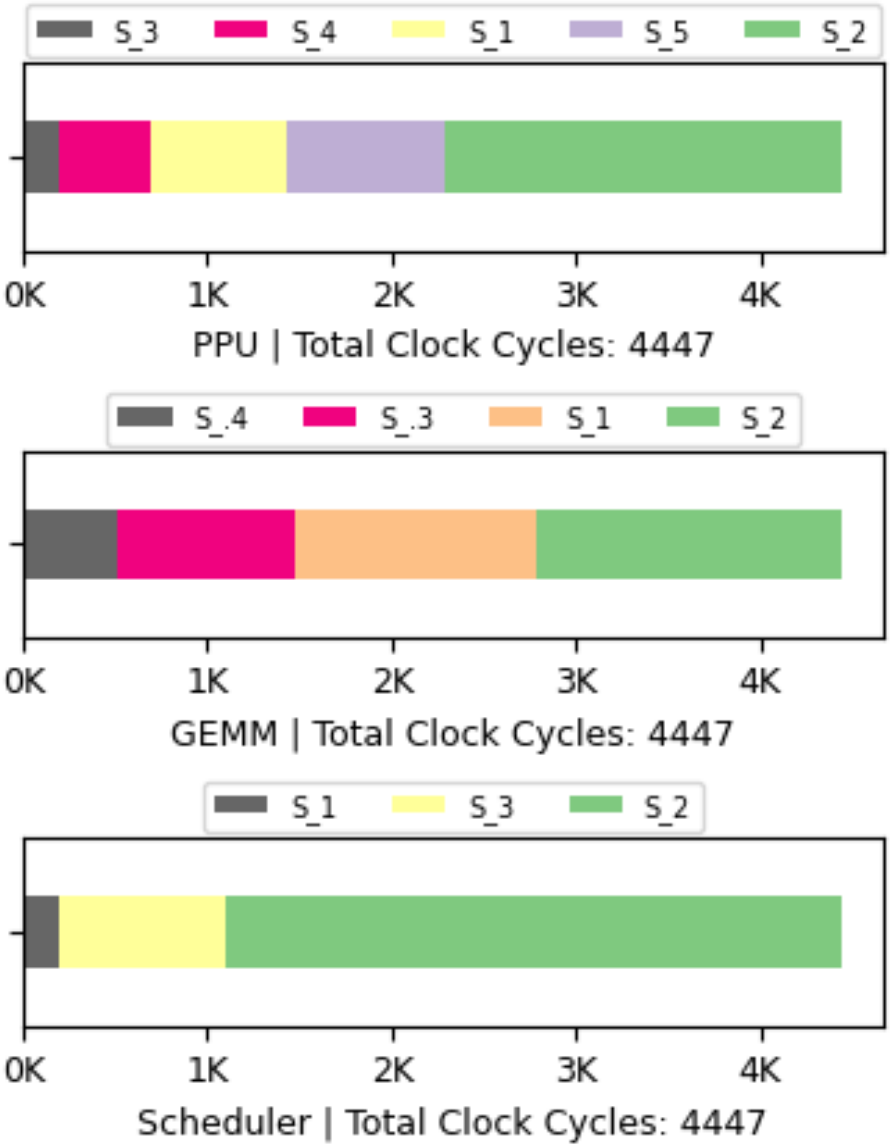


Figure 5.4: Example of simulation profile visualisation tool.

been deprecated. Thus, our accelerators use signed 8-bit quantised DNN models, a popular machine learning optimisation that can reduce the inference time with a low accuracy penalty [Zho+17].

We accelerate convolutional layers, which in TFLite are implemented using the *GEMM convolution* algorithm. Thus, we develop the aforementioned custom accelerators and their respective drivers to reduce the model's inference time. For models in the BERT family, we note that they contain high-level DNN layer structures commonly referred to as *transformer layers*. However, these transformer layers can be decomposed into several Matrix Multiplication operations, which are represented as Fully Connected (FC) layers within TFLite models. From our experiments, these FC layers are the most expensive in terms of computational requirements, taking up to 66% of the overall

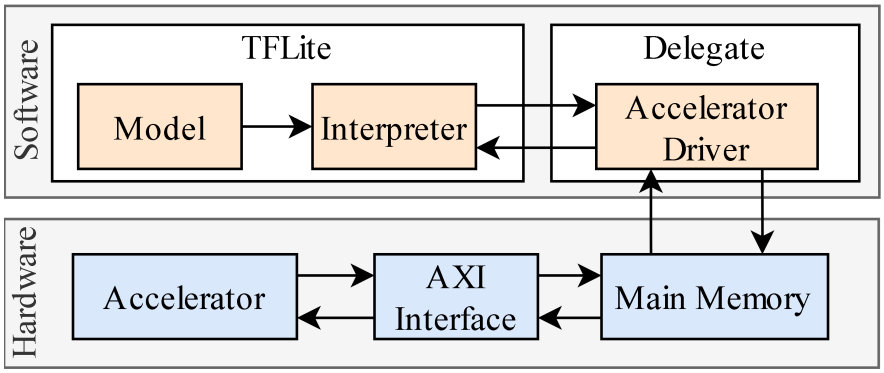


Figure 5.5: SECDA-TFLite runtime model, common for all accelerators and DNN model types.

inference time. Thus, we develop a new GEMM-based accelerator design for BERT-based models to accelerate the FC layers within TFLite.

Figure 5.5 shows the execution flow when performing DNN inference using our custom accelerators. We integrate the offloading of computation to our accelerators through the SECDA-TFLite delegates. We describe the improved development environment made possible through the SECDA-TFLite toolkit and how it is used throughout the case study, and provide details of the designs in the following sections.

Section 5.4.1 gives a brief overview of how the SECDA design methodology is followed when using SECDA-TFLite. Section 5.4.2 discusses our three accelerator designs, with Section 5.4.3 giving details of their components. Finally, in Section 5.4.4, we briefly discuss the relevant features of the accelerators' supporting software.

5.4.1 SECDA-TFLite Workflow

To develop new accelerators using the SECDA methodology, we need to instantiate the development environment within the application framework so that we can load and run models, as well as run simulations and synthesis of our candidate hardware designs. As discussed in Section 5.2, SECDA-TFLite provides the instantiation of the SECDA methodology within TFLite, thus avoiding many manual steps. Once this occurs, the developer can follow the SECDA methodology to develop and define their chosen accelerators, leveraging the utilities available in SECDA-TFLite. The following sections give an overview of how we did this for our case study.

Initialisation

After setting up the environment using the utilities provided by SECDA-TFLite (as described in Section 5.2.2), the developer can use the SECDA-TFLite simulation delegate to make a delegate for their target accelerator, specifying features such as the operation type(s) they want to accelerate.

As in a typical SECDA workflow, for the first iteration of development, the developer may define a native C++ implementation of the target operation. The native C++ implementation can be a useful starting point for the designer to develop the hardware design. Over time, the developer can replace this stub with SystemC hardware definitions, producing a viable first accelerator design.

SystemC Simulation Co-Design/Co-Verification

After adding simple SystemC constructs to our accelerator module, we develop hardware components such as hardware buffers or processing units in SystemC, to replace the initial implementation. As part of the SECDA methodology, we developed the simulation testbench and implemented hardware components for computing the target operation (e.g., weight buffers and multipliers). Using the testbench and the end-to-end simulation environment, we go through several iterations where we fine-tune the design of our accelerator components. This fine-tuning ensures that each hardware component is efficient in terms of the hardware resources utilised and clock cycles spent. This also includes ensuring that the overall behaviour of the accelerator architecture is efficient and that no component is creating a bottleneck.

To help perform this fine-tuning, we can use the profiling tool within SECDA-TFLite, as discussed in Section 5.2.2. Our pre-defined metrics, such as buffer utilisation or clock cycle counts can be used, or custom metrics can be defined. Using these metrics, the developer can adapt the hardware design and accelerator driver iteratively. For example, by tracking processing element (PE) utilisation, we can ensure that there is little to no idle time.

Design Loop

SECDA-TFLite provides developers with the utilities required to instantiate an accelerator development environment quickly. Once this is done, the main design loops of SECDA begin. Developers can switch between evaluating simulated and FPGA-synthesised versions of their hardware designs, noting bottlenecks and making design iterations. One key benefit of using the SECDA-TFLite toolkit is that leveraging the delegate system simplifies switching between simulation and FPGA evaluation.

Table 5.1: PYNQ Z1 FPGA resources utilised per accelerator design.

Accelerator \ Resource	BRAM	DSP	FF	LUT
Vector Mac	221	188	61127	50298
Systolic Array	160	196	59585	33139
FC-GEMM	224	164	35580	26585

5.4.2 Accelerators Designs

Following the workflow described in Section 5.4.1, we developed and integrated three hardware accelerator designs within SECDA-TFLite. All three accelerator designs follow an output-stationary dataflow approach [Kwo+19], which was chosen to remove the need to store many intermediate results on valuable on-chip memory or incur latency and power overheads associated with storing them off-chip. From the original SECDA case study, both the Vector Mac (VM) and Systolic Array (SA) designs have been updated to support per-axis quantisation and the signed-integer quantisation scheme required by newer versions of TFLite. The new FC-GEMM accelerator used for the transformer model was developed entirely from scratch using the SECDA-TFLite toolkit. Table 5.1 contains the resource utilisation for each accelerator design, which was estimated after HLS; all three designs were configured to run at 200MHz. Note that although the number of DSPs varies across designs, all three designs contain the same number of MAC PEs. Due to DSP limitations, a portion of the PEs are instantiated using LUTs instead of DSPs. The following section gives a brief overview of FC-GEMM accelerator design, with more detailed descriptions of the VM and SA designs in Section 4.4.3.

Fully Connected GEMM Accelerator (FC-GEMM)

Figure 5.6 shows an overview of the FC-GEMM accelerator design, which we use to accelerate FC layers within transformer models such as BERT. The AXI-MM interface performs data movement between main memory and on-chip memory, allowing the accelerator to access the main memory directly to load and store data. The AXI-MM interface decreases the need for complicated data packing and ordering from the accelerator driver, which is otherwise required by the two previous designs that used the AXI-Stream interface. However, the AXI-MM interface comes at the cost of potentially higher data transfer latency.

The design contains five key hardware modules: Fetch, Load, Scheduler, Compute, and Store Units. For further details of these modules, refer to Sections 5.4.3. The computation is performed by a single *Compute Unit*, that contains a $4 \times 4 \times 16$ MAC array. For FC-GEMM, we opt for a single large *Compute Unit* as the BERT models we

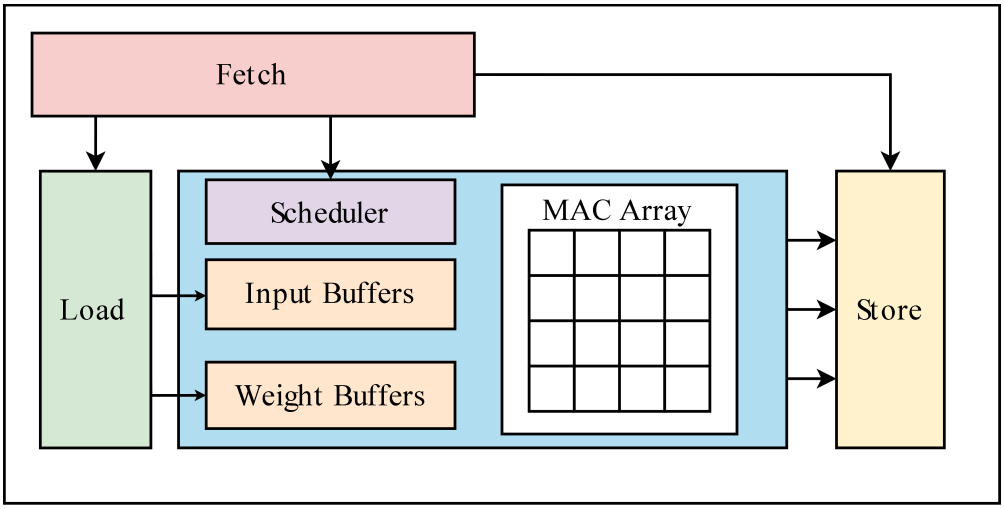


Figure 5.6: FC-GEMM accelerator design, featuring a single Compute Unit.

target consistently contain large MatMul dimensions, which ensure the *Compute Unit* is fully utilised during GEMM, as opposed to multiple smaller compute units which are more efficient for smaller sized GEMM more prevalent in CNN models. This, additionally simplifies the work of the *Scheduler Unit* as it only needs to decode the compute instructions for a single *Compute Unit*.

5.4.3 Accelerator Components

The accelerator designs are built around a common set of components used to perform the key operations required for DNN inference. Here, we describe the components used within the FC-GEMM accelerator design and the updated *Post Processing Unit* (PPU) used in the VM and SA designs. The rest of the components are described in Section 4.4.3.

FC-GEMM Specific Hardware

The FC-GEMM design differs significantly from the VM and SA designs, and it is a completely new accelerator design in this work. It features five key components: Fetch, Load, Scheduler, Compute, and Store Units.

- The **Fetch** Unit receives control signals to start the accelerator along with the number of instructions to load from main memory.
- The **Load** Unit receives instructions from the *Fetch* Unit to load data into either input, weight or bias buffers. The instruction contains the main memory address along with the size and stride of the data that needs to be loaded.

- The *Scheduler* for the FC-GEMM design decodes the compute instructions and directs the MAC Array to perform the GEMM computation.
- The **Compute** Unit is a $4 \times 4 \times 16$ MAC array, which simplifies the data orchestration within the accelerator.
- The **Store** Unit receives output data from the Compute Unit and, similarly to the PPU, performs post-processing steps, including quantisation, bias addition, and application of the ReLU activation function. The *Store* Unit reads instructions from the *Fetch* Unit to get the destination address within the host-side memory to write back the processed data and finally coordinates this data transfer to the main memory using AXI-MM.

Post Processing Unit (PPU)

The PPU is only used in the VM and SA designs and has been updated from the original case study. The PPU receives `int32` output tiles from their adjacent processing unit and applies the post-processing pipeline to obtain the quantised `int8` result tiles. Due to the change in the quantisation scheme from earlier TFLite versions, the PPU was updated to read additional quantisation parameters (e.g., the axis scaling factor) and apply them per output tile. By performing the post-processing steps within the accelerator rather than the CPU, we reduce the size of our output data by a factor of 4, which translates into significant inference time savings. Note the FC-GEMM's *Store* component performs some of the same functions.

5.4.4 Accelerator Drivers

For each accelerator design, the driver needs to communicate with the accelerator and ensure the correct instructions and data are passed to the accelerator. Careful co-design of the accelerator driver and accelerator hardware is required to optimise the performance effectively. This section gives a brief overview of the accelerator drivers used across the three accelerator designs.

Convolutional Layers

The VM and SA accelerators target the same operation type, namely convolutional layers. Hence, the two designs share very similar driver software. The accelerator driver applies the *im2col* operation to the input tensor, a required step for our definition of GEMM convolution. For most parts, the new drivers for the VM and SA

designs within the SECDA-TFLite delegate are similar to the original GEMM driver presented in Section 4.4.2. The key difference is that the new drivers are fully contained within the SECDA-TFLite delegate, unlike previous version which was connected to the *Gemmlowp* library. Additionally, the new drivers support signed 8-bit quantisation, which requires more metadata to be passed to the accelerator.

Fully Connected Layers

The FC-GEMM accelerator driver has several key differences since it targets a different DNN operation from the VM and SA designs. However, it ultimately performs the same function, namely processing data passed between TFLite and the accelerator. In addition, since the accelerator reads and writes host memory directly, the driver needs to identify and pass the relevant memory addresses to the accelerator.

Overall, the key responsibilities of the FC-GEMM accelerator's driver are:

(i) Computing quantisation parameters and managing the bias and activation function metadata, which will be passed to the accelerator; (ii) Padding and copying TFLite inputs tensors to memory-mapped input buffers; (iii) Generating the instructions required for the accelerator to compute the given FC layer; (iv) Copying back computed results from the memory-mapped output buffer to the TFLite outputs tensors; (v) Managing all the accelerator control signals, such as initialising the accelerator and waiting for the accelerator to finish the computation.

5.5 Evaluation

5.5.1 Experimental Setup

We evaluated the three accelerator designs (see Section 5.4.2) on the PYNQ-Z1 board, which includes an edge FPGA and a 650MHz dual-core ARM Cortex-A9 CPU. At the time of the experiments, we utilised the TFLite Model Benchmarking tool[2] to run all experiments; later once the SECDA-TFLite benchmarking suite was developed we were able to use it to re-run and extend the experiments.

First, we benchmarked seven widely used CNN models quantised to signed 8-bit integers: MobileNetV1 [How+17], MobileNetV2 [San+18], InceptionV1 [Sze+15], InceptionV3 [Sze+16], ResNet18 [He+16], ResNet50, and EfficientNet-Lite [TL19], all trained for the ImageNet dataset [Rus+15]. We evaluated each CNN model's CPU-only inference times in TFLite using 1 and 2 CPU threads. We compared the median

[2] `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools`

Table 5.2: Details on the DNN models used throughout our evaluation. For CNN models, CONV layers are shown and for BERT models, FC layers are shown.

Model	CONV/FC Layers	MACs (Million)	Parameters (Million)	Model Size (MBs)
MobileNetV1	14	569	4.2	11.4
MobileNetV2	35	300	3.8	12.8
InceptionV1	57	1502	6.6	20.5
InceptionV3	94	5725	23.9	75.2
ResNet18	20	1800	11.5	30.5
ResNet50	54	3,800	25.6	61.5
EfficientNet-L	61	2,550	13.0	42.6
MobileBert	362	2,850	25.3	345.4
Albert	554	2,609	11.0	178.3

of 100 runs against our VM and SA designs implemented through the TFLite delegate system. In addition to comparisons against our CPU-only baseline, we reproduced and extended our original comparison (see Section 4.5.3) against the state-of-the-art VTA accelerator on the same device, as discussed later in Section 5.4.4.

Similarly, we benchmarked two popular BERT models also quantised to signed 8 bits integers: MobileBert [Sun+20] and Albert [Lan+19], both defined for the SQAUD dataset [Raj+16] with sequence lengths of 384 and 128 for the respective models. For each BERT model, we again evaluated CPU-only inference times in TFLite using 1 and 2 CPU threads, taking the median (to not let outlier runs affect the results) of 100 runs to compare against our FC-GEMM accelerator. All reported runs in this section have a standard deviation of less than 0.7%. For more details on the experimental hardware setup, refer back to Section 1.1.2.

Table 5.2 contains key details of all the DNN models used throughout our experiments. Note that non-accelerated TFLite layers for both sets of benchmarks use their native C++ implementations and are compiled with the recommended TFLite optimisations for the target platform. This included enabling NEON-based vector instructions, multi-threading, and utilising the Gemmlowp [Goo24] backend library. Conversely, our accelerated layers use custom-designed CPU-side accelerator drivers (see Section 5.4.4) that include handwritten software optimisations (e.g., vector instructions) developed using the SECDA-TFLite toolkit. We gather energy metrics using a COOWOO digital USB power meter [COO]. Figure 5.7 shows the performance speedup of the SECDA-TFLite accelerators against single thread CPU-only inference for the CNN and BERT models under study. Rest of the section goes into these results in more detail.

5.5.2 CNN Results

Table 5.3 shows the breakdown of inference time and energy consumption for the CNN models under study for a single image using the CPU (1 and 2 threads) and

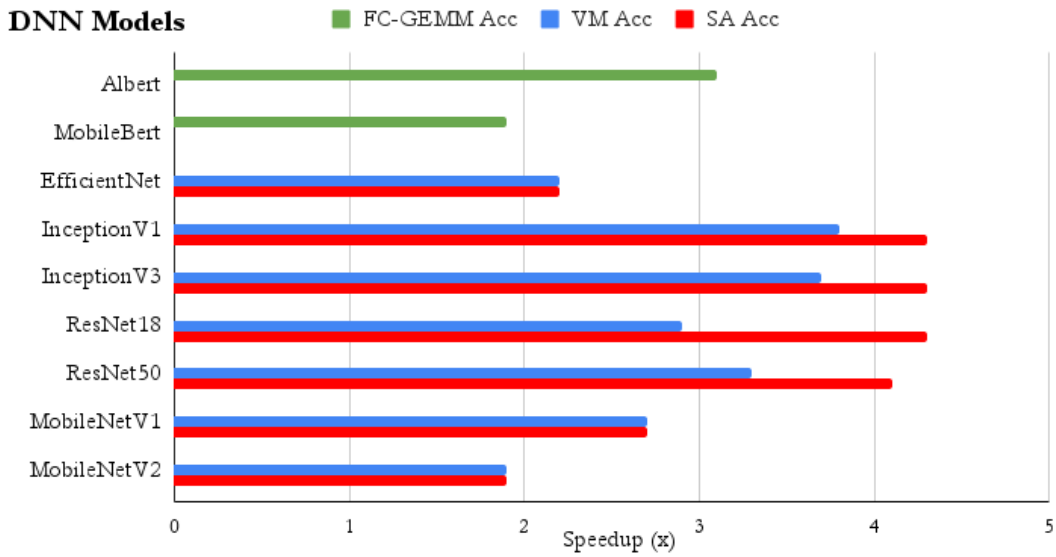


Figure 5.7: Performance speedup of the SECDA-TFLite accelerators against single thread CPU-only inference for the CNN and BERT models under study.

the two accelerator designs (VM and SA) with a stddev of 0.5%. The time is split between convolutional layers (T_{CONV}), which our accelerators target, and all other layers ($T_{Non\text{-}CONV}$), which run on the CPU. Note this table is an updated and more comprehensive version of Table 4.2 from Section 4.5.2 for the updated VM and SA designs.

Performance Across Models

For the VM accelerator, we observe an average speedup across models of 2.9× and 1.8× and an average energy saving of 2.6× and 1.7× for one and two threads, respectively, in each case when compared to CPU-only inference. Similarly, for the SA accelerator, we observe an average speedup across models of 3.4× and 2.0× and an average energy saving of 2.9× and 1.9× for one and two threads, respectively, in each case when compared to CPU-only inference.

We also observe that InceptionV1 achieves the best speedup relative to the CPU-only version, with 3.8× and 2.1× speedup for one and two threads, respectively for VM, and 4.3× and 2.4× respectively for SA. Compared to MobileNetV1 and MobileNetV2, which feature depthwise separable convolutions (meaning that each convolutional layer performs fewer MACs per input), InceptionV1's standard convolutions have greater potential for GEMM acceleration since the relative time-cost of its data preparation stage is smaller. Additionally, for InceptionV1, InceptionV3, ResNet18, and ResNet50, we observed negligible speedup for multithreaded execution relative to the other models due to the larger number of GEMM operations, coupled with our pipelined execution.

Table 5.3: Inference time (milliseconds, speedup) and energy consumption (joules, speedup) for the 7 CNN models under study when using 1/2 CPU threads and accelerator designs.

DNN	Hardware setup	T_{CONV}	$T_{Non\text{-}CONV}$	Total time		Energy	
MobileNetV1	CPU (1 thr)	566	165	732	1.0x	2.05	1x
	CPU (1 thr) + VM	107	164	271	**2.7x**	0.79	**2.6x**
	CPU (1 thr) + SA	107	164	271	**2.7x**	0.83	2.5x
	CPU (2 thr)	288	103	391	1.0x	1.26	1.0x
	CPU (2 thr) + VM	107	102	209	1.9x	**0.65**	1.9x
	CPU (2 thr) + SA	107	101	**208**	1.9x	**0.65**	1.9x
MobileNetV2	CPU (1 thr)	430	204	634	1.0x	1.84	1.0x
	CPU (1 thr) + VM	124	203	327	**1.9x**	0.94	**2x**
	CPU (1 thr) + SA	126	199	325	**1.9x**	0.94	**2x**
	CPU (2 thr)	219	128	347	1.0x	1.08	1.0x
	CPU (2 thr) + VM	124	128	**252**	1.4x	**0.76**	1.4x
	CPU (2 thr) + SA	126	127	253	1.4x	**0.76**	1.4x
InceptionV1	CPU (1 thr)	1300	70	1370	1.0x	3.64	1.0x
	CPU (1 thr) + VM	291	67	358	3.8x	1.08	3.4x
	CPU (1 thr) + SA	254	66	**320**	**4.3x**	**1.01**	**3.6x**
	CPU (2 thr)	687	72	759	1.0x	2.05	1.0x
	CPU (2 thr) + VM	291	67	358	2.1x	1.08	1.9x
	CPU (2 thr) + SA	255	66	321	2.4x	**1.01**	2.0x
InceptionV3	CPU (1 thr)	5182	312	5494	1.0x	15.44	1.0x
	CPU (1 thr) + VM	1213	283	1496	3.7x	4.72	3.3x
	CPU (1 thr) + SA	1001	279	**1280**	**4.3x**	4.32	**3.6x**
	CPU (2 thr)	2673	310	2983	1x.0	9.04	1.0x
	CPU (2 thr) + VM	1215	281	1496	2.0x	4.75	1.9x
	CPU (2 thr) + SA	1003	277	**1280**	2.3x	**4.25**	2.1x
ResNet18	CPU (1 thr)	1680	53	1733	1x	3.67	1.0x
	CPU (1 thr) + VM	558	48	606	2.9x	1.80	2.0x
	CPU (1 thr) + SA	356	46	**402**	**4.3x**	**1.30**	**2.8x**
	CPU (1 thr) + VTA	-	-	1369	1.3x	3.28	1.1x
	CPU (2 thr)	876	53	929	1.0x	2.45	1.0x
	CPU (2 thr) + VM	558	49	607	1.5x	1.76	1.4x
	CPU (2 thr) + SA	356	46	**402**	2.3x	**1.30**	1.9x
	CPU (2 thr) + VTA	-	-	737	1.3x	1.51	1.6x
ResNet50	CPU (1 thr)	3200	475	3675	1.0x	10.08	1.0x
	CPU (1 thr) + VM	798	328	1126	3.3x	3.35	3.0x
	CPU (1 thr) + SA	588	324	912	**4.1x**	2.84	**3.5x**
	CPU (1 thr) + VTA	-	-	1759	2.1x	4.39	2.3x
	CPU (2 thr)	1648	410	2058	1.0x	5.76	1.0x
	CPU (2 thr) + VM	799	328	1127	1.8x	3.31	1.7x
	CPU (2 thr) + SA	586	324	**910**	2.3x	2.84	2.0x
	CPU (2 thr) + VTA	-	-	1036	2.0x	**2.81**	2.1x
EfficentNet-L	CPU (1 thr)	2665	1230	3895	1.0x	10.98	1.0x
	CPU (1 thr) + VM	566	1241	1807	**2.2x**	5.33	2.1x
	CPU (1 thr) + SA	555	1225	1780	**2.2x**	5.44	2.0x
	CPU (2 thr)	1335	667	2002	1.0x	6.12	1.0x
	CPU (2 thr) + VM	567	650	1217	1.6x	**3.78**	1.6x
	CPU (2 thr) + SA	555	651	**1206**	1.7x	3.92	1.6x

This means that the CPU-side latency due to data format conversions is 'hidden" by the accelerator's computation, resulting in minimal benefits from two threads.

Updated Accelerator Performance

Overall, we observe that the updated VM and SA designs decrease performance compared to the original designs. This is noticeable when comparing T_{CONV} times for the VM and SA designs in Table 5.3 to the original results in Table 4.2. This decrease in performance is as expected, and is due to the additional computational demand introduced by the per-axis quantisation and the signed 8-bit integer data format, as opposed to the original unsigned 8-bit integer and per-tensor quantisation.

Conversely, the updated results still present the trends we observed in the original results. For example, we observe less speedup and energy consumption with dual-thread and the bottleneck analysis remains similar to as described in Section 4.5.2.

VM vs SA Performance Comparison

Comparing our two designs, SA achieves slightly better performance, 16% on average in latency and up to 9% in energy savings. This performance difference between SA and VM can be attributed to the different strategies used to perform GEMM and the different configurations of the MAC PEs. The SA design contains a large array of PEs, whereas the VM design consists of four smaller GEMM units, introducing higher overhead for scheduling GEMM operations. While VM's smaller GEMM units allow for smaller tile sizes, the SA design's single large systolic array leads to higher data reuse, lowering the number of BRAM reads.

Note that both accelerator designs could still be further refined; however, the purpose of the case study is to highlight that by using SECDA-TFLite, we were able to quickly port, redesign, and further iterate upon the previously defined VM and SA accelerators while improving inference time performance and energy consumption against the CPU-only case.

5.5.3 BERT Results

Table 5.4 shows the breakdown of inference time and energy consumption for the two BERT models under study using the CPU (1 and 2 threads) and the FC-GEMM accelerator with a stddev 0.1%. The time is split between Fully Connected layers (T_{FC}) which our accelerator targets, and all other layers ($T_{Non\text{-}FC}$) which run on the CPU.

Table 5.4: Inference time (seconds, speedup) and energy consumption (joules, speedup) for the 2 BERT models under study when using 1/2 CPU threads and the FC-GEMM accelerator.

DNN	Hardware setup	T_{FC}	$T_{Non\text{-}FC}$	Total time		Energy	
MobileBert	CPU (1 thr)	8.62	3.42	12.04	1.0x	3.02	1.0x
	CPU (1 thr) + FC-GEMM	2.74	3.41	6.15	**2.0x**	**1.58**	**1.9x**
	CPU (2 thr)	4.44	3.47	7.91	1.0x	2.09	1.0x
	CPU (2 thr) + FC-GEMM	2.64	3.39	**6.04**	1.3x	1.62	1.3x
Albert	CPU (1 thr)	9.88	2.59	12.46	1.0x	3.13	1.0x
	CPU (1 thr) + FC-GEMM	1.42	2.57	4.00	**3.1x**	**1.08**	**2.9x**
	CPU (2 thr)	4.47	3.07	7.54	1.0x	2.02	1.0x
	CPU (2 thr) + FC-GEMM	1.40	2.57	**3.96**	1.9x	**1.08**	1.9x

For our FC-GEMM accelerator, we observe an average speedup across models of 2.5× and 1.6× and an average energy saving of 2.4× and 1.6× for one and two threads, respectively, in each case when compared to CPU-only inference.

Similar to the CNN experiments, we observe less speedup and energy consumption with dual-thread execution as expected since the compute capacity of the CPU doubles while the accelerator designs remain the same.

Comparing the performance of the two models, we find that the accelerator provides a 3.1× speedup for Albert while only providing a 1.9× speedup for MobileBert for single thread execution. We perform further analysis of the single thread inference of the two models to identify the reason for the difference in performance improvement between the two models while using the accelerator.

We observe that the time spent during inference on FC layers goes from 79% to 36% when comparing CPU-only inference against accelerated inference for Albert. Similarly, FC layers inference time percentage goes from 72% to 46% for MobileBert. This highlights that when accelerated, the remaining layers of the models become the majority of the inference time, especially for Albert.

We investigate further and break down how the FC layers are delegated during inference utilising the accelerator. Note that the FC delegate used for the FC-GEMM accelerator is capable of grouping consecutive FC layers within a given model before inference. We observe that by using the accelerator delegate, we reduce the number of FC layers by 5× and 2.5× for Albert and MobileBert, respectively. Hence, this grouping optimisation is the key reason behind the difference in the performance boost achieved by utilising the accelerator for Albert and MobileBert.

5.5.4 Comparison with state-of-the-art DNN accelerators

As a final evaluation, we compare our designs against the state-of-the-art VTA accelerator [Che+18b] using the TVM compiler framework, similar to the SECDA case study. The results show that the designs developed using SECDA and the SECDA-TFLite toolkit are competitive with VTA. In single thread comparison with the VTA accelerator, the VM and SA accelerators are 1.61x and 2.1x faster on average across the models, respectively, and similarly, the VM and SA accelerators report 39% and 12% less energy consumption on average, respectively. In dual thread comparison, our VM design is on par for ResNet18 and only 16% worse for ResNet50 in terms of latency, while VTA reports 43% and 23% less energy consumption for ResNet18 and ResNet50, respectively. Our SA design outperforms VTA by 35% and 6% in terms of latency for ResNet18 and ResNet50, respectively, but VTA reports 14% and 5% less energy consumption for ResNet18 and ResNet50, respectively.

For BERT acceleration, to the best of our knowledge, and when this case study was conducted, there were no publicly available FPGA-based DNN accelerator inference designs targeting transformer models that we could compare against. Hence, we only compare against CPU-only inference.

5.6 Summary

In this chapter, we presented SECDA-TFLite, an open-source toolkit that eases developing new hardware accelerators for edge DNN inference, following the hardware-software co-design SECDA methodology. SECDA-TFLite provides the initial environment setup within the TFLite DNN framework, as well as a set of tools and utilities to aid in the development of accelerator designs. It provides the infrastructure to initialise and follow the SECDA design loop within TFLite. As a result, developers are able to co-design new accelerator designs for TFLite, bypassing many of the initial setup challenges and overheads.

The SECDA-TFLite toolkit enables tight integration of accelerator designs with TFLite while enabling the developer to easily follow the SECDA design loop within TFLite, thus improving opportunities for co-design of the accelerator delegate and driver. We provide utilities for SystemC interfacing, simulation profiling, data communication, and multi-threading for the driver. Additionally, we provide tools to automate the hardware synthesis, benchmark new accelerator designs, and visualise profiled data.

As a case study, we updated two existing GEMM-based accelerator designs using the SECDA-TFLite toolkit and developed a new one for BERT models. We evaluated

the accelerators' performance against the CPU baseline and demonstrated that they outperformed the CPU in all cases across seven CNN models and two BERT models. Overall, we demonstrated through SECDA-TFLite that the SECDA methodology is suitable for developing FPGA-based DNN accelerators using a target Machine Learning framework such as TFLite. Our toolkit provides additional features to aid in the development of new CNN and transformer-based accelerators.

In the next chapter, we will build upon the SECDA methodology and the SECDA-TFLite toolkit to develop a new hardware platform for LLM inference, utilizing the tools we formalised within SECDA-TFLite, we define the SECDA design platform for LLMs.

6 SECDA-LLM

Through the rise of large language models (LLMs), the need for efficient inference accelerators for edge devices has become more apparent. In this chapter, we introduce SECDA-LLM, a platform that simplifies the creation of new FPGA-based hardware accelerators for edge LLM inference using the hardware/software co-design SECDA methodology within the *llama.cpp* environment. SECDA-LLM follows suit with the SECDA-TFLite toolkit, providing a seamless connection between the SECDA design environment and the target application framework, *llama.cpp*.

This chapter is structured as follows: Section 6.1 introduces the chapter and motivates the need for the SECDA-LLM platform. Section 6.2 presents the design of the SECDA-LLM platform and the integration of SECDA tools within the *llama.cpp* framework. Section 6.3 presents a short case study using the SECDA-LLM platform. Finally, Section 6.4 summarises the chapter.

6.1 Introduction

Large language models (LLMs) are an emerging class of machine learning (ML) systems geared toward learning from huge text-based datasets. LLMs such as GPT-3[Bro+20] have revolutionised the ability of Artificial Intelligence (AI) systems to understand and generate human language. Due to innovative changes in model architecture and training methods, and through the help of the popularity of online services like ChatGPT [Ray23], the field of LLMs is evolving rapidly.

The number of everyday users is also growing rapidly due to the myriad of use cases from translation [Yao+24], classification [Sun+23], code generation [Liu+23] to healthcare [Clu+23]. Additionally, cloud-based LLM services are currently the go-to method of access to LLMs for everyday users. However, as the availability of open-source LLMs and datasets increases, especially over the last few years, the need for edge-based, localised access and execution of LLMs has become more sought after. Massive community-driven pushes have facilitated easy access to LLMs and rapid prototyping

of new models and optimisations to enable efficient LLM inference on edge devices. At the forefront of these pushes is the GPT-Generated Model Language [Ger24a] (GGML). GGML is a tensor library for ML that specialises in enabling large models and high performance on commodity hardware. Furthermore, *GGML*'s *llama.cpp* project [Ger24b] is specialised towards running LLMs on edge devices, supporting LLM inference on commodity CPUs and GPUs.

Unfortunately, LLMs can be very computationally demanding, even for inference. In addition, due to their large memory footprint, they require high memory capacity and bandwidth. These properties of LLMs make them challenging to execute on resource-constrained edge devices. For example, running LLMs on mobile phones or Internet-of-Things devices (IoT) devices is, in some cases, impossible due to memory constraints. Hence, there is a great demand for developing and deploying custom hardware accelerators to run these LLMs efficiently on resource-constrained edge devices. Fortunately, FPGAs are ideal for designing new flexible and power-efficient accelerators in order to employ optimisations to improve LLM performance, such as block floating point quantisation. While some FPGA-based accelerators [Kha+21; Lu+20] already exist for LLM inference at the edge, with constant changes to LLM architectures and optimisations, we are in need of new specialised FPGA-based accelerators.

To create new and innovative FPGA-based accelerator architectures for LLM inference at the edge, we need ways to quickly prototype and evaluate LLM-based inference accelerators to reduce development time and increase design space exploration. Hence, we present SECDA-LLM, a new platform for designing, integrating and deploying specialised accelerators for LLMs at the edge. SECDA-LLM employs the SECDA design methodology, and similar to SECDA-TFLite, it allows the user to quickly prototype accelerator designs with the target application framework, in this case, *llama.cpp* project. Our SECDA-LLM platform enables the designer to consider hardware-software co-design optimisations in terms of both algorithmic and hardware implementations [Gib+25] and makes deployment of LLMs through FPGA-based accelerators effortless. Note that the SECDA-LLM platform and the work presented in this chapter is ongoing, as such, the current contributions of this chapter are as follows:

- *SECDA-LLM*, a design platform using the SECDA methodology which enables the design, integration and deployment of FPGA-based accelerators for LLMs on resource-constrained edge devices.
- A case study to demonstrate *SECDA-LLM*, where we prototype and deploy a new accelerator to efficiently execute quantised MatMul operations during LLM inference.

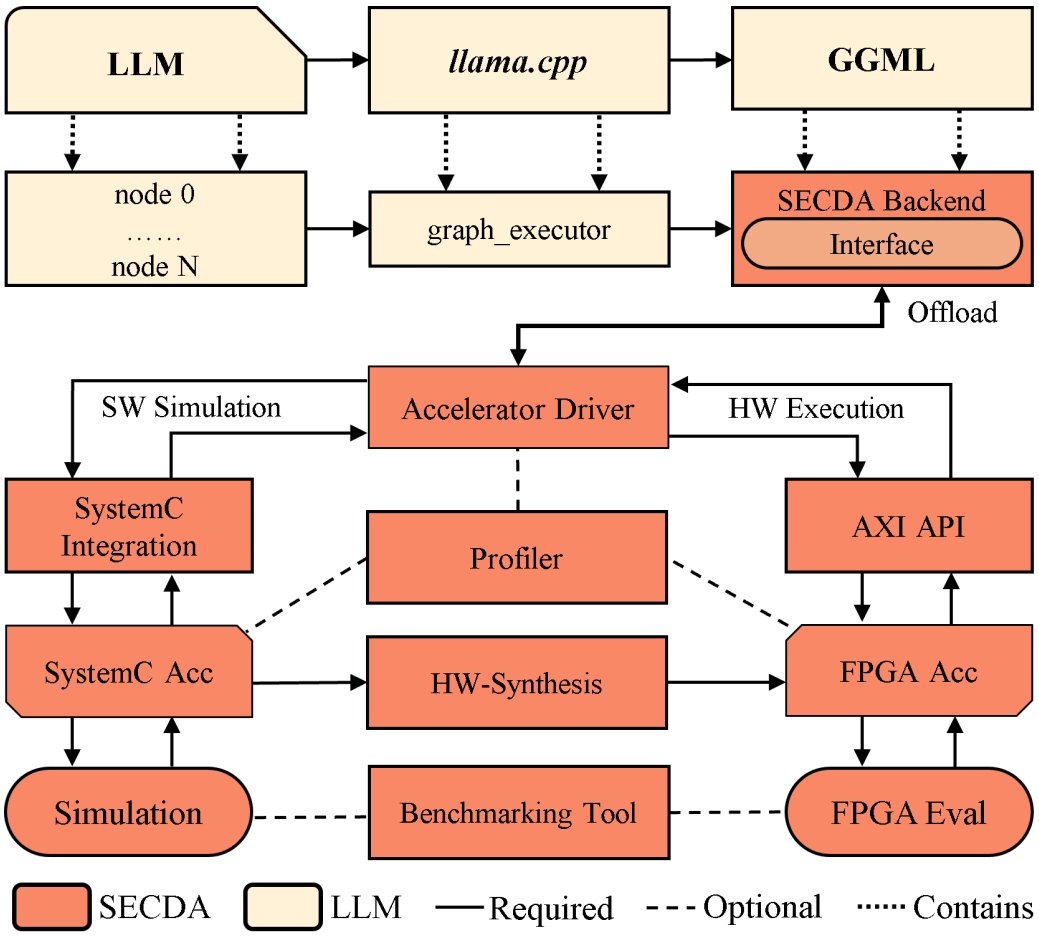


Figure 6.1: Overview of the SECDA-LLM. Key SECDA components are highlighted in orange, and the LLM components are highlighted in beige.

- Evaluation of our initial accelerator design executing the TinyLlama [Zha+24] model on the PYNQ-Z1 [Dig] board, where we achieve a 11× speedup over dual-core ARM NEON-based CPU execution for the compute-intensive MatMul operation.

6.2 SECDA-LLM Platform

SECDA-LLM is a specialised platform for creating FPGA-based LLM accelerators for edge devices using the SECDA methodology within the *llama.cpp* environment. Figure 6.1 outlines the main components of SECDA-LLM. The platform simplifies the accelerator design process by integrating the SECDA tools, thus allowing a seamless connection between the SECDA design environment and the target application framework, *llama.cpp*. This integration enables developers to begin prototyping and integrating their new designs with minimal setup overhead.

The rest of this section provides details on SECDA-LLM and: (i) how it is integrated with *llama.cpp*; (ii) how it enables the accelerator designer to prototype and simulate new designs with SystemC [Des23] simulation; (iii) the ease of hardware evaluation; (iv) the profiling and performance analysis capabilities of SECDA-LLM.

6.2.1 Integration with *llama.cpp*

Figure 6.1 shows that the SECDA-LLM platform builds upon the core *llama.cpp* project inference. Our current integration is through *llama.cpp*'s main example project, which enables users to run LLM models with minimal overhead. We can connect into *llama.cpp* once it calls any of the *GGML*'s operations, such as matrix multiply, convolution, etc.

Depending on our target operation(s), we create additional connection points from the GGML library to the SECDA environment. During these connections, we ensure the creation of a context handle to pass from the GGML environment to the SECDA environment; the context handle includes pointers memory, memory-mapped model data, access to relevant inputs tensors, quantisation, and layer parameters.

6.2.2 SECDA Environment

Within the SECDA environment, shown in Figure 6.1, the accelerator designer can start quickly prototyping the initial accelerator design and driver code. First, the user is required to create the initial driver, a simple C++ class that will gain access to the context handle provided by the offload call from within *GGML*. Second, the developer must create an initial SystemC description of their accelerator. Then, the user can instantiate their desired data communication channels between the driver and accelerator using data interfaces provided within the SECDA environment (e.g, AXI-S, AXI-MM and AXI-Lite). The developer can use these data channels for SystemC end-to-end simulation.

6.2.3 SystemC Simulation

SystemC end-to-end simulation is a crucial step in the SECDA methodology; therefore, SECDA-LLM provides access to SystemC simulation. The simulation-based design loop is shown on the bottom left half of the figure 6.1. Once the driver and accelerator are connected through the desired data communication channels, the user can perform end-to-end simulations of LLMs using SECDA-LLM. With simulation enabled, the designer can quickly prototype new driver and accelerator features, verifying correctness,

profiling performance and modelling control flow behaviour within their design. The hardware developer is able to rapidly iterate through their design process, through end-to-end simulation, to meet their target performance.

6.2.4 Hardware Evaluation

With simulation-based evaluation, the designer can quickly make fast, broad design changes. Once satisfied with their design, the designer can quickly take their SystemC-based design and perform High-level synthesis (HLS) and logic synthesis (HLX) through the hardware automation tool provided by SECDA-LLM to map it to their target FPGA, as shown on the bottom right of figure 6.1. Additionally, as SECDA-LLM is integrated with the *llama.cpp* project, we can leverage the *llama.cpp* project's compilation flow to generate pre-defined applications that use the LLMs through the *llama.cpp*'s interface. These generated applications will now have complete access to the driver and accelerator for execution on an FPGA-enabled device; see Section 6.3.3 for details.

A major benefit of the SECDA methodology, and therefore SECDA-LLM, is that we can reuse the driver and accelerator completely. For actual FPGA evaluation, the designer does not need to make any changes to the driver to enable real hardware execution, as the SECDA data interfaces switch between simulation and FPGA execution through a simple *'SYSC'* compiler flag. Once the accelerator is mapped to the target FPGA, the designer can evaluate its performance with their target applications.

6.2.5 Profiler

Through SECDA-LLM, we provide two types of profiling: simulation profiling and execution time profiling. The *profiler* module shown in figure 6.1 highlights how the profiling interacts with both the accelerator design and driver. Additionally, we are able to leverage any additional profiling tools enabled by the *llama.cpp* project.

The simulation profiling allows the designer to quickly evaluate the potential performance impact of changes to the accelerator design's hardware and software components and verify the implementation's correctness. Meanwhile, execution profiling can provide execution time for the custom driver and accelerator. Additionally, execution profiling can be used during a simulation run to profile driver execution times, which can be combined with SystemC-reported simulation times for the accelerator. This would estimate end-to-end execution time in terms of both CPU and accelerator. At the high-level, these profiling tools are the same as ones used for SECDA-TFLite, hence Section 5.2.2 provides more details.

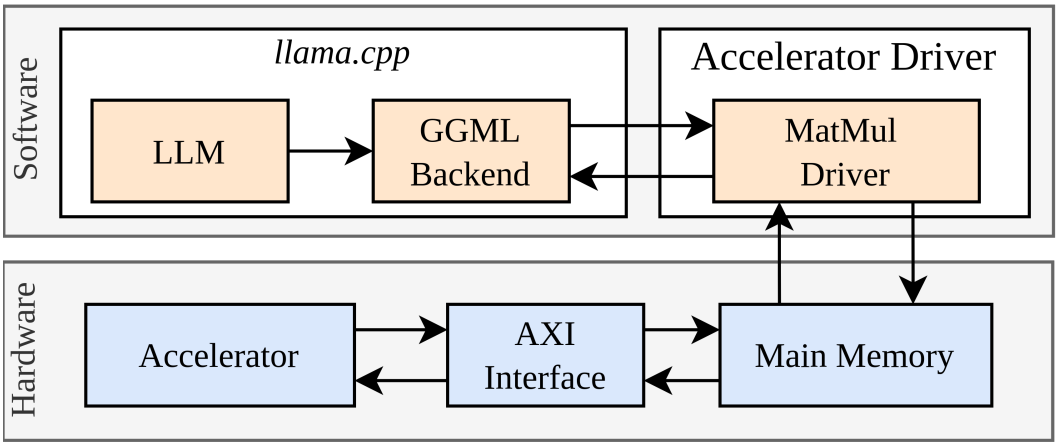


Figure 6.2: SECDA-LLM runtime model.

6.3 Case study

To demonstrate our SECDA-LLM platform and how it provides a quick and efficient design flow for developing LLM accelerators for edge devices using the SECDA methodology, we develop a new custom FPGA-based accelerator for block floating point (BFP) quantised LLM inference. Figure 6.2 highlights the execution flow when performing LLM inference with our specialised accelerators using SECDA-LLM within this case study. Note that we integrate the offloading of matrix-multiply operations through the GGML backend, which calls our MatMul driver to activate our accelerator.

6.3.1 Target Problem

For our case study, we target the acceleration of MatMul operations within our target LLM, as MatMul represents about 97% of the computations. Specifically, we accelerate the GGML's *MatMul_Q3_K_Q8_K* kernel, which uses 3-bit weights and 8-bit inputs with BFP quantisation.

Both weights and inputs are stored in what is called 'super-blocks' (SBs); these SBs are critical in maintaining LLM accuracy by adjusting mathematical scaling during computation. Figure 6.3 contains an visual representation of the *Q3_K* BFP format used for weights, where each SB can represent 256 weights (N_w) and is partitioned into 16 tiles (N_{tiles}) and each tile contains a scaling factor (6-bits) and 16 weights (3-bits). Additionally, each SB has one super-scaling factor (16-bits). By summing up the total number of bits required for the weights and the scaling factors and then dividing by the number of weights, we can determine that the BFP format requires ∼3.5 bits-per-weight, which is a significant reduction from the typical 32-bit floating-point format used in DNNs. With the *Q8_K* format used for inputs, each SB contains

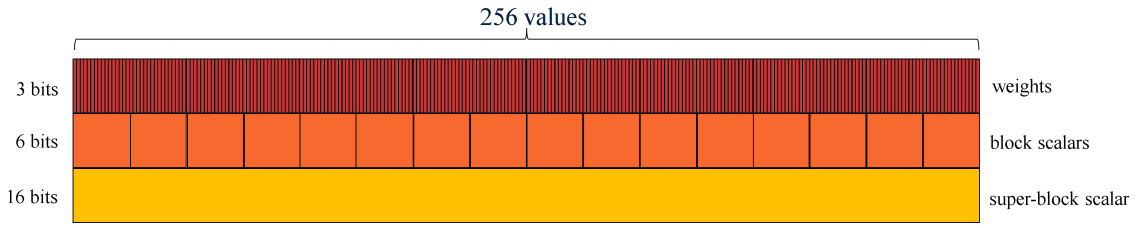


Figure 6.3: Q3_K super-block Data Format.

256 inputs (8-bits) and a single super-scaling factor *SSF* (16-bits), which equates to 8∼ bits-per-input.

6.3.2 Accelerator Design

Our accelerator design, shown in Figure 6.4, contains an *instruction decoder*, a *data mapper*, a *scheduler* and the *Super-Block Vector Processor (SBVP)*:

- The *instruction decoder* loads and decodes instructions from the AXI-Stream and then communicates the instruction throughout the rest of the accelerator.
- The *data mapper* parses the incoming data stream and maps the weight and input super-blocks into their respective weight and input buffers. Our mapping scheme enables efficient data access, so that the *SBVP* can compute without stalling the computation pipeline.
- The *SBVP* efficiently computes the dot product between the SB of weights and inputs while scaling the computation according to the SB scaling factors.
- The *scheduler* tiles the MatMul problem according to the dimension of the target layer. Additionally, it synchronises and accumulates the output data produced by the SBVP and sends the results back to the main memory using the AXI-Stream.

6.3.3 Evaluation

We evaluate our accelerator design on the PYNQ-Z1 [Dig] FPGA board. We execute inference for the TinyLlama model [Zha+24], which contains 1.1B parameters (460 MB), trained on the Guanaco dataset [Jos23]. This model contains various BFP quantisation levels, but most layers are quantised to *Q3_K*. Note that with *llama.cpp*, you can apply different quantisation levels to reduce the model size as required. For more details on the experimental hardware setup, refer back to Section 1.1.2.

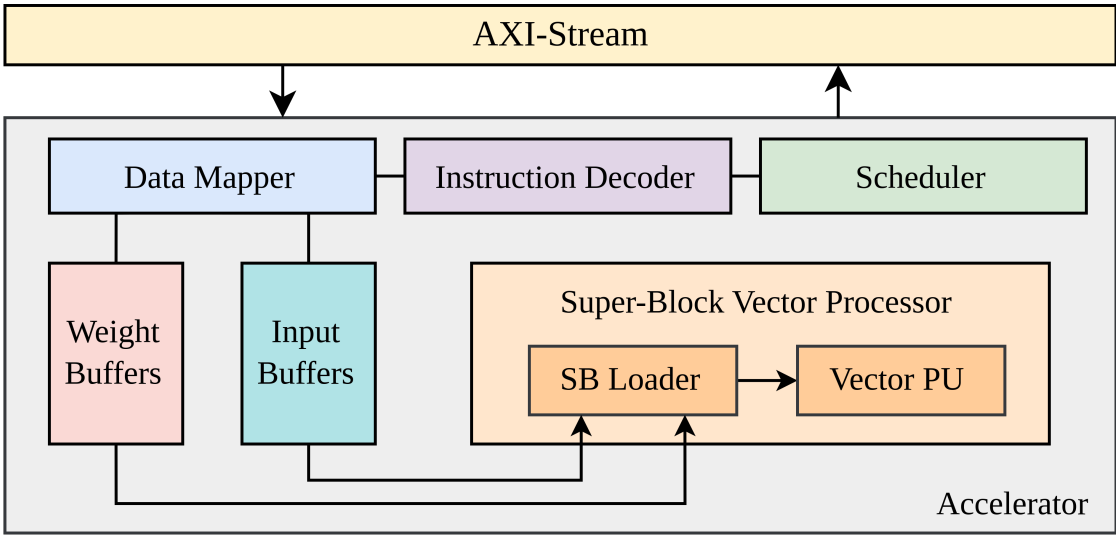


Figure 6.4: Overview of our block floating point quantised accelerator design for GGML's *MatMul_Q3_K_Q8_K* kernel.

For our experiments, we use the *llama.cpp* project's 'main' program cross-compiled for our target CPU architecture, ARMv7a, with Neon vector instructions enabled alongside our accelerator driver. We execute the TinyLlama model utilising our FPGA-mapped accelerator to offload the *MatMul_Q3_K_Q8_K* layers to obtain an initial speed of 1.7 seconds per token ($\sim$ 2 seconds per word). Figure 6.5 compares the performance of the CPU and our accelerator across single and dual-thread execution. Currently, both our accelerator and CPU-only execution do not gain performance improvement with the additional thread. We suspect this is because the workload is memory-bound. This provides a 11$\times$ speedup over CPU-only inference for execution of the MatMul kernel, drastically improving the usability of LLMs on such a resource-constraint device

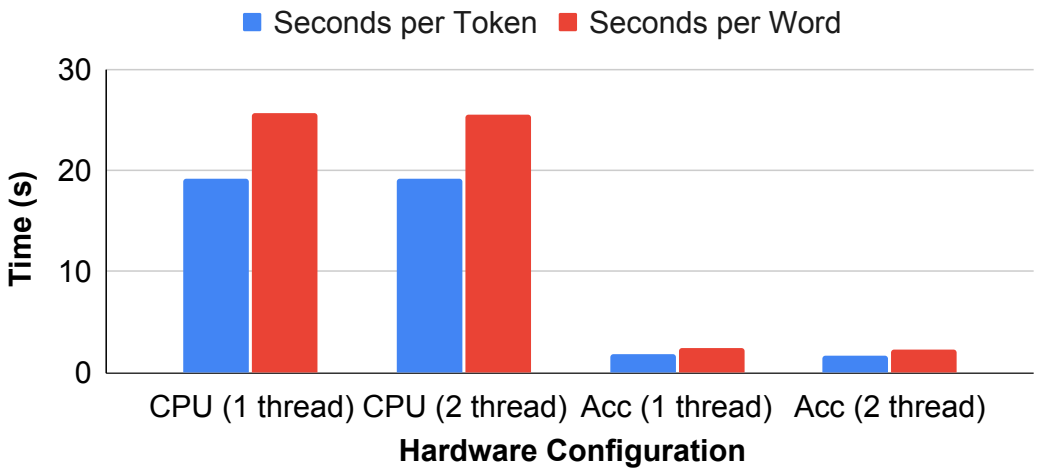


Figure 6.5: Performance of our BFP accelerator compared to CPU (1 & 2 threads).

Design Iterations

We iterated through several design changes during the design process to improve the accelerator's performance. We initially started with a simple design (v1) that only computed the dot product between the weights and inputs while considering quantisation. After profiling the design, we identified that the accelerator was underutilised due to its lack of parallelism. Hence, we designed and added the SBVP (v2) to the accelerator to process multiple SBs in parallel. Finally, we introduced the scheduler (v3) to support tiling and improve data reuse across weight tiles, reducing the number of main memory accesses needed to fetch the weights. Figure 6.6 shows the performance improvements achieved with each design iteration.

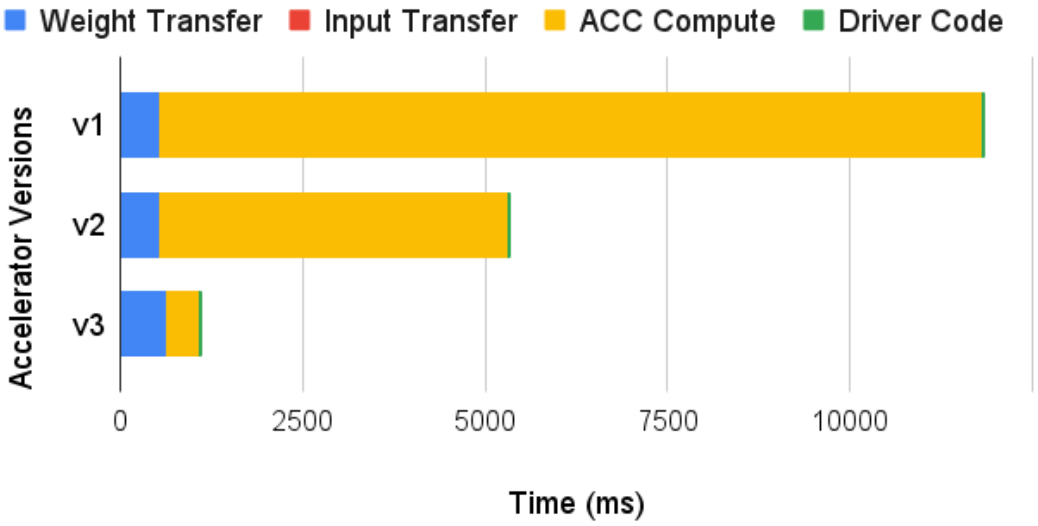


Figure 6.6: Performance improvements achieved with each design iteration of the accelerator.

6.3.4 Discussion

This short case study demonstrates the utility of the SECDA-LLM platform for developing FPGA-based accelerators for LLM inference. SECDA-LLM as a platform is still in the early stages of development, and we plan to extend it with more features to support a broader range of LLM models and optimisations. Our goal is to enable research and development of FPGA-based accelerators for LLM inference when resources are constrained and the model is too large to fit in the memory of the target device.

Our initial accelerator design for the *MatMul_Q3_K_Q8_K* kernel demonstrates the potential of FPGA-based accelerators for LLM inference in a resource-constrained edge

device. We plan to extend our accelerator design to support more LLM models and, ideally, support more quantisation levels.

6.4 Summary

This chapter introduced a novel platform, SECDA-LLM, that simplifies the creation of specialised hardware accelerators for LLM inference on resource-constrained edge devices. SECDA-LLM integrates the SECDA methodology within *llama.cpp*, enabling developers to access the SECDA tools (e.g., AXI-API, profiler), which can be used to effectively co-design new FPGA-based accelerators for LLMs with ease. As a case study, we presented a quantised MatMul accelerator design that optimises LLM inference for the TinyLlama model. We also highlight the key potential of the SECDA methodology, which is the possible performance improvements through quick design iterations.

Overall, we demonstrated that SECDA-LLM is the ideal platform for developing FPGA-based LLM accelerators, providing a tool which integrates the SECDA methodology with the *llama.cpp* framework, allowing design space exploration of LLM accelerators for resource-constrained edge FPGAs.

7 GAN Acceleration with SECDA-TFLite

One of the key challenges discussed in Section 1.2 is the need for problem-specific design and optimisation for efficient DNN acceleration. Here, we tackle the second key objective of the thesis by trying to solve this challenge for Generative Adversarial Networks (GANs).

As discussed in Chapter 2, GANs are a popular class of DNN models used for generative AI-based applications. However, the unique computational requirements of GANs make them challenging to deploy on resource-constrained devices. In this chapter, we fully utilise the capabilities of the SECDA design methodology, and the SECDA-TFLite toolkit to fully design, integrate, and evaluate a custom FPGA-based accelerator for GAN inference.

The chapter is structured as follows: Section 7.1 provides an introduction to the problem and the target GAN model. Section 7.2 outlines the key observations which led to the design of the accelerator. Section 7.3 presents the design of the accelerator and Section 7.4 evaluates the accelerator's performance. Finally, Section 7.5 summarises the chapter.

7.1 Introduction

Generative models are used in various applications, including image-super resolution [DLT16], style transfer [JAF16] and object detection [Liu+19a]. Generative models such as Generative Adversarial Networks (GANs) and Fully Convolutional Neural networks (FCNs) contain an 'upscaling' mechanism to generate new data. For example, generator modules in GANs contain specialised layers to upscale input feature maps, where the Transposed Convolution (TCONV) layer is the core of this 'upscaling' mechanism. However, this can be much more challenging that executing them on devices with powerful CPUs/GPUs, which have much higher computational and memory

capabilities [Gib+25].

When compared to the traditional convolution operation, which has been extensively studied and accelerated, the complex computing properties of the TCONV operation, such as the overlapping sum problem [Zha+17], make it challenging to design accelerator architectures that can efficiently process it, especially on edge devices with limited computational and memory capabilities.

As discussed in Section 2.4.3, there are serval methods of implementing TCONV, including the Zero-Insertion, Transforming Deconvolution to Convolution (TDC) and the Input-Orientated-Mapping (IOM) method. However, they all have their drawbacks, which are extensively discussed in Section 2.4.3 and 3.2.2. To address these drawbacks, there have been multitudes of research on developing hardware accelerators for TCONV, especially using the IOM method, as discussed in Section 3.2.2.

In this chapter, we present MM2IM, an accelerator architecture for TCONV that merges **M**atrix **M**ultiplication (**M**at**M**ul) with col**2IM** [Devc], a matrix transformation operation that rearranges data columns into blocks. Following the SECDA methodology, we designed a hardware-software co-designed approach that enables the IOM method on resource-constrained edge devices by efficiently handling key challenges, including the overlapping sum problem, ineffectual computations due to cropped outputs, and tiling TCONV computations. We developed our hardware design using SECDA-TFLite and evaluated its performance on a resource-constrained edge FPGA for various TCONV configurations, including the DCGAN model. Furthermore, we compare the performance of MM2IM against similar TCONV accelerators for resource-constrained edge devices, demonstrating our superior throughput per DSP. The contributions of this chapter are as follows:

- **MM2IM**: a new accelerator architecture that efficiently processes TCONV operations on resource-constrained edge devices using our IOM-based tiling strategy.
- **MM2IM mapper hardware module**: a compute and output mapping engine to tackle the overlapping sum efficiently and the cropped output problem on-the-fly without requiring additional memory or bandwidth.
- **Integration and evaluation of MM2IM**: using SECDA-TFLite, we compare MM2IM against our ARM Neon optimised CPU baseline. We obtain an average speedup of 84× across 261 TFLite TCONV problem configurations; and similarly achieve up to 2.6× speedup and 2× energy reduction across the DCGAN model.

7.2 Efficient Transposed Convolution

Here, we discuss the efficient execution of the Transposed Convolution (TCONV) operation using the Input-Oriented Mapping (IOM) method. While other methods are viable, we focus on the IOM method due to its efficiency, compared to the computations and transformation overheads faced by the Zero-Insertion and TDC methods, respectively.

7.2.1 Optimising Input-Oriented Mapping

Figure 7.1 highlights the TCONV operation using the IOM method implemented using MM and col2IM for the TCONV problem $tconv(2, 2, 2, 3, 2, 1)$, refer back to Section 2.4.3 for a detailed description of the TCONV operation. Note the dimensions are as follows for this example: $I_h(input\_height) = 2$, $I_w(input\_width) = 2$, $I_c(input\_channels) = 2$, $K_s(kernel\_size) = 3$, $O_c(output\_channels) = 2$, $S(stride) = 1$, and that the output dimensions, output height O_h and width O_w are defined as: $O_{hw} = S \times I_{hw} = 2$.

The MM operations are performed on input features and weights to produce the partial output matrix. Translating the TCONV dimensions to MM dimensions, we get the following: $M = I_h * I_w$, $N = K_s^2 * O_c$, and the depth dimension $K = I_c$. Hence, the partial output matrix can be represented by dimensions M and N, the rows and columns of the MM operation. Therefore, the number of operations required for the IOM method is equivalent to $I_h * I_w * I_c^2 * K_s^2 * O_c$ or simply $M * N * K$. Once the partial outputs are calculated, the final output is determined through the col2IM operation, which accumulates the partial outputs into the final output feature maps.

Note that the IOM method produces padded output feature maps, so the perimeter of the output feature map is cropped, as shown by the grey squares in Figure 7.1. Each partial output requires a dot product of the input row and filter column. The partial output is then summed to produce the final output; hence, each grey square computed represents K ineffectual computation. Additionally, the IOM method would require the intermediate results to be stored in memory, as the partial output of the MM operation needs to be coalesced to produce the TCONV outputs.

IOM Inefficient Computation

The baseline IOM method has two inefficiencies: ineffectual computations and the storage of partial outputs. The number of ineffectual computations, i.e., the dropped

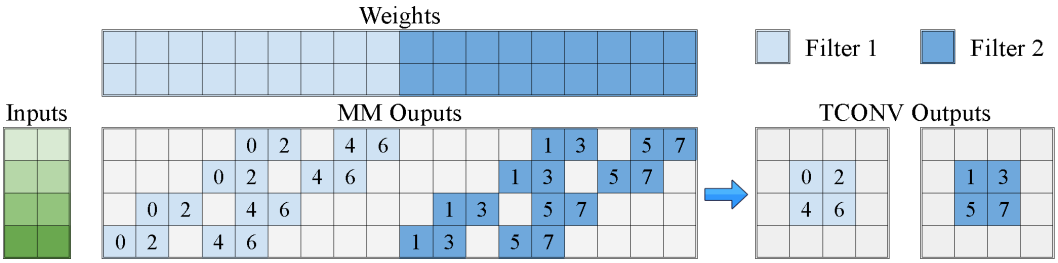


Figure 7.1: Implementing TCONV using MatMul + col2IM.

outputs D_o per TCONV problem, can be statically determined using the col2IM algorithm [Devc]. Overall, for a given TCONV problem, IOM efficiency can be determined by looking at the drop rate: $D_r = D_o/M * N$. Therefore, for a given TCONV problem with drop rate D_r, the level of computation inefficiency can be expressed as $((M * N) - D_o)/D_o$. In the case of the example in Figure 7.1, where $D_o = 12$ and $M * N = 36$, and therefore $D_r = 0.33$, the standard IOM method performs 50% more computations than needed for the example TCONV problem.

IOM Inefficient Storage

In terms of storing partial outputs, we can formulate the wasted buffer space W_s as the number of final outputs F_{outs} minus the number of partial outputs P_{outs}, calculated assuming we do not skip ineffectual computations; where $F_{outs} = O_c * O_h * O_w$ and $P_{outs} = M * N$. In an ideal scenario, we can completely skip storing partial outputs and simply accumulate them to the final output, improving buffer space efficiency by P_{outs}/F_{outs}. In the case of the example in Figure 7.1, where $P_{outs} = 36$ and $F_{outs} = 4$, this would improve space efficiency $4.5x$.

Solving IOM Inefficiencies

To solve IOM inefficiencies, we must first define the *output mapping* and the *compute mapping*. In Figure 7.1, each square in the *MM Outputs* matrix represents a partial TCONV output, and the values inside these squares represent the output index of the final TCONV outputs (shown on the right); this *output mapping* is a function of S and the input dimensions (I_h, I_w). For example, all the '0' index partial outputs are summed and stored in the '0' index of the final output feature maps. Additionally, calculating the index map of the (light and dark) blue squares in the *MM Outputs*, we derive the *compute mapping* for the given TCONV problem, that is, the index map of partial outputs that are not dropped out via col2IM.

Our **first key insight** is that by using the *output mapping* and the *compute mapping*, we can solve the IOM inefficiencies and enable an efficient accelerator architecture that

can: (i) Skip ineffectual computation of the dropped partial outputs (grey squares); (ii) Remove the need for storing partial sums in temporary memory and to be summed later; (iii) Map the outputs of the MM operation directly to the final output feature map.

7.2.2 Resource-Constrained Acceleration Dataflow

The data transfer between off-chip and on-chip memory can become a bottleneck especially on resource-constrained edge devices. Hence, we co-designed *Tiled MM2IM*, a specialised tiling strategy for MM2IM that enables weight and output stationary dataflow minimising data transfer redundancy, highlighted in Algorithm 6. Tiled MM2IM loads $filer\_step$ filters and produces the corresponding output channels within the outer loop. We load a dynamic number of input rows within the inner loop to calculate one output row per iteration. We pre-calculate the i_end_row array that holds the number of input rows required to compute the current output row.

Our **second key insight** is that using this dataflow can increase/decrease hardware parallelism depending on the resource constraints by adjusting $filer\_step$. Additionally, with *Tiled MM2IM*, we preemptively calculate partial outputs for later output rows depending on the input rows being processed.

Algorithm 6: Tiled MM2IM

Data: Initialise $filter\_step$, i_end_row

```
1  foreach c ← 0 to O_c by filter_step do
2      SendWeightFilters(c, filter_step)
3      starting ← 0
4      foreach h ← 0 to O_h do
5          rows_to_send ← i_end_row[h] + 1 − starting
6          if i_end_row[h] ≠ starting − 1 then
7              SendInputRows(starting, rows_to_send)
8          ComputeOutRow(h, c, filter_step)
9          StoreOutRow(h, c, filter_step)
10         starting ← i_end_row[h] + 1
```

7.2.3 Performance Model

We built an analytical model for our MM2IM architecture to estimate performance and guide further design choices. Our performance model accounts for the problem size and the properties of our accelerator design to assess the overall performance. Additionally,

we combine the accelerator analysis with data movement analysis to estimate the end-to-end performance for a given TCONV layer. In this section, we provide an overview of the performance model.

First, we calculate problem-specific metrics such as the number of MACs and the number of cropped MatMul outputs for the given TCONV problem. Then, we calculate the accelerator processing time, finding the processing time for each of the Processing Module (PM) and its components, the Compute Unit (CU) and the Accumulation Unit (AU), which are discussed in detail in Section 7.3.5:

$$T_{PM} = T_{CU\_compute} + T_{CU\_load} + T_{CU\_store} + T_{AU} \tag{7.1}$$

Then, we calculate the data transfer time required between the main memory and the accelerator: Note that data redundancy due to tiled inputs is taken into account.

$$T_{Data} = (W_{size} + I_{size} + O_{size} + OMap_{size}) * BW \tag{7.2}$$

Finally, we calculate the end-to-end latency of the TCONV problem based on the data movement required between the main memory and the accelerator:

$$T_{total} = T_{PM} + T_{Data} \tag{7.3}$$

Note that the total processing time only considers a single PM, assuming that all PMs are processing in parallel, which is the case in our accelerator architecture.

Our **third key insight**, through performance modelling, is that up to 35% of the end-to-end latency (T_{total}) for a given TCONV problem was due to transferring output mapping data between main memory and the accelerator. Hence, we developed the MM2IM mapper and a hardware module that completely removes the need for output mapping data transfers, discussed in Section 7.3.6.

7.3 Accelerator Architecture

Figure 7.2 overviews MM2IM, our proposed stream-based and scalable accelerator architecture, which utilises simple instructions to configure, load data and execute TCONV operations. These instructions enable MM2IM to dynamically tile TCONV operations using Algorithm 6. MM2IM exploits two dimensions of parallelism at the processing module (PM) level, splitting O_c computation across the X number of PMs (used for $filter\_steps$ in Algorithm 6) and unrolling I_c within the compute units with

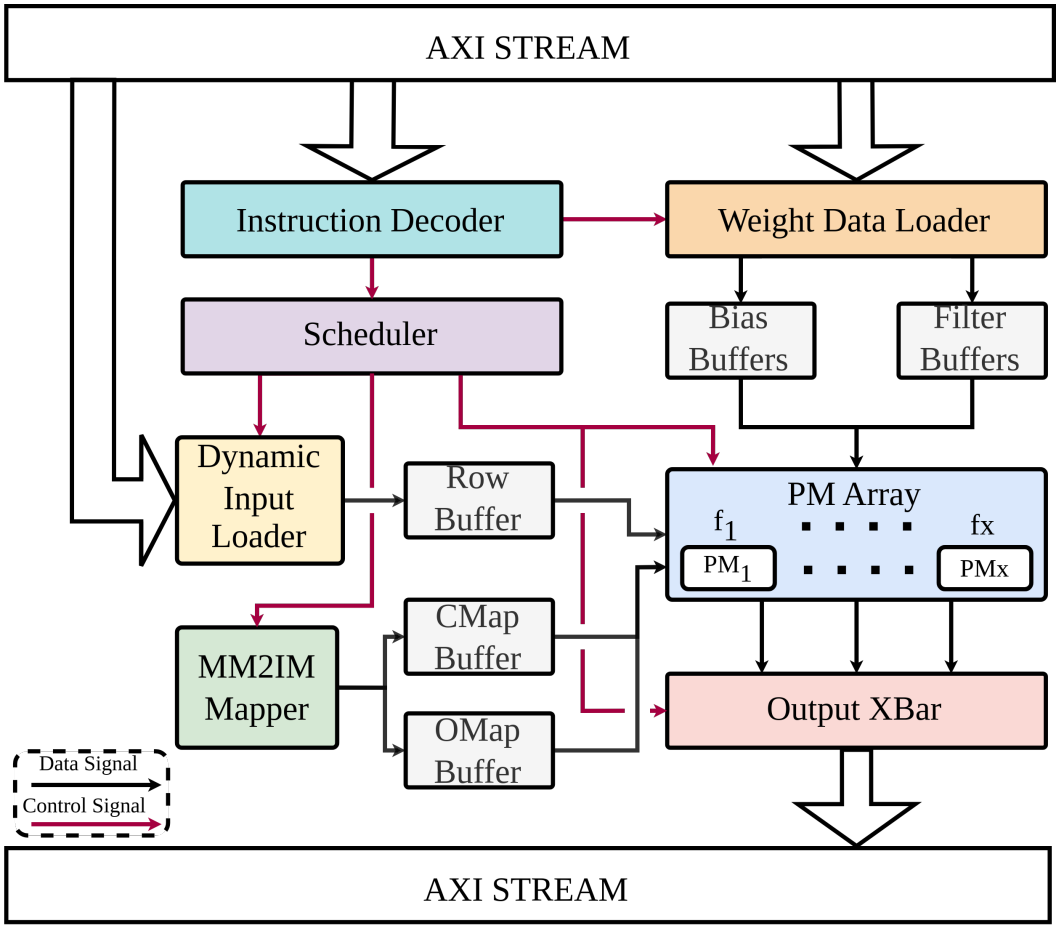


Figure 7.2: MM2IM Accelerator Architecture. The accelerator is connected to main memory via AXI-Stream buses, which used to receive instructions and send/receive data.

an unrolling factor of *UF*. Note that we have set $X = 8$ and $UF = 16$ for our instantiation. The following sections discuss the key components of the accelerator.

7.3.1 Instruction Decoder

The *instruction decoder* allows for the reconfiguration of the accelerator and execution of TCONV problems within the accelerator by decoding instructions and sending control signals to the *scheduler* and *weight data loader*. Table 7.1 shows the opcode set of micro-ISA for the accelerator and a brief description of each instruction. These instructions are generated and sent to the accelerator by the host-side driver code during the execution of a given TCONV problem. Note that some opcodes, for example opcode '0x01', are immediately followed by operand data which the accelerator expects once the instruction is decoded; this enables dynamic reconfiguration of the accelerator or loading of new sets of data to accelerator buffers.

Table 7.1: Micro-ISA Opcode Set.

Opcode	Description
0x01	Configure TCONV (Sets configuration registers)
0x02	Loads Bias and Filter (Activates Weight Data Loader)
0x04	Load Input (Activates Dynamic Input Loader)
0x08	Schedule TCONV (Activates Scheduler)
0x10	Store Output (Activates Output Crossbar)

7.3.2 Scheduler

The *Scheduler* is the main control unit within the accelerator. Once it is activated, it orchestrates the execution of the entire TCONV layer. First it activates the *MM2IM Mapper* alongside the *Dynamic Input Loader*, and then it activates the array of the *Processing Modules* to execute the TCONV operation. Additionally, the *Scheduler* continuously monitors the *Instruction Decoder* for new instructions to either load the next row of input data to the *Row Buffer* or to send back the output data to main memory by activating the *Output Crossbar*.

7.3.3 Data Loaders

There are two data loaders, the *Weight Data Loader* and the *Dynamic Input Loader*. The *Weight Data Loader* loads batches of filter and bias data from main memory (via AXI-Stream) to their respective buffers. Once the batch is loaded into the buffers, the *Scheduler* allocates the filter and the corresponding bias data across the PMs. The *Dynamic Input Loader* loads new rows of inputs data dynamically to store within the *Row Buffer*, which at the request of the *Scheduler* broadcasts the new row of input data to all the PMs via dedicated FIFOs.

7.3.4 Output Crossbar

The *Output Crossbar* is a simple interface module which combines the output streams from each of the PMs, and at the request of the *Scheduler* sends the combined output data back to main memory.

7.3.5 Processing Module

Each processing module contains an accumulation and a compute unit connected via a FIFO stream. Figure 7.3 provides a detailed view of the PM architecture with a

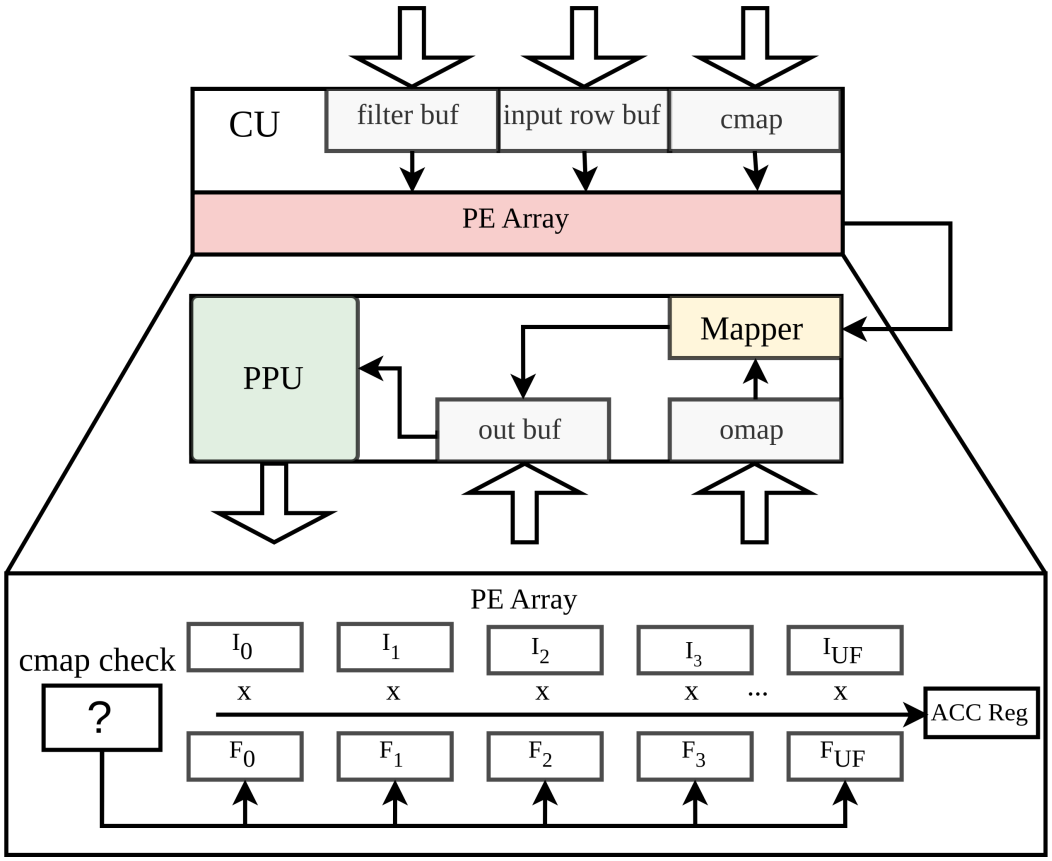


Figure 7.3: Processing Module Architecture with a detailed view of the PE array. The wide arrows represent data movement from and to the rest of the accelerator.

fine-grained view of the PE array; note that the wide arrows represent data movement between the rest of the accelerator and the PM.

During each TCONV layer, X filters are partitioned along the PMs. Once all the PMs load their respective filters, rows of input data are streamed to all the PMs. Additionally, the PMs receive the compute map (*cmap*) and output mapping (*omap*) from the *MM2IM Mapper* (described in Section 7.2.1).

Compute Unit: Due to the TCONV's complex computing nature, the Compute Units (CUs) require additional logic to ensure that only the required dot product is processed. This additional logic, the 'cmap check' within the *PE Array*, takes the *cmap*, the input row, and the filter data and computes the dot product of the *selected* input row and filter column. The computed partial results are stored in the accumulation register ('ACC Reg' in Figure 7.3) and then streamed into the accumulation unit for further processing. Note that the 'ACC Reg' is only used to temporarily hold a single accumulated value of the current dot-product calculation within the PE array.

Additionally, CUs are scalable and the unrolling factor (UF) defines the number of MACs within the PE array per CU. The UF is used to tile the I_c dimension of the

Algorithm 7: MM2IM Mapper

```
1  row_id ←load(row_id), row_width ←load(row_width)
2  foreach r in MM_rows do
3      h_pad = −padding_top + (S ∗ (row_id % row_width))
4      w_pad = −padding_left + (S ∗ (row_id ÷ row_width))
5      im_dex = h_pad ∗ O_w + w_pad
6      col = 0
7      foreach ih in Ks do
8          foreach iw in Ks do
9              if (ih + h_pad >= 0 && ih + h_pad < O_h
10              && iw + w_pad >= 0 && iw + w_pad < O_w) then
11                 PMs_cmap.broadcast_write(col)
12                 PMs_omap.broadcast_write(im_dex)
13             col++, im_dex++
14         im_dex+ = O_w − Ks
15     row_id++
```

given TCONV problem. Hence, to execute dot-product from I_c, the PE array will take I_c/UF number of cycles. Increasing UF will directly increase the number of MACs per cycle per CU while increasing the hardware resources required.

Accumulation Unit: The partial sums calculated by the CUs are stored within the *output buffers* in the correct output indices; the *Mapper* ensures this by using the *omap*. Subsequent partial sums for the same output accumulate with existing results, avoiding the need for extra buffer space. Once the output is fully calculated for an entire output row, the post-processing unit (PPU) processes the row. The PPU is a specialised processing engine used to perform the post-processing steps required by a given DNN model and then send the output data to the *Output Crossbar*. We have instantiated the PPU with the required stages to execute TFLite's re-quantisation scheme [TFL].

7.3.6 MM2IM Mapper

The *MM2IM Mapper* is a key component of our accelerator architecture, as shown in Algorithm 7. It generates the *cmap* and *omap* corresponding to the row of partial results (i.e., the output row of MM) and streams them to the PMs. The MM2IM Mapper takes the current row_{id} and the number of rows as parameters to ensure the cmap/omap are generated only for the required rows. This also allows the MM2IM Mapper to support partitioning/processing of the data in a tiled manner, since the row_{id} can be initialised to the starting row of the tile instead of the starting row of the output data. Overall, the MM2IM Mapper generates the compute and output

mappings only once per row, and each map is broadcast to all the PMs, thus saving hardware resources and additional computational overhead.

7.4 Evaluation

7.4.1 Experimental setup

To design, validate, and evaluate our MM2IM accelerator, we utilised the SECDA-TFLite toolkit to quickly design and integrate its architecture following the SECDA methodology. For the experiment, we opted to use the PYNQ-Z1 board, a suitable resource-constrained edge device with an FPGA, to map our accelerator All reported runs in this section have a standard deviation of less than 0.5%. For more details on the experimental hardware setup, refer back to Section 1.1.2.

To further elaborate on the implementation, we used SECDA-TFLite to develop a custom MM2IM delegate for TFLite; this custom delegate first marks any TCONV layers within the target TFLite model for offloading to our accelerator. Note that we fine-tuned our accelerator design specifically for TFLite's signed INT8 quantisation scheme. During inference, our MM2IM delegate would process the marked layers, offloading all TCONV-related metadata and pointers to the MM2IM driver code. Then, the host driver orchestrates the tiling strategy for accelerating TCONV as described in Algorithm 6, offloading the relevant input and weight data as required by the accelerator to calculate the output feature maps for the layer. As the accelerator finishes calculating each output feature map, the driver code generates the store opcode, which makes the accelerator send back the output data, which the driver then stores in the TFLite allocated memory space.

7.4.2 Synthetic benchmarks

First, we evaluated the performance of our MM2IM accelerator across varying sets of transposed convolution problems. Using the benchmarking suite available within SECDA-TFLite, we were able to generate TFLite models with a single-layer of TCONV, which we used to benchmark the MM2IM's performance across 261 different transposed convolution configurations.

We permuted the TCONV parameter with the following values: (i) $O_c = [16, 32, 64]$; (ii) $K_s = [3, 5, 7]$; (iii) $I_h = [7, 9, 11]$; (iv) $I_c = [32, 64, 128, 256]$; (v) $S = [1, 2]$. We discussed these parameters in Section 2.4.3.

Figure 7.4 shows the normalised speedup against dual-thread CPU execution (with NEON-vector instructions enabled) of the same problems within the PYNQ-Z1 board. We group similar problems for ease of visualisation of the results. The key take-aways from these experiments are the following: (i) On average, MM2IM achieves a 84× speedup against the dual-thread CPU; (ii) The accelerator's performance scales with I_c, I_h and K_s; (iii) Higher stride values result in lower speedup as expected due to less cropped outputs.

To highlight the impact of cropped outputs on the speedup, we generated Figure 7.5, which highlights the % of cropped outputs or the 'drop rate', for the various TCONV problems benchmarked within Figure 7.4. The drop rate is calculated as the ratio of cropped outputs to the total number of outputs. Looking at Figure 7.5, we can see that increasing K_s results in higher drop rates, while higher I_h and S result in lower drop rates. Comparing the drop rate to the speedup, we can conclude: (i) Increased kernel size results in higher drop rates and greater speedup; (ii) Increased stride results in lower drop rates and speedup, as expected; (iii) Decreased drop rate with increased I_h does not hamper the speedup. We theorise that this is due to the increased utilisation of the processing modules, as more computation is required for the larger input height (and width).

Additionally, we compared the performance improvement of MM2IM against other resource-constrained TCONV accelerators. The accelerator designs proposed by Liu et al. [Liu+18] and Zhang et al. [Zha+17] serve as good points of comparison, as they target similar resources-constrained devices. We compare the overall GOPs/DSP reported in both works (across synthetic workloads) to our performance. Overall, MM2IM outperforms the accelerators proposed by Liu et al. and Zhang et al. by 1.2× and 3.4× in GOPs/DSP, respectively.

7.4.3 End-to-end evaluation

To better understand the performance of our MM2IM accelerator, we evaluated the DCGAN model [RMC16] with end-to-end TFLite inference. We accelerate the TCONV layers and the post-layer quantisation using our MM2IM design. The rest of the layers are executed on the board's CPU. We achieve a latency improvement of 1.7× and an energy reduction of 1.4× . Note that since the DCGAN model contains different types of layers, the potential end-to-end performance improvement with our accelerator is limited to the TCONV layers within the model. For TCONV-only layers, we achieve a latency improvement of 2.3×.

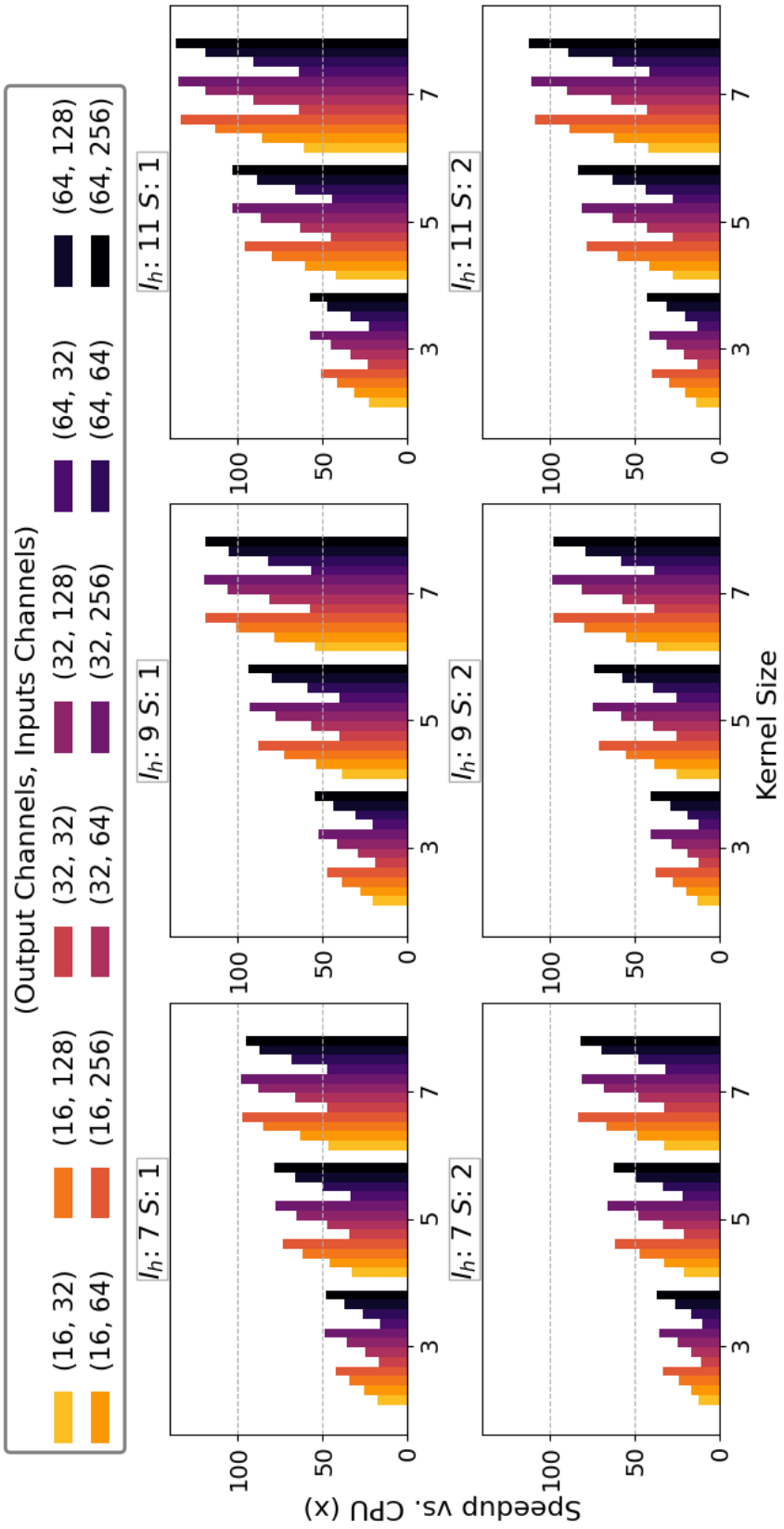


Figure 7.4: Speedup vs. CPU across various TCONV problems.

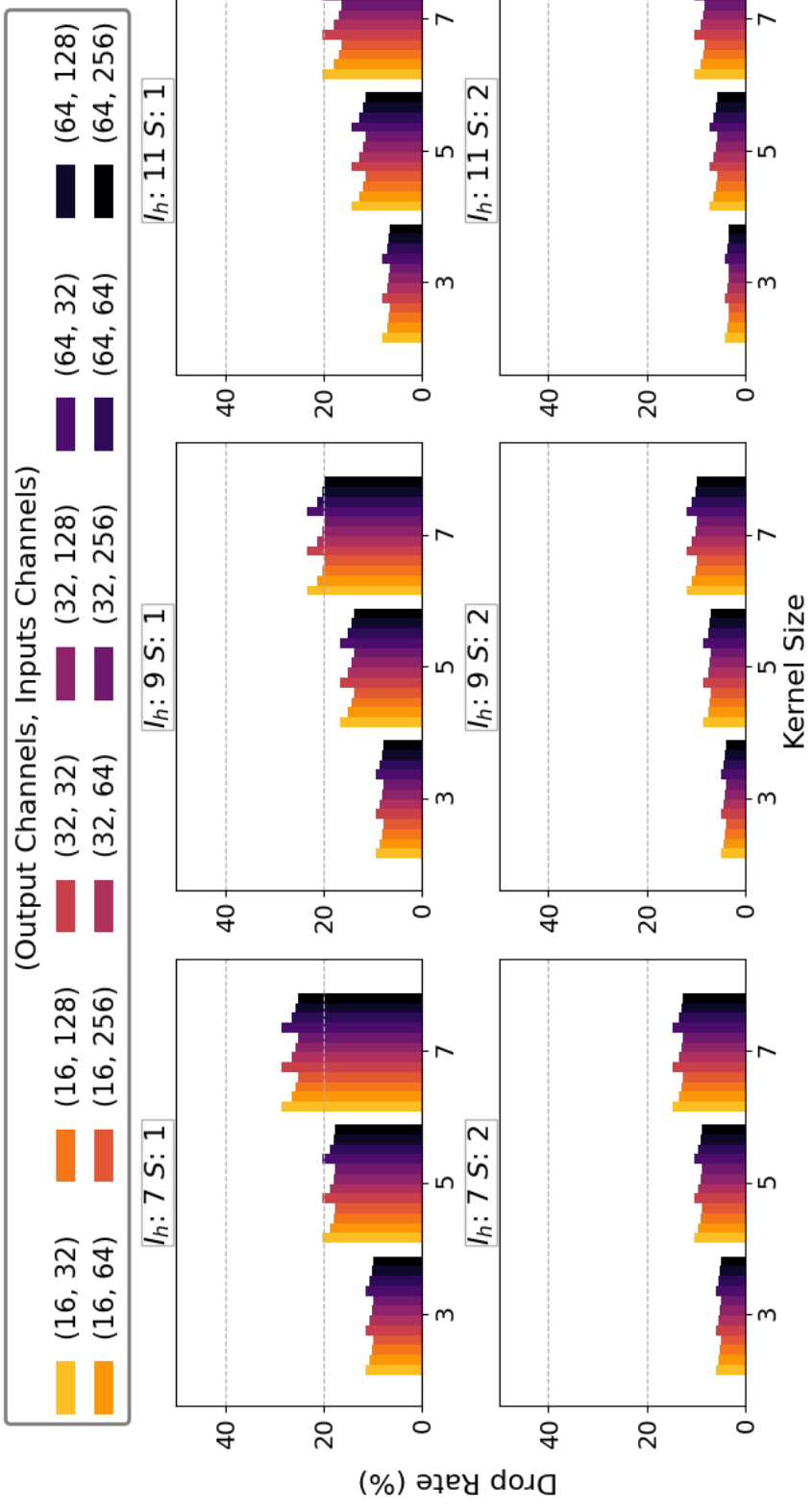


Figure 7.5: Highlights the % of cropped outputs for different the various TCONV problems benchmarked within Figure 7.4.

7.4.4 Discussion

As mentioned in Section 7.2.3, we used our performance model to estimate and guide the design choices of our MM2IM accelerator. To validate our performance model, we compare its expected performance to our accelerator's actual performance. On average, the model estimates the actual performance within 10% of our MM2IM accelerator. More importantly, applying the TCONV decoder optimisation to our performance model predicts the expected performance improvement within 1% deviation of the actual performance improvement that the optimisation provides. This demonstrates the utility of our performance model in guiding design choices through estimated performance improvements per proposed optimisations.

7.5 Summary

In this chapter, we proposed MM2IM, a novel accelerator architecture for accelerating TCONV operations. Our efficient hardware-software co-designed solution solves three key challenges: (i) the overlapping sum mapping problem; (ii) ineffectual computation and cropped output mapping; and (iii) the need for efficient dataflow strategies for resource-constrained edge devices.

We implemented our proposed hardware design on an edge FPGA using SECDA-TFLite. We evaluated the performance across a large variety of configurations for TCONV problems, achieving an average speedup of 84× against a dual-thread CPU. Furthermore, we also compared MM2IM against other resource-constrained TCONV accelerators and achieved 2.3× higher GOPs/DSP. Finally, we evaluated a full model (DCGAN) and achieved a 1.7× speedup and 1.4× energy reduction compared with the CPU.

8 | Automatic Host Code Generation for Specialised Accelerators

The third challenge presented in Section 1.2 is the need for efficient host-accelerator communication. Hence, our final objective is to develop a solution to enable efficient host-accelerator communication. In this chapter, we present AXI4MLIR, a novel code generation tool that automatically generates host-accelerator communication code for custom hardware accelerators. This work was conducted in close collaboration with colleagues at Northeastern University, Boston, USA; and the Pacific Northwest National Laboratory, USA. For completeness, we provide a detailed look at the entirety of the work conducted, not just the parts that were directly contributed by myself. The following sections are attributed as my main contributions to the work:

- Section 8.2.1, the development of the AXI DMA Runtime Library for AXI4MLIR.
- Section 8.2.2, the defining architectural features of the supported accelerators within AXI4MLIR and integration with existing ARM platforms.
- In Section 8.3, the experiments were performed in joint efforts, with my focus on developing the hardware/software infrastructure and running the experiments; this included developing the baseline experiments, designing custom accelerators for the matrix multiplication experiments (Section 8.3.2 and Section 8.3.3) and convolutional experiments (Section 8.3.4), whereas my colleagues focused on the analysis and discussion of the results within these sections.

The chapter is structured as follows: Section 8.1 introduces and motivates the need for AXI4MLIR. Section 8.2 presents AXI4MLIR in detail, highlighting the key features and contributions of the tool. Section 8.3 highlights the experiments we conducted to evaluate the AXI4MLIR tool. Finally, Section 8.4 summarises the chapter.

8.1 Introduction

We can define key operations in DNNs as tensor algebra operations. These operations are also widely used in scientific computing and data analytics [AK23; Jum+21; RF18]. Hence, the work presented in this chapter can be applied to a wide range of applications.

Tensor operations tend to be computationally intensive and require high memory bandwidth, making them suitable for specialised hardware implementations. Automated tools have been proposed [Kwo+20; Xu+20; Ye+20; Zha+20a] to help explore new classes of custom domain-specific accelerators targeting tensor computations, and are currently the best path available to obtain performance gains in scientific workloads and machine learning applications.

However, designing and fully exploiting custom hardware accelerators for tensor operations is a challenging task [Gib+25]. When co-designing these devices, we need to generate efficient architectures, and we must optimise the communication between the host CPU and the accelerator. In particular, the host-accelerator interaction involves several aspects, including data transfers, synchronisation, and the accelerator's control flow. These aspects depend on the characteristics of the host CPU microarchitecture, the host-accelerator interface, the accelerator design, and the application code. Manually rewriting the host driver code for each accelerator and application scenario can be very tedious and error-prone. Furthermore, most of the prior work proposing new accelerators [Ago+20; Che+19; Mor+19; Nga+20; Ska+18] only considers a simple offload model or assumes that the required data is already placed in the accelerator's internal buffers, falling short in providing insights into how host-to-accelerator transfers should be performed or generated. Additionally, complex accelerators, exemplified by Google's TPUs and Nvidia's GPUs, benefit from large teams that can collaboratively engineer dedicated compilers to address some of these issues. However, smaller development teams may lack expertise or available time resources to invest in compilers. Consequently, custom accelerator designers typically manually implement driver code and instruction streams to validate and deploy their designs for a subset of synthetic workloads.

We argue that all major features of a system-on-chip (SoC) must be considered to implement or generate efficient host-to-accelerator communication. Figure 8.1, a high-level version of the system model presented in Section 1.1.2, highlights a typical system and the core attributes that should be considered when generating efficient host-accelerator communication code. This includes the AXI [Deva] interconnection between the CPU and a custom accelerator, which is a common choice in many designs [Liu+18]. The host-code implementation should exploit CPU, interconnect, and accelerator features

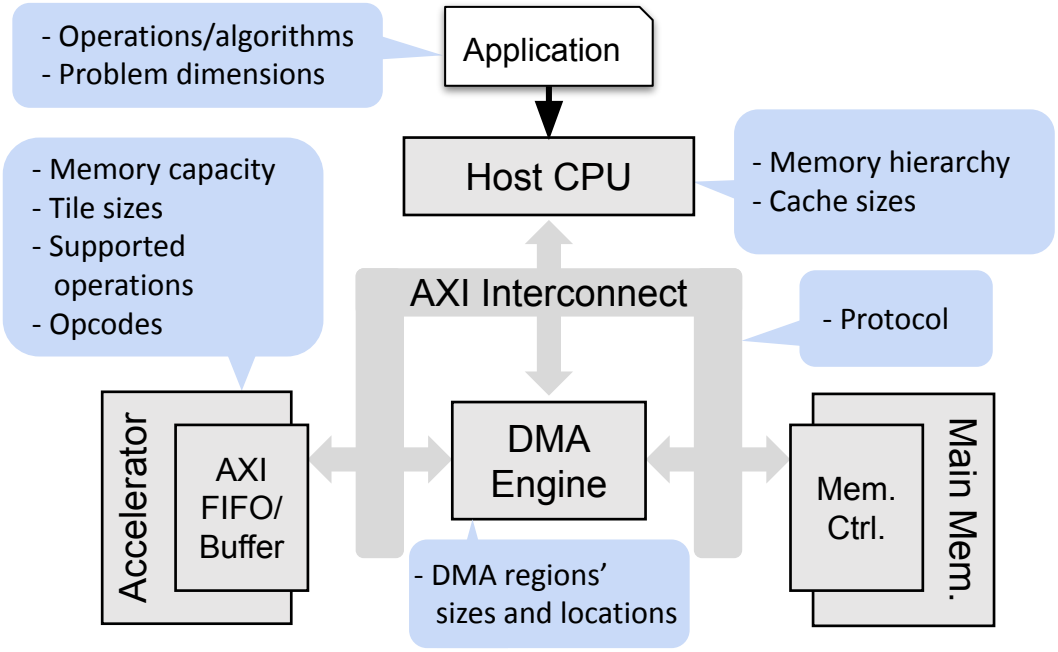


Figure 8.1: Typical host-accelerator system design, highlighting (blue colour) relevant parameters that should be considered for efficient generation of host-accelerator communication code.

(see Figure 8.1) to maximise accelerator performance.

To effectively consider each of the key system features described in Figure 8.1 while also delivering efficient and automated CPU-accelerator driver code generation, we propose **AXI4MLIR**, an extension to the MLIR compiler framework [Lat+21] that enables efficient and automated CPU-accelerator driver code generation for accelerators targeting linear algebra applications. AXI4MLIR takes a high-level application description in the MLIR's linear algebra (`linalg`) abstraction [Dev20] as input and introduces custom MLIR attributes to describe the target accelerator capabilities. These attributes provide accelerator-specific information to custom transformation passes that can effectively specialise and generate accelerator-aware host driver code. Our extensions facilitate hardware-software co-design by allowing developers to automatically generate driver code with varying configurations, more easily explore their design space, and use the designed accelerator in applications that can be compiled with the MLIR framework. While leveraging the new attributes of AXI4MLIR, the compiler entirely automates our user-directed host code generation. This provides a significant advantage in terms of productivity and maintainability.

The contributions of this chapter are as follows:

- We present AXI4MLIR, an extension to the MLIR compiler framework that enables the generation of efficient host code for custom accelerators targeting tensor

algebra operations.

- New MLIR attributes that provide a standardised and extensible approach to represent accelerators that can implement a range of linear algebra algorithms supported by the MLIR *linalg* abstraction.
- The ability to describe and explore accelerator-specific tiling and dataflow strategies for the target linear algebra operation, which can improve computation efficiency within the accelerator and reduce data movement overheads between the accelerator and CPU.
- An analysis of our compiler optimisations on a suite of benchmarks representing key linear algebra applications, demonstrating the effectiveness of our approach in achieving significant performance gains (up to 1.65× speedup and 56% fewer cache references) when compared to optimised manual driver code implementations.

8.2 AXI4MLIR

To support efficient host code generation for AXI-based custom accelerators, we extended the MLIR compiler framework with the added capabilities presented in Figure 8.2. AXI4MLIR can be used as part of the design loop while following the SECDA methodology, however for the sake of clarity and since our focus in this chapter is on the automated host code generation aspect, we assume that the custom accelerator has already been designed and validated. After the custom accelerator is designed and the host CPU system is selected, the user creates a configuration file with the host CPU system details (e.g., number and size of the caches) and a high-level description of the accelerator capabilities (i.e., supported operations and dimensions), the available opcodes (simple instructions), and possible opcode flows ① (sequence of opcodes). This information is parsed ② by the compiler and used to find ③ suitable `linalg.generic` operations with the desired operation traits (algorithm implemented, previously shown in Figure 2.2a-L1 to L9), that can be executed on the accelerator. These operations will require host-accelerator driver code generation. Subsequently, with user-provided information on the total size of the CPU caches, the compiler transforms the code to efficiently exploit the CPU memory hierarchy and the accelerator size ④, performing the appropriate set of tiling transformations to leverage temporal locality in the CPU caches and to map the problem on the accelerator. In the final step, the compiler generates the runtime calls ⑤ that leverage the accelerator features based on user-directed

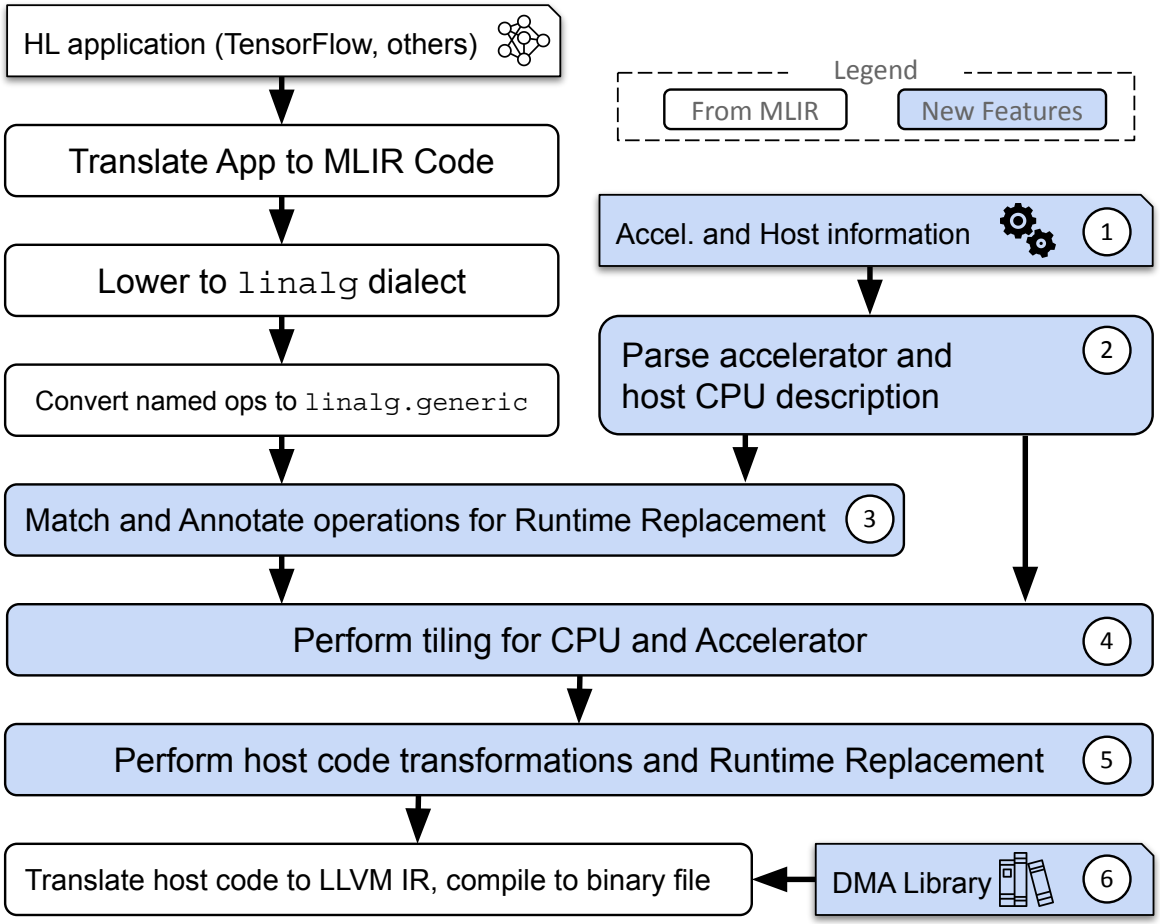


Figure 8.2: AXI4MLIR Compiler Flow. The numbered elements are the contributions of this work.

dataflow description (e.g., avoiding redundant host-accelerator data transfers when the algorithm and accelerator functionality permits).

The following sections discuss the class of supported accelerators and the key features of our AXI4MLIR DMA library. We provide details on how to describe new accelerators within AXI4MLIR, introduce `linalg.generic` trait extensions, describe our new MLIR dialect that provides support for runtime call replacement of opcodes and data transfers, and finally present some key optimisations that can be performed (depending on the available features of the host system and the custom accelerator).

8.2.1 The Custom AXI DMA Library

The AXI4MLIR *DMA library* ⑥ (Figure 8.2) exposes low-level DMA calls working at a privileged level to enable data movement between the main memory and the accelerator.

We designed this library to be lightweight (55 bytes in size for our target ARM SoC)

so that it can be deployed on both resource-constrained and non-constrained systems; bare-metal systems can also execute it. During the compilation process, the AXI runtime issues calls to initialise the DMA engine(s) before entering the workload's computation kernel. First, a library call initialises the DMA engine, mapping memory for the input and output buffers, which act as temporary staging buffers between the CPU and the accelerator.

After DMA initialisation, the accelerator is accessible via AXI-based data transfers. Any data that needs to be transferred to the accelerator during workload execution is first copied to a DMA input buffer. This staging copy acts as a packing optimisation (similar to [Sal+23]), contributing to an increased cache-hit ratio during communication. Then, the AXI "send" function call requests the DMA engine to start the data transfer and waits for it to finish. Note that the data sent to the accelerator can be either accelerator instructions or raw input data that must be processed. Similarly, AXI4MLIR generates 'recv' function calls to wait for computation completion and to obtain output data from the DMA output buffer.

In Section 8.2.3, Figure 8.7 presents the lowering of different high-level operations into our DMA library calls. Function `copy_to_dma_region(...)` implements data movement from a `memref` to the DMA-accessible memory region intended for transmission to the accelerator. The `offset` argument allows for efficient batching of different data transfers after computing the total length and executing a single 'send' operation. Appropriate offset values prevent overwriting existing data in the DMA region. Function `dma_start_send(...)` instructs the DMA engine to transmit a `size` of X bytes to the connected accelerator, commencing from a specified `offset` within the DMA space allocated. Function `dma_wait_send_completion(...)` instructs the CPU to wait for the DMA's signal informing the transaction's completion. When receiving data from the accelerator, the CPU waits for the data to be placed in the DMA-accessible memory to later copy it into a `memref`.

8.2.2 Supported Accelerators

In matrix multiplication and similar algorithms, the term *stationary* refers to a slice of data that can be reused across many iterations of an algorithm's computation. A *stationary* strategy attempts to maximise data reuse and minimise data movement, which can significantly benefit accelerators that require efficient memory accesses. We want to enable the programmer to easily control accelerators that support *stationary* flows.

Next, we discuss the types of accelerators that AXI4MLIR can support. Then, we

propose a standardised approach to concisely define the class of supported accelerators in a configuration file.

Finally, we show how the AXI4MLIR parser can take user-defined configurations, extract essential attributes of the target accelerator, and populate a trait specification to guide our MLIR compiler transformations.

Accelerator Designs

The AXI4MLIR compiler transformations support linear algebra kernels implemented as accelerators using the AXI interconnect. In addition, the AXI-S data transfers within AXI4MLIR facilitate support for accelerators that use a micro-ISA (Instruction Set Architecture) with opcodes, which consist of instructions that the host CPU sends to the accelerator. Generally, the following three actions are used to categorise the actions within an instruction: *send*, *compute*, and *receive*. Issuing a combination of these three actions can complete any accelerator's instructions that require external communication (e.g., data transfers, or activation/reset/configuration of the accelerator compute modules). In addition, each action can have additional meta-data (e.g., opcode literal, data, length, dimensions, and indexes), which are used to guide compiler transformations during accelerator host code generation. Further, specific traits of the accelerator, such as internal buffer space (or accelerator tile sizes) and data types, are supported and must be defined within the accelerator configuration file.

Accelerator Configuration File

```
{"cpu" = { "cache-levels": [32K,512K],
           "cache-types": [data,shared] }
 "accelerators" = [
   { "name": ..., "version": x.x, "description": ...,
     "dma_config" : {...}, "kernel": "linalg.matmul",
     "accel_size": [4,4,4], "data_type": int32,
     "dims": ["m", "n", "k"],
     "data": { "A": [m,k], "B": [k,n], "C": [m,n]},
     "opcode_map" : "<opcode_map string - see S8.3.4>",
     "opcode_flow_map" : { "flowID01" :
           "<opcode_flow string - see S8.3.4>", ...},
     "selected_flow" : "flowID01" }]}
```

Figure 8.3: Accelerator and CPU configuration file.

Once an AXI-based accelerator is fully designed, the accelerator developer can quickly integrate it with our AXI4MLIR compiler transformations by providing *Accelerator and*

Host information ① (Figure 8.2) through a configuration file for the new accelerator and the target host system. Figure 8.3 shows a sample configuration file defined in the standard JSON format. The developer must specify the accelerator's architectural features, such as supported tile sizes, data type, and input and output data with related dimensions. Additionally, the developer should describe any micro-ISA that the accelerator can execute. The developer should define 'opcode IDs', captured by the 'opcode_map string', which are comprised of actions to describe the memory operations and related data transfers. Finally, the developer should define the possible "opcode flow IDs" and select the desired flow for the particular operation. The configuration file does not capture the internal behaviour of the accelerator, which has been the focus of other works [Che+19; Xu+20]; instead, we seek to optimise the communication with the accelerator. Thus, the configuration file contains information about the I/O interface for sending data and instructions to the accelerator. Similar to the accelerator information, the CPU information, shown in Figure 8.3-L1 to L2, needs to contain basic architectural details such as the number and size of caches.

Configuration Parsing

The parser implemented in ② (Figure 8.2) is responsible for providing the information from the configuration file to the MLIR IR and the AXI4MLIR transformation passes. To this end, the *kernel* and *cache information*, paired with a simple heuristic that identifies the dimensions of the target MLIR operation, are used to schedule tiling transformations (Figure 8.2 - ④) that leverage the CPU memory hierarchy sizes and increase temporal locality of the memory accesses.

Additionally, the parser validates the `opcode_map` and the user selected `opcode_flow`, which are then translated into new MLIR attributes to the target `linalg.generic` operation trait. Their syntax and functionality are described in Section 8.2.3.

Supported Systems

Our work is focused on SoCs with accelerators connected to ARM CPUs via an AXI-S interconnect. AXI4MLIR seamlessly integrates with a diverse set of Xilinx platforms, although we also anticipate similar applicability to other FPGA-SoC devices. Changing the cross-compiler would allow support for other processors. Adapting our DMA library implementation to other standards would be required to support other types of interconnects. AXI4MLIR currently supports AXI-Stream accelerators, which do not communicate via direct memory requests. Thus, AXI4MLIR does not require support for host-accelerator coherence protocols since the host manages the DMA engine

transfers.

8.2.3 MLIR Extensions and Optimisations

To implement *match and annotate operations for runtime replacement* ③ (Figure 8.2), and to offload the computation onto the accelerator, we implemented passes to identify the target algorithms supported by the accelerator and extended the `linalg.generic` operation `trait` with additional information, as shown in Figure 8.4a. In particular, we introduced two new types of attributes to MLIR, `opcode_map` and `opcode_flow`, which follow the syntax described in Figure 8.5 and Figure 8.6, respectively. We elaborate more on each attribute in the operation `trait` below.

Extensions to linalg.generic Traits

- `dma_init_config`: defines the parameter values used to configure a DMA engine associated with a specific accelerator. If multiple or different accelerators are present, they would have different values in this field. Figure 8.4a-L2 to L4 shows the available parameters. The code generated for the DMA initialisation is executed by the CPU only once per application.

- `init_opcodes`: defines a flow of opcodes that should be sent to initialise or reset the accelerator for a new kernel execution. During application runtime, these opcodes are sent N times, where N is the number of kernels in an application that can be mapped onto the custom accelerator. In Figure 8.4a-L7, we define that the reset opcode must be included to support the described accelerator. The opcode's functionality is derived from the `opcode_map` parameter below.

- `accel_dim`: defines the size of the accelerator for each dimension of the implemented algorithm. Figure 8.4a-L9 shows an example, specifying that the *accelerator* supports a tiled $\text{MatMul}_{4\times4\times4}$ version of the implemented algorithm.

- `permutation_map`: defines the order in which nested loops execute. In Figure 8.4a-L12, we switch the order of the two innermost loops, potentially enabling the data structure that uses *[m,k]* indices to be stationary, as the other data structures are streamed in/out of the accelerator. In our MatMul example (Figure 2.2b presented in Section 2.2.3), this enables an `A` stationary dataflow (Figure 8.4b).

- `opcode_map`: describes accelerator opcodes as key-value pairs. Following the syntax scheme shown in Figure 8.5, the key or *opcode_entry*, is an identifier that maps to a list of actions, or *opcode_list*, which represents sequential memory operations that have to be performed to drive the accelerator.

```
#matmul_accel_trait = {
  dma_init_config = {         id = 0x0,
    inputAddress = 0x42,      inputBufferSize = 0xFF00,
    outputAddress = 0xFF42,   outputBufferSize = 0xFF00 },

  // Opcodes sent once. Tokens defined in opcode_map.
  init_opcodes = init_opcodes < (reset) >,

  accel_dim = map<(m, n, k) -> (4, 4, 4)>, // Tiling

  // Permutation and who can be stationary.
  permutation_map = affine_map<(m, n, k) -> (m, k, n)>,

  opcode_map = opcode_map < // Valid Opcodes
    sA        = [send_literal(0x22), send(0)],
    sB        = [send_literal(0x23), send(1)],
    cC        = [send_literal(0xF0)],
    rC        = [send_literal(0x24), recv(2)],
    sBcCrC    = [send_literal(0x25), send(1), recv(2)],
    reset     = [send_literal(0xFF)] >,

  // Flow to implement. Tokens defined in opcode_map.
  opcode_flow = opcode_flow < (sA (sBcCrC)) > // As
  // Example of other      < ((sA sB cC) rC) > // Cs
  //       valid flows     < (sB sA cC rC) >   // Ns
}
```

(a) New Attributes for Accelerator Description.

```
func.func @matmul_call(...) {
  // Declare constants (loop bounds and literals): %cX, ...
  accel.dma_init(%c0,%c66,%c65280,%c65346,%c65280) : ...
  accel.sendLiteral(%c0xFF, %c0) : i32,i32->i32 // reset
  // Tiling by 4,4,4
  scf.for %m = %c0 to %c60 step %c4 { // first loop
    scf.for %k = %c0 to %c80 step %c4 { // second loop
      %sA = memref.subview %A[%m, %k] [4, 4] [1, 1] : ...
      %offset0 = accel.sendLiteral(%c0x22,%c0):i32,i32->i32
      accel.send(%sA, %offset0) : !mr4x4_0, i32 -> i32
      scf.for %n = %c0 to %c72 step %c4 { // innermost
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1] :
        %sC = memref.subview %C[%m, %n] [4, 4] [1, 1] :
        %offset1 = accel.sendLiteral(%0x25,%c0): ...
        %offset2 = accel.send(%sB, %offset1) :
                              !mr4x4_0, i32 -> i32
        accel.recv {mode="accumulate"}(%sC, %c0) :
                              !mr4x4_0, i32 -> i32
  } } } } } }
  return }
```

(b) IR to drive the MatMul accelerator with an A-stationary flow.

Figure 8.4: Information added to the linalg.generic traits to capture accelerator behaviour in MLIR and IR with accel operations.

```
opcode_dict   ::=
    `opcode_map` `<` opcode_entry (`,` opcode_entry)* `>`
opcode_entry ::= (bare_id | string_literal) `=` opcode_list
opcode_list   ::= `[` opcode_expr (`,` opcode_expr)* `]`
opcode_expr   ::= `send` `(` bare_id `)`
                | `send_literal` `(` integer_literal `)`
                | `send_dim` `(` bare_id `)`
                | `send_idx` `(` bare_id `)`
                | `recv` `(` bare_id `)`
```

Figure 8.5: Opcode Map Syntax. A dictionary for accelerator opcodes and actions.

```
opcode_flow_entry ::= `opcode_flow` `<` flow_expr >
flow_expr ::= `(` flow_expr `)` | bare_id (` ` bare_id)*
```

Figure 8.6: Opcode Flow Syntax. The sequence of opcodes to implement a specific dataflow of host-accelerator communication.

Each action, or *opcode_expr* (`send`, `send_literal`, `send_dim`, `send_idx`, `recv`), implements different types of copies to/from the DMA memory-mapped region. The `send` and `recv` actions take an input. The input is a number that is used to represent one of the arguments to the `linalg.generic` operation; e.g., 0, 1, or 2 would map to A, B, or C, respectively in the MatMul example (Figure 2.2a-L12-13). During code generation, this information is used to copy the needed tile to the memory-mapped region. For example, Figure 8.4a-L15 shows an opcode with identifier *"sA"* that issues copies *to* the accelerator for the *literal* value `0x22` and then for the *data* associated with the tile of argument 0. Furthermore, `send_dim` and `send_idx` can be used to send tile dimensions or tile indices, which could be used to drive more complex accelerators. Subsequent text will refer to an *opcode_entry*, such as *"sA"*, simply as *opcode.*

- `opcode_flow`: represents valid opcode/data transfer *flows* and respects the syntax scheme shown in Figure 8.6. Figure 8.4a-L23 shows an example, which defines an *input A stationary* (associated with argument 0) valid flow implemented with two opcodes, using the identifiers defined in the `opcode_map`. Additional valid examples for *output C stationary* and *nothing stationary* flows are shown in lines 24 and 25 of Figure 8.4a. The information in `opcode_flow` is parsed, and the set of parentheses is understood as a proxy to specify multiple scopes for sequential or nested *for* loops in the algorithm. Following this flow, logic related to *"sA"* would be transmitted inside of the second loop (Figure 8.4b-L8 to L10), and logic related to *"sBcCrC"* would appear in the innermost loop (Figure 8.4b-L12 to L18).

Suppose the user decides to forego the opportunity to specify *input A* as stationary, then the opcode flow could become *"(sA sB cC rC)"*, and all communication driver

Figure 8.7: Semantics and lowering of `accel` dialect operations.

logic would be generated in the innermost loop.

The `accel` Dialect

Before generating function calls for *runtime replacement* to the DMA runtime library (described in Section 8.2.1), we perform *host code transformations* ⑤ (Figure 8.2) by lowering the `linalg.generic` operation, with the proposed `trait`, to standard MLIR dialects (`scf`, `arith`, `memref`) and a new dialect that we call `accel`. The operations in the `accel` dialect abstract away host-accelerator transactions, such as initialisation, memory transfers, and synchronisation.

Figure 8.7 presents the core `accel` operations and their semantics, providing examples of how these operations map onto our custom AXI DMA library calls. Additionally, Figure 8.4b shows how the `accel` operations are used in our MatMul example.

Note that it is easier to perform analysis and transformations of operations when they are expressed in our `accel` dialect, as opposed to using a lower-level abstrac-

tion. With lower-level abstractions such as `llvm`, function calls and additional logic have already been exposed: additional instructions must be present in the IR to implement buffer slicing, size/offset calculations, and function calls to copy data to/from the DMA regions. Performing analysis and transformations in the `llvm` abstraction is more challenging, as traversal of control flow blocks and LLVM instructions are necessary. Instead, operations in the intermediate `accel` dialect encode the relevant information, and are easily relocated during transformation passes, respecting dependencies without requiring complex compiler analysis. This approach facilitates implementing communication flows that consider one of the data structures to be stationary by simply hoisting the `accel` operations up to the right loop nest level, while considering the flow patterns. Finally, the `accel` dialect provides an intermediate step before runtime call replacement. In AXI4MLIR we target our AXI DMA runtime library described in Section 8.2.1, but further extensions could implement the transformation of `accel` operations into other runtime libraries such as OpenCL [SGS10] or SYCL [Rey+20], which are commonly used to interface with SoC FPGA accelerators.

8.3 Experiments and Results

To evaluate AXI4MLIR, we continue to use the PYNQ-Z1 [Dig] board with the Zynq-7000 SoC. For more details on the experimental hardware setup, refer back to Section 1.1.2. We also use a library of tile-based accelerators developed using SECDA-TFLite implemented with the AXI-S interface and opcodes with a micro-ISA. For workloads, we target a suite of kernels covering a range of dimensions, as well as an end-to-end machine learning application.

We leverage hand-written baselines, which we discuss in Section 8.3.1. Section 8.3.2 evaluates accelerators implementing MatMul, comparing inference performance against a hand-written baseline, identifying potential bottlenecks, and showcasing the benefits of our optimised dataflows. Section 8.3.3 highlights the value of AXI4MLIR by demonstrating how to handle accelerators with configurable parameters such as tile sizes and dataflows. We showcase how to use AXI4MLIR with a convolution-based accelerator in Section 8.3.4. Finally, Section 8.3.5 shows how AXI4MLIR can work in the context of a complete application, evaluating the TinyBERT model [Jia+20].

8.3.1 Hand-written Baselines

The following experiments employ hand-written optimised driver code developed and tested using the SECDA-TFLite following the SECDA methodology to establish per-

Table 8.1: Description of the MatMul accelerators used in the experiments. Synthesised with Xilinx Vivado at 200MHz.

Type	Possible Reuse	Opcode(s)	Configurations (Size, OPs/Cycle)
$v1_{size}$	Nothing	sAsBcCrC	(4, 10)
$v2_{size}$	Inputs	sA, sB, cCrC	(8, 60)
$v3_{size}$	Ins/Out	sA, sB, cC, rC	(16, 112)
$v4_{size}$	Ins/Out (flex size)	sA, sB, cC, rC	

formance baselines. With host-driver code written in C++, these manual baselines will be labelled as *cpp_MANUAL*. All baselines are implemented with various tiling strategies, with no additional data transfer overheads and with the fewest number of data transfer calls for the selected dataflow.

8.3.2 Matrix-Multiplication Experiments

The tile-based accelerators used here resemble vector MAC engines [Alb+16; Che+14b; Zha+15; Zha+16a] implementing MatMul algorithms. They vary in input/output buffer size and supported dataflow. From the CPU-host perspective, some of them can support varying degrees of data reuse when the appropriate opcode stream drives the accelerator. Table 8.1 presents a short summary of their functionality, where *size* stands for the supported tile size of the accelerator. For example, $v1_4$ is a $MatMul_{4x4x4}$ accelerator that does not support data reuse and only supports $tM, tN, tK == 4, 4, 4$ tiles. For $v1_4$, AXI4MLIR will tile the algorithm's loops in the host code, taking into account the accelerator size of 4 and all the data movement will happen in the innermost loop - '`opcode_flow <(sA sB cCrC)>`'. For $v2_8$, AXI4MLIR will tile the computation by a factor of 8 and generate code to maximise the reuse of one of the inputs. In v2, a stationary (As) is implemented with `opcode_flow <(sA (sB cCrC))>`.

Accelerators v3 and v4 can also reuse their output data structures. Accelerator v4, marked with *flex size*, supports computations of non-square tiles, i.e., $v4_{16}$ can process a MatMul of $tM, tN, tK = 32, 16, 64$, as long as tM, tN, tK are divisible by 16 and fit in the accelerator's memory.

All accelerators were implemented using HLS pipelining and unrolling to maximise the number of internal processing elements instantiated and their arithmetic throughput. The last column of Table 8.1 reports throughput (OPs/cycle) for each accelerator, highlighting that many arithmetic operations are executed in parallel at each cycle. Different types of accelerators with the same size have the same throughput, and accelerators with bigger sizes provide higher throughput. All bar graphs presented in this section represent the average of 5 independent runs with the same configuration.

Accelerator Relevance

In order to evaluate the performance of the accelerators defined in Table 8.1, we conducted experiments to compare the runtime of the CPU execution (*mlir_CPU*) against the manual C++ implementation (referred to as *cpp* for short) of the driver code using the accelerators. The task clock was used as a metric to measure the execution time of the benchmarks. We present the results of the experiments in Figure 8.8, which plots the task clock on the y-axis (smaller is better) and only includes the "Nothing Stationary flow", which means that the data transfers happen in the innermost loop.

Looking at Figure 8.8, we can see that the accelerator offload only becomes relevant (i.e., executes faster than the CPU) for problems with $dims \geq 64$, where $dims = M = N = K$. For problems with smaller dimensions, CPU execution will be faster than the accelerator. In addition, the results in Figure 8.8 suggest that accelerators only become relevant if $accel\_size = tM = tN = tK \geq 8$. For smaller accelerator sizes, the CPU execution is faster than the accelerator.

These observations suggest that the performance benefits of using accelerators are limited to a range of problem sizes and accelerator sizes. Therefore, it is essential to carefully choose the appropriate accelerator configuration for a given problem to achieve the best performance. Consequently, for the next experiments we will limit our focus to problems with $dims \geq 64$ and accelerators with $accel\_size \geq 8$.

AXI4MLIR Generated vs. Manual Implementation

AXI4MLIR provides several benefits. First, our passes automatically tile data mapped to the CPU memory hierarchy, leveraging spatial and temporal locality. The second benefit is the ability to automatically generate specific flows, such as the Nothing Stationary (Ns) flow, which can be tedious and error-prone when done manually. Additionally, AXI4MLIR provides an efficient path to flow strategies that can potentially improve performance, such as input A or B stationary (As, Bs) flows. Figure 8.9 presents these results.

First, we compare the differences in execution time between a *manual implementation* (see Section 8.3.1) of an Ns flow strategy and an *AXI4MLIR generated* Ns flow strategy, represented by the first two bars in each group of bars in Figure 8.9. The remaining bars in each group of bars show results for automatically generated flow strategies, with As and Bs for v2 accelerators and As, Bs, and Cs for v3 accelerators. Looking at Figure 8.9, we see that some flows, especially Cs, provide improvements. To achieve this, the user simply has to encode the information for Cs (or other flows) during compilation. For

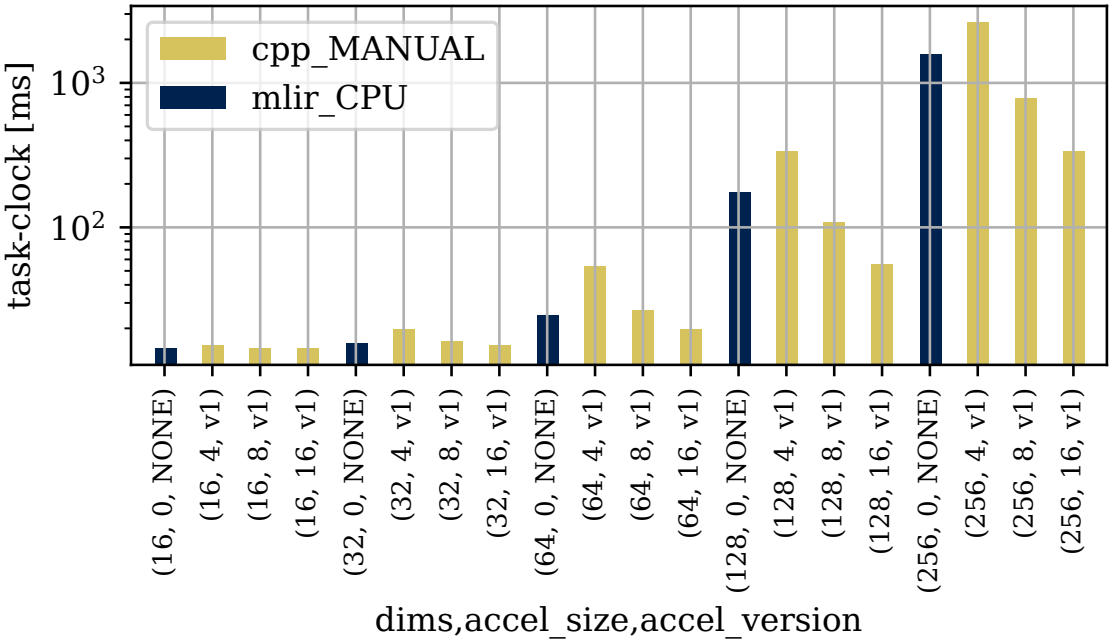


Figure 8.8: Runtime characterisation CPU vs. Accelerator execution for Matrix Multiplication problems. Note how an accelerator only becomes relevant for problems with $dims \geq 64$ and $accel\_size \geq 8$.

example, we can encode Cs using the opcode_flow previously presented in Figure 8.4a-L25 in the operation's trait.

Next, in Figure 8.9, we focus on the results with the v3 accelerator. Here, we see that AXI4MLIR generated Cs performs better than the manually generated Ns, although the other flows are not performing as expected. First, we would expect the performance of AXI4MLIR generated Ns to have similar/closer task clock performance than manual Ns. And second, we would also expect As and Bs flows to always outperform Ns due to the degree of reuse, as they copy less data and can keep the accelerator better utilised. Hence, this first implementation has room for improvement and, in the following experiment, we *identified and fixed the bottlenecks* by analysing performance counters and implementing optimisations that specialise memory copies.

Identifying Bottlenecks and Improving AXI4MLIR Codegen

Next, we identify performance bottlenecks in AXI4MLIR-generated copies and improve upon them to enhance the performance of the workloads when using the custom hardware accelerators. Specifically, the experiment compares the performance of manually implemented host-accelerator driver code with AXI4MLIR-generated code for Ns, As, Bs, and Cs flows in terms of branch instructions, cache reference counters, and the task

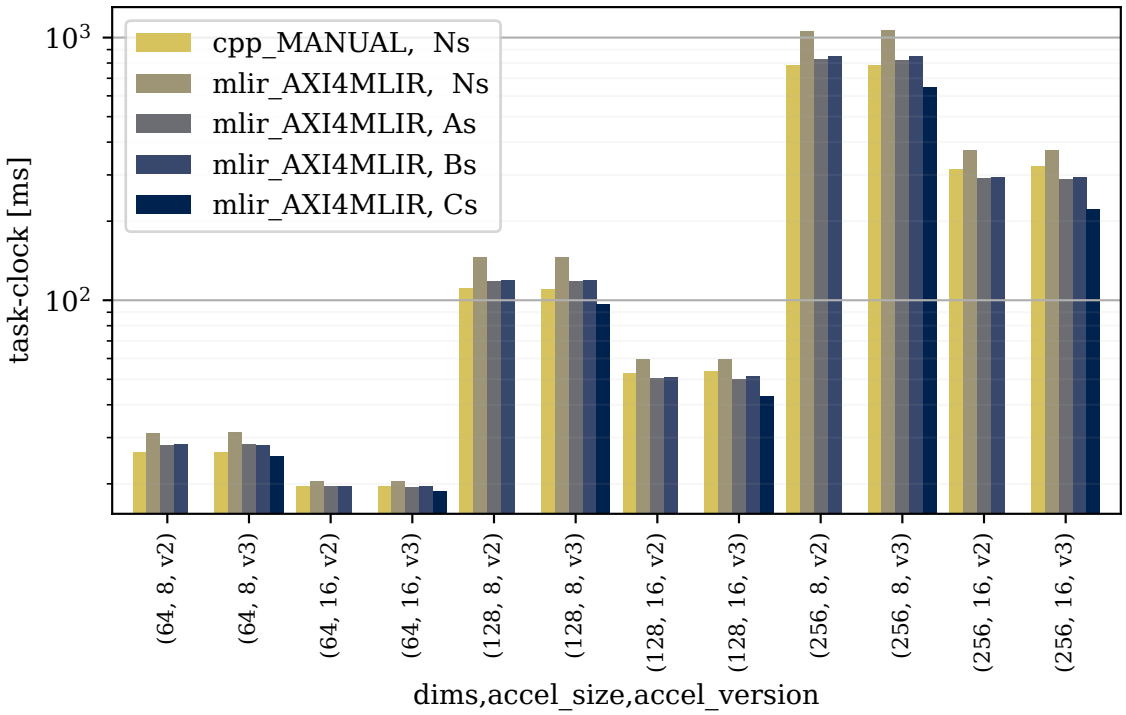
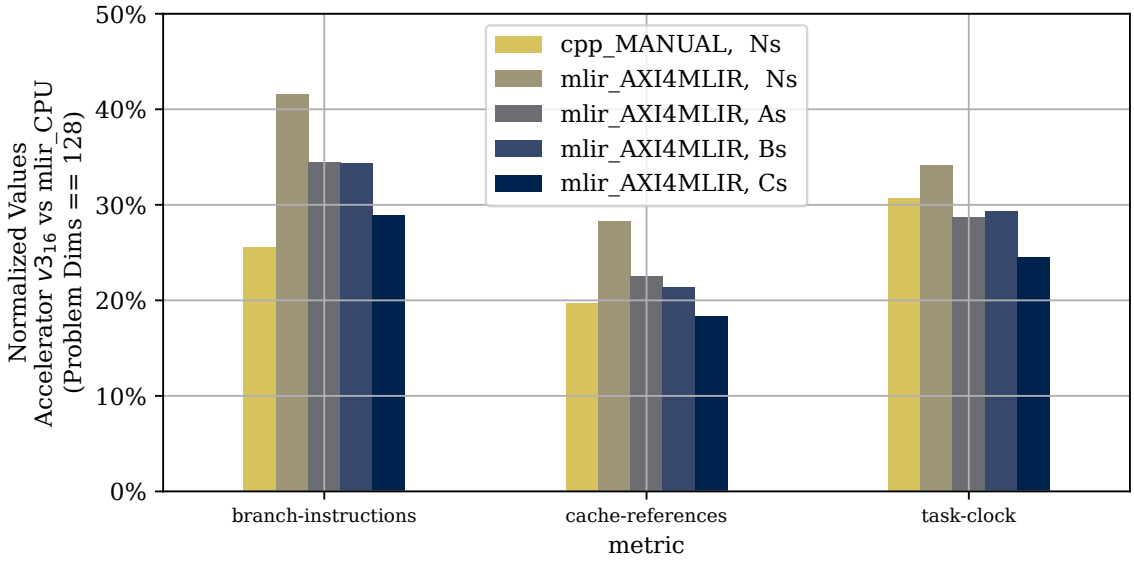


Figure 8.9: Runtime results on Matrix Multiplication kernels. Manual implementation of Ns flow vs. AXI4MLIR generated driver code for different flow strategies, Ns, As, Bs, Cs. All bar groups follow similar trends. Ns, As, and Bs **bottlenecks are analysed and addressed** in the following experiments.

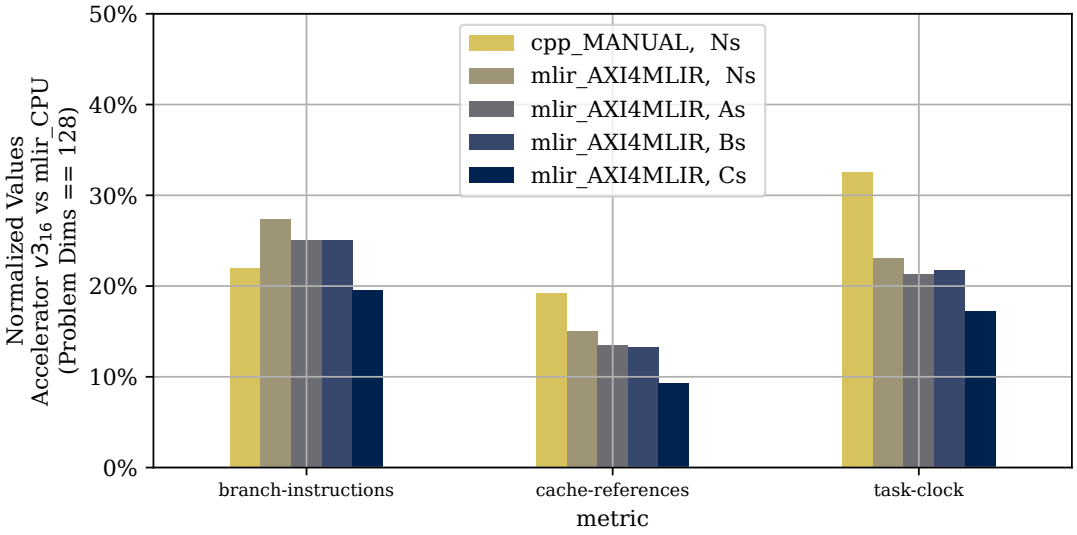
clock. These metrics were obtained using the `perf` tool [The] to profile the application and retrieve counters for CPU perf_events over 5 runs.

Figure 8.10a shows branch instructions, cache reference counters, and the task clock for $dims == 128$, for the $v3_{16}$ accelerator that supports input and output stationary flows. The trends are similar to other problem and accelerator sizes. Our results are normalised to the same counters collected on a CPU-only execution of the same problem size. In each group, we show results for AXI4MLIR automatically generated code for Ns, As, Bs, and Cs flows and compare them against manual implementations (first bar of a group) for copying the necessary data through the DMA memory-mapped region. MLIR applications have to consider MLIR memory references (presented in Section 2.2.3), but manual implementations use bare C-arrays. To support generality, MLIR copies between MemRef and the raw array (DMA buffer region) are implemented with a recursive call, loading and storing one element at a time. This is necessary to support $rank = N$ MemRefs, where strides in all dimensions differ from 1.

In order to address this issue, we implemented an optimisation for when $strides[N - 1] == 1$ (i.e., elements in $N - 1$ dimension are adjacent to each other in memory) and specialised MemRef copies for some known rank sizes, such as $rank == 2$. For



(a) Without the MemRef-DMA buffer copy optimisation. Generated host-accelerator code has overheads if not specialised.



(b) With MemRef-DMA buffer copy optimisation. AXI4MLIR improves accelerator performance over manual Ns implementation.

Figure 8.10: Cache, branch, and runtime metrics of different tools and strategies using v3$_{16}$ accelerator with problem size (*dims* == 128). Normalised values against CPU (without accelerator) executions of same problem size.

this scenario, we leverage the spatial locality and implement the copy not with individual load and store instructions but by calling `std::memcpy(src, dst, size)`. When compiling this function for our platform, the compiler will inline the assembly, implementing a vectorised copy that improves the performance of the copy operation. The implications of this optimisation are twofold. First, it reduces the number of branch references because there is no need for branching to handle non-unitary strides or to

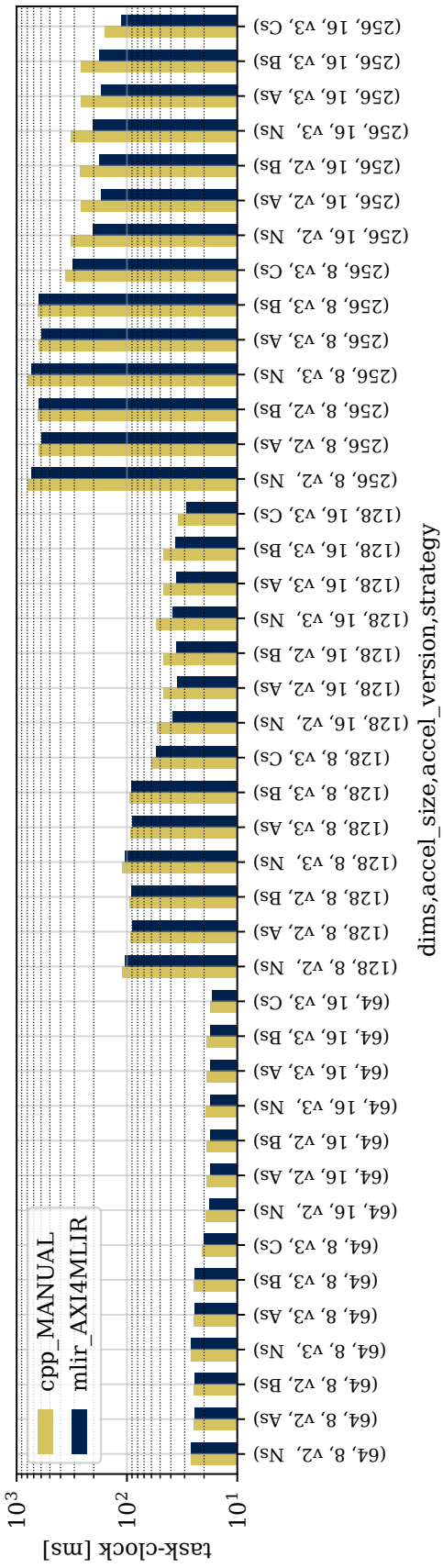


Figure 8.11: Runtime comparison of the manual implementation of driver code and AXIMLIR-generated code. Each set of two bars has a matching Accelerator Type, Accelerator Size, and Flow Strategy (Ns, As, Bs, Cs). AXI4MLIR is better in all cases.

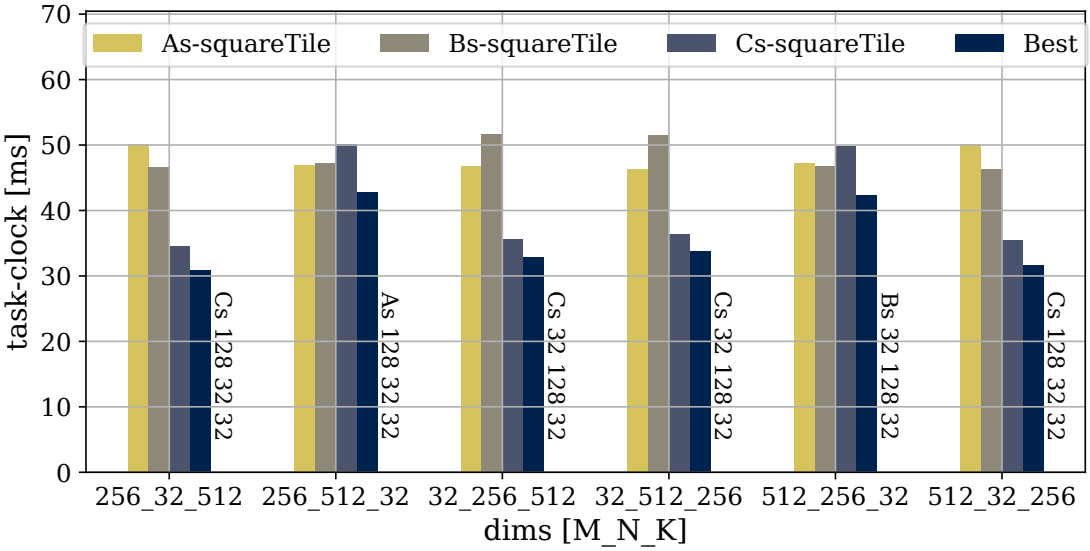


Figure 8.12: MatMul problem permutations (v4 accelerator) for different strategies. For the "Best" strategies we annotate the chosen flow and tiling values.

iterate over an arbitrary number of dimensions, resulting in better control flow and branch prediction. Second, the vectorised code reduces the number of cache references because the data is accessed sequentially in memory.

Therefore, there will only be two cache references to fetch the cache line containing the requested data, and subsequent loads within the same cache line will not require additional cache references as they are read from the vector VFP registers [ARM23]. The results for this optimisation are presented in Figure 8.10b.

After incorporating this optimisation, the AXI4MLIR-generated driver code executed faster on all accelerators than their respective manual implementations. In Figure 8.11, we compare AXI4MLIR against manual implementations for Ns, As, Bs, and Cs and found that the compiled generated driver code provided by AXI4MLIR is consistently faster (1.18× average speedup and 1.65× max speedup), thanks to its ability to leverage proper tiling for the CPU's memory hierarchy, resulting in a 10% average and 56% max reduction in cache references.

8.3.3 Matrix-Multiplication with flexible sizes

Runtime configurable accelerators allow for fine-grained hardware tuning for specific problems. With AXI4MLIR, we can generate host code to configure and optimise flexible accelerators for the target problem. To demonstrate this capability, we evaluated multiple permutations of a MatMul problem on the v4 accelerator. The v4 accelerator supports multiple dataflow strategies and adjustable tile sizes for its tM, tN,

and tK dimensions. The intuition is that scientific and machine learning workloads present problem sizes with different values for each dimension, sometimes resulting in tall/skinny matrices during execution. Tiling the problem in the accelerator with different dimensions for tM, tN, and tK, and selecting the appropriate flow strategy can be beneficial for the application.

When using AXI4MLIR, a developer is *not* limited to one configuration of an accelerator. Based on the user's knowledge of the application, AXI4MLIR can automatically generate the driver for accelerators with adjustable dimensions. This flexibility allows for a more thorough design space exploration, enabling the developer to find the best sizes for tM, tN, tK, and the best flow strategy for each problem instance.

In Figure 8.12, we compare four different heuristics and use them to choose the best tiling and dataflow configuration for a MatMul problem. We evaluated performance in terms of execution time. We profile the problem with M,N, and K dimensions permuted from the following values: $[32, 256, 512]$. Hence, the theoretical minimum number of multiply-accumulate operations required for all permutations is the same. Here, the *As-squareTile*, *Bs-squareTile*, and *Cs-squareTile* heuristics try to find the best configuration to reduce the total memory access count given the constraint of tiling the MatMul with square tiles (i.e., $tM = tN = tK = T$), with A, B, and C stationary dataflow, respectively. The fourth heuristic, *Best*, chooses between all dataflows and flexible tiling options, only sharing the choice of the accelerator. In Figure 8.12, we annotate the '*Best*' configuration found for each problem.

Square tiling. We observe that as we change the problem permutation, the best flow between *As-squareTile*, *Bs-squareTile*, and *Cs-squareTile* tiling strategies changes. The best flow depends on the problem shape, the size, and the available accelerator buffer space. $T = 32$ was selected for all square flows because it is the biggest value, so the tiles fit inside the accelerator's internal memory.

Flexible tiling. The *Best* heuristic, selected from non-square strategies, outperforms square tiling by leveraging flexible tiling sizes. AXI4MLIR can generate code to utilise larger tile sizes in various dimensions, taking advantage of the v4 accelerator's unrestricted tiling factors and improving the accelerator's internal memory utilisation.

Configurations. Manually implementing all configurations' driver code for even a simple accelerator such as v4 is very time-consuming. AXI4MLIR can quickly generate the host code for configurable accelerators easily, enabling the developer to specify an accelerator configuration per problem instance.

```
accel_dim = map<(B,H,W, iC,oC,fH,fW) ->
                (0,0,0,256, 1, 3, 3)>, // Tiling
opcode_map<
  sIc0=[send_literal(70), send(0)], // send 3D input window
                                    //  and compute
  sF=[send_literal(1), send(1)],    // send 3D filter
  r0=[send_literal(8), recv(2)],    // recv 2D output slice
  rst=[send_literal(32), send_dim(1,3),  // set filter size
       send_literal(16), send_dim(0,1)]> // set iC size
opcode_flow <(sF (sIc0) r0)> // filter+output stationary
init_opcodes <(rst)>
```

(a) Opcode Map and Flow for Conv2D accelerator.

```
func.func @conv_call(...) {
  // With %I: !mrI_1_256_7_7; %W: !mrW_64_256_3_3
  // and  %O: !mrO_1_64_5_5
  // Declare constants (loop bounds and literals): %cX, ...
  accel.dma_init(%c0,%c66,%c65280,%c65346,%c65280) : ...
  accel.sendLiteral(%c32, %c0) : i32,i32->i32 // send inst
  accel.sendDim(%W,%c3,%c0) : !mrW,i32,i32->i32 // send %fH
  accel.sendLiteral(%c16, %c0) : i32,i32->i32 // send inst
  accel.sendDim(%I,%c1,%c0) : !mrI,i32,i32->i32 // send %iC

  // Tile dims by (B,H,W,iC,oC,fH,fW) -> (-,-,-,256,1,3,3)
  scf.for %b = %c0 to %c1 step %c1 { // B loop
    scf.for %oc = %c0 to %c64 step %c1 { // OC loop
      %sW = memref.subview %W[%oc,0,0,0][1,256,3,3] ...
      %offset0 = accel.sendLiteral(%c1, %c0) : i32,i32->i32
      %offset1 = accel.send(%sW, %offset0) :
                            !mrSubWx256x3x3, i32 -> i32
      scf.for %oh = %c0 to %c5 step %c1 { // OH loop
        scf.for %ow = %c0 to %c5 step %c1 { // OW loop
          %xoffset = ... // index calculation
          %yoffset = ... // index calculation
          %sI = memref.subview %I[0,0,%xoffset,%yoffset]
                            [1,256,3,3] ...
          %offset2 = accel.sendLiteral(%c70, %c0) : ...
          %offset3 = accel.send(%sI, %offset2) :
                            !mrSubIx256x3x3, i32 -> i32
          // inner product of sW and sI computed in HW
      } }
      %sO = memref.subview %O[0,%oc,0,0] [1,1,5,5] ...
      %offset4 = accel.sendLiteral(%c8, %c0) : ...
      accel.recv {mode="accumulate"}(%sO, %c0) :
                            !mrSubO_5x5_0, i32 -> i32
} } } } return }
```

(b) IR to drive the Conv2D accelerator with an output-stationary flow.

Figure 8.13: Information added to the linalg.generic traits to capture convolution accelerator behaviour in MLIR and IR with accel operations.

8.3.4 Convolution

We show the flexibility of AXI4MLIR by generating driver code for a convolution-based accelerator executing different problem sizes. This accelerator supports varying input channel (*iC*) and filter (*fHW*) sizes, computing one output slice (all elements in one output channel - *oC*) per iteration. Note that multiple instructions have to be sent to the accelerator to orchestrate the execution of the convolution operations.

This orchestration is achieved by compiling the driver code derived from the MLIR `accel` code (Figure 8.13b). The `accel` code is generated after a transformation pass takes into account the attributes shown in Figure 8.13a and MLIR's `linalg.conv_2d_nchw_fchw` operations. Note that if the convolution operation has *iC*, *fH*, *fW* dimensions that are smaller than the dimensions in `accel\_dim`, no tiling will be performed across these dimensions. In the convolution example (Figure 8.13), upon accelerator reset, we use `send\_dim(1,3)` to send to the accelerator the filter size as the dimension '3' of data structure '1' (i.e., the filter), and we use `send\_dim(0,1)` to send the input channel size as the dimension '1' of the data structure '0' (i.e., the input).

We evaluated the performance of AXI4MLIR during the execution of all convolution layers of ResNet18 [Wan+17]. Figure 8.14 presents performance metrics normalised to the runtime of layer-specific manual C++ driver code. The results observed here present similar trends to those observed in the MatMul experiments. Only one layer (*56_64_1_128_2*) presented a 10% slowdown, contrary to previous trends. The slowdown happened because *fHW (1)* and *iC (64)* were too small, and the overhead of dealing with small MemRefs was not overcome since we could not leverage the *strided copy optimisation* presented in Section 8.3.2. Smaller AXI4MLIR speedups are observed every time that *fHW*== **1**. That said, AXI4MLIR achieves better runtime performance on 10 out of 11 ResNet18 layers, with 1.28× and 1.54× average and max speedup, respectively, thanks to the improved CPU cache performance.

8.3.5 End-To-End Analysis

Finally, we evaluated AXI4MLIR when compiling a natural language processing model to co-execute on both the CPU and the $v4_{16}$ accelerator. We benchmarked the TinyBERT [Jia+20] model, a compact transformer [Wol+20] model for Masked Language Modelling and Next Sentence Prediction targeted at mobile and embedded devices. We translate TinyBERT to MLIR IR using Torch-MLIR [Tor24] and compare the inference performance of CPU execution (using -O3 during compilation) against co-execution using the *"Ns"* offloading approach and the *'Best'* approach, which employs the heuristics

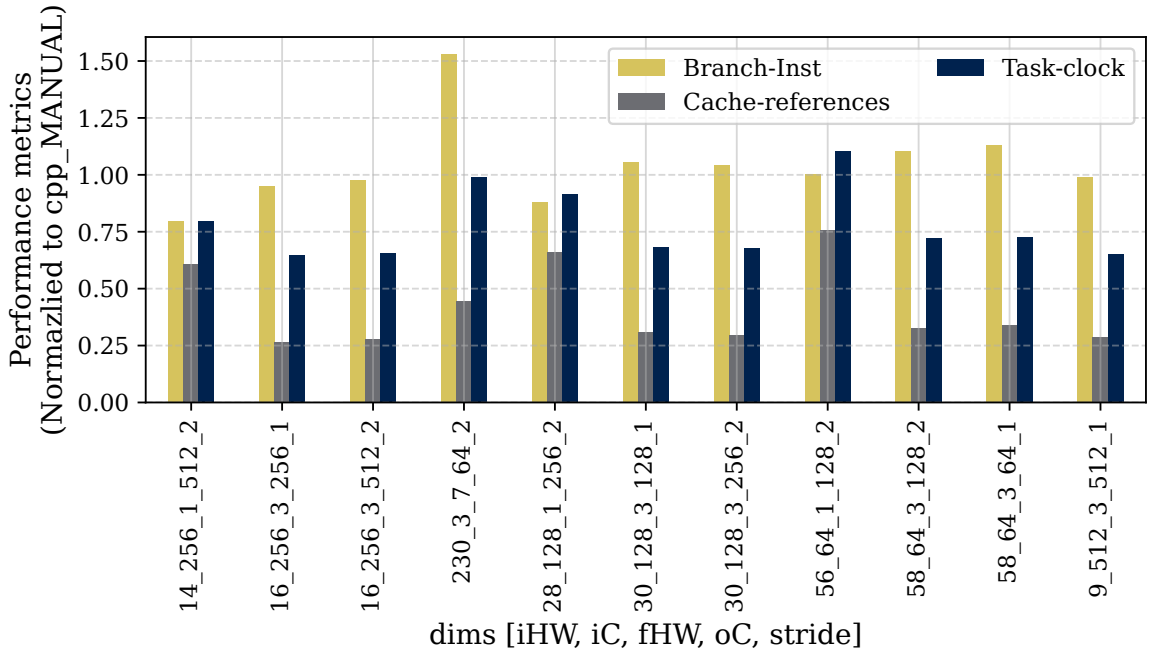


Figure 8.14: ResNet18 convolution layers: AXI4MLIR vs. Manual.

presented in Section 8.3.3.

As we can see in Figure 8.15, AXI4MLIR achieves a 3.4× speedup in end-to-end execution, with an 18.4× speedup in the accelerated MatMul layers that represent 75% of the original CPU runtime. This experiment showcases how AXI4MLIR can be used to evaluate and optimise natural language processing models on embedded devices. This study highlights that developers can easily co-design the accelerators when targeting full workloads, enabling efficient exploration and utilisation of CPU and accelerator resources.

8.4 Summary

This chapter presented AXI4MLIR, an MLIR extension to enable automatic code generation of host driver code for new custom accelerators. We present the new MLIR attribute that enable a standardised approach to describe the accelerator that supports tensor linear algebra operations. Within our code generation we are able to replace standard MLIR linalg operations with custom functions which enable specific tiling and dataflows, and generate the custom accelerator instructions/opcodes to configure and control the accelerator. Additionally, our code generation creates calls to our custom AXI enabled DMA library to perform data transfers between the host and the accelerator. We demonstrate the effectiveness of AXI4MLIR through a set of experiments which used a library of custom accelerators developed using the SECDA methodology

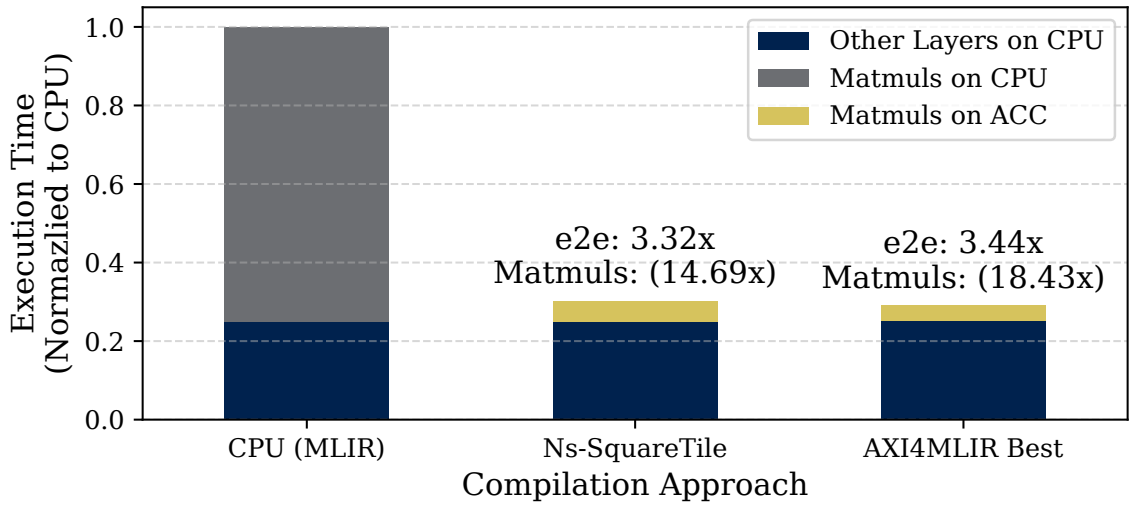


Figure 8.15: Execution time of the TinyBERT model with batch_size == 2. Each bar represents a compilation strategy. The speedups for end-to-end ($e2e$) and for accelerated MatMul layers are shown as annotations.

and SECDA-TFLite toolkit. The experiments show that the generated code is able to achieve similar, if not better performance, compared to manually written code while reducing the development time and effort to write the host driver code. Additionally, the experiments highlight the potential for increased design space exploration of host driver code, as the code generation can be easily modified to explore different dataflows and tiling strategies.

9 | Conclusions

This thesis was motivated by the need for efficient hardware accelerators for DNNs, given a resource-constrained environment. As we explored the process of designing and implementing hardware accelerators for DNNs, we identified several key challenges that need to be addressed in order to facilitate the design and implementation of efficient hardware accelerators. The challenges presented in Section 1.2, provoked the need for a guideline, a path to follow, and a way to systematically approach the process of designing and implementing hardware accelerators for DNNs. Chapter 4 addressed this need by introducing the SECDA methodology, which provides a systematic approach to designing and implementing hardware accelerators for DNNs. During the development of the SECDA methodology, we identified tangible bottlenecks in the design process which could be solved through tools and frameworks that ease the adoption of the methodology for the target application framework. This led to the development of the SECDA-TFLite toolkit and the SECDA-LLM platform presented in Chapter 5 and Chapter 6 respectively.

Once the process of designing and implementing hardware accelerators was streamlined, we identified potential interesting workloads for hardware acceleration. Chapter 7 explored the challenges of accelerating GANs, specifically looking at the generative aspect of GANs, which is essentially the Transposed Convolution operation. Finally, Chapter 8 presented AXI4MLIR as way to tackle the challenge of developing host driver code for new custom accelerators through automatic code generation using the MLIR compiler framework.

This chapter is structured as follows: Section 9.1 summarises the main contributions of this thesis, Section 9.2 outlines potential future work that can be built upon the work presented in this thesis, and finally, Section 9.3 reflects on the challenges faced, the lessons learned and self-critiques on the presented work.

9.1 Contributions

The main contributions of this thesis can be related to the main challenges identified in Section 1.2. The following sections summarise the main contributions: first, reducing the development time of designing new hardware accelerators for DNNs; second, design and optimisation for accelerating Transposed Convolution operations in GANs; and finally, a tool for automated host driver code generation to enable efficient host-accelerator communication.

9.1.1 Designing Process for FPGA-based DNN Accelerators

The main contribution and core of this thesis is about reducing the development time of designing new hardware accelerators for DNNs, specifically targeting resource-constrained edge devices. The SECDA methodology, as presented in Chapter 4, was developed to simplify the design process and provide a high-level design approach to enable a fast and iterative design loop. While the SECDA methodology was initially developed as part of the learning process to design FPGA-based DNN accelerators, it matured into a design process that can easily be applied to different types of DNN accelerators and potentially other types of accelerators. This maturity in the design process was first achieved through the development of the SECDA-TFLite toolkit, which provided a simple connection between a hardware design environment and a high-level application framework (TensorFlow Lite). Utilising the SECDA-TFLite toolkit has enabled exciting research opportunities that have been demonstrated through the SECDA-TFLite case study in Section 5.4. In fact, it was a significant factor in enabling new ideas and prototyping designs, as presented in Chapter 7 and Chapter 8. Additionally, as the SECDA methodology had shown promise through SECDA-TFLite, where adapting the methodology to the LLM domain was a natural progression resulting in the SECDA-LLM platform as presented in Chapter 6. While the SECDA-LLM platform is still in its early stages, with our short case study in Section 6.3, it has shown promise in enabling the exploration of FPGA-based accelerator designs and optimisations for LLMs. As a whole, the focus of this thesis has been on reducing the time-effort of research and enabling the exploration of novel ideas. Hence, the contributions presented in this thesis are the initial steps enabling further research and development in the area of DNN accelerators for resource-constrained edge devices.

9.1.2 Accelerating Transposed Convolution for GANs

Exploring the efficient execution of GAN models has led to the second contribution of this thesis, which is the design and optimisation of the *MM2IM* hardware design for accelerating Transposed Convolution operations in GANs as described in Chapter 7. The initial exploration of the methods of implementing Transposed Convolution operations as described in Section 2.4.3 led to the understanding that there are inefficiencies. From the additional compute and memory requirements for the padded input data in the Zero-Insert method to the transformation overhead of the TDC method or the overlapping sum problem of the IOM method, it was clear that while research has progressed in this area, the focus on optimising for resource-constrained edge devices was lacking. Focusing on the IOM method, we noted three main requirements for efficient execution on resource-constrained edge devices: efficient processing of overlapping sums, storing of intermediate results, and handling of cropped outputs. Existing solutions especially neglect the issue of ineffectual computations due to cropped outputs. Hence, in order to address these requirements, we developed the *MM2IM* accelerator architecture. The key components of this architecture include: (i) the use of the *output mapping* and the *compute mapping* as described in Section 7.2.1; (ii) the tiled MM2IM dataflow as described in Section 7.2.2; and (iii) the on-the-fly MM2IM mapper as described in Section 7.3.6. While our study in Chapter 7 provides improved results, as it is still a work in progress, there are still areas that need further exploration. Most importantly, a more comprehensive evaluation of the MM2IM accelerator architecture is required, including a more extensive comparison with other resource-constrained TCONV accelerators (albeit limited) and an evaluation of a broader range of GAN models.

9.1.3 Automated Host Driver Code Generation

The automatic code generation of host driver code for AXI-based accelerators, as presented in Chapter 8, tackles the challenge of enabling efficient host-accelerator communication. With AXI4MLIR, we have developed an extension to the MLIR compiler framework, which includes new MLIR attributes that enable the representation of accelerators to be defined and used to generate efficient host driver code. One crucial aspect is that AXI4MLIR enables the description of the CPU and custom accelerator architecture, so our transformations create CPU and accelerator-aware host driver code, such as tiling and dataflow strategies supported by the accelerator. As a consequence, the AXI4MLIR compilation flow generates code to perform efficient data transfers and also generates code containing accelerator-specific instructions/opcodes

that can be used to control the accelerator. To enable this, we have developed the 'accel' dialect, which connects the algorithmic implementation of offloading computations to a new custom accelerator with the low-level AXI DMA library, which enables the transfer of data between the host CPU and the AXI-based accelerator. Through our experiments, we have demonstrated that AXI4MLIR can act as: first, the bridge between the high-level algorithmic description of a DNN and the low-level host driver code required to offload computations to a custom accelerator; and second, an efficient way to explore the design space of different dataflow strategies for the target accelerator. AXI4MLIR, combined with the SECDA methodology, can enable the design space exploration of new classes of custom domain-specific accelerators targeting DNNs and even tensor computations in general.

9.2 Future Work

Here, we explore the potential future work that can spring forth from the research presented in this thesis. Also, we discuss the limitations and how they can be addressed in future work. Some of these ideas are partly in progress, while others are interesting research directions that can be explored.

9.2.1 The Potential of SECDA

The SECDA methodology, combined with the subsequent tools developed, SECDA-TFLite and SECDA-LLM, have shown promise in enabling the exploration of FPGA-based accelerator designs and optimisations for DNN inference on resource-constrained edge devices. One of the main avenues for improvement is to further reduce the development time of designing new hardware accelerators. An exciting and potentially impactful direction is to explore the use of templated-based design space exploration tools such as the ones described in Section 3.3.2 with the SECDA methodology. SECDA would enable the high-level design process, which could be used to generate the architectural templates for the design space exploration tools. This would enable the exploration of a wider range of fine-grained hardware parameter tuning of the accelerator designs produced by SECDA and its subsequent tools.

Enabling SECDA for different types of accelerators, such as Processors-in-Memory (PIM) accelerators or accelerators for Genome Analysis to name a few, would also be an interesting research direction. This work would entail similar approaches to SECDA-TFLite and SECDA-LLM but with a focus on the specific requirements of the target domain. To enable such an endeavour, we are currently working on a library

of SECDA-based tools that are application-agnostic and could enable the integration of SECDA with different application frameworks with minimal effort. Similarly, the existing SECDA tools are still under development. While SECDA-TFLite is more mature and publicly available, SECDA-LLM is still in its early stages and requires further development.

Most importantly, the SECDA methodology and tools are designed to enable the exploration of novel ideas in the space of DNN accelerations. The exploration of these ideas within the context of this thesis has been limited. Hence, future work is dependent upon the exploration of critical research questions in this field, such as the exploration of quantisation techniques, the acceleration of different types of DNN models, efficient execution of sparse DNNs, and heterogeneous and reconfigurable accelerators, to name a few. Incidentally, one of the complementary works [RJJ24] listed in Section 1.4 focuses on the acceleration of power-of-two (PoT) quantisation techniques on resource-constrained edge devices. This work was able to utilise the SECDA methodology and SECDA-TFLite to explore the design space of different PoT quantisation techniques and propose a new hardware design for accelerating PoT-quantised DNN models.

9.2.2 Accelerating Generative DNN Models

The MM2IM accelerator architecture, as presented in Chapter 7, has shown promise in accelerating Transposed Convolution operations in GANs. A key area for future work is a more comprehensive evaluation of the MM2IM accelerator architecture. This evaluation should include a more fine-grained understanding of the accelerator's performance and, overall, a broader set of GAN models. Additionally, a study on the scalability of the MM2IM architecture in terms of hardware resources and performance would be beneficial. The architecture is inherently scalable due to modular design, but a more in-depth study is required to understand the limits and potential for scaling the architecture. This study could enable insights into the design of MM2IM and also support TCONV operations in edge devices with even more stringent resource constraints like microcontrollers.

9.2.3 Extensions to AXI4MLIR

Concerning AXI4MLIR, we have formalised the potential extensions to the work within a short paper [Har+24a] presented in the C4ML Workshop at the International Symposium on Code Generation and Optimization (CGO) 2024. The three main extensions include: (i) DMA-based data allocation, i.e., the ability to allocate data directly to the DMA buffers; (ii) Data coalescing, i.e., the ability to coalesce multiple data transfers

into a single transfer; and (iii) Software pipelining of data transfers, which enables overlapping data transfers with computation. These extensions aim to improve the overall performance of the generated host driver code by reducing the data transfer overheads and enabling more efficient data transfers between the host CPU and the accelerator. Appendix A.1 covers these extensions in more detail. Finally, as stated before, integrating AXI4MLIR with the SECDA methodology could enable wider exploration of the design space. As such, developing a fully integrated toolchain enabling acceleration of MLIR defined workloads to FPGA-based accelerators would be an interesting research direction.

9.3 Reflection

To reflect on the work presented in this thesis, we will discuss the challenges faced, the lessons learned, and provide a self-critique of the work. And to conclude, we will provide some final thoughts on the thesis and on the PhD journey as a whole.

9.3.1 Challenges Faced

Throughout the thesis, I have faced several challenges that have shaped the direction of the research and ultimately the contribution presented within this thesis. To summarise, the first and foremost of these challenges was the development time of designing new hardware accelerators for DNNs. At the beginning of the journey in understanding how to develop hardware accelerators for DNNs on FPGAs, I was faced with the understanding of the complexity and importance of the design process. Realising the need for an efficient and simple approach to prototype and evaluate new designs and ideas was the critical point in this field of research where the requirements and workloads are constantly evolving.

Another challenge faced was essentially out of the control of the researcher, which was the deprecation of SystemC High Level Synthesis (HLS) in newer versions of the Xilinx Vitis toolchain. While this has limited the exploration of new HLS-based optimisations, it has not affected the overall contributions of the thesis. As one could argue, research should be more resilient to such changes, hence while the tools presented in this thesis have been tested with Xilinx based FPGAs, nothing prevents the tools from being adapted to other FPGA vendors.

9.3.2 Lessons Learned

As the research journey progressed, a key lesson learned but not emphasised in the thesis was the importance of writing efficient synthesisable code. Many, many hours were spent debugging and optimising the hardware designs to ensure they were synthesised correctly and efficiently. While this is a common challenge in hardware design, the core scope of this thesis did not cover the difficulties and complexities of HLS, and it would be another thesis in itself to cover this topic. Hence, the lesson learned is to understand the limitations of the HLS tools and to write efficient, synthesisable and modular code. Additionally, the importance of HLS 'pragmas' cannot be understated, as they are crucial in guiding the HLS tools to generate efficient hardware designs, while architectural ideals that can be developed and designed with pen and paper to formalise them into a hardware description is a different challenge.

Another lesson learned was the importance of tooling, automation, and reproducibility in research. Whether it is a small task, such as loading up and uploading a new design to the FPGA, or a more time-consuming task, such as running a set of experiments, it is crucial to have a set of scripts and tools to automate these tasks. This strengthens the research's reproducibility, as the methods can be easily replicated using the tools and scripts developed. A notable example is the development of AXI4MLIR, one of the later projects in the PhD journey; we set a high standard for automating the process of evaluating the generated host driver code. This pre-developed automation enabled us to quickly provide a reproducible environment, which enabled us to pass the artefact evaluation for the paper submission to the International Symposium on Code Generation and Optimization (CGO) 2024.

9.3.3 Self-Critique

As discussed earlier in this chapter, the future work section covers potential extensions to the work presented in this thesis. As such, some aspects of the work have been limited and could be improved. Firstly, the work presented in this thesis has been evaluated using a single FPGA platform, the PYNQ-Z1. While the ideas presented in this thesis are platform-independent, evaluating different FPGA platforms could have highlighted any platform-specific limitations or advantages of the tools presented within this thesis.

One avenue of optimisation not explored in this thesis is the potential for multi-layer hardware acceleration, i.e., the acceleration of multiple heterogeneous layers of a DNN model within a single accelerator without sending data back to the host processor. The lack of exploration in this area was due to the self-imposed hardware resource

constraints within the works presented in this thesis. To explain, a broader exploration of the varying levels of resources 'constraints' could explode the possibilities of hardware acceleration. While it would be interesting to explore this widely, it would be a significant undertaking.

9.3.4 Final Thoughts

The design of hardware accelerators for DNNs is an expansive and complex problem space. In addition, as the field rapidly evolves, new types of workloads and applications based around DNNs emerge, making hardware-software co-design more critical than ever. Furthermore, as the need for cost-effective, power-efficient, and high-performant solutions becomes more prevalent, pushing the boundaries of what is possible with as few resources becomes even more important. Therefore, the need for efficient approaches to explore the design space of hardware accelerators for DNNs is crucial.

A Appendix

A.1 Data Transfer Optimisations in AXI4MLIR

Here we discuss the proposed extensions to the AXI4MLIR compilation flow which extensive discussed in Chapter 8. While AXI4MLIR can provide considerable speedup over a manual implementation of the host-driver code, we observe potential areas for further optimisations, which can improve the utilisation of the accelerator and reduce overall latency.

Figure A.1 shows a breakdown of the cycles spent inside the accelerator while executing MatMul problems for a range of dimensions and tile sizes. Ideally, the accelerator's compute cores should be fully utilised, but our experiments show that we are achieving on average less than 10% utilisation. Additionally, profiling the execution of this MatMul problem from the CPU's perspective, we observed that the CPU-side bottleneck is caused by copying data allocated within the memory heap to the DMA buffers. To tackle this accelerator under-utilisation and host-side bottlenecks, we propose extending AXI4MLIR with three key optimisations, discussed in detail in the following section.

A.1.1 Proposed Data Transfer Optimisations

The following data movement optimisations extend the AXI4MLIR transformation and lowering pipeline to mitigate and hide the time spent on transferring data. We demonstrate and apply the proposed optimisations to the tiled MatMul problem with a flexible stream-based accelerator that supports double buffering. Figure A.2 shows the baseline code generated by AXI4MLIR for this accelerator and algorithm.

DMA-based data allocation

Any data that needs to be transferred via AXI DMA engines must first be placed in DMA-mmapped buffers. Current AXI4MLIR implementations copy data between

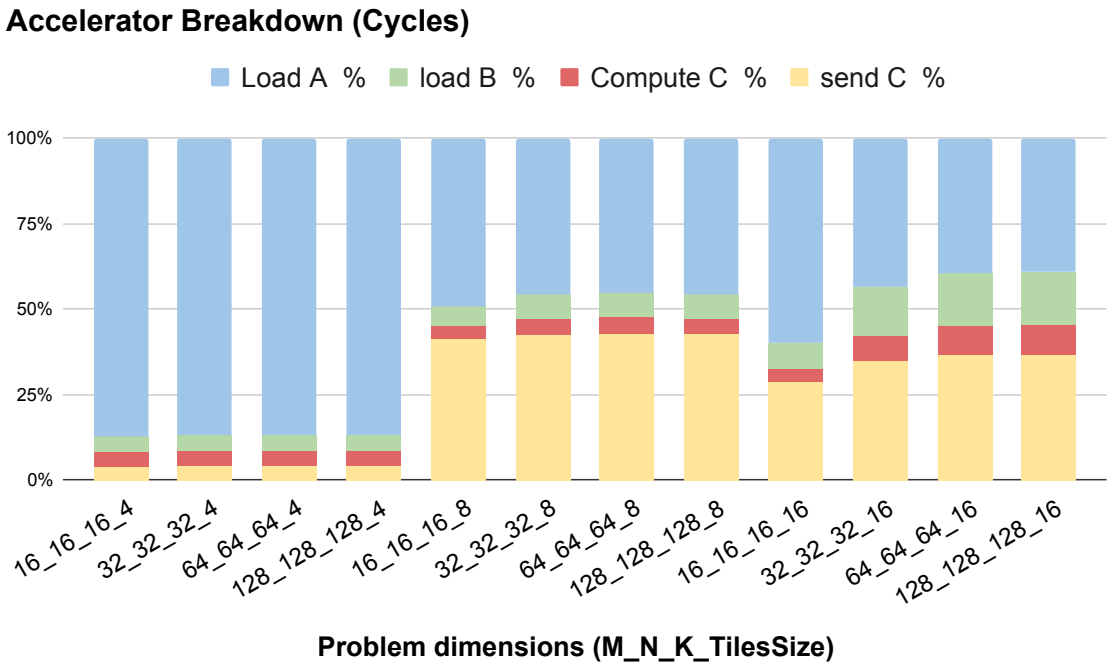


Figure A.1: Breakdown of clock cycles spent inside a simple MatMul accelerator. Red segments (Compute C %) represents the time when the accelerator's processing elements are active.

buffers (`memrefs`) allocated in the heap to the mmapped region (dma buffers) while communicating with the accelerator. This can incur significant overhead. To mitigate the extra staging transfers, we propose a new attribute and set of transformations that trigger allocation of `memrefs` needed by the accelerator directly in the DMA region.

Figure A.3 demonstrates how the optimisation will tag the `memref` with the `#dma` tag, which ensures that the required data is allocated within the DMA buffers, hence avoiding additional copying. To support this feature, the lowering of `accel.send` operations is simplified, skipping any staging copy and immediately initiating data transfers to the accelerator.

Data coalescing

A detailed analysis of the accelerator performance highlights that the initial data load latency is a bottleneck within the accelerator, which explains why *load A* takes more cycles than *load B* within Figure A.1. The first data packet, loaded from memory to the accelerator incurs additional latency, whereas the following data can be loaded in a FIFO-like manner with, in some cases, a 1-cycle delay. To reduce the initial transfer overhead, we propose a data coalescing strategy. This involves combining multiple data sends within the same loop body into a single DMA transfer operation, replacing

```
func.func @main (%input, %output) {
  // Allocation of %A and %B as outputs of other operation
  %A = memref.alloc () : () -> memref<60x80xfp32>
  %B = memref.alloc () : () -> memref<80x72xfp32>
  // ..Other operations happen...
  // %A, %B, and % are needed by the accelerator
  %C = memref.alloc () : () -> memref<60x72xfp32>
  func.call @matmul_call(%A, %B, %C) // ...
}

func.func @matmul_call(...) {
  // Declare constants (loop bounds and literals): %cX, ...
  accel.sendLiteral(%c0xFF) // reset opcode
  // Tiling by 4,4,4
  scf.for %m = %c0 to %c60 step %c4 { // first loop
    scf.for %k = %c0 to %c80 step %c4 { // second loop
      scf.for %n = %c0 to %c72 step %c4 { // innermost
        %sA = memref.subview %A[%m, %k] [4, 4] [1, 1]
        accel.sendLiteral(%c0x22)
        accel.send(%sA)
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1]
        %sC = memref.subview %C[%m, %n] [4, 4] [1, 1]
        accel.sendLiteral(%0x25)
        accel.send(%sB)
        accel.recv {mode="accumulate"}(%sC)
  } } } return }
```

Figure A.2: Pseudo-MLIR code of a tiled MatMul problem showcasing the baseline code generation.

multiple synchronisation operations with a single.

Figure A.4 shows how we propose to represent the coalescing of multiple data transfers into one. We will transform `accel.send` into a *variadic* operation, which takes multiple `memref` arguments, indicating that only one synchronisation is necessary.

Software pipelining & double-buffering

To further reduce the accelerator idle time and overall latency, we propose host-driver support to pipeline the accelerator load, compute, and store stages. Figure A.5 demonstrates how the initial iterations of the inner loop within the `matmul_call` are moved outside the loop to enable software-level pipelining. Note that this driver code optimisation can only be enabled when the accelerator provides double buffering support to overlap loading data with computation.

```
func.func @main (%input, %output) {
  // ...
  %A = memref.alloc () : () -> memref<60x80xfp32, #dma>
  %B = memref.alloc () : () -> memref<80x72xfp32, #dma>
  %C = memref.alloc () : () -> memref<60x72xfp32, #dma>
  func.call @matmul_call(...)
  // ...
return }
```

Figure A.3: Pseudo-MLIR code of a tiled MatMul problem with DMA-based data-allocation optimisation.

```
func.func @matmul_call(...) {
  // Declare constants (loop bounds and literals): %cX, ...
  accel.sendLiteral(%c0xFF) : i32,i32->i32 // reset
  scf.for %m = %c0 to %c60 step %c4 { // first loop
    scf.for %k = %c0 to %c80 step %c4 { // second loop
      scf.for %n = %c0 to %c72 step %c4 { // innermost
        %sA = memref.subview %A[%m, %k] [4, 4] [1, 1]
        %op0 = accel.load_opcode(%0x22)
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1]
        %sC = memref.subview %C[%m, %n] [4, 4] [1, 1]
        %op1 = accel.load_opcode(%0x25)
        accel.send([%op0,%sA,%sB,%op1])
        accel.recv {mode="accumulate"}(%sC)
  } } } return }
```

Figure A.4: Pseudo-MLIR code of a tiled MatMul problem with data-coalescing optimisation.

```
func.func @matmul_call(...) {
  // Declare constants (loop bounds and literals): %cX, ...
  accel.sendLiteral(%c0xFF) // reset opcode
  scf.for %m = %c0 to %c60 step %c4 { // first loop
    scf.for %k = %c0 to %c80 step %c4 { // second loop
      %sA = memref.subview %A[%m, %k] [4, 4] [1, 1]
      accel.sendLiteral(%c0x22)
      accel.send(%sA)
      %sB = memref.subview %B[%k, %c0] [4, 4] [1, 1]
      accel.sendLiteral(%0x25)
      accel.send(%sB)
      scf.for %n = %c4 to %c68 step %c4 { // innermost
        %sA = memref.subview %A[%m, %k] [4, 4] [1, 1]
        accel.sendLiteral(%c0x22)
        accel.send(%sA)
        %n_last = arith.subi(%n, %c4)
        %sB = memref.subview %B[%k, %n] [4, 4] [1, 1]
        %sC = memref.subview %C[%m, %n_last] [4, 4] [1, 1]
        accel.sendLiteral(%0x25)
        accel.send(%sB)
        accel.recv {mode="accumulate"}(%sC)
      }
      %sC = memref.subview %C[%m, %c68] [4, 4] [1, 1]
      accel.recv {mode="accumulate"}(%sC)
  } } return }
```

Figure A.5: Pseudo-MLIR code of a tiled MatMul problem with software pipelining of data transfers.

Glossary

accelerators A hardware device which is designed to efficiently execute a specific operation or set of operations. Where efficiency is measured in terms of latency, throughput, power consumption, and area. 1

activation functions Activation functions are used in neural networks to compute the output of a node based on the inputs and the pre-existing/learned weights. 3

bandwidth The amount of data that can be transmitted in a fixed amount of time. 1

co-design Design process which involves simultaneous design of multuple components in a system, considering the interactions between them. 2

covolutions Refers to convolution tensor operation used in neural networks, where the input tensor is convolved with a weight tensor to produce an output tensor. 3

data-level parallelism Form of parallelisation that allows for the same instructions to be executed across multiple subset of the input data across parallel processing elements. 3

DNN inference DNN inference is the process of executing a trained DNN model using new input data to make predictions/classifications. 1

edge In our context, the 'edge' refers to computation (physically) close to the source of the data and away from server/cloud-based computation. This often means avoiding the costs of network delays and a lack of security and privacy. 1

edge platforms This refers to any hardware device that is running on the edge. 3

end-to-end In our context, end-to-end refers to execution of the entire workload usually the inference of a DNN model. 2

host-side driver The software component that runs on the host processor which manages the communication to the hardware accelerator. 9

mmapped Memory mapping enables mapping of a file or device into virtual memory, allowing the file or device to be accessed by a user program.

opcode A numeric code representation of an computer instruction, abbreviated from 'operation code'. 139

resource-constrained In our context, resource-constrained refers to limited hardware resources (computational hardware, memory capacity, memory bandwidth), physical area, and power/energy budget. 1

Acronyms

AI Artificial Intelligence. 1,

ASIC Application-Specific Integrated Circuit. 3,

BERT Bidirectional Encoder Representations from Transformers. 18,

BFP Block Floating Point. 10, 117,

BRAM Block RAM. 6,

CGRAs Coarse-Grained Reconfigurable Arrays. 1, 4, 23,

CLBs Configurable Logic Blocks. 4,

CNNs Convolutional Neural Networks. 2,

CONV Convolutional. 14,

DMA Direct Memory Access. 4,

DNNs Deep Neural Networks. 1, 14,

DSE Design Space Exploration. 54,

DSP Digital Signal Processor. 100,

DSPs Digital Signal Processors.

FC Fully Connected. 17,

FC-GEMM Fully Connected General Matrix Multiply. 10,

FFs Flip-Flops. 4,

FPGAs Field-Programmable Gate Arrays. 2,

GANs Generative Adversarial Networks. 2,

GEMM General Matrix Multiply. 63,

GOPS Giga Operations Per Second. 48,

GPUs Graphics Processing Units. 1,

HDL Hardware Description Language. 28,

HLS High-Level Synthesis. 29,

IoT Internet of Things. 1,

LLMs Large Language Models. 2,

LUTs Look-Up Tables. 4,

MACs Multiply-Accumulates. 50,

ML Machine Learning. 1, 14,

MLIR Multi-Level Intermediate Representation. 18,

MM2IM MatMul to col2IM. 10,

mmapped memory mapped. 5

MMIO Memory-Mapped Input/Output. 40,

NLP Natural Language Processing. 1,

NPUs Neural Processing Units. 1,

NVDLA NVIDIA Deep Learning Accelerator. 1,

PE Processing Element. 26,

PoT Power-of-Two. 165,

PUs Processing Units. 25,

QoR Quality of Results. 54,

RTL Register Transfer Level. 8, 28,

SA Systolic Array. 10,

SIMD Single Instruction, Multiple Data. 26,

TCONV Transposed Convolution. 10,

TOPS Tera Operations Per Second. 50,

TPU Tensor Processing Unit. 1,

VM Vector Mac. 10,

Bibliography

[Aba+16] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Nov. 2, 2016, pp. 265–283. ISBN: 978-1-931971-33-1.

[Abd+14] Ossama Abdel-Hamid et al. "Convolutional Neural Networks for Speech Recognition". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 22.10 (Oct. 2014), pp. 1533–1545. ISSN: 2329-9304. DOI: 10.1109/TASLP.2014.2339736.

[acc16] accellera. *SystemC Synthesizable Subset Version 1.4.7*. Version 1.4.7. 2016. URL: https://www.accellera.org/images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_4_7.pdf.

[Ago+20] Nicolas Bohm Agostini et al. "Design Space Exploration of Accelerators and End-to-End DNN Evaluation with TFLITE-SOC". In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD'20. Sept. 2020, pp. 10–19. DOI: 10.1109/SBAC-PAD49847.2020.00013.

[Ago+24] Nicolas Bohm Agostini et al. "AXI4MLIR: User-Driven Automatic Host Code Generation for Custom AXI-Based Accelerators". In: *IEEE/ACM International Symposium on Code Generation and Optimization*. CGO'24. Mar. 2024, pp. 143–157. DOI: 10.1109/CGO57630.2024.10444801.

[AK23] Milad Abolhasani and Eugenia Kumacheva. "The Rise of Self-Driving Labs in Chemical and Materials Sciences". In: *Nature Synthesis* 2.6 (June 2023), pp. 483–492. ISSN: 2731-0582. DOI: 10.1038/s44160-022-00231-0.

[Alb+16] Jorge Albericio et al. "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. ISCA'16. June 2016, pp. 1–13. DOI: 10.1109/ISCA.2016.11.

[Alt11] Altera. *Implementing FPGA Design with the OpenCL Standard*. 2011.

[Alw+16] Manoj Alwani et al. "Fused-Layer CNN Accelerators". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'16. Oct. 15, 2016, pp. 1–12.

[Ama+13] Filippo Amato et al. "Artificial Neural Networks in Medical Diagnosis". In: *Journal of Applied Biomedicine* 11.2 (Jan. 1, 2013), pp. 47–58. ISSN: 1214-021X. DOI: 10.2478/v10136-012-0031-x.

[AMDa] AMD. *AMD Versal AI Core Series Adaptive SoCs*. AMD. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/ai-core-series.html.

[AMDb] AMD Developers. *AMD Zynq™ 7000 SoCs*. AMD. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html.

[AMD24] AMD Developers. *AMD XDNA™ Architecture*. AMD. 2024. URL: https://www.amd.com/en/technologies/xdna.html.

[Ami+20] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (July 2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[ARM23] ARM Developers. *NEON Registers*. 2023. URL: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview/NEON-registers.

[Asa+16] Krste Asanovic et al. *The Rocket Chip Generator*. Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[Bac+12] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. June 3, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228584.

[BKC17] Vijay Badrinarayanan et al. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (Dec. 2017), pp. 2481–2495. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2016.2644615.

[Bou+20] Andrew Boutros et al. "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs". In: *2020 International Conference on Field-Programmable Technology*. ICFPT'20. Dec. 2020, pp. 10–19. DOI: 10.1109/ICFPT51103.2020.00011.

[Bro+20] Tom Brown et al. "Language Models Are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. NIPS'20. 2020, pp. 1877–1901. URL: https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[BSF94] Y. Bengio et al. "Learning Long-Term Dependencies with Gradient Descent Is Difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1941-0093. DOI: 10.1109/72.279181.

[Cha+20] Jung-Woo Chang et al. "Towards Design Methodology of Efficient Fast Algorithms for Accelerating Generative Adversarial Networks on FPGAs". In: *2020 25th Asia and South Pacific Design Automation Conference*. ASP-DAC'20. Jan. 2020, pp. 283–288. DOI: 10.1109/ASP-DAC47756.2020.9045214.

[Cha+23] Kit Yan Chan et al. "Deep Neural Networks in the Cloud: Review, Applications, Challenges and Research Directions". In: *Neurocomputing* 545 (Aug. 7, 2023), p. 126327. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2023.126327.

[Che+14a] Tianshi Chen et al. "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Feb. 24, 2014, pp. 269–284. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541967.

[Che+14b] Yunji Chen et al. "DaDianNao: A Machine-Learning Supercomputer". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'14. Dec. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[Che+17] Yu-Hsin Chen et al. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (Jan. 2017), pp. 127–138. ISSN: 1558-173X. DOI: 10.1109/JSSC.2016.2616357.

[Che+18a] Tianqi Chen et al. "Learning to Optimize Tensor Programs". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Dec. 3, 2018, pp. 3393–3404.

[Che+18b] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Oct. 8, 2018, pp. 579–594. ISBN: 978-1-931971-47-8.

[Che+19] Yu-Hsin Chen et al. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. May 20, 2019. URL: http://arxiv.org/abs/1807.07928.

[CHZ11] Wang Chong et al. “Hardware/Software Co-Design of Embedded Image Processing System Using Systemc Modeling Platform”. In: *2011 International Conference on Image Analysis and Signal Processing*. IASP’11. Oct. 2011, pp. 524–528. DOI: 10.1109/IASP.2011.6109098.

[CJM20] Lu Chi et al. “Fast Fourier Convolution”. In: *Advances in Neural Information Processing Systems*. Vol. 33. NIPS’20. 2020, pp. 4479–4488. URL: https://papers.nips.cc/paper_files/paper/2020/hash/2fd5d41ec6cfab47e32164d5624269b1-Abstract.html.

[CKK20] Jung-Woo Chang et al. “An Energy-Efficient FPGA-Based Deconvolutional Neural Networks Accelerator for Single Image Super-Resolution”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 30.1 (Jan. 2020), pp. 281–295. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2018.2888898.

[Clu+23] Jan Clusmann et al. “The Future Landscape of Large Language Models in Medicine”. In: *Communications Medicine* 3.1 (Oct. 10, 2023), pp. 1–8. ISSN: 2730-664X. DOI: 10.1038/s43856-023-00370-1.

[Coc+21] Marco Cococcioni et al. “Novel Arithmetics in Deep Neural Networks Signal Processing for Autonomous Driving: Challenges and Opportunities”. In: *IEEE Signal Processing Magazine* 38.1 (Jan. 2021), pp. 97–110. ISSN: 1558-0792. DOI: 10.1109/MSP.2020.2988436.

[COO] COOWOO. *COOWOO USB Digital Power Meter Tester Multimeter Current and Voltage Monitor*. URL: http://www.coowootech.com/coowoo-usb-digital-power-meter-tester-multimeter-current-and-voltage-monitor-dc-51a-30v-amp-voltage-power-meter-test-speed-of-chargers-cables-capacity-of-power-banks-black.html.

[Cor] Coral. *Edge TPU Compiler*. Coral. URL: https://coral.ai/docs/edgetpu/compiler/.

[CPS06] Kumar Chellapilla et al. “High Performance Convolutional Neural Networks for Document Processing”. In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. IWFHR’06. Oct. 23, 2006. URL: https://inria.hal.science/inria-00112631.

[Dav+19] Shail Dave et al. "dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators". In: *ACM Trans. Embed. Comput. Syst.* 18 (5s Oct. 8, 2019), 70:1–70:27. ISSN: 1539-9087. DOI: 10.1145/3358198.

[Des06] Design Automation Standards Committee. "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (Apr. 2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[Des19] Design Automation Standards Committee. "IEEE Standard for VHDL Language Reference Manual". In: *IEEE Std 1076-2019* (Dec. 2019), pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

[Des23] Design Automation Standards Committee. "IEEE Standard for Standard SystemC® Language Reference Manual". In: *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (Sept. 2023), pp. 1–618. DOI: 10.1109/IEEESTD.2023.10246125.

[Deva] ARM Developers. *AMBA AXI-Stream Protocol Specification*. URL: https://developer.arm.com/documentation/ihi0051/b/.

[Devb] Bazel Developers. *Bazel*. Bazel. URL: https://bazel.build/.

[Devc] MATLAB Devs. *Col2im - Rearrange Matrix Columns into Blocks - MATLAB*. URL: https://uk.mathworks.com/help/images/ref/col2im.html.

[Dev+19] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. NAACL-HLT'19. June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[dev11] ONNX Runtime developers. *ONNX Runtime — Home*. 2011. URL: https://onnxruntime.ai/.

[Dev20] MLIR Developers. *'linalg' Dialect - MLIR*. 2020. URL: https://mlir.llvm.org/docs/Dialects/Linalg/.

[Dha+20] Prafulla Dhariwal et al. *Jukebox: A Generative Model for Music*. Apr. 30, 2020. URL: http://arxiv.org/abs/2005.00341.

[Di+20] Xinkai Di et al. "Exploring Efficient Acceleration Architecture for Winograd-Transformed Transposed Convolution of GANs on FPGAs". In: *Electronics* 9.2 (2 Feb. 2020), p. 286. ISSN: 2079-9292. DOI: 10.3390/electronics9020286.

[Dig] Digilent. *PYNQ-Z1 Reference Manual - Digilent Reference*. URL: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual.

[DLT16] Chao Dong et al. "Accelerating the Super-Resolution Convolutional Neural Network". In: *European Conference on Computer Vision*. ECCV'16. 2016, pp. 391–407. ISBN: 978-3-319-46475-6. DOI: 10.1007/978-3-319-46475-6_25.

[Don+18] Shi Dong et al. "Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Mar. 30, 2018, pp. 96–106. ISBN: 978-1-4503-5095-2. DOI: 10.1145/3184407.3184423.

[Don+21] Shi Dong et al. "Spartan: A Sparsity-Adaptive Framework to Accelerate Deep Neural Network Training on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (Oct. 2021), pp. 2448–2463. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3067825.

[Esm+12] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'12. Dec. 2012, pp. 449–460. DOI: 10.1109/MICRO.2012.48.

[Fah+21] Farah Fahim et al. *Hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices*. Mar. 23, 2021. DOI: 10.48550/arXiv.2103.05579.

[Fri+19] Geoffrey A. Fricker et al. "A Convolutional Neural Network Classifier Identifies Tree Species in Mixed-Conifer Forest from Hyperspectral Imagery". In: *Remote Sensing* 11.19 (19 Jan. 2019), p. 2326. ISSN: 2072-4292. DOI: 10.3390/rs11192326.

[Fuk+18] Tomohiro Fukuda et al. "DNN-Based Assistant in Laparoscopic Computer-Aided Palpation". In: *Frontiers in Robotics and AI* 5 (June 19, 2018). ISSN: 2296-9144. DOI: 10.3389/frobt.2018.00071.

[GC20] Perry Gibson and José Cano. "Orpheus: A New Deep Learning Framework for Easy Deployment and Evaluation of Edge Inference". In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Aug. 2020, pp. 229–230. DOI: 10.1109/ISPASS48437.2020.00042.

[GC23] Perry Gibson and José Cano. "Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. Jan. 27, 2023, pp. 28–39. ISBN: 978-1-4503-9868-8. DOI: 10.1145/3559009.3569682.

[Gen+21] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *2021 58th ACM/IEEE Design Automation Conference*. DAC'21. Dec. 5, 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[Ger24a] Georgi Gerganov. *Ggerganov/Ggml*. Aug. 20, 2024. URL: https://github.com/ggerganov/ggml.

[Ger24b] Georgi Gerganov. *Ggerganov/Llama.Cpp*. July 31, 2024. URL: https://github.com/ggerganov/llama.cpp.

[Ghe05] Frank Ghenassia. *Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. 2005. ISBN: 978-0-387-26232-1 978-0-387-26233-8. DOI: 10.1007/b137175.

[Gib+20] Perry Gibson et al. "Optimizing Grouped Convolutions on Edge Devices". In: *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP). July 2020, pp. 189–196. DOI: 10.1109/ASAP49362.2020.00039.

[Gib+25] Perry Gibson et al. "DLAS: A Conceptual Model for Across-Stack Deep Learning Acceleration". In: *ACM Trans. Archit. Code Optim.* 22.1 (Mar. 21, 2025), 1:1–1:28. ISSN: 1544-3566. DOI: 10.1145/3688609.

[GK22] In Gim and JeongGil Ko. "Memory-Efficient DNN Training on Mobile Devices". In: *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. MobiSys '22. June 27, 2022, pp. 464–476. ISBN: 978-1-4503-9185-6. DOI: 10.1145/3498361.3539765.

[Gon+14] Yunchao Gong et al. "Compressing Deep Convolutional Networks Using Vector Quantization". Dec. 18, 2014. URL: http://arxiv.org/abs/1412.6115.

[Goo+14] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Vol. 27. NIPS'14. 2014. URL: https://papers.nips.cc/paper_files/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

[Goo24] Google. *Gemmlowp*. Google, Aug. 19, 2024. URL: https://github.com/google/gemmlowp.

[Goz+20] Andrew Gozillon et al. "triSYCL for Xilinx FPGA". In: *Proceedings of the 2020 International Conference on High Performance Computing & Simulation (HPCS)* (Dec. 22, 2020).

[Gua+17] Yijin Guan et al. "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM'17. Apr. 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25.

[Gup+15] Suyog Gupta et al. "Deep Learning with Limited Numerical Precision". In: Feb. 9, 2015. URL: http://arxiv.org/abs/1502.02551.

[Had+18] Ramyad Hadidi et al. "Real-Time Image Recognition Using Collaborative IoT Devices". In: *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*. ReQuEST '18. June 20, 2018, p. 1. ISBN: 978-1-4503-5923-8. DOI: 10.1145/3229762.3229765.

[Had+19] Ramyad Hadidi et al. "Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices". In: *2019 IEEE International Symposium on Workload Characterization*. IISWC'19. Nov. 2019, pp. 35–48. DOI: 10.1109/IISWC47752.2019.9041955.

[Ham+20] Tae Jun Ham et al. "A^3: Accelerating Attention Mechanisms in Neural Networks with Approximation". In: *2020 IEEE International Symposium on High Performance Computer Architecture*. HPCA'20. Feb. 1, 2020, pp. 328–341. ISBN: 978-1-7281-6149-5. DOI: 10.1109/HPCA47549.2020.00035.

[Hao+19] Cong Hao et al. "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. June 2, 2019, pp. 1–6. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317829.

[Har+21] Jude Haris et al. "SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference". In: *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing*. SBAC-PAD'21. Oct. 2021, pp. 33–43. DOI: 10.1109/SBAC-PAD53543.2021.00015.

[Har+23] Jude Haris et al. "SECDA-TFLite: A Toolkit for Efficient Development of FPGA-based DNN Accelerators for Edge Inference". In: *Journal of Parallel and Distributed Computing* 173 (Mar. 1, 2023), pp. 140–151. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.11.005.

[Har+24a] Jude Haris et al. "Data Transfer Optimizations for Host-CPU and Accelerators in AXI4MLIR". In: *Compilers for Machine Learning Workshop*. C4ML'24. Feb. 29, 2024. DOI: 10.48550/arXiv.2402.19184.

[Har+24b] Jude Haris et al. "Designing Efficient LLM Accelerators for Edge Devices". In: *Addressing the Computing Requirements of LLMs and GNNs Workshop*. ARC-LG'24. Aug. 1, 2024. URL: http://arxiv.org/abs/2408.00462.

[HB20] Danny Hernandez and Tom B. Brown. *Measuring the Algorithmic Efficiency of Neural Networks*. May 8, 2020. DOI: 10.48550/arXiv.2005.04305.

[He+16] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR'16. June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[HGC23] Wenhao Hu et al. "ICE-Pick: Iterative Cost-Efficient Pruning for DNNs". In: ICML 2023 Workshop Neural Compression: From Information Theory to Applications. July 11, 2023. URL: https://openreview.net/forum?id=fWYKVtf7lu.

[HHC24] Wenhao Hu et al. "DQA: An Efficient Method for Deep Quantization of Deep Neural Network Activations". In: NeurIPS 2024 Workshop Machine Learning with New Compute Paradigms. Oct. 17, 2024. URL: https://openreview.net/forum?id=esMnmm2VJh.

[HMD16] Song Han et al. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *4th International Conference on Learning Representations*. ICLR'16. Feb. 15, 2016. URL: http://arxiv.org/abs/1510.00149.

[How+17] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Apr. 16, 2017. DOI: 10.48550/arXiv.1704.04861.

[Hsi+23] Samuel Hsia et al. "MP-Rec: Hardware-Software Co-design to Enable Multi-path Recommendation". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages*

and Operating Systems, Volume 3. ASPLOS'23. Mar. 25, 2023, pp. 449–465. ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582068.

[HVD15] Geoffrey Hinton et al. *Distilling the Knowledge in a Neural Network*. Mar. 9, 2015. DOI: 10.48550/arXiv.1503.02531.

[Int24] Intel Developers. *Overview of Intel's Neural Processing Unit (NPU)*. 2024. URL: https://intel.github.io/intel-npu-acceleration-library/npu.html.

[JAF16] Justin Johnson et al. "Perceptual Losses for Real-Time Style Transfer and Super-Resolution". In: *European Conference on Computer Vision*. ECCV'16. 2016, pp. 694–711. ISBN: 978-3-319-46475-6. DOI: 10.1007/978-3-319-46475-6_43.

[Jan+21] Jun-Woo Jang et al. "Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*. ISCA'21. June 2021, pp. 15–28. DOI: 10.1109/ISCA52012.2021.00011.

[JD18] John Hennessy and David Patterson. "A New Golden Age for Computer Architecture: Domain-specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*. ISCA'18. June 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011.

[Jia+14] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22nd ACM International Conference on Multimedia*. MM'14. Nov. 3, 2014, pp. 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889.

[Jia+20] Xiaoqi Jiao et al. "TinyBERT: Distilling BERT for Natural Language Understanding". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. EMNLP'2020. Nov. 2020, pp. 4163–4174. DOI: 10.18653/v1/2020.findings-emnlp.372.

[Jos23] Joséphus Cheung. *GuanacoDataset (Revision 892e57a)*. Hugging Face, 2023. DOI: 10.57967/hf/1423.

[Jou+17] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA'17. June 24, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[Jud+16a] Patrick Judd et al. "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets". Jan. 8, 2016. URL: http://arxiv.org/abs/1511.05236.

[Jud+16b] Patrick Judd et al. "Stripes: Bit-serial Deep Neural Network Computing". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'16. Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783722.

[Jum+21] John Jumper et al. "Highly Accurate Protein Structure Prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[Kar+18a] Sagar Karandikar et al. "Firesim: FPGA-accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA'18. June 2, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014.

[Kar+18b] Tero Karras et al. "Progressive Growing of GANs for Improved Quality, Stability, and Variation". In: *International Conference on Learning Representations*. ICLR'18. Feb. 15, 2018. URL: https://openreview.net/forum?id=Hk99zCeAb.

[KH08] Mohamed Khalil-Hani and Y.W. Hau. "SystemC HW/SW Co-Design Methodology Applied to the Design of an Elliptic Curve Crypto System on Chip". In: *2008 International Conference on Microelectronics*. ICM'08. Dec. 2008, pp. 147–150. DOI: 10.1109/ICM.2008.5393532.

[Kha+21] Hamza Khan et al. "NPE: An FPGA-based Overlay Processor for Natural Language Processing". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA'21. Feb. 17, 2021, p. 227. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439477.

[KLA21] Tero Karras et al. "A Style-Based Generator Architecture for Generative Adversarial Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.12 (Dec. 1, 2021), pp. 4217–4228. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2020.2970919.

[KLL23] Bokyung Kim et al. "INCA: Input-stationary Dataflow at Outside-the-box Thinking about Deep Learning Accelerators". In: *2023 IEEE International Symposium on High-Performance Computer Architecture*. HPCA'23. Feb. 2023, pp. 29–41. DOI: 10.1109/HPCA56546.2023.10070992.

[KSH17] Alex Krizhevsky et al. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 24, 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[KSK18] Hyoukjun Kwon et al. “MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS’18. Mar. 19, 2018, pp. 461–475. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173176.

[Kwo+19] Hyoukjun Kwon et al. “Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO’52. Oct. 12, 2019, pp. 754–768. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358252.

[Kwo+20] Hyoukjun Kwon et al. “MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings”. In: *IEEE Micro* 40.3 (May 2020), pp. 20–29. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2985963.

[Lai+19] Yi-Hsiang Lai et al. “HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA’19. Feb. 20, 2019, pp. 242–251. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293910.

[Lan+19] Zhenzhong Lan et al. “ALBERT: A Lite BERT for Self-supervised Learning of Language Representations”. In: *International Conference on Learning Representations*. ICLR’19. Sept. 25, 2019. URL: https://openreview.net/forum?id=H1eA7AEtvS.

[Lat+21] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO’21. Feb. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[LCO18] Manolis Loukadakis et al. “Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures”. In: Eleventh International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2018). Jan. 2018. URL: https://eprints.gla.ac.uk/183819/.

[Lec+98] Y. Lecun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[LG16] Andrew Lavin and Scott Gray. "Fast Algorithms for Convolutional Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR'16. June 2016, pp. 4013–4021. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.435.

[Li+20] Bingbing Li et al. "FTRANS: Energy-Efficient Acceleration of Transformers Using FPGA". In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED'20. Aug. 10, 2020, pp. 175–180. ISBN: 978-1-4503-7053-0. DOI: 10.1145/3370748.3406567.

[Liu+11] Ling Liu et al. "Automatic SoC Design Flow on Many-Core Processors: A Software Hardware Co-Design Approach for FPGAs". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA'11. Feb. 27, 2011, pp. 37–40. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950424.

[Liu+18] Shuanglong Liu et al. "Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA". In: *ACM Transactions on Reconfigurable Technology and Systems* 11.3 (Sept. 30, 2018), pp. 1–22. ISSN: 1936-7406, 1936-7414. DOI: 10.1145/3242900.

[Liu+19a] Lanlan Liu et al. "Generative Modeling for Small-Data Object Detection". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. ICCV'19. Oct. 2019, pp. 6072–6080. ISBN: 978-1-7281-4803-8. DOI: 10.1109/ICCV.2019.00617.

[Liu+19b] Leibo Liu et al. "A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications". In: *ACM Computing Surveys* 52.6 (Oct. 16, 2019), 118:1–118:39. ISSN: 0360-0300. DOI: 10.1145/3357375.

[Liu+23] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Advances in Neural Information Processing Systems* 36 (Dec. 15, 2023), pp. 21558–21572. URL: https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html.

[LLC21] Zejian Liu et al. “Hardware Acceleration of Fully Quantized BERT for Efficient Natural Language Processing”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition*. DATE'21. Feb. 2021, pp. 513–516. DOI: 10.23919/DATE51398.2021.9474043.

[Lu+17] Wenyan Lu et al. “FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks”. In: *2017 IEEE International Symposium on High Performance Computer Architecture*. HPCA'17. Feb. 2017, pp. 553–564. DOI: 10.1109/HPCA.2017.29.

[Lu+20] Siyuan Lu et al. “Hardware Accelerator for Multi-Head Attention and Position-Wise Feed-Forward in the Transformer”. In: *2020 IEEE 33rd International System-on-Chip Conference*. SOCC'20. Sept. 2020, pp. 84–89. DOI: 10.1109/SOCC49529.2020.9524802.

[Ma+22] Zhengzheng Ma et al. “An Intermediate-Centric Dataflow for Transposed Convolution Acceleration on FPGA”. In: *ACM Transactions on Embedded Computing Systems* (Sept. 1, 2022). ISSN: 1539-9087. DOI: 10.1145/3561053.

[Man+20] Paolo Mantovani et al. “Agile SoC Development with Open ESP”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD'20. Nov. 2, 2020, pp. 1–9. ISBN: 978-1-4503-8026-3. DOI: 10.1145/3400302.3415753.

[Mar+18] Stefano Markidis et al. “NVIDIA Tensor Core Programmability, Performance & Precision”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2018), pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[MMM17] Fabio Martinelli et al. “Evaluating Convolutional Neural Network for Effective Mobile Malware Detection”. In: *Procedia Computer Science*. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 21st International Conference, KES-20176-8 September 2017, Marseille, France 112 (Jan. 1, 2017), pp. 2372–2381. ISSN: 1877-0509. DOI: 10.1016/j.procs.2017.08.216.

[Mor+19] Thierry Moreau et al. “A Hardware–Software Blueprint for Flexible Deep Learning Specialization”. In: *IEEE Micro* 39.5 (Sept. 2019), pp. 8–16. ISSN: 1937-4143. DOI: 10.1109/MM.2019.2928962.

[Muñ+21] Francisco Muñoz-Martínez et al. “STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators”. In: *IEEE*

Computer Architecture Letters 20.2 (July 2021), pp. 122–125. ISSN: 1556-6064. DOI: 10.1109/LCA.2021.3097253.

[Muñ+23] Francisco Muñoz-Martínez et al. "Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS'23. Mar. 25, 2023, pp. 252–265. ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582069.

[Nga+20] Jennifer Ngadiuba et al. "Compressing Deep Neural Networks on FPGAs to Binary and Ternary Precision with Hls4ml". In: *Machine Learning: Science and Technology* 2.1 (Dec. 2020), p. 015001. ISSN: 2632-2153. DOI: 10.1088/2632-2153/aba042.

[Now+17] Tony Nowatzki et al. "Stream-Dataflow Acceleration". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA'17. June 24, 2017, pp. 416–429. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080255.

[NVIa] NVIDIA. *NVIDIA Deep Learning Accelerator*. URL: http://nvdla.org/.

[NVIb] NVIDIA. *NVIDIA TensorRT*. NVIDIA Developer. URL: https://developer.nvidia.com/tensorrt.

[Ott+20] G. Ottavi et al. "A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference". In: *2020 IEEE Computer Society Annual Symposium on VLSI*. ISVLSI'20. July 2020, pp. 512–517. DOI: 10.1109/ISVLSI49217.2020.000-5.

[Pas+17] Adam Paszke et al. "Automatic Differentiation in PyTorch". In: *Advances in Neural Information Processing Systems*. NIPS'17. Oct. 28, 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ.

[Pat+22] Suchita Pati et al. "Demystifying BERT: System Design Implications". In: *2022 IEEE International Symposium on Workload Characterization*. IISWC'22. Nov. 2022, pp. 296–309. DOI: 10.1109/IISWC55918.2022.00033.

[Pel+16] Maxime Pelcat et al. "Design Productivity of a High Level Synthesis Compiler versus HDL". In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. SAMOS'16. July 2016, pp. 140–147. DOI: 10.1109/SAMOS.2016.7818341.

[PG16] Lerrel Pinto and Abhinav Gupta. "Supersizing Self-Supervision: Learning to Grasp from 50K Tries and 700 Robot Hours". In: *2016 IEEE International Conference on Robotics and Automation*. ICRA'16. May 2016, pp. 3406–3413. DOI: 10.1109/ICRA.2016.7487517.

[Pra+17] Raghu Prabhakar et al. "Plasticine: A Reconfigurable Architecture For Parallel Paterns". In: *ACM SIGARCH Computer Architecture News* 45.2 (June 24, 2017), pp. 389–402. ISSN: 0163-5964. DOI: 10.1145/3140659.3080256.

[Qin+20] E. Qin et al. "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training". In: *2020 IEEE International Symposium on High Performance Computer Architecture*. HPCA'20. Feb. 2020, pp. 58–70. DOI: 10.1109/HPCA47549.2020.00015.

[Rad+19] Valentin Radu et al. "Performance Aware Convolutional Neural Network Channel Pruning for Embedded GPUs". In: *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 2019 IEEE International Symposium on Workload Characterization (IISWC). Nov. 2019, pp. 24–34. DOI: 10.1109/IISWC47752.2019.9042000.

[Raj+16] Pranav Rajpurkar et al. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. EMNLP'16. Nov. 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264.

[Ray23] Partha Pratim Ray. "ChatGPT: A Comprehensive Review on Background, Applications, Key Challenges, Bias, Ethics, Limitations and Future Scope". In: *Internet of Things and Cyber-Physical Systems* 3 (Jan. 1, 2023), pp. 121–154. ISSN: 2667-3452. DOI: 10.1016/j.iotcps.2023.04.003.

[RBA] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: ().

[RCO19] Simon Rovder et al. "Optimising Convolutional Neural Networks Inference on Low-Powered GPUs". In: Twelfth International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2019). 2019. URL: https://eprints.gla.ac.uk/183820/.

[Rey+20] Ruyman Reyes et al. “SYCL 2020: More than Meets the Eye”. In: *Proceedings of the International Workshop on OpenCL*. IWOCL’20. Apr. 27, 2020, p. 1. ISBN: 978-1-4503-7531-3. DOI: 10.1145/3388333.3388649.

[RF18] Qing Rao and Jelena Frtunikj. “Deep Learning for Self-Driving Cars: Chances and Challenges”. In: *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems*. SEFAIAS’18. May 2018, pp. 35–38. URL: https://ieeexplore.ieee.org/document/8452728.

[RFB15] Olaf Ronneberger et al. “U-Net: Convolutional Networks for Biomedical Image Segmentation”. In: *Medical Image Computing and Computer-Assisted Intervention*. MICCAI’15. 2015, pp. 234–241. ISBN: 978-3-319-24574-4. DOI: 10.1007/978-3-319-24574-4_28.

[RJJ24] Rappy Saha et al. “Accelerating PoT Quantization on Edge Devices”. In: *IEEE International Conference on Electronics Circuits and Systems. 2024*. ICECS’24. 2024.

[RMC16] Alec Radford et al. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *4th International Conference on Learning Representations*. ICLR’16. 2016. URL: http://arxiv.org/abs/1511.06434.

[Rus+15] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision* 115.3 (Dec. 1, 2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y.

[Sal+23] Caio Salvador Rohwedder et al. “To Pack or Not to Pack: A Generalized Packing Analysis and Transformation”. In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. CGO’23. Feb. 22, 2023, pp. 14–27. ISBN: 979-8-4007-0101-6. DOI: 10.1145/3579990.3580024.

[San+18] Mark Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. CVPT’18. June 2018, pp. 4510–4520. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00474.

[Ses+22] Kiran Seshadri et al. “An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks”. In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 2022 IEEE International Symposium on Workload Characterization (IISWC). Nov. 2022, pp. 79–91. DOI: 10.1109/IISWC55918.2022.00017.

[SFM17] Yongming Shen et al. "Maximizing CNN Accelerator Efficiency Through Resource Partitioning". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA'17. June 24, 2017, pp. 535–547. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080221.

[SGC22] Axel Stjerngren et al. "Bifrost: End-to-End Evaluation and Optimization of Reconfigurable DNN Accelerators". In: 2022 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS'22. May 1, 2022, pp. 288–299. ISBN: 978-1-6654-5954-9. DOI: 10.1109/ISPASS55109.2022.00042.

[SGS10] John E. Stone et al. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73. ISSN: 1558-366X. DOI: 10.1109/MCSE.2010.69.

[Sha+16a] Yakun Sophia Shao et al. "Co-Designing Accelerators and SoC Interfaces Using Gem5-Aladdin". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'16. Oct. 15, 2016, pp. 1–12.

[Sha+16b] Hardik Sharma et al. "From High-Level Deep Neural Models to FPGAs". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'16. Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783720.

[Sha+18] Hardik Sharma et al. "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA'18. June 2, 2018, pp. 764–775. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00069.

[Sha+23] Hesam Shabani et al. "HIRAC: A Hierarchical Accelerator with Sorting-based Packing for SpGEMMs in DNN Applications". In: *2023 IEEE International Symposium on High-Performance Computer Architecture*. HPCA'23. Feb. 2023, pp. 247–258. DOI: 10.1109/HPCA56546.2023.10070977.

[She20] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network". In: *Physica D: Nonlinear Phenomena* 404 (Mar. 1, 2020), p. 132306. ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306.

[Sim] Willison Simon. *Stanford Alpaca, and the Acceleration of on-Device Large Language Model Development*. URL: https://simonwillison.net/2023/Mar/13/alpaca/.

[Ska+18] Sam Skalicky et al. "Hot & Spicy: Improving Productivity with Python and HLS for FPGAs". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*. FCCM'18. Apr. 2018, pp. 85–92. DOI: 10.1109/FCCM.2018.00022.

[Son+20] Zhuoran Song et al. "DRQ: Dynamic Region-Based Quantization for Deep Neural Network Acceleration". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA'20. May 30, 2020, pp. 1010–1021. ISBN: 978-1-7281-4661-4. DOI: 10.1109/ISCA45697.2020.00086.

[SPS23] Cristian Sestito et al. "FPGA Design of Transposed Convolutions for Deep Learning Using High-Level Synthesis". In: *Journal of Signal Processing Systems* (Aug. 4, 2023). ISSN: 1939-8115. DOI: 10.1007/s11265-023-01883-7.

[Sun+20] Zhiqing Sun et al. "MobileBERT: A Compact Task-Agnostic BERT for Resource-Limited Devices". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. ACL'20. July 2020, pp. 2158–2170. DOI: 10.18653/v1/2020.acl-main.195.

[Sun+23] Xiaofei Sun et al. "Text Classification via Large Language Models". In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. EMNLP'23. Dec. 2023, pp. 8990–9005. DOI: 10.18653/v1/2023.findings-emnlp.603.

[SVL14] Ilya Sutskever et al. "Sequence to Sequence Learning with Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 27. NIPS'14. 2014. URL: https://proceedings.neurips.cc/paper_files/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

[Sze+15] Christian Szegedy et al. "Going Deeper with Convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). June 2015, pp. 1–9. ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298594.

[Sze+16] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR'16. June 2016, pp. 2818–2826. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.308.

[Sze+17] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (Dec. 2017), pp. 2295–2329. ISSN: 1558-2256. DOI: 10.1109/JPROC.2017.2761740.

[Ten] TensorFlow. *TensorFlow Lite Delegates*. TensorFlow. URL: https://www.tensorflow.org/lite/performance/delegates.

[TFL] TFLite Developers. *TensorFlow Lite 8-Bit Quantization Specification*. TensorFlow. URL: https://www.tensorflow.org/lite/performance/quantization_spec.

[The] The Linux Perf Team. *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[TK12] Vaishali Tehre and Ravindra Kshirsagar. "Survey on Coarse Grained Reconfigurable Architectures". In: *International Journal of Computer Applications* 48.16 (June 30, 2012), pp. 1–7. ISSN: 09758887. DOI: 10.5120/7429-0104.

[TL19] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. International Conference on Machine Learning. ICML'19. May 24, 2019, pp. 6105–6114. URL: https://proceedings.mlr.press/v97/tan19a.html.

[Tor24] Torch-MLIR Developers. *The Torch-MLIR Project*. Aug. 21, 2024. URL: https://github.com/llvm/torch-mlir.

[Tur+18] Jack Turner et al. "Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks". In: *2018 IEEE International Symposium on Workload Characterization*. IISWC'18. Sept. 2018, pp. 101–110. DOI: 10.1109/IISWC.2018.8573503.

[Umu+17] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA'17. Feb. 22, 2017, pp. 65–74. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021744.

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Dec. 4, 2017, pp. 6000–6010. ISBN: 978-1-5108-6096-4.

[Wan+16] Ying Wang et al. "DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. DAC'16. June 2016, pp. 1–6. DOI: 10.1145/2897937.2898002.

[Wan+17] Fei Wang et al. "Residual Attention Network for Image Classification". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR'17. July 2017, pp. 6450–6458. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.683.

[Wan+21] Yu Emma Wang et al. "Exploiting Parallelism Opportunities with Deep Learning Frameworks". In: *ACM Trans. Archit. Code Optim.* 18.1 (Dec. 30, 2021), 9:1–9:23. ISSN: 1544-3566. DOI: 10.1145/3431388.

[Wen+20] Jian Weng et al. "DSAGEN: Synthesizing Programmable Spatial Accelerators". In: *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. ISCA'20. May 30, 2020, pp. 268–281. ISBN: 978-1-7281-4661-4. DOI: 10.1109/ISCA45697.2020.00032.

[Wol+20] Thomas Wolf et al. "Transformers: State-of-the-Art Natural Language Processing". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. EMNLP'2020. Oct. 2020, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6.

[Xi+20] Sam (Likun) Xi et al. "SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads". In: *ACM Transactions on Architecture and Code Optimization* 17.4 (Nov. 10, 2020), 39:1–39:26. ISSN: 1544-3566. DOI: 10.1145/3424669.

[Xia+22] Shaojie Xiang et al. "HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA'22. Feb. 13, 2022, pp. 78–88. ISBN: 978-1-4503-9149-8. DOI: 10.1145/3490422.3502369.

[Xila] Xilinx. *Introduction • AXI DMA LogiCORE IP Product Guide (PG021) • Reader • AMD Technical Information Portal*. URL: https://docs.amd.com/r/en-US/pg021_axi_dma.

[Xilb] Xilinx. *Vivado Design Suite*. AMD. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[XLZ15] Jungang Xu et al. "An Overview of Deep Generative Models". In: *IETE Technical Review* 32.2 (Mar. 4, 2015), pp. 131–139. ISSN: 0256-4602. DOI: 10.1080/02564602.2014.987328.

[Xu+18] Dawen Xu et al. "FCN-Engine: Accelerating Deconvolutional Layers in Classic CNN Processors". In: *2018 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD'18. Nov. 2018, pp. 1–6. DOI: 10.1145/3240765.3240810.

[Xu+20] Pengfei Xu et al. "AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA'20. Feb. 24, 2020, pp. 40–50. ISBN: 978-1-4503-7099-8. DOI: 10.1145/3373087.3375306.

[Yan+18] Jiale Yan et al. "GNA: Reconfigurable and Efficient Architecture for Generative Network Acceleration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (Nov. 2018), pp. 2519–2529. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2857258.

[Yan+20] Xuan Yang et al. "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS'20. Mar. 9, 2020, pp. 369–383. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378514.

[Yao+24] Binwei Yao et al. *Benchmarking LLM-based Machine Translation on Cultural Awareness*. Mar. 22, 2024. DOI: 10.48550/arXiv.2305.14328.

[Ye+20] Hanchen Ye et al. "HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation". In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. DAC'20. Nov. 18, 2020, pp. 1–6. ISBN: 978-1-4503-6725-7.

[Yu+20] Yunxuan Yu et al. "Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.7 (July 2020), pp. 1545–1556. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2020.2995741.

[ZB19] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-Nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[Zha+15] Chen Zhang et al. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA'15. Feb. 22, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689060.

[Zha+16a] Shijin Zhang et al. "Cambricon-X: An Accelerator for Sparse Neural Networks". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO'16. Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723.

[Zha+16b] Ying Zhang et al. "Towards End-to-End Speech Recognition with Deep Convolutional Neural Networks". In: *Interspeech 2016* (Sept. 8, 2016), pp. 410–414. DOI: 10.21437/Interspeech.2016-1446.

[Zha+17] Xinyu Zhang et al. "A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA". May 7, 2017. URL: http://arxiv.org/abs/1705.02583.

[Zha+18] Xiaofan Zhang et al. "DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD'18. Nov. 2018, pp. 1–8. DOI: 10.1145/3240765.3240801.

[Zha+20a] Xiaofan Zhang et al. "DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD'20. Dec. 17, 2020, pp. 1–9. ISBN: 978-1-4503-8026-3. DOI: 10.1145/3400302.3415609.

[Zha+20b] Jerry Zhao et al. "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V*. CARRV'20. May 2020.

[Zha+22] Jin Zhao et al. "TDGraph: A Topology-Driven Accelerator for High-Performance Streaming Graph Processing". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA'22. June 11, 2022, pp. 116–129. ISBN: 978-1-4503-8610-4. DOI: 10.1145/3470496.3527409.

[Zha+24] Peiyuan Zhang et al. *TinyLlama: An Open-Source Small Language Model*. Jan. 4, 2024. DOI: 10.48550/arXiv.2401.02385.

[Zhe+22] Size Zheng et al. “AMOS: Enabling Automatic Mapping for Tensor Computations on Spatial Accelerators with Hardware Abstraction”. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA’22. June 18, 2022, pp. 874–887. ISBN: 978-1-4503-8610-4. DOI: 10.1145/3470496.3527440.

[Zho+17] Aojun Zhou et al. “Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights”. In: *5th International Conference on Learning Representations*. ICLR’17. 2017.