TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Implementation of complex Signal Processing applications using High Level Synthesis tools on a VERSAL platform.

Author:
Alexandros Andreas
STAVROPOULOS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Sotirios IOANNIDIS
Prof. Dimitrios SOUDRIS (NTUA)

A thesis submitted in fulfillment of the requirements
for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

October 7, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Implementation of complex Signal Processing applications using High Level Synthesis tools on a VERSAL platform.

by Alexandros Andreas STAVROPOULOS

The growing demand for efficient and scalable hardware architectures to support complex machine learning (ML) and signal processing applications has led to the exploration of versatile platforms like the AMD/Xilinx Versal Adaptive Compute Acceleration Platform (ACAP). This thesis investigates the performance of the Versal VCK190, focusing on its Network-on-Chip (NoC) architecture and its potential for accelerating convolutional neural networks (CNNs) and matrix multiplication tasks. By leveraging the Vitis and Vitis AI development environments, benchmarks for CNN models and matrix operations were deployed to evaluate the throughput and efficiency of the NoC. The results demonstrate the strengths of the Versal platform in managing high-performance AI workloads, while also revealing areas where optimizations could improve performance, particularly in real-time applications.

Key contributions of this work include the development of custom benchmarks for CNNs and matrix multiplication, a thorough evaluation of NoC throughput and performance analysis based on real-time AI inference. The findings offer valuable insights into the capabilities and limitations of the Versal architecture for heterogeneous computing applications.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Implementation of complex Signal Processing applications using High Level Synthesis tools on a VERSAL platform.

by Alexandros Andreas STAVROPOULOS

Η αυξανόμενη ζήτηση για αποδοτικές και κλιμακούμενες αρχιτεκτονικές υλικού για την υποστήριξη σύνθετων εφαρμογών μηχανικής μάθησης (ΜΜ) και επεξεργασίας σήματος έχει οδηγήσει στην εξερεύνηση ευέλικτων πλατφορμών όπως η πλατφόρμα AMD/Xilinx Versal Adaptive Compute Acceleration Platform (ACAP). Η παρούσα διπλωματική διερευνά τις επιδόσεις του Versal VCK190, εστιάζοντας στην αρχιτεκτονική του Network-on-Chip (NoC) και στις δυνατότητές του για την επιτάχυνση των συνελικτικών νευρωνικών δικτύων (ΣΝΔ) και των εργασιών πολλαπλασιασμού πινάκων. Αξιοποιώντας τα περιβάλλοντα ανάπτυξης Vitis και Vitis AI, αναπτύχθηκαν δείκτες αναφοράς για μοντέλα ΣΝΔ και πράξεις πινάκων για την αξιολόγηση της απόδοσης και της αποδοτικότητας του NoC. Τα αποτελέσματα καταδεικνύουν τα πλεονεκτήματα της πλατφόρμας Versal στη διαχείριση φορτίων εργασίας ΜΜ υψηλής απόδοσης, ενώ παράλληλα αποκαλύπτουν περιοχές όπου οι βελτιστοποιήσεις θα μπορούσαν να βελτιώσουν τις επιδόσεις, ιδίως σε εφαρμογές πραγματικού χρόνου.

Οι βασικές συνεισφορές αυτής της εργασίας περιλαμβάνουν την ανάπτυξη προσαρμοσμένων δοκιμών για ΣΝΔ και πολλαπλασιασμό πινάκων, ενδελεχή αξιολόγηση της απόδοσης του NoC και ανάλυση επιδόσεων με βάση την εξαγωγή συμπερασμάτων σε πραγματικό χρόνο. Τα ευρήματα προσφέρουν πολύτιμες πληροφορίες σχετικά με τις δυνατότητες και τους περιορισμούς της αρχιτεκτονικής Versal για εφαρμογές ετερογενών υπολογιστών.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Professor Apostolos Dollas for believing in me and providing the opportunity to undertake this thesis. His unwavering support and passion for hardware have been a source of inspiration and have significantly deepened my enthusiasm for this field. I am also deeply thankful to Professor Dimitrios Soudris for placing his trust in me and welcoming me into his lab, as well as for his invaluable contributions and insightful suggestions throughout my work. In addition, I am sincerely grateful to Professor Sotirios Ioannidis for his time and effort in reviewing this thesis.

A special note of appreciation goes to Professor Giorgos Lentaris, who has been an exceptional mentor from the very beginning. His consistent guidance and steadfast support have been instrumental in my journey; without his mentorship, this thesis would not have come to fruition. I am equally grateful to the Microprocessor Lab for providing a supportive and collaborative environment. In particular, I extend my heartfelt thanks to Ilias Papalabrou and Panagiotis Chaidos for their generous assistance and for helping me overcome numerous challenges with their expertise and dedication. I also wish to acknowledge Vasilis Leon, who supported me during the early stages of this work, and though our paths diverged, he has remained a trusted friend and source of guidance.

I would also like to extend my gratitude to Alexandros Paterakis, whose recent experience in completing his own thesis proved to be invaluable. His advice and problem-solving insights greatly facilitated the successful completion of this project.

Last but certainly not least, I would like to express my deepest appreciation to my family and friends for their unwavering support, encouragement and mostly patience. Their belief in me, especially during difficult times, has been a cornerstone of my success and personal growth.

Contents

List of Figures

List of Tables

List of Abbreviations

ACE	**A**XI **C**oherent **E**xtension
ACAP	**A**daptive **C**ompute **P**latform
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
AI	**A**rtificial **I**ntelligence
APU	**A**pplication **P**rocessing **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced **X**temsible **I**nterface
CAN-FD	**C**ontroller **A**rea **N**etwork **F**lexible **D**ata-Rate
CLB	**C**onfigurable **L**ogic **B**lock
CPU	**C**entral **P**rocessing **U**nit
DFx	**D**esigned **F**or e**X**ellence
DDRMC	**D**ouble **D**ata **R**ate **M**emory **C**ontroller
DPU	**D**eep-Learning **P**rocessing **U**nit
DSP	**D**igital **S**ignal **P**rocessor
ECC	**E**rror **C**orrection **C**ode
FPGA	**F**ield **P**rogrammable **G**ate **A**rrays
FIFO	**F**irst **I**n **F**irst **O**ut
GEMM	**G**eneral **M**atrix **M**ultiplication
GPU	**G**raphics **P**rocessing **U**nit
HNOC	**H**orizontal **N**etwork **O**n **C**hip
HLS	**H**igh **L**evel **S**ynthesis
LUT	**L**ook **U**p **T**able
MAC	**M**ultiply **AC**ummulate
ML	**M**achine **L**earning
NMU	**N**oC **M**aster **U**nit
NPD	**N**oC **P**acketised **D**ata
NPI	**N**oC **P**acket **I**nterface
NPP	**N**oC **P**acket **P**rotocol
NPS	**N**oC **P**acket **S**witch
NSU	**N**oC **S**lave **U**nit
OCM	**O**n **C**hip **M**emory
PCIe	**P**eripheral **C**omponent **I**nterconnect **E**xpress

PLL	**P**hase **L**ocked **L**oop
PMC	**P**latform **M**anagement **C**ontroller
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
RPU	**R**eal-time **P**rocessing **U**nit
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTL	**R**egister **T**ransistor **L**evel
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SPI	**S**erial **P**eripheral **I**nterface
TCM	**T**ightly **C**oupled **M**emory
TPU	**T**ensor **P**rocessing **U**nit
USB	**U**niversal **S**erial **B**us
VNOC	**V**ertical **N**etwork **O**n **C**hip
VLIW	**V**ery **L**ong **I**nstruction **W**ord
XRT	**X**ilinx **R**un **T**ime

To those closest to my heart, with gratitude and love.

Chapter 1

Introduction

Rapid advances in Artificial Intelligence (AI) and Machine Learning (ML) have revolutionised many devices and systems, driven by significant improvements in algorithms and specialised hardware architectures. Traditional CPU advances, driven by Moore's Law, have reached their physical limits, leading to the rise of GPUs for ML due to their many simple processing units while companies such as Google have developed specialised hardware such as the Tensor Processing Unit (TPU). However, the rapid scaling of technology nodes poses significant challenges, particularly in the semiconductor industry, where AI training requirements are doubling every 3.5 months, far outpacing Moore's Law. This surge in algorithmic innovation and data proliferation has exponentially increased computing requirements, creating an urgent need for advanced hardware solutions. The end of the hardware optimisation renaissance marked by Dennard's scaling and Amdahl's law necessitates the exploration of domain-specific architectures such as DSPs, GPUs and FPGAs. Despite their theoretical computational power, these architectures face performance limitations due to memory system gaps and the need for specialised expertise [1].

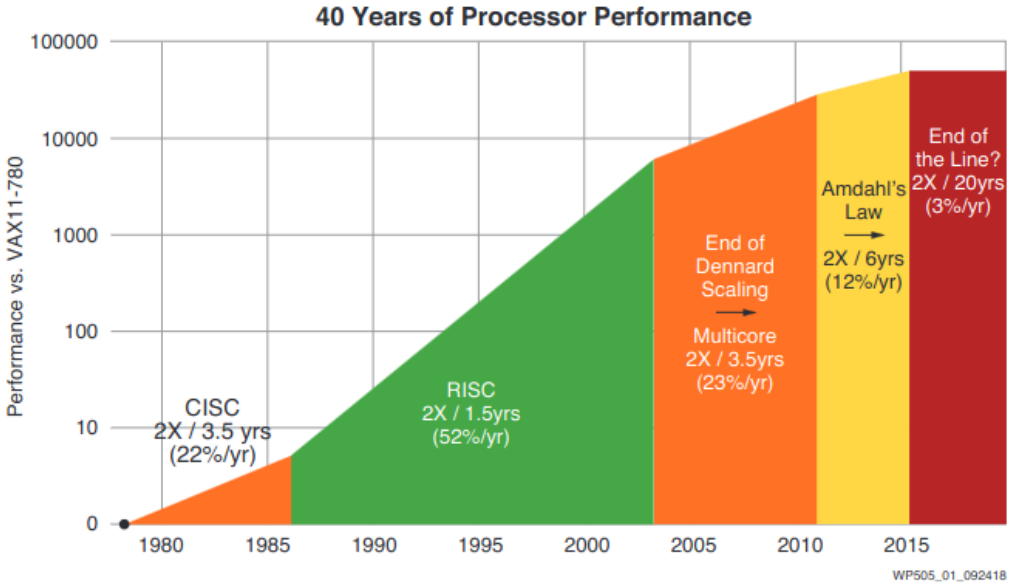


FIGURE 1.1: Processor Performance vs Time (Source:[1])

1.1 Motivation

This increasing demand for more advanced systems that run ML applications has led to the prioritisation of heterogeneous architectures and hardware accelerators in the context of technological advancement. Among these, AMD/Xilinx's Adaptive Compute Acceleration Platform (ACAP), exemplified by the Versal family, stands out. Versal ACAPs integrate Scalar Engines, Adaptable Engines and Intelligent Engines, all connected via a high-bandwidth Network-on-Chip (NoC). This architecture provides software programmability, heterogeneous acceleration and dynamic adaptability, significantly improving performance in applications such as AI, ML, 5G, and automotive systems. The unified tool-chain simplifies development, enabling both software and hardware developers to leverage the platform's capabilities without extensive hardware expertise. Versal's AI engines accelerate AI inference and complex data processing tasks, which are critical for edge devices that require real-time predictive capabilities. By providing a programmable memory hierarchy and customisation options, Versal's ACAPs address economic, technological and ease-of-use challenges, facilitating high-volume adoption across multiple domains.

However, the practical effectiveness of Versal's NoC, particularly for real-time applications with stringent timing constraints, requires thorough investigation. In this thesis, a throughput-based evaluation of the Network-on-Chip (NoC) is conducted. The evaluation involves deploying parts of a Convolutional Neural Network (CNN) as well as entire CNN models using both Vitis and Vitis AI tools. The performance is assessed by measuring the throughput achieved during the deployments, comparing it to AMD/Xilinx specifications.

1.2 Thesis Contributions

- Investigated and designed architectures simulating real-world applications, with a primary focus on advanced mapping techniques in the highly heterogeneous AMD/Xilinx Versal VCK190 platform.
- Evaluated the Network on Chip (NoC) performance using both AMD/Xilinx tools, optimizing for real-time AI inference within complex, heterogeneous system architectures.
- Performed in-depth performance analysis and throughput benchmarking, specifically targeting the hard-IP NoC to assess its efficiency under AI workloads.
- Provided critical insights into the strengths and constraints of the Versal NoC architecture when applied to sophisticated AI processing tasks.

1.3 Thesis Outline

- **Chapter 2 - Background:** Chapter 2 provides an in-depth analysis of the Versal VCK190 hardware architecture, including AI Engines, Programmable Logic and Network on Chip architecture. Additionally, this chapter introduces and contextualizes the software frameworks and tools (Vitis AI, Vitis) employed for benchmarking and testing.
- **Chapter 3 - Related Work:** Chapter 3 provides a comprehensive review of existing literature and technologies relevant to the thesis. It explores key developments in convolutional neural networks (CNNs), hardware-accelerated matrix multiplication and Network-on-Chip (NoC) architectures highlighting the advancements that have shaped the design of modern AI accelerators.
- **Chapter 4 - Methodology:** Chapter 4 presents the methodologies used in both approaches. It details the selection process for the Convolutional Neural Network (CNN) in the first approach and the matrix multiplication algorithm in the second. Additionally, it addresses key considerations such as data types and provides an outline of the testing methodology employed.
- **Chapter 5 - Development and Implementation:** Chapter 5 outlines the detailed implementation process for each developed system, including both CNN-based and matrix multiplication designs. The chapter highlights modifications made to AI Engines, Programmable Logic, and system components, with visual aids to support the explanation.
- **Chapter 6 - Evaluation:** Chapter 6 offers a comprehensive overview of the board and the test setup. It presents the results from each individual test case and includes a comparison with the expected values.
- **Chapter 7 - Conclusions and Future work:** Chapter 7 presents the concluding insights and outcomes of this thesis, while also outlining potential future work to further enhance this research.

Chapter 2

Background

This chapter introduces the essential background needed for the thesis. It covers the Versal SoC architecture and introduces the tools and frameworks used for testing, explaining their application. This chapter also covers basic concepts of neural networks.

2.1 Versal Adaptive SoC: Hardware Architecture

2.1.1 Overview

The Versal Adaptive Compute Acceleration Platform (ACAP) from AMD/Xilinx has been designed to provide cutting-edge heterogeneous acceleration for a wide range of applications, from edge to cloud. These ACAPs tightly integrate Scalar Engines (Processor System), Adaptable Engines (Programmable Logic), and Adaptive Intelligent Engines (AI Engines), all seamlessly interconnected through a hard-IP Network-on-Chip (NoC). To support such complex system, a suite of tools, libraries and APIs is provided, enhancing the programmability of the device to achieve optimal performance. This thesis focuses on the Versal VCK190, a device delivering high AI inference and signal processing throughput for both cloud and edge applications [2]. The following sections provide a detailed description of its major resource blocks.

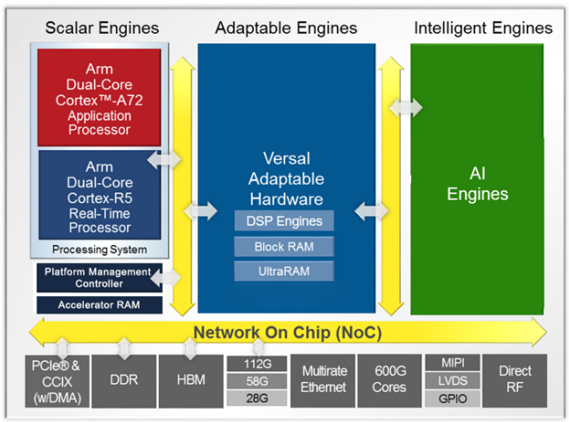


FIGURE 2.1: Versal Top-Level Architecture (Source: [2])

2.1.2 AI Engines (Adaptable Intelligent Engines)

The *AI Engines* in the Versal VCK190 exemplify AMD/Xilinx's innovative response to the growing demand for computing power as Moore's Law nears its limits. These engines are organized into an array of AI Engine tiles and Interface tiles. Specifically, the VCK190 board features 400 AI Engine tiles arranged in an 8x50 2D array configuration, alongside 50 Interface tiles [3].

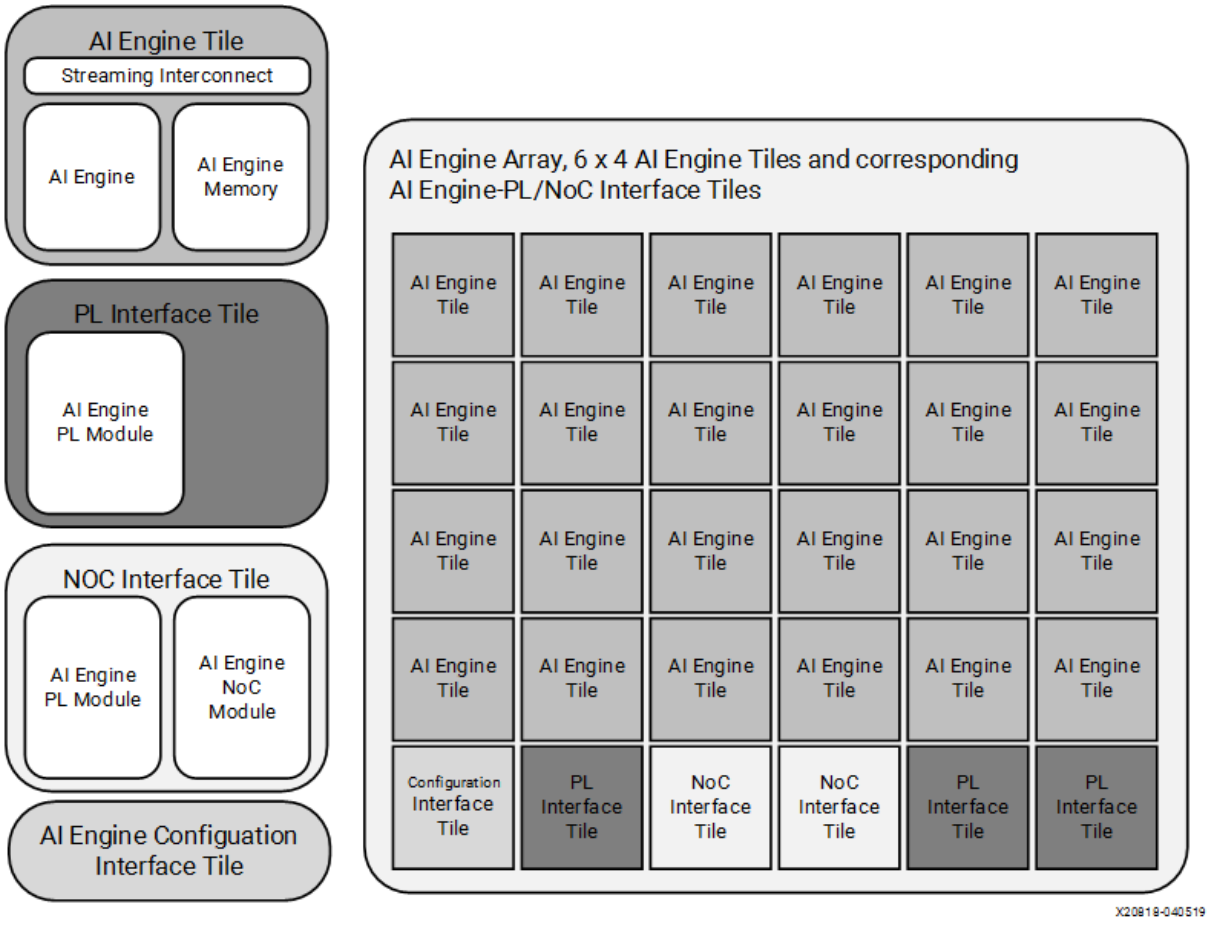


FIGURE 2.2: Hierarchy of Tiles in an AI Engine Array (Source: [3])

AI Engine Tiles

Each AI Engine tile integrates a 32-bit scalar RISC processor capable of non-linear functions and datatype conversion between scalar fixed-point and scalar floating-point numbers (scalar unit) as well as a 512-bit vector processor capable of both fixed-point and floating-point operations (vector unit). The system comprises three address generation units (AGU), dedicated program memory (16KB) and data memory (32KB divided into eight banks), both incorporating error handling and correction. Additionally, it includes a dedicated instruction fetch and decode unit capable of issuing up to six operations in parallel using a VLIW (Instruction Fetch and Decode Unit) [3].

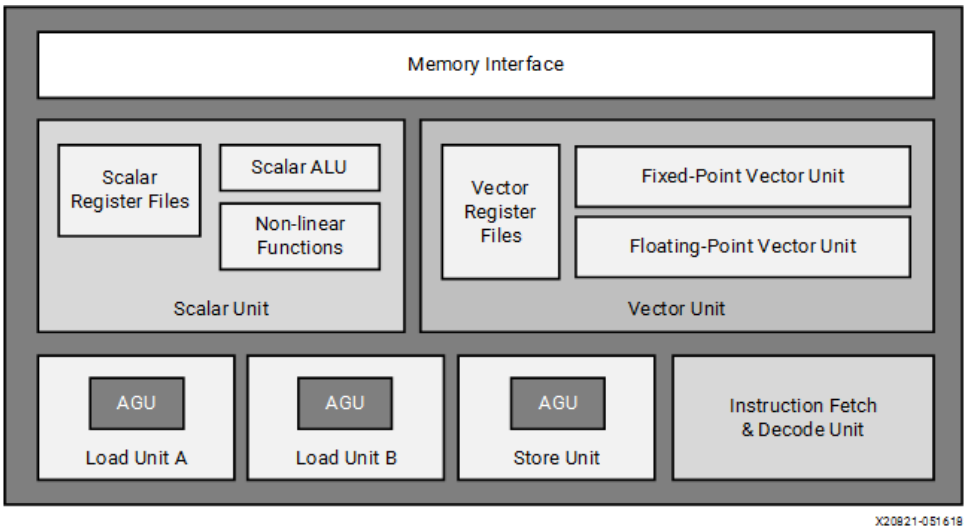


FIGURE 2.3: Inner Architecture of Tiles in a AI Engine Array (Source: [3])

The AI Engines employ a Single Instruction Multiple Data (SIMD) and Very Long Instruction Word (VLIW) architecture, clocked at 1GHz+ (capability to reach up to 1.25GHz). This architecture enables the execution of up to six concurrent instructions per clock cycle, allowing for the simultaneous execution of multiple scalar operations, vector load and write operations, and fixed or floating-point vector operations. [3].AI Engine iles

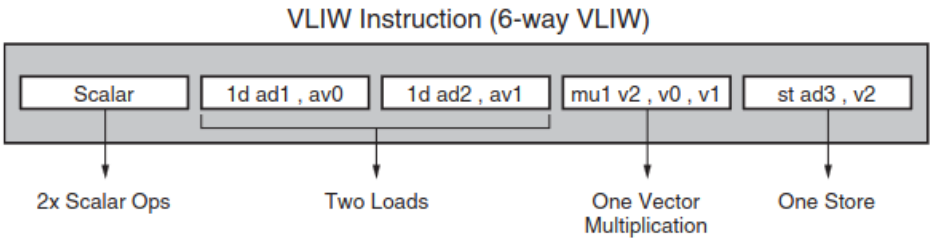
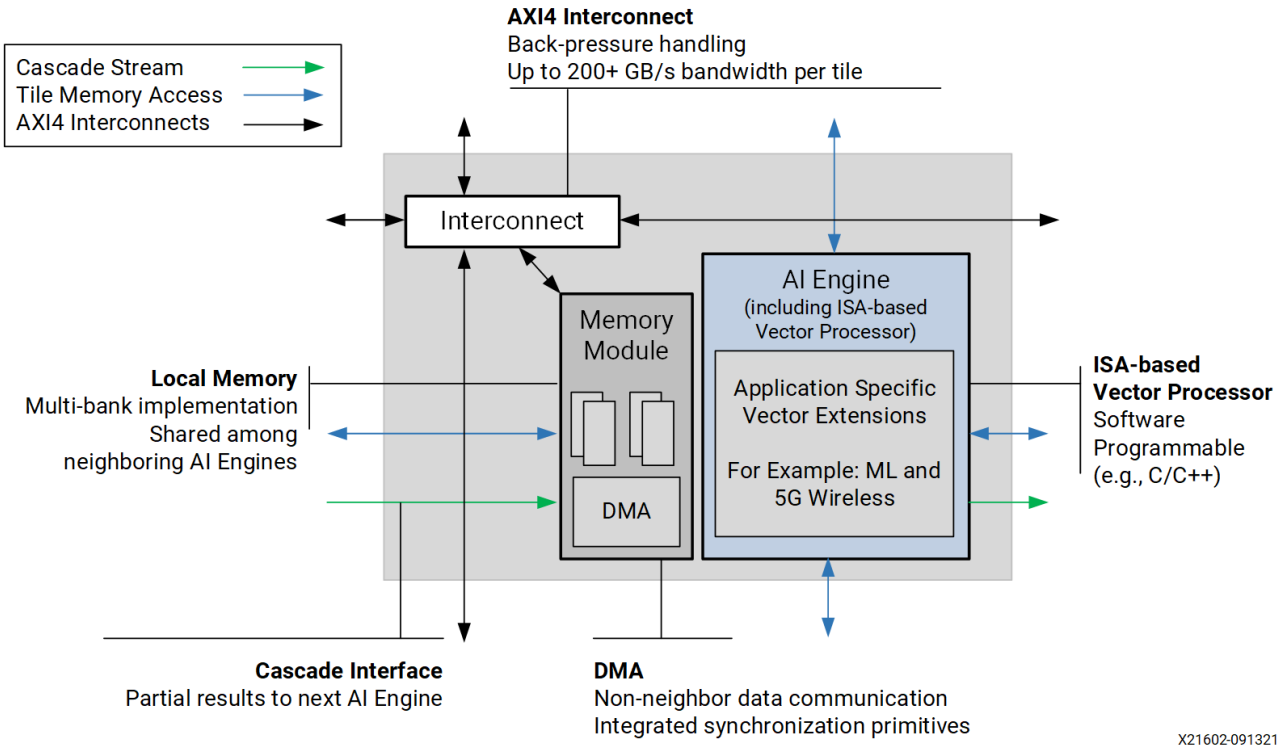


FIGURE 2.4: AI Instruction Level Parallelism (Source: [3])

Interconnection of AI Engine tiles is achieved through a combination of dedicated AXI4 bus routing and direct links to neighbouring tiles, greatly enhancing trace and debug capabilities. Each AI Engine has access to its own memory and can connect to up to three neighbouring modules (north, south, east or west), although edge engines are constrained by a checkerboard pattern, resulting in fewer options. This architecture also enables seamless data transfer between non-neighbouring AI Engine tiles. Additionally, between the AI Engine and external components such as Programmable Logic (PL) or Network-on-Chip (NoC) connections are handled via interface tiles managed by AXI4-Stream and memory-mapped interfaces. This flexible interconnect design ensures optimal performance and scalability. In addition, the 384-bit cascade streams traverse horizontally, changing direction at the edge of each row, alternating the placement of AI engines and memory modules until reaching the top row where the stream ends [3].



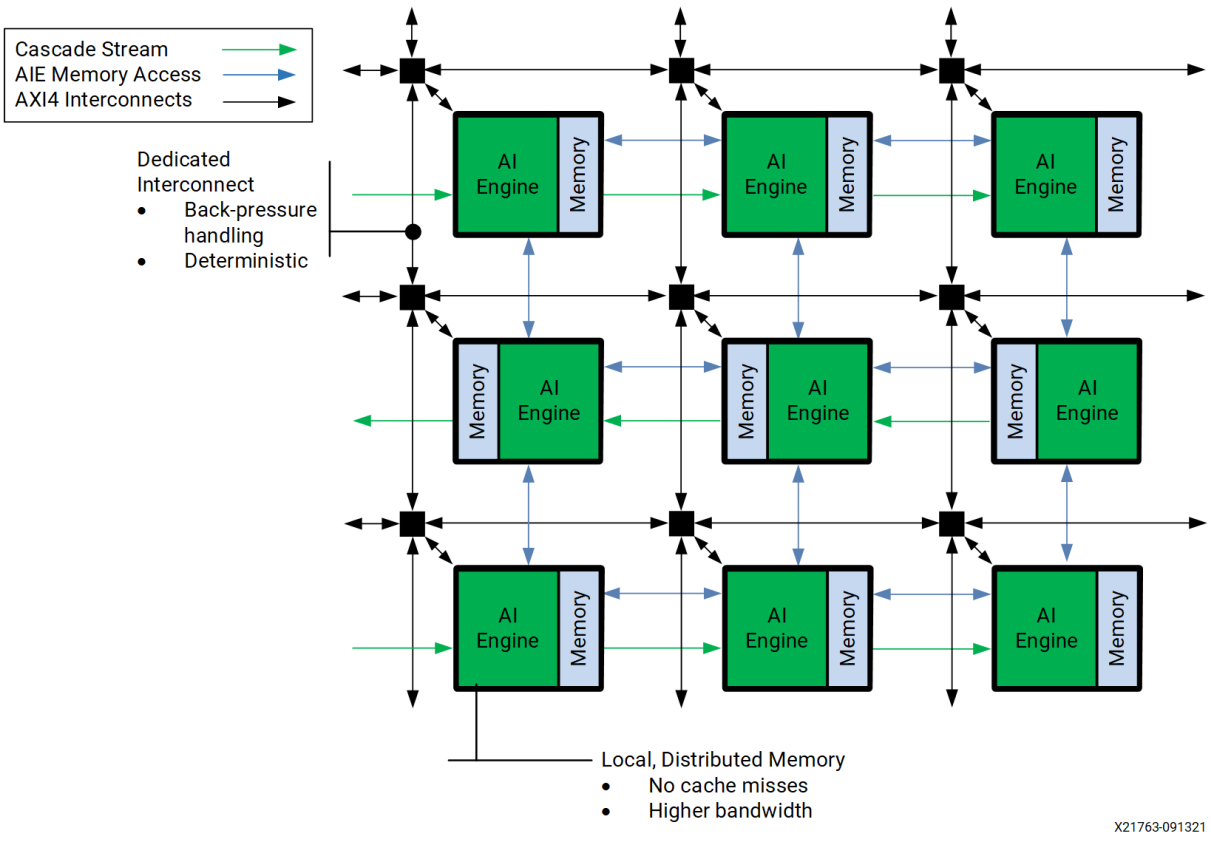


FIGURE 2.5: AI Engine Tile Interconnects (Source: [3])

AI Engine Interface Tiles

As illustrated in Figure 2.2, the bottom row houses specialised interface tiles that are crucial for seamless device integration. All three types of tiles, utilise AXI4-Stream interconnects in order to enable efficient communication within the system [3].

The *PL Interface Tile* houses a PL module with a Memory-mapped AXI4 and an AXI4-Stream switch, alongside control, debug and trace units, aided by asynchronous FIFOs for clock domain crossing. It manages six streams from AI Engine to PL and eight from PL to each AI Engine column. Each AXI4-Stream interface supports significant bandwidth: 24 GB/s from AI Engine to PL and 32 GB/s from PL to AI Engine. With 39 array interface tiles, the PL interface achieves impressive throughput of 1.0 TB/s from AI Engine to PL and 1.3 TB/s from PL to AI Engine, assuming a 1 GHz clock [3].

The *NoC Interface Tile* serves to connect the horizontal NoC (HNoC) with four streams between the AI Engine and the NoC. It integrates a PL module and a NoC module with interfaces to the NoC master unit (NMU) and NoC slave unit (NSU). The tile facilitates interrupt handling and activates interrupt lines within the AI Engine array interface, linking the AI Engine configuration interface tile [3].

The *AI Engine configuration interface tile*, found once per AIE array, serves as a central hub for critical operations. It is responsible for the generation of the AI Engine clock via a PLL and oversees the essential global controls, encompassing interrupt handling, global reset control and DFx logic [3].

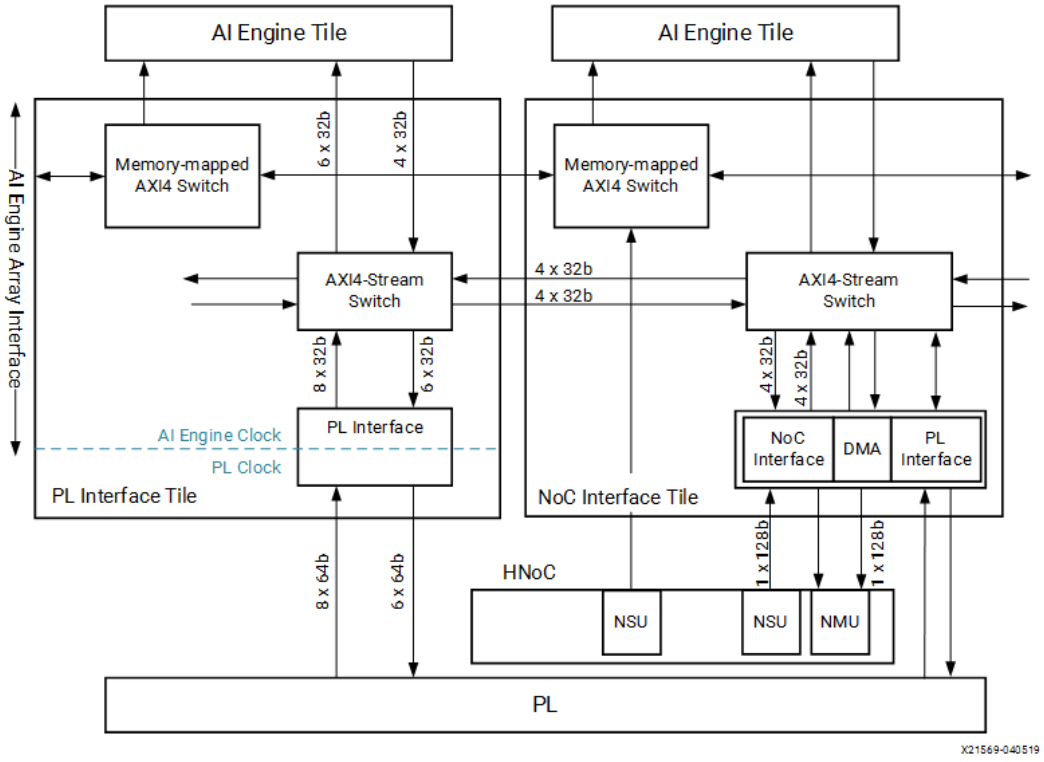


FIGURE 2.6: AI Engine Interface Tile Architecture (Source: [3])

2.1.3 Programmable Logic (PL)

The *Programmable Logic* (PL) component within Versal ACAPs incorporates FPGA functionality, providing a versatile fabric for efficient hardware implementation of specialised functions. Versal's PL, with a maximum clock speed of 500MHz, contains sophisticated Configurable Logic Blocks (CLBs), internal memory resources, DSP engines and a variety of interfaces for seamless integration with other hardware components such as the AI engines and processing System (PS) [4].

Configurable Logic Block (CLB)

Each *Configurable Logic Block* (CLB) in Versal ACAPs comprises 32 Look-Up Tables (LUTs) and 64 flip-flops, providing a robust foundation for diverse logic operations. LUTs can function individually as a 6-input LUT or as dual 5-input LUTs with shared inputs or can be registered within flip-flop. The CLB also incorporates arithmetic carry logic, multiplexers and flexible configuration options for RAM and shift registers, utilising LUTs. Additionally, there are dedicated interconnect paths between LUTs within the block, enabling flexible carry chain initiation at any bit position [4].

Memory Resources

The PL component exhibits a diverse array of internal memory options, demonstrating the versatility and robust data processing capabilities. Particularly noteworthy is the incorporation of on-chip memory (OCM) equipped with error correction code (ECC) and accelerator RAM, which ensures efficient storage and error correction mechanisms. Furthermore, the incorporation of true dual-port block RAMs (36 Kb with ECC, configurable into two independent 18 Kb RAMs), facilitating seamless data handling. These integrated memory features enhance the performance and adaptability, rendering it an optimal choice for a diverse range of computing tasks [4].

Digital Signal Processing (DSP)

The *Digital Signal Processing* (DSP) engines within the Versal devices provide a robust solution to signal processing tasks, combining speed, compactness and flexibility. Each engine includes a 27×24-bit multiplier and 58-bit accumulator with dynamic bypass capability. Additional features such as pre-adders, XOR functions and pipelining enhance performance. In addition to fixed-point operations, these engines support various modes such as vector dot products and single-precision floating-point arithmetic, extending their application beyond traditional signal processing [4].

2.1.4 Processor System (PS)

The *Processing System* (PS) in the Versal VCK190 comprises multiple sophisticated units, each designed for specific processing tasks and efficient communication. Key components include the Application Processing Unit (APU), the Real-Time Processing Unit (RPU) and connectivity peripherals such as a USB 2.0 controller and Ethernet MACs. These components integrate seamlessly with the Programmable Logic (PL) through various high-speed interfaces, ensuring robust and versatile system performance [4].

Application Processing Unit (APU)

The *Application Processing Unit* (APU) in the Versal VCK190 employs a dual-core Arm Cortex-A72 processor, leveraging the 64-bit Arm-v8A architecture. Each Cortex-A72 core is equipped with 48 KB of instruction L1 cache and 32 KB of data L1 cache, NEON SIMD engines and floating point units, thereby enhancing the APU's computational capability. The APU incorporates a snoop control unit and a 1 MB L2 cache with ECC protection, which enhances system-level performance and coherency. It communicates with the remainder of the PS through a 128-bit AXI coherent extension (ACE) port [4].

Real-Time Processing Unit (RPU)

The *Real-Time Processing Unit* (RPU) is comprised of dual-core Arm Cortex-R5F processors based on the 32-bit Arm-v7R architecture. Each core has 32 KB of L1 instruction and data caches and a 128 KB tightly coupled memory (TCM) interface for real-time single-cycle access. The RPU can operate in split or lock-step mode and communication with the PS is conducted via 128-bit AXI-4 ports [4].

Connectivity

The Versal VCK190 offers an extensive range of connectivity options, including CAN-FD, SPI, USB, Ethernet, I2C, and UART interfaces, making it highly versatile for various communication needs. It integrates essential components such as gigabit Ethernet controllers and a USB 2.0 controller capable of operating in both host and device modes, supporting data transfer speeds of up to 480 Mb/s. Additionally, the platform includes two tri-speed Ethernet MACs, supporting 10 Mb/s, 100 Mb/s, and 1 Gb/s connections, with advanced features like jumbo frame support to enhance network performance.[4].

DDR Memory Controller (DDRMC)

The integrated *DDR Memory Controllers* (DDRMCs) in the Versal VCK190 support DDR4 and LPDDR4/4X memory interfaces with four NoC interface ports. They offer five Quality of Service (QoS) classes for command prioritisation and handle burst transactions with command reordering for efficiency. The controller are designed to accommodate a variety of memory topologies and data widths and it features ECC support, address parity and a range of reliability features, including on-the-fly/background scrubbing, error logging and error injection for writes. The controllers guarantee efficient memory operations by implementing read-after-write and write-after-write hazard checking, as well as maintaining AXI ID ordering [5].

2.1.5 Deep-Learning Processing Unit (DPU)

The *Deep Learning Processing Unit* (DPU) found on the Versal VCK190 (DPUCVDX8G) represents a highly configurable computation engine, optimised for the processing of convolutional neural networks. The DPUCVDX8G has been designed for incorporation into Versal ACAP devices and offers considerable flexibility in terms of parallelism, which can be adapted to align with the specific requirements of the device and the intended application. The device is compatible with a multitude of neural network models, including VGG, ResNet, GoogLeNet, YOLO and MobileNet among others. The unit features a variety of interfaces, including AXI4-Lite, which facilitates access to configuration and status registers, with comprehensive support for a diverse range of operations, including convolution, pooling, activation functions and element-wise operations [6].

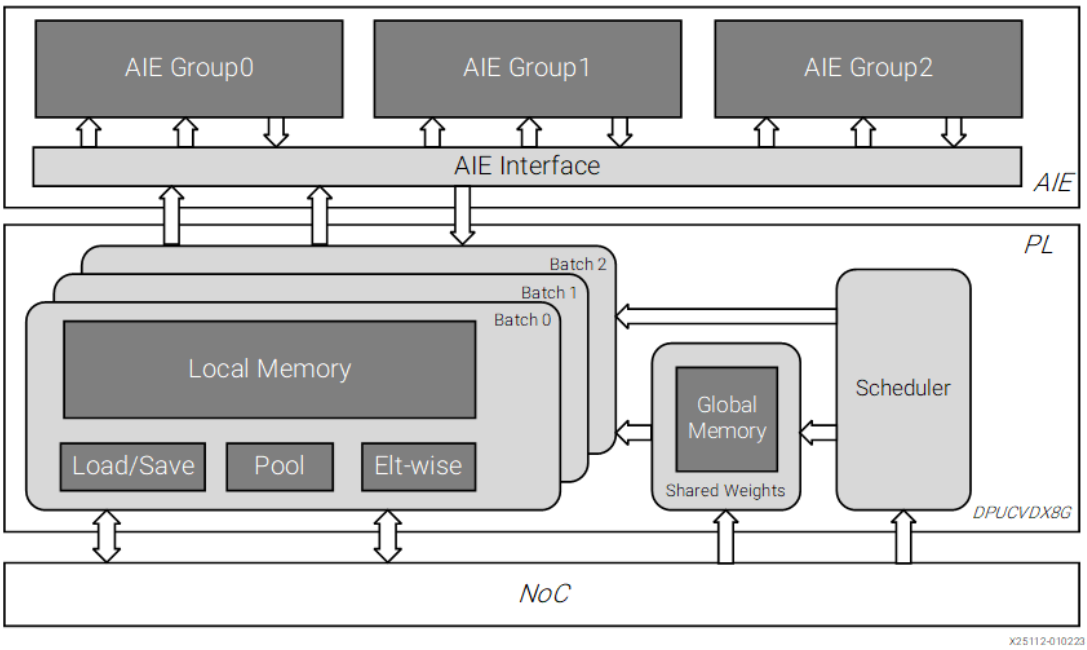


FIGURE 2.7: DPU Block Diagram (Source: [6])

2.1.6 Network-on-Chip (NOC)

The Versal VCK190, is equipped with a cutting-edge *Network-on-Chip* (NoC) infrastructure. This infrastructure is designed for seamless data sharing among integrated IP endpoints within its Programmable Logic (PL), processing system (PS), AI Engines (AIE) and other blocks. The high-speed, integrated data path, with dedicated switching, facilitates efficient communication while preserving PL resources while End-to-end Quality of Service (QoS) ensures effective traffic management, balancing latency and bandwidth requirements. The scalability of the NoC is achieved through a network of interconnected horizontal (HNoC) and vertical (VNoC) paths, complemented by customisable hardware components. This architecture embodies the advanced capabilities of the Versal VCK190 for adaptable and efficient system-level integration [5].

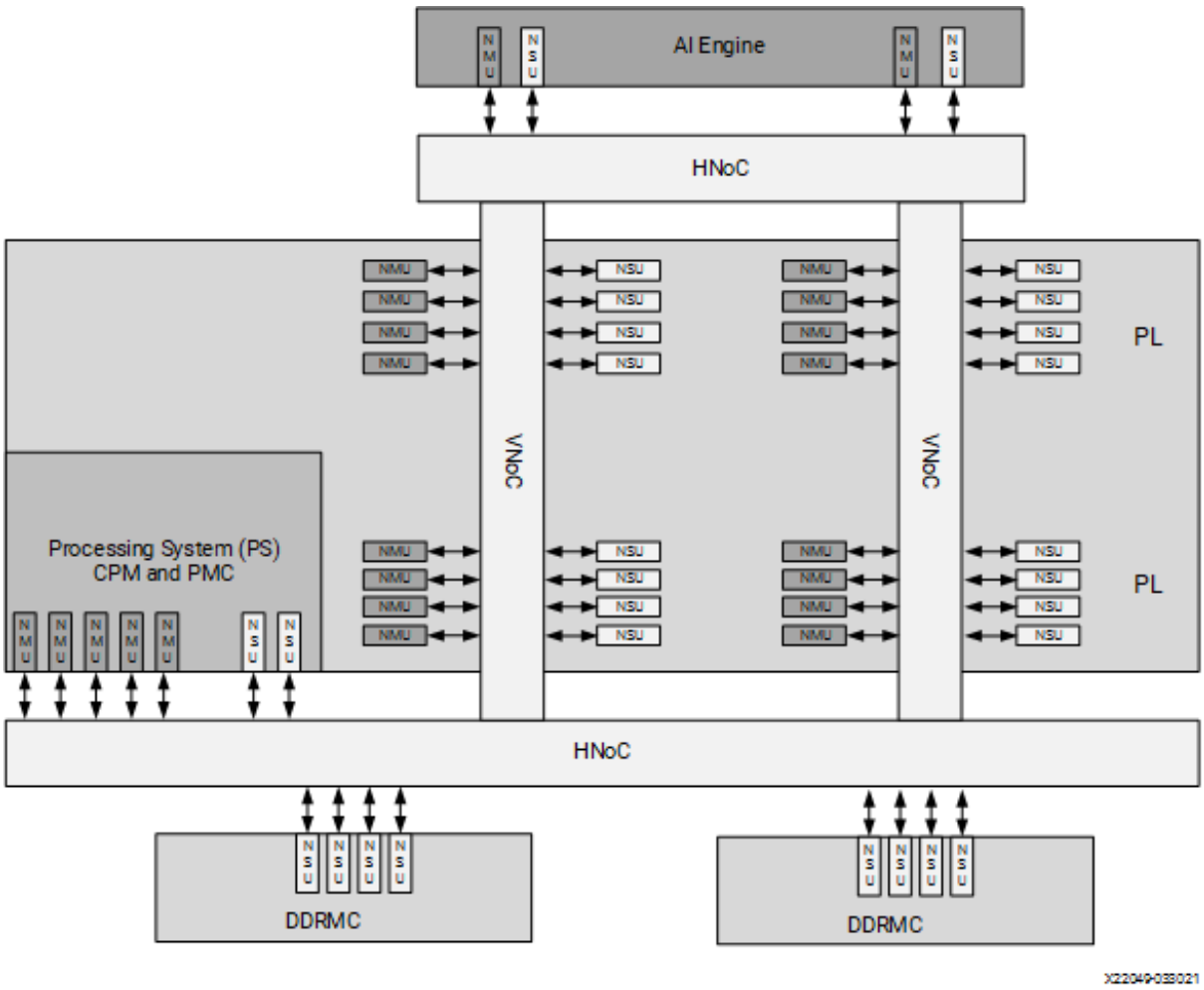


FIGURE 2.8: NoC Block Diagram (Top level) (Source: [5])

Horizontal and Vertical NoC (HNoC and VNoC)

The *horizontal and vertical network-on-chip* (HNoC and VNoC) architectures play a pivotal role in modern system-on-chip (SoC) designs, offering high-bandwidth and scalable interconnect solutions. HNoCs are positioned on both the top and bottom of the die and include multiple bi-directional physical NoC channels. These connect various blocks, including the Processing System (PS), the Programmable Logic (PL), the Adaptive Intelligence Engines (AI Engines) as well as the Platform Management Controller (PMC), PCIe and the DDRMC [5].

In contrast, VNoCs are vertical columns with bi-directional channels that primarily connect to the Programmable Logic (PL) areas. The interconnected paths are designed to handle diverse traffic types efficiently, maintaining low latency and high throughput across the chip, thus forming a comprehensive NoC system when combined [5].

NoC Function Blocks

NoC Master Units (NMU)

The *NoC Master Unit* (NMU) serves as the gateway for data entering the NoC, facilitating interconnection between the AXI master and the NoC through asynchronous clock domain crossing and rate matching. It translates AXI protocol transactions into NoC Packet Protocol (NPP) and supports various burst types with interface widths from 32 to 512 bits. The NMU is capable of handling up to 64 AXI reads and writes, featuring a 512-byte write buffer and DDR controller interleaving support. It performs address matching, route control and manages ingress Quality of Service (QoS), ensuring prioritised and efficient data traffic management [5].

NoC Slave Unit (NSU)

The *NoC Slave Unit* (NSU) serves as the egress point from the NoC, converting NoC packetised data (NPD) to and from AXI protocol data. It supports asynchronous clock domain crossing and rate matching between the AXI slave and the NoC. As the NMU, the NSU is capable of handling access with interface widths from 32 to 512 bits (AXI) and 128 to 512 bits (AXI4-Stream) interfaces. It is also equipped to support AXI4, AXI4-Stream, and AXI-ID compression, with up to 32 outstanding read and write transactions simultaneously. The NSU de-packetises received NoC data packets, converts them into AXI transactions, and manages the asynchronous data crossing to ensure efficient communication with the AXI master interface [5].

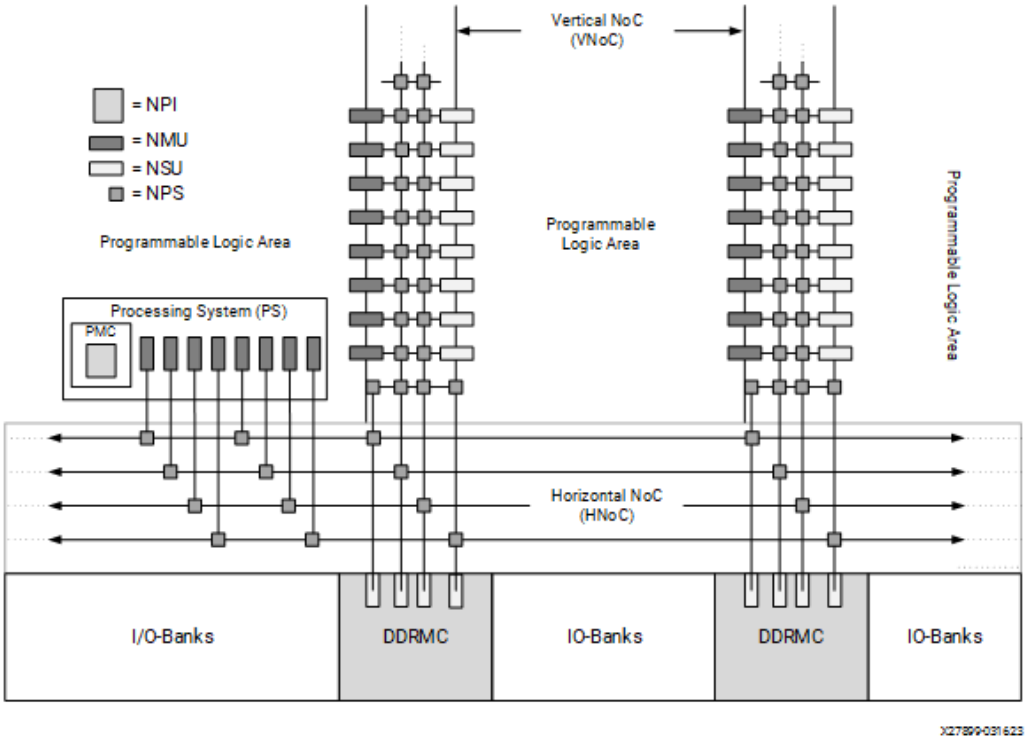


FIGURE 2.9: NoC Block Diagram (Source: [5])

NoC Packet Switch (NPS)

The *NoC Packet Switch* (NPS) serves to connect the NMU and NSU interfaces within the NoC. Each NPS is a full-duplex 4x4 switch, with each port supporting eight virtual channels in both directions. Each port is fully buffered and contains eight FIFOs, one per virtual channel, and employs credit-based flow control. The switch ensures a minimum latency of two cycles and supports configurable Quality of Service (QoS). The routing table, which determines the output port for incoming packets based on their destination ID, is programmable at boot time via the NPI and can be reprogrammed when the NoC is quiescent. The HNoC ports are equipped with a seven-deep FIFO, while the VNoC ports have a five-deep FIFO [5].

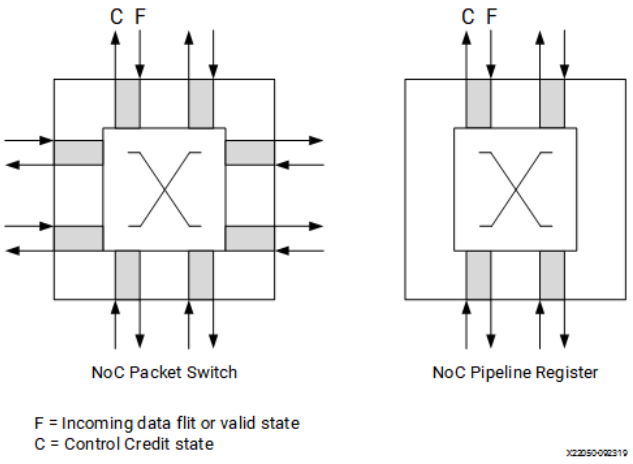


FIGURE 2.10: NoC Packet Switch (Source: [5])

2.2 Benchmarking Designs

Since the primary objective of this thesis is to evaluate the NoC performance of the VCK190, it is essential to develop a robust test methodology focused on measuring achievable throughput. Throughput is a critical metric for assessing the efficiency and performance of the NoC, making it a key factor in the overall evaluation process. The following chapter presents a comprehensive examination of both the specialized benchmarks provided by AMD/Xilinx as well as custom benchmarking approaches that are independent of the target platform. This includes an in-depth analysis of matrix multiplication and the implementation of a convolutional neural network (CNN), both serving as key benchmarks for evaluating throughput.

2.2.1 Benchmarks Provided by AMD/Xilinx

In order to achieve the objective of measuring the NoC's throughput, it is essential to generate a substantial amount of traffic over the NoC. AMD/Xilinx offers a range of pre-existing IPs, with the Performance AXI Traffic Generator (TG) representing a fundamental instrument developed with such specific purpose. This IP supports AXI3, AXI4 and AXI4-Stream protocols and is available in both non-synthesizable and synthesizable versions, thereby offering flexibility for both simulation and hardware testing. The TG is capable of generating configurable AMBA traffic patterns. Its key features include the maximisation of bandwidth, configurable transaction delays and data integrity checking, enabling comprehensive and accurate NoC performance testing [7].

The effectiveness of the TG is further enhanced when used in conjunction with the NoC AXI Performance Monitor (AXI_pmon) IP, which collects and reports performance metrics such as bandwidth and latency. This combination provides a detailed evaluation of NoC performance, allowing designers to optimise their systems for maximum efficiency and throughput [7].

2.2.2 Custom Benchmarking Approaches

While built-in tools, such as the TG, can yield valuable insights, they often rely on isolated tests that fail to capture the complexity of real-world applications. To achieve a more accurate performance evaluation, it is essential to consider more realistic scenarios as benchmarks. Given that Versal ACAPs are designed for machine learning applications, deploying a representative machine learning application would serve as an effective benchmark.

Neural Networks

Convolutional Neural Networks (CNNs) have become the cornerstone of many modern machine learning applications, particularly in the field of image and video recognition, natural language processing and various other domains. CNNs are designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers.

At the core, convolutional layers apply filters to the input image, capturing features like edges and textures. Pooling layers follow, reducing the spatial dimensions of the representation to decrease the number of parameters and computation. Finally, fully connected layers perform high-level reasoning to output class scores or other outcomes.

Beyond image recognition, CNNs excel in natural language processing for tasks like text classification and sentiment analysis, and in medical imaging for detecting abnormalities. They are also pivotal in autonomous driving for object detection and in recommendation systems for analyzing visual content.

Given their target applications, Convolutional Neural Networks (CNNs) often need to process large amounts of data under stringent timing requirements, especially in real-time applications. This necessitates a high level of optimization for the functions used in CNNs, making them an ideal subject for achieving the objectives of this thesis.

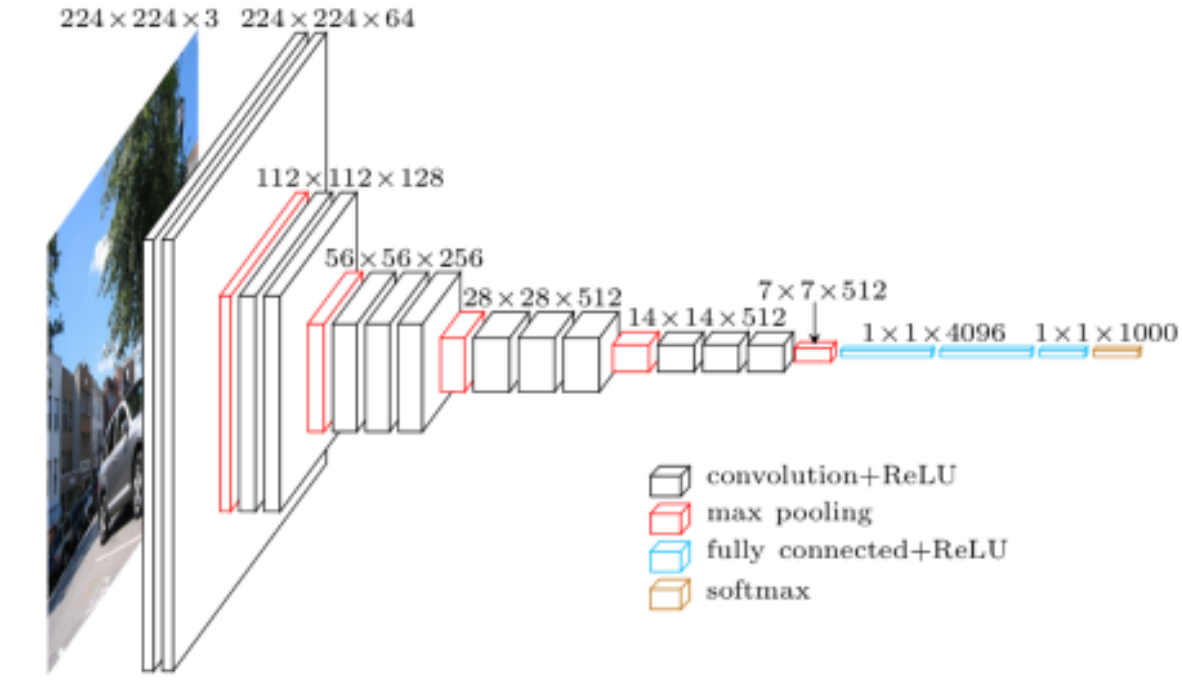


FIGURE 2.11: Neural Network Layer Architecture (Source: https://arxiv.org/abs/1409.1556v6)

Matrix Multiplication

Matrix multiplication plays a pivotal role in the performance of Convolutional Neural Networks (CNNs), often accounting for a significant portion of the execution time. This is primarily due to its heavy utilization in core operations such as weight matrix multiplications and convolution layers, which are commonly reformulated as matrix operations to improve computational efficiency. This transformation has spurred significant efforts to optimize matrix multiplication and boost overall performance.

As a mathematical operation it is straightforward. Given two matrices—one with dimensions $n \times m$ and the other with dimensions $m \times k$—matrix multiplication can be performed since the number of columns in the first matrix matches the number of rows in the second.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}_{n \times m} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} \\ b_{21} & b_{22} & \cdots & b_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mk} \end{bmatrix}_{m \times k}$$

Then for each value of the output matrix, the sum between a row from the first matrix and a column from the second, is calculated. This operation results in a new matrix with dimensions $n \times k$.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}_{n \times m} \quad \text{and} \quad B = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mk} \end{bmatrix}_{m \times k}$$

The resulting matrix C from the multiplication $C = A \times B$ will be:

$$C = A \times B = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1k} \\ c_{21} & c_{22} & \cdots & c_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nk} \end{bmatrix}_{n \times k} \quad \text{where element } c_{ij} \text{ is computed as: } c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj}$$

In the field of hardware, the implementation of matrix multiplication, commonly referred to as MAC (multiply and accumulate), involves several steps. First, the two operands are fetched, then their product is computed and finally, the product is added to an internal accumulator. The accumulator stores the cumulative result of previous MAC operations or starts with an initial value. This process is repeated m times and the final accum value represents the outcome.

$$\text{accum} = \text{accum} + (\text{oper\_A} \cdot \text{oper\_B})$$

In order to further optimize it, firstly, as each output value in the matrix is independent of the others, parallelising multiple MACs can be advantageous. Secondly, by vectorising both matrices and using entire vectors instead of individual values, the number of required memory accesses is significantly reduced. Thirdly, the most optimised approach involves tiling the matrix multiplication. In this method, the initial matrices are divided into smaller tiles, allowing for more efficient computation.

$$A = \begin{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} & \begin{bmatrix} A_{13} & A_{14} \\ A_{23} & A_{24} \end{bmatrix} \\ \begin{bmatrix} A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix} & \begin{bmatrix} A_{33} & A_{34} \\ A_{43} & A_{44} \end{bmatrix} \end{bmatrix} \text{ and } \quad B = \begin{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} & \begin{bmatrix} B_{13} & B_{14} \\ B_{23} & B_{24} \end{bmatrix} \\ \begin{bmatrix} B_{31} & B_{32} \\ B_{41} & B_{42} \end{bmatrix} & \begin{bmatrix} B_{33} & B_{34} \\ B_{43} & B_{44} \end{bmatrix} \end{bmatrix}$$

The output matrix C is comprised in this case by four individual tiles. Each tile is calculated by summing the products of the corresponding tiles from A and B. For example:

$$C_{T11} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} + \begin{bmatrix} A_{13} & A_{14} \\ A_{23} & A_{24} \end{bmatrix} \cdot \begin{bmatrix} B_{31} & B_{32} \\ B_{41} & B_{42} \end{bmatrix}$$

which essentially result to:

$$C_{T11} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix} + \begin{bmatrix} A_{13}B_{31} + A_{14}B_{41} & A_{13}B_{32} + A_{14}B_{42} \\ A_{23}B_{31} + A_{24}B_{41} & A_{23}B_{32} + A_{24}B_{42} \end{bmatrix}$$

Finally, the output matrix is calculated as:

$$C = \begin{bmatrix} C_{T11} & C_{T12} \\ C_{T21} & C_{T22} \end{bmatrix} \text{ where } \quad C_{T11} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

This approach effectively parallelizes the computation while simultaneously minimizing memory accesses, thereby enhancing the overall efficiency of the process.

2.3 Software Tools and Frameworks

This chapter introduces the fundamental tools provided by AMD/Xilinx for the development of applications on the Versal platform as well as the PyTorch and TensorFlow frameworks. Vitis Unified Software Platform represents a central component development process, offering a comprehensive environment for integrating the various aspects of software and hardware development for heterogeneous computing applications. Vitis facilitates a software-centric approach, allowing developers to work at their preferred level of abstraction, whether for embedded software or hardware acceleration. Furthermore, the Vitis AI development environment has been designed with AI inference in mind, offering a range of libraries that facilitate the development of AI applications. Additionally, the PyTorch and TensorFlow frameworks provide a dynamic and flexible platform for developing machine learning models. Both frameworks are compatible with Vitis AI further enhancing the workflow and enabling seamless deployment of models onto the VCK190.

2.3.1 Vitis Unified Software Platform

Overview

Vitis Unified Integrated Development Environment (IDE) by AMD/Xilinx is a comprehensive platform designed to streamline the development of applications for AMD/Xilinx hardware, including FPGAs, SoCs and Versal ACAPs. This unified software platform integrates multiple tools to facilitate the creation, analysis and optimization of applications, catering to both software and hardware engineers [8].

Vitis provides advanced tools for designing and optimizing custom accelerators. The Vitis HLS (High-Level Synthesis) tool enables the creation of hardware modules from high-level code, significantly reducing development time compared to traditional RTL (Register-Transfer Level) design methods. The IDE's comprehensive debugging, profiling and performance analysis tools are essential for optimizing both software and hardware components. Xilinx Run-Time library (XRT) abstracts the complexities of hardware management, providing a consistent programming model across various platforms [8].

The Vitis Unified IDE's integration with the Xilinx Vivado Design Suite ensures a seamless transition from design to implementation, allowing detailed exploration and fine-tuning of hardware architecture. This integration, combined with the IDE's robust feature set, accelerates the development cycle, enhances productivity and enables developers to fully utilize AMD/Xilinx hardware for a wide array of applications, from high-performance computing to embedded systems [8].

Workflow

In order to actually implement an application, a specific workflow must be followed. A high level diagram helps understand the required steps - complexity of such design:

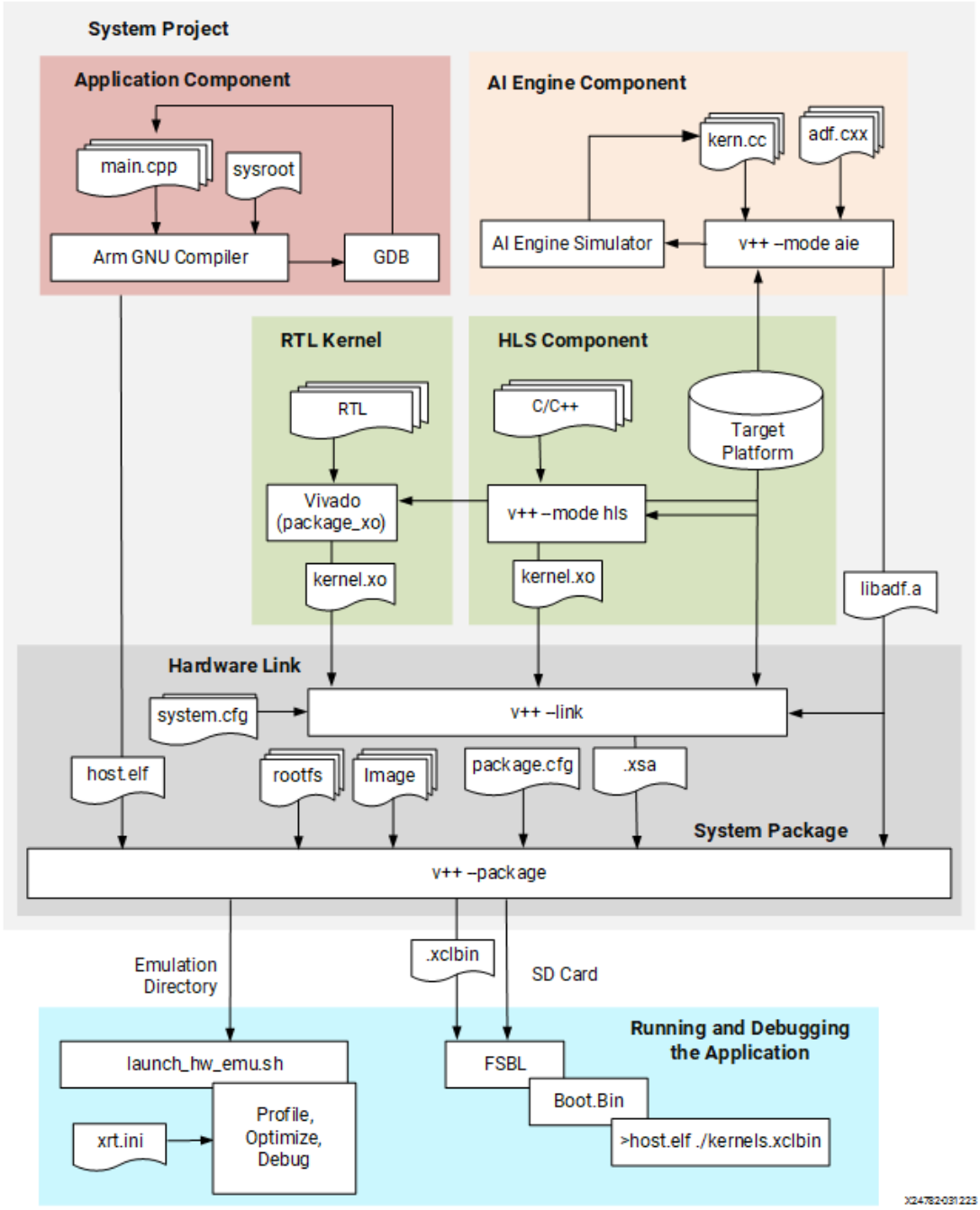


FIGURE 2.12: Vitis Workflow (Source: [8])

1. **Identify Performance-Critical Sections:** Identifying the application segments that require acceleration. Subsequently, it is necessary to evaluate which of these segments should be implemented on the Programmable Logic (PL) and which on the AI Engines. This decision is dependent on the specific characteristics of each task and the distinctive capabilities of each building block.

2. **Design and Implement Accelerators (PL and AIE):** Developing accelerated components using the following approaches:
    - For the PL components, it is necessary to use C/C++ High-Level Synthesis (HLS). In order to optimise performance, it is possible to use specialised pragmas. Each component should be validated rigorously using dedicated test benches in order to ensure correct functionality.
    - For the AI Engine graph, representing the component running on the AI Engines, it is necessary to use C/C++ and ideally intrinsics, low-level machine instruction enabling highly optimised and efficient kernel execution. AMD/Xilinx offers a comprehensive API that includes a large number of intrinsics, for a variety of task. The compilation of the graph is performed using the V++ (AI Engines compiler). Each AIE graph should undergo extensive testing through the AI Engines simulator.
3. **Host Application (PS):** The development of the host application for the Processor System is also a crucial element of the project. The host application is responsible for the initialization, control, and termination of all the required buffers, Programmable Logic (PL) components, and AI Engine (AIE) graphs. Additionally, it handles data indexing and executes all tasks that are not accelerated. Written in C/C++, the host application utilizes several libraries provided by AMD/Xilinx, including the specialized XRT library.
4. **Creating the System Project:** Combining the AI Engine graph, the PL components, the host application and performing the proper indexing between PL components and AI Engine graph, results in the final system application. This application facilitates all the necessary components for the system to function.
5. **Test Through Emulation:** Testing the entire system is essential. Software emulation simulates components within a software environment, thereby providing rapid feedback on functionality and identifying logical or linking errors. In contrast, hardware emulation enhances testing by increasing the accuracy of the hardware, uncovering more hardware-centric issues and offering a closer approximation of real-world performance.
6. **Hardware Deployment:**The process of compiling the system for hardware deployment is a time-consuming one, requiring the synthesis of components for physical implementation using Vitis and Vivado tools. The resulting Linux image file is then written to an SD card, allowing the application to be deployed on the Versal device for comprehensive real-world testing and validation.

2.3.2 Vitis AI

Overview

Vitis AI is an AI inference development platform by AMD/Xilinx designed to accelerate deep learning applications on their devices, including the Versal ACAP devices. It offers optimized IP, libraries, frameworks and example designs to streamline AI model creation, optimization and deployment [9].

Key features include the Vitis AI Compiler and XRT Library for optimizing models from popular frameworks, including TensorFlow and PyTorch, for high-performance execution on AMD/Xilinx hardware. The Vitis AI Model Zoo provides ready-to-use models, while the Optimizer and Quantizer offer advanced pruning and quantization to reduce computational demands with minimal accuracy loss [9].

The platform's debugging and profiling tools, such as the Vitis AI Profiler, help developers fine-tune models for optimal performance. The XRT library simplifies hardware complexity with a unified programming model for seamless AI application deployment across devices [9].

Workflow

The Vitis AI development workflow consists of several stages, each crucial for successfully deploying AI models on Xilinx hardware. Below are the detailed steps:

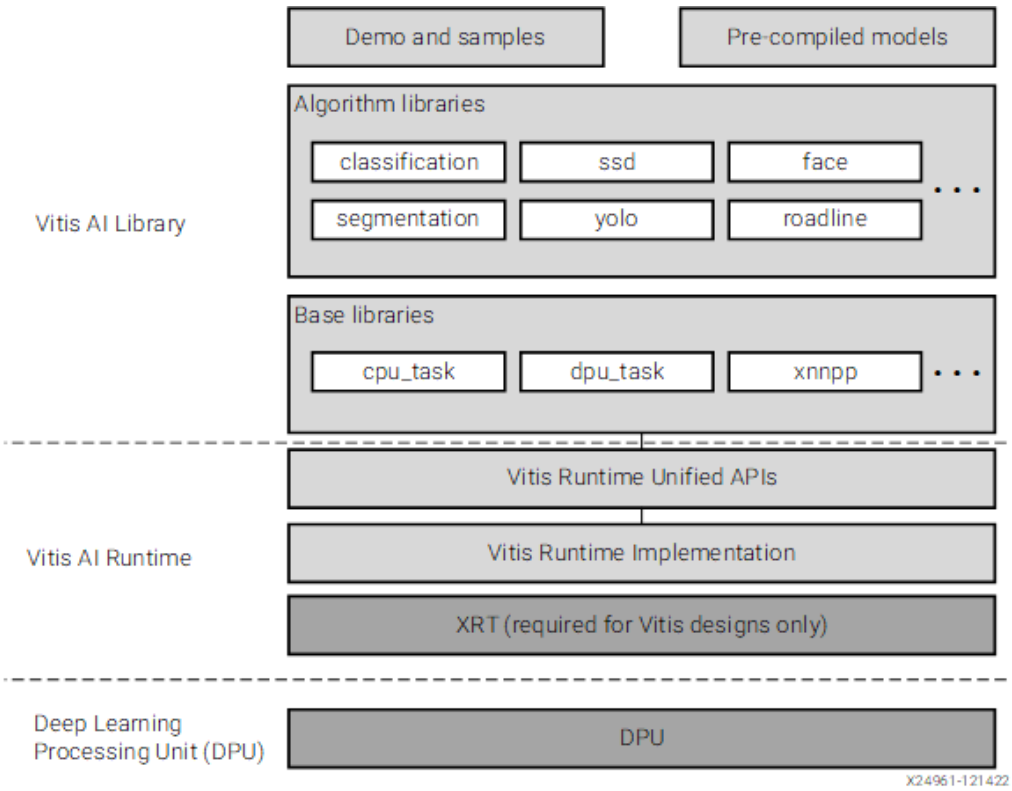


FIGURE 2.13: Vitis AI Workflow (Source: [9])

1. **Prepare the Pre-trained Model:** Acquire a pre-trained floating-point model from a widely-used framework such as TensorFlow or PyTorch. Ensure compatibility with Vitis AI by adhering to the specified guidelines for supported frameworks and operations.
2. **Quantize the Model:** Prior to deployment of the model on the Versal's DPU, it must first undergo quantisation. This process is facilitated by the Vitis AI Quantizer, converts the floating-point model to a fixed-point format, specifically INT8.
3. **Calibrate the Model:** Following the quantisation process, the model is calibrated and fine-tuned in order to minimise any potential loss of accuracy that may have been incurred as a result of the quantisation procedure.
4. **Compile the Model:** Subsequently, the Vitis AI compiler is employed to compile and optimise the quantised model for the target DPU architecture. This process generates the final x_model, along with all the requisite files for deployment.
5. **Develop the Accompanying Control Application:** To evaluate the model, it is necessary to create an application in C++/Python that will control the DPU and supply it with the appropriate data. This application will utilise a variety of AMD/Xilinx libraries that have been specifically designed for each device.
6. **Deploy and Run the Model:** To deploy the model on the Versal device, the compiled model and the associated application are transferred to the device. Once transferred, the model is executed on the Versal device to perform inference, utilizing the deployed model for real-time predictions.
7. **Profile the Model:** Finally, through Vitis AI Profiler, the model's performance on the the target hardware is assessed. Further optimization of the model and hardware configuration can be performed based on the profiling results to enhance overall performance.

2.3.3 PyTorch

PyTorch is an open-source deep learning framework. It is widely acknowledged for its dynamic computational graph, which permits flexible and intuitive model construction. The architecture of PyTorch facilitates the debugging of models and the efficient performance of operations, which has contributed to its popularity among researchers and practitioners for both academic research and production-level applications. The framework supports an extensive range of functionalities, including tensor computations, automatic differentiation, and GPU acceleration, which facilitate the development and training of complex neural networks [10].

2.3.4 TensorFlow

TensorFlow is a powerful open-source deep learning framework widely used for machine learning and artificial intelligence applications. Known for its scalability and flexibility, TensorFlow enables users to build and deploy machine learning models across various platforms, from edge devices to large-scale distributed systems. Its computational graph-based architecture allows for efficient execution of complex operations, while features like automatic differentiation, tensor computations, and hardware acceleration (GPU/TPU support) make it suitable for both research and production environments [11].

Chapter 3

Related Work

In this chapter, key developments in convolutional neural networks (CNNs), hardware acceleration for matrix multiplication, and network-on-chip (NoC) architectures are reviewed. These areas are critical to understanding the foundation of this work, with a focus on how they contribute to improving computational efficiency and performance in modern FPGA designs.

3.1 Convolutional Neural Networks

In recent years, Convolutional Neural Networks (CNNs) have emerged as one of the most powerful and widely used architectures in the field of deep learning, particularly for tasks involving image recognition, classification and segmentation. The development of CNN models has led to significant advancements in computer vision, enabling breakthroughs in diverse applications ranging from autonomous vehicles to medical imaging. Several key CNN architectures have become foundational in the evolution of the field, each contributing unique innovations and optimizations that have shaped the design of modern deep learning models. In the following sections, the defining characteristics of the most influential CNN models will be presented.

3.1.1 Most Known Models

LeNet, introduced in 1998, was one of the earliest convolutional neural networks, specifically designed for digit recognition tasks. Operating on 2D grayscale images, LeNet features a simple architecture with only 5 layers. Despite its simplicity, it laid the foundation for future CNN developments, demonstrating the potential of convolutional networks for image-based tasks. Its low computational complexity made it well-suited for simpler problems and smaller datasets, solidifying its influence on subsequent architectures [12].

AlexNet, the winner of 2012 ImageNet competition, marked a significant leap in CNN development by introducing deeper architectures with 8 layers and utilizing ReLU activations. Designed to process 2D color images, AlexNet revolutionized the field by leveraging GPUs for training, accelerating the performance of deep learning models. While it advanced the depth of neural networks, it required considerable computational resources, especially during training, due to its increased complexity [13].

VGGNet, developed in 2014, is recognized for its uniform architecture that employs small 3x3 convolutional filters throughout the network. Processing 2D color images, and consisting of 16 to 19 layers, making it significantly deeper than its predecessors. Despite its straightforward design, VGGNet's depth leads to a high number of parameters, resulting in increased computational complexity and memory demands [14].

DenseNet, introduced in 2016, presented a groundbreaking architecture that connects each layer directly to every other layer in a dense connectivity pattern. This enhances feature propagation and promotes efficient reuse of features, significantly improving parameter efficiency compared to traditional architectures. By alleviating redundancy and ensuring stronger gradient flow through the network, DenseNet achieves impressive accuracy with fewer parameters making it particularly suitable for tasks requiring high performance without the computational overhead of deeper networks [15].

ResNet, introduced in 2015, addressed the challenge of training very deep networks by implementing residual learning, a technique that adds shortcut connections to ease the flow of gradients. This innovation allowed for the effective training of much deeper architectures, such as ResNet-50, which process 2D color images. It's ability to maintain performance with depth makes it a milestone in deep learning, although it comes at the cost of heightened computational demands, particularly in deeper models [16].

MobileNet, developed in 2017, brought a new focus on efficiency by introducing depthwise separable convolutions to reduce the number of parameters and computational load. Optimized for mobile and embedded vision applications, MobileNet processes 2D color images with minimal complexity, making it ideal for deployment in resource-constrained environments. Its balance between performance and efficiency has made it a go-to solution for mobile and low-power devices [17].

EfficientNet, introduced in 2019, revolutionized CNN design by proposing a novel scaling method that balances network depth, width and resolution to optimize performance and computational efficiency. Unlike traditional models that scale these dimensions arbitrarily, EfficientNet systematically adjusts them based on a compound coefficient, leading to superior accuracy with fewer parameters and lower computational cost. This makes it highly effective for tasks requiring both high performance and efficiency, such as image classification in cloud environments or edge devices. Its scalable nature has set new benchmarks in the field of computer vision [18].

All the aforementioned CNN architectures are widely supported by popular deep learning frameworks such as TensorFlow and PyTorch, enabling easy development, train and fine-tune these models for a variety of applications.

3.1.2 Quantization of CNN Models

Quantization is a widely adopted technique in model deployment, especially for edge devices like embedded systems and FPGAs, where hardware resources such as memory and computational power are limited. By reducing the precision of the numerical representation of model parameters and activations—commonly from 32-bit floating point (FP32) to 8-bit integers (INT8)—quantization significantly decreases the model's memory footprint and computational complexity. This results in faster inference times and reduced energy consumption, which are crucial for real-time applications and resource-constrained environments.

However, quantization entails a trade-off in reduced precision, which can result in a loss of information and potentially diminish the overall accuracy of the model. The extent of this accuracy drop depends on the model architecture and the dataset, so designers must carefully balance between performance gains and acceptable accuracy levels. To mitigate this, techniques such as post-training quantization and quantization-aware training are employed to minimize accuracy degradation while still leveraging the benefits of reduced precision.

In recent years, INT8 quantization has become a standard technique in modern deep learning frameworks, offering significant performance improvements while maintaining accuracy close to FP32 levels. Widely adopted in frameworks like TensorFlow and PyTorch, INT8 quantization reduces the computational and memory requirements by lowering the precision of weights and activations from 32-bit floating-point (FP32) to 8-bit integer (INT8). Studies such as NVIDIA's *Achieving FP32 Accuracy for INT8* [19] and PyTorch's *INT8 Quantization for x86 CPUs* [20] have shown that INT8 quantization can deliver substantial performance gains with minimal accuracy loss.

Moreover, ongoing academic research is exploring its application in training and inference as well as. In particular, Kang Zhao's *Distribution Adaptive INT8 Quantization for Training CNNs* [21] introduces a novel INT8 quantization framework that enhances training speed while preserving near-lossless accuracy. Similarly, Sumin Kim's *Performance Evaluation of INT8 Quantized Inference on Mobile GPUs* [22]. demonstrates that INT8 quantization can achieve up to 2.45× faster inference on mobile GPUs with only a minimal accuracy reduction of 0.31%.

3.2 Matrix Multiplication

Matrix multiplication is a fundamental operation, particularly crucial in computational tasks like Convolutional Neural Networks (CNNs), where it can account for up to 90% of the execution time [23]. In CNNs, matrix multiplication is heavily utilized, especially in the multiplication of input features with weight matrices and in the transformation of convolution layers into matrix operations. This transformation has even spurred the development of dedicated hardware units to optimize the im2col operation [24]. Given its significance, optimizing matrix multiplication is essential for achieving high performance, motivating significant research efforts across algorithmic and hardware-specific optimizations.

3.2.1 Matrix Multiplication Efficiency

Algorithmic Advancements

Matrix multiplication efficiency has long been a critical area of study, particularly due to its widespread application in fields such as machine learning, scientific computing, and signal processing. Early efforts to improve matrix multiplication centered around reducing the time complexity of standard methods. A major breakthrough occurred in 1969 with the introduction of *Strassen's algorithm*, which reduced the complexity from $O(n^3)$ to approximately $O(n^{2.81})$ [25]. This marked the first substantial departure from classical approaches, and since then, numerous advancements have continued to push these boundaries.

One of the most notable recent advancements is *AlphaTensor*, a deep reinforcement learning model designed to discover matrix multiplication algorithms that outperform traditional, human-designed methods. AlphaTensor's discovery of more efficient algorithms, particularly for small matrices such as 4x4, highlights the potential of machine learning to automate and improve both the theoretical and practical aspects of matrix multiplication [26]. In addition to automated algorithm discovery, sparse factorization techniques have emerged as powerful methods for enhancing matrix multiplication performance, particularly in cases involving large matrices or low-resolution quantization, providing both speed and hardware adaptability [27].

Hardware-Specific Optimizations

Optimizing matrix multiplication for specific hardware platforms has been another critical focus of research. As modern hardware architectures evolve, tailored optimizations significantly enhance performance. One such approach is the use of asymmetric hashing with tensor power structures to build on the *Coppersmith-Winograd* algorithm, which accelerates computations by reducing the computational complexity [28].

Furthermore, communication-optimal parallel recursive algorithms have been developed to minimize data transfer, which is essential for improving performance on multicore and GPU platforms [29].

On CPU-GPU architectures, applying advanced algorithms such as Strassen's has led to substantial performance improvements, showcasing the potential of heterogeneous computing environments for efficient matrix operations [30]. Meanwhile, FPGA-based accelerators have demonstrated notable gains by reducing latency and area through customized hardware designs, making them highly efficient for matrix multiplication tasks [31].

Matrix Multiplication on the Versal Platform

Recent research has increasingly focused on optimizing matrix multiplication for the AMD/Xilinx Versal platform, which integrates AI Engines and adaptable hardware accelerators. For example, the study "*Toward Matrix Multiplication for Deep Learning Inference on the Xilinx Versal*" explores ways to leverage the Versal architecture to optimize deep learning inference through efficient matrix multiplication, targeting both performance and energy efficiency [32].

Another significant advancement comes from "*CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture*," which demonstrates how the flexibility of the Versal ACAP architecture allows for the integration of various accelerators to achieve substantial improvements in matrix multiplication performance [33]. In addition, "*MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine*" presents innovative strategies for maximizing computational efficiency, specifically utilizing the AI Engine to accelerate matrix operations and achieve high performance [23].

Furthermore, AMD/Xilinx provides an optimized matrix multiplication implementation through its AI Engine API, which uses specialized intrinsics to access low-level hardware instructions, ensuring highly efficient execution [34].

These advancements underline the growing importance of hardware-aware optimization techniques for achieving cutting-edge performance in matrix multiplication tasks across various platforms.

3.3 Platforms for Accelerating Matrix Multiplication

In the pursuit of optimizing matrix multiplication, various hardware platforms have been used to accelerate these computations, each offering distinct advantages. General-purpose platforms like GPUs leverage parallelism to handle large-scale computations efficiently, while FPGAs provide customizable hardware acceleration tailored to specific tasks. Moreover, dedicated accelerators such as Google's Tensor Processing Unit (TPU) offer specialized solutions that deliver unparalleled performance for deep learning tasks. These platforms have been instrumental in meeting the computational demands of applications where matrix multiplication often represents a significant bottleneck in terms of performance and energy consumption.

Graphics Processing Units (GPUs)

Initially developed for rendering graphics, *Graphics Processing Units* (GPUs) have since evolved into powerful computing devices capable of handling highly parallelizable workloads. Their architecture, characterized by a large number of cores, is well-suited for tasks such as matrix multiplication, which is fundamental to many machine learning algorithms, particularly Convolutional Neural Networks (CNNs).

GPUs excel in General Matrix Multiplication (GEMM) by utilizing a tiling approach, where the large matrices are divided into smaller tiles that fit into the GPU's memory, allowing for highly parallel computations [35]. This makes them one of the go-to platforms for both training and inference in CNNs. However, despite their high throughput, GPUs are proprietary, power-hungry and sometimes struggle with real-time processing, particularly for floating-point models when timing constraints are critical.

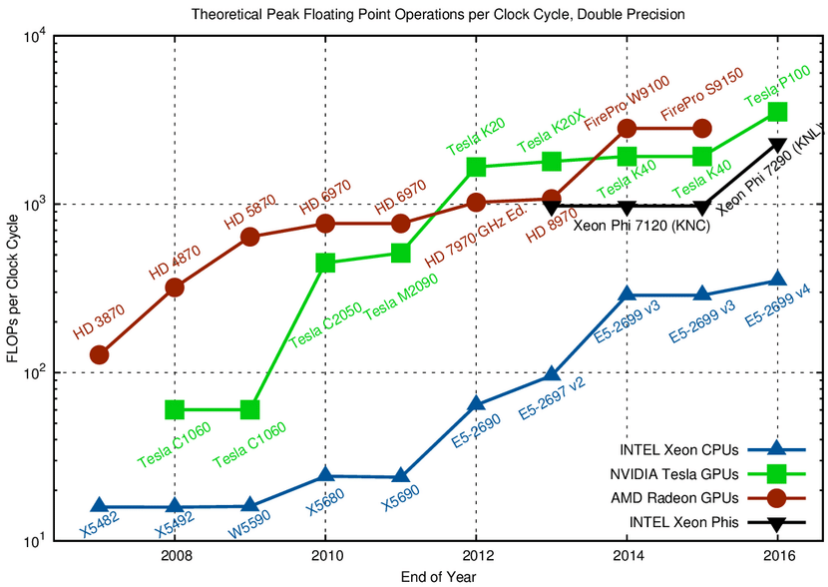


FIGURE 3.1: CPU vs GPU performance for floating-point operations (Source: [36])

Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) offer a highly flexible and reconfigurable platform that excels in accelerating tasks such as matrix multiplication and CNN processing. Unlike general-purpose processors like CPUs or GPUs, FPGAs can be customized at the hardware level to create highly specialized circuits tailored to specific tasks. This allows for parallel execution, optimized memory access, and reduced latency, making FPGAs a compelling choice for high-performance computing and machine learning applications.

In the context of CNNs, FPGAs are particularly adept at handling convolutional layers and pooling operations. By leveraging parallelism and optimizing the data flow, they can achieve lower energy consumption and faster computation times compared to GPUs. For matrix multiplication, custom-designed arithmetic units and data paths further enhance performance, making FPGAs ideal for energy-efficient, low-latency applications that require high-throughput computation.

One of the key advantages of FPGAs is their ability to strike a balance between flexibility and efficiency, allowing them to outperform traditional hardware platforms when properly optimized. This is particularly important in applications like real-time inference, where power consumption and processing speed are critical factors.

Tensor Processing Units (TPUs)

Google's *Tensor Processing Unit* (TPU) was developed to address the computational challenges posed by deep learning workloads, particularly in their data centers. Introduced around 2013, the TPU is an Application-Specific Integrated Circuit (ASIC) designed explicitly to accelerate inference tasks. At the core of its architecture is a 128x128 systolic array of Multiply-Accumulate (MAC) units, optimized for 8-bit operations on both signed and unsigned integers.

The TPU delivers significant performance improvements over general-purpose hardware, achieving 15 to 30 times the throughput of an Nvidia K80 GPU or a server-grade Intel Haswell CPU. In addition, it offers impressive energy efficiency, with performance-per-watt metrics up to 80 times higher than that of GPUs and 200 times higher than that of CPUs [37].

This performance boost is largely due to the TPU's highly specialized architecture, which is optimized for the specific needs of deep learning operations, such as matrix multiplication in CNNs. However, despite its impressive capabilities, the TPU remains a proprietary solution, which limits its adaptability and availability for a broader range of applications outside Google's ecosystem.

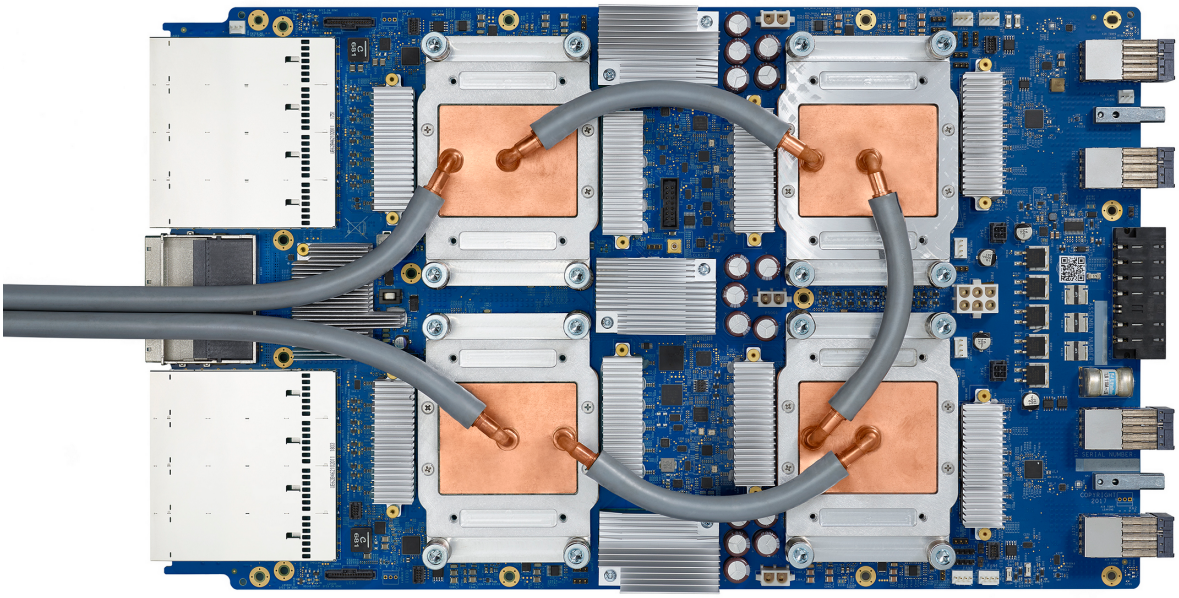


FIGURE 3.2: Google's TPU (Source: [37])

3.3.1 Comparison of Hardware Platforms

Each of these platforms—GPUs, FPGAs, and TPUs—offers unique advantages depending on the specific application. GPUs are widely used for their versatility and parallel processing power, especially in tasks like matrix multiplication for deep learning. FPGAs, while requiring more effort to program, offer unmatched flexibility and energy efficiency, making them ideal for custom implementations. TPUs, on the other hand, provide extraordinary performance for deep learning inference, but their proprietary nature limits their broader adoption.

In summary, the choice of platform for accelerating matrix multiplication depends on the specific requirements of the application. For real-time, energy-efficient processing, FPGAs stand out as a strong contender. For large-scale, high-throughput machine learning workloads, TPUs and GPUs are more commonly employed, with TPUs excelling in inference and GPUs being more versatile for training.

3.4 Architecture of a Hard-IP NoC

The architecture of Hard-IP Network-on-Chip (NoC) systems has been a critical area of research in both academia and industry. As systems-on-chip (SoCs) have evolved, the need for scalable, efficient, and low-latency communication solutions has driven the development of NoC architectures. These solutions provide high-bandwidth, reliable data transfer between various components, making them integral to modern computing platforms, especially in high-performance and FPGA-based systems.

3.4.1 Network-on-Chip

The concept of a Hard-IP Network-on-Chip (NoC) predates its commercial implementation, having been explored extensively in academic research. Initial studies laid the groundwork for scalable interconnects that address the growing complexity of system-on-chip designs.

One of the seminal works in this area is Dally and Towles's 2001 paper, "*Route Packets, Not Wires: On-Chip Interconnection Networks*", which presented the foundational principles of NoC architecture for system-on-chip applications, emphasizing scalability and low-latency communication [38]. This early work influenced subsequent research focused on FPGA-based NoCs, such as Marescaux et al.'s "*Network-on-Chip for Reconfigurable Systems*" [39], which addressed unique design challenges, including the need for reconfigurability and optimization for FPGA platforms.

Further exploration into the NoC efficiency for FPGAs is presented in "*Exploring Hard and Soft Networks-on-Chip for FPGAs*", which analyzed different NoC implementations —hard, soft, and mixed—on FPGA platforms, providing valuable insights into the trade-offs between silicon area, bandwidth, and communication performance [40]. Hilton and Nelson's "*PNoC: A Flexible Circuit-Switched NoC for FPGA-Based Systems*" [41] expanded on these ideas, investigating the performance and flexibility benefits of circuit-switched NoCs for FPGA implementations.

Research into low-latency NoCs is also critical for enhancing FPGA performance, as shown in Mullins et al.'s "*The Design and Implementation of a Low-Latency On-Chip Network*", which detailed design trade-offs that optimize latency without compromising throughput [42]. The concept of embedding NoCs within FPGAs to improve interconnect efficiency was further explored in "*Augmenting FPGAs with Embedded Networks-on-Chip*", which proposed enhancements to FPGA architectures for improved system-level performance [43].

Fault tolerance is another important aspect of NoC design. A study on adaptive routing algorithms, "A New Fault-Tolerant and Congestion-Aware Adaptive Routing Algorithm for Regular Networks-on-Chip", focused on addressing the reliability and congestion challenges in NoC-based systems [44]. Junshi Wang et al.'s "Design of Fault-Tolerant and Reliable Networks-on-Chip" [45] further explored these topics, proposing fault-tolerant strategies that are particularly relevant for FPGA-based systems.

3.4.2 The Versal NoC

The AMD/Xilinx Versal Adaptive Compute Acceleration Platform (ACAP) is one of the most advanced commercial implementations of a Hard-IP Network-on-Chip. The Versal ACAP integrates a programmable NoC that enhances data transfer efficiency, enabling high-performance communication between the platform's various compute engines.

Research on the Versal NoC has garnered significant attention, with studies such as "*Network-on-Chip Programmable Platform in Versal ACAP Architecture*" [46] highlighting how the integration of a programmable NoC optimizes data flow and boosts system performance. This NoC architecture enables the Versal platform to handle complex workloads across multiple applications, including machine learning, data processing, and high-performance computing.

Another important contribution to understanding the Versal NoC is Ian Elmor Lang's work, "*Worst-Case Latency Analysis for the Versal Network-on-Chip*", which provides a methodology for calculating upper bounds on network latency using Recursive Calculus, further validated by simulation-based evaluations [47]. This analysis is crucial for applications requiring guaranteed performance under worst-case conditions, such as safety-critical systems.

Max Wierse's thesis, "*Evaluation of Xilinx Versal Device*", examines the performance, API, and toolchain of the Versal platform, focusing on its unique features that drive parallelism and computational efficiency [48]. In addition, Chen et al.'s "*Exploiting On-chip Heterogeneity of Versal Architecture for GNN Inference Acceleration*" [49] explores the Versal platform's heterogeneous computing capabilities, particularly its adaptability for Graph Neural Network (GNN) inference tasks, showcasing the versatility of the Versal NoC.

3.4.3 Other Commercial NoC Solutions

Beyond AMD/Xilinx, other companies have developed SoCs incorporating Hard-IP NoCs to enhance communication efficiency. Intel's Stratix 10 FPGAs [50] feature Arm Cortex-A53 cores and advanced transceivers, alongside a Hard-IP NoC optimized for

data center and networking applications. Intel's Agilex FPGAs [51], built on 10nm technology, also integrate a Hard-IP NoC that supports high-speed I/O for AI acceleration and data processing applications.

NVIDIA's Jetson AGX Xavier [52], designed for AI, robotics and edge computing, incorporates an 8-core Arm CPU, Volta GPU, and deep learning accelerators. Its Hard-IP NoC supports the intensive data exchange required for real-time AI inference and robotic control, demonstrating the role of NoCs in heterogeneous computing environments where multiple processing elements need to work in tandem.

Chapter 4

Methodology

In this chapter, a comprehensive overview of the methodologies used in both approaches is presented. It details the selection process for the Convolutional Neural Network (CNN) in the first approach and the matrix multiplication algorithm in the second. Additionally, it discusses key considerations such as data types and outlines the testing methodology employed.

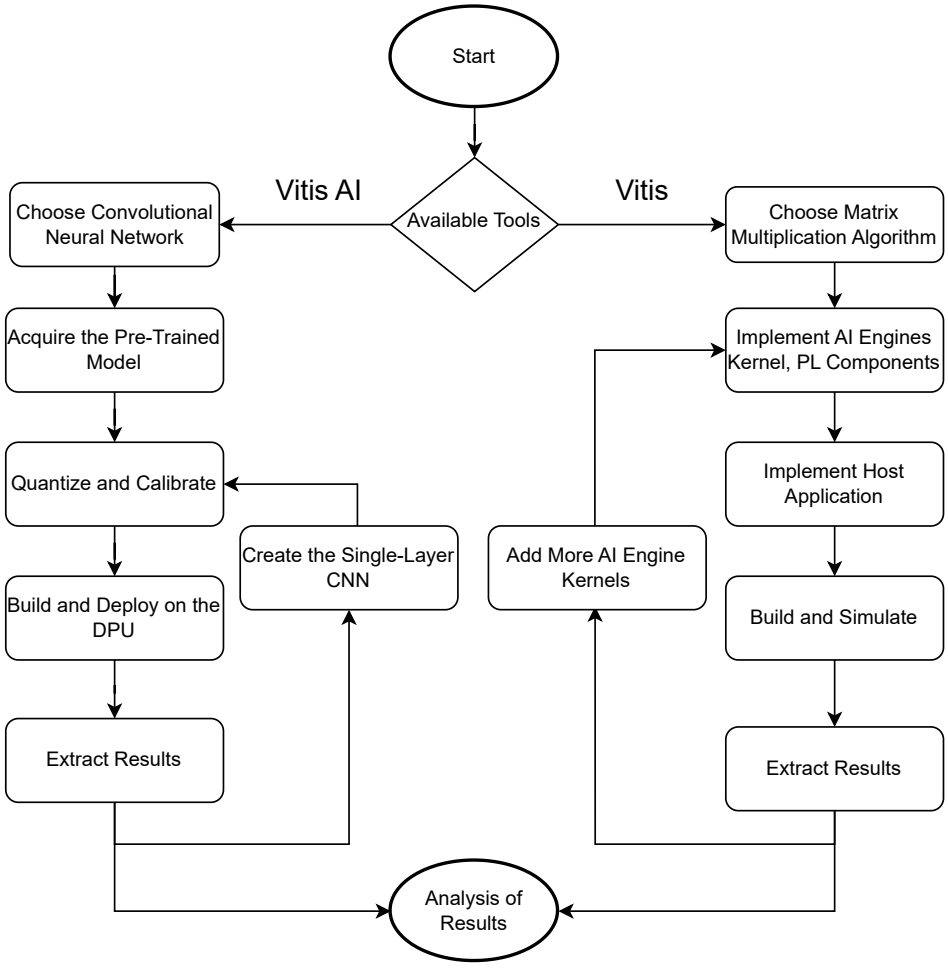


FIGURE 4.1: General Methodology Flowchart

4.1 Vitis AI approach

As mentioned before in the subsection 2.3.2, Vitis AI has been developed to ensure compatibility with widely used frameworks such as PyTorch and TensorFlow and to optimize the execution of commonly deployed models like MobileNet and ResNet.

To achieve the primary objective of measuring throughput, in order to evaluate the device, several key factors must be considered when selecting an appropriate model. First, the model should facilitate easy alterations to input data with minimal modifications, allowing for the simulation of various scenarios. Second, the complexity of the model should be minimized, focusing on data transfers over the NoC rather than the intricate details of the CNN. Lastly, the model must be optimized for the DPU and capable of running on an optimized framework to further reduce the computation time.

After careful consideration of the main characteristics mentioned in the subsection 3.1.1, the ResNet architecture and specifically ResNet-20 was selected as the CNN model for this thesis. ResNet-20's acceptance of images as input enables straightforward modifications to the data input with only minor adjustments to the layers. Although it does not have the smallest processing requirements, its relative simplicity and numerous applications makes it suitable for this study. Additionally, ResNet-20 operates within the PyTorch framework, which is optimized for the Versal DPU, making it an ideal choice for evaluating throughput in this context.

4.1.1 Data Type and Test Methodology

The ResNet-20 model was initially designed for images of dimensions 32×32 pixels. The specific model selected was pre-trained using the CIFAR10 dataset. CIFAR-10 is a widely used benchmark dataset in the field of computer vision, consisting of 60,000 color images divided into 10 distinct classes, such as airplanes, automobiles, birds and cats. Each image in the dataset is 32x32 pixels in size, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 test images, making it a valuable resource for evaluating the performance of machine learning models, particularly in tasks related to image classification.

However, to accommodate larger images, as it is intended, both the model and the data had to be adapted. Regarding the model, this can be achieved by incorporating a max-pooling layer. Max-pooling layers effectively change the dimensions of a matrix by merging a number of pixels according to specific parameters. Since the images are represented by a matrix, this additional layer effectively allows the model to process images of any size by reducing them to the required dimensions of 32×32. Following the pooling layer, the existing layers of ResNet-20 remain unchanged, allowing the model to achieve the desired results with minimal modifications.

Regarding the dataset, since the accuracy of the model is not the primary focus of this work, larger images are randomly generated with the appropriate dimensions. Despite the variation in input images, the CNN consistently produces the same classification vector as output. This approach enables us to focus on evaluating the hardware performance and architectural efficiency, rather than the model's accuracy in classification tasks.

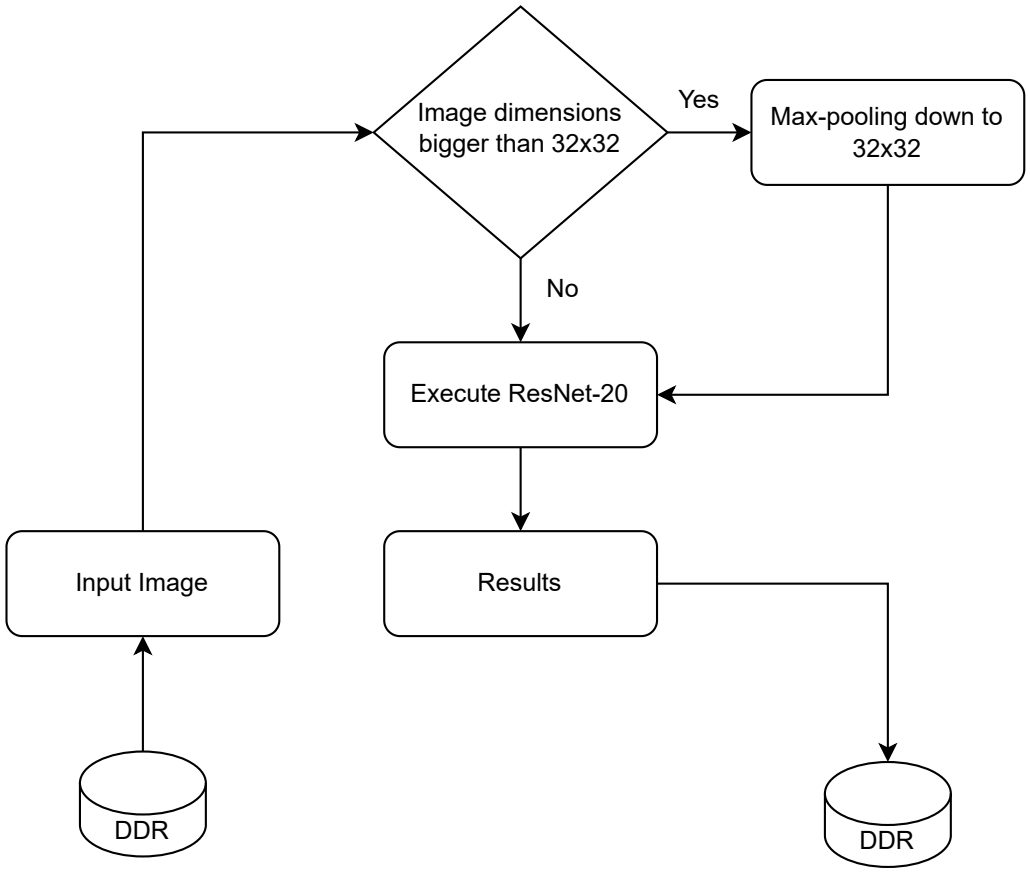


FIGURE 4.2: ResNet-20 System Design

In the proposed testing methodology, the CNN is deployed on the DPU and evaluated using images of varying sizes. For each test, the time taken to load the image from the DDR, process it through the CNN on the DPU and write the result vector back to the DDR memory is recorded. This measured time serves as the basis for calculating throughput, which reflects the data transfer rate across the NoC. Throughput is determined by dividing the total number of bits transferred—comprising both input and output data—by the elapsed time for each test, expressed in seconds.

$$\text{Throughput} = \frac{\text{data (bits)}}{\text{time (s)}} \tag{4.1}$$

4.1.2 Single-Layer CNN Modification

Since the measurements are time-based, it is essential to account for the actual runtime of the CNN. To enhance accuracy, the processing on the DPU must be minimized. This can be achieved by removing all layers except the one responsible for max pooling. This modification results in a system that takes an input image with specific dimensions, reduces it to half its width and height—effectively shrinking it to one-fourth of its original size—and writes the downsized image to the DDR. While the processing time is still included in the measurement, it becomes negligible in this scenario.

Moreover, this approach significantly increases the volume of data transferred through the NoC during each iteration. Unlike the previous case, where the CNN's output was a single classification result, this method outputs a substantial portion of the image itself. The test data remains consistent with earlier tests and the throughput is calculated in the same manner as before.

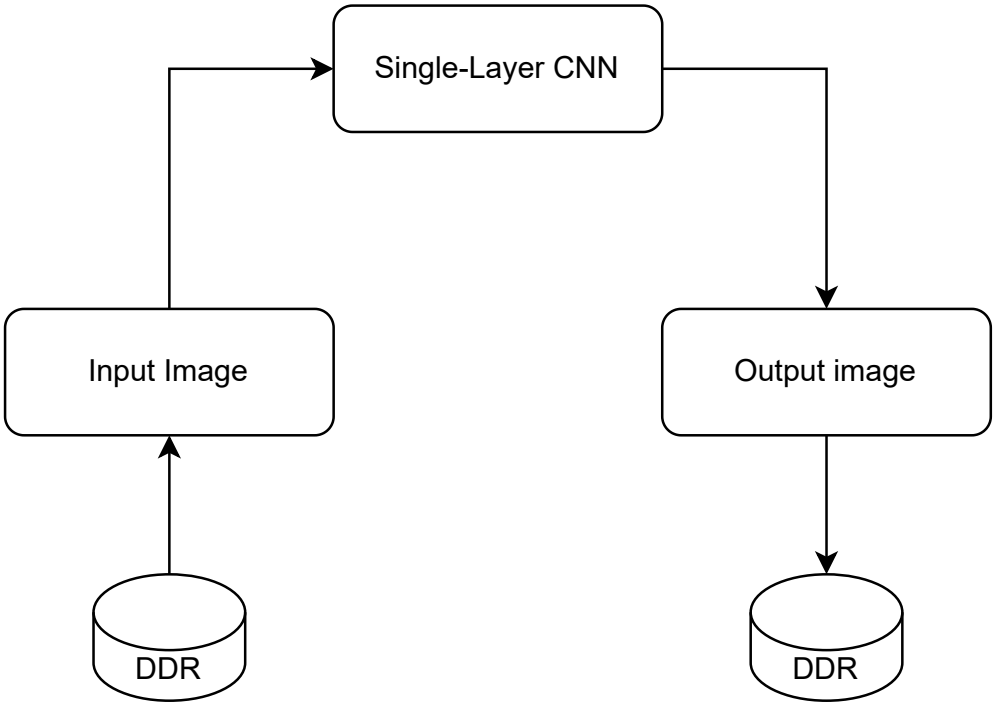


FIGURE 4.3: Single-Layer CNN System Design

4.2 Vitis Approach

As previously discussed in the subsection 2.2.2, a Convolutional Neural Network is a sophisticated structure comprised of numerous interconnected layers, each playing a crucial role in the network's functionality. This complexity necessitates careful design and optimization to ensure efficient performance. Additionally, as detailed in subsection 2.3.1, the Vitis workflow encompasses several distinct steps, each requiring significant effort and expertise though these steps are essential for leveraging the full potential of the target platform.

In this case, the objective of measuring the NoC's throughput can be effectively achieved by developing a more streamlined system, composed of only a select number of key components. Adopting such an approach allows for a more focused and controlled evaluation of the NoC's performance, without the added complexity and overhead associated with implementing a full-scale CNN acceleration system. This simplified system ensures that the measurements remain accurate and are not influenced by unnecessary factors, providing a clearer insight into the NoC's efficiency.

As previously discussed in Subsection 2.2.2, matrix multiplication is a critical operation within CNNs, playing a fundamental role in their overall performance. Subsections 3.2.1 and 3.2.1 further highlight the numerous algorithms developed over the years to optimize matrix multiplication, including specialized implementations for the Versal platform. In this study, the optimized versions provided by AMD/Xilinx were chosen due to their use of intrinsics, which greatly enhance the algorithm's efficiency over alternative implementations. This selection also enables a direct performance comparison between the NoC and the benchmark results reported by AMD/Xilinx.

4.2.1 Data Type and Test Methodology

The system's performance, particularly in terms of data throughput, will be evaluated by measuring the total time required for the entire process—from the initial loading of data from DDR, its processing on the AI engines, to the final generation and storage of the output matrix back into DDR. To ensure highly accurate measurements, it is crucial to once again minimize the workload handled by the AI Engines. Therefore, the simplest supported data type, INT8, will be utilized to streamline operations and minimize any potential overhead. As for the actual matrices, two randomly generated matrices will be created and stored in the DDR before processing begins, allowing for a consistent and efficient evaluation of the system's throughput.

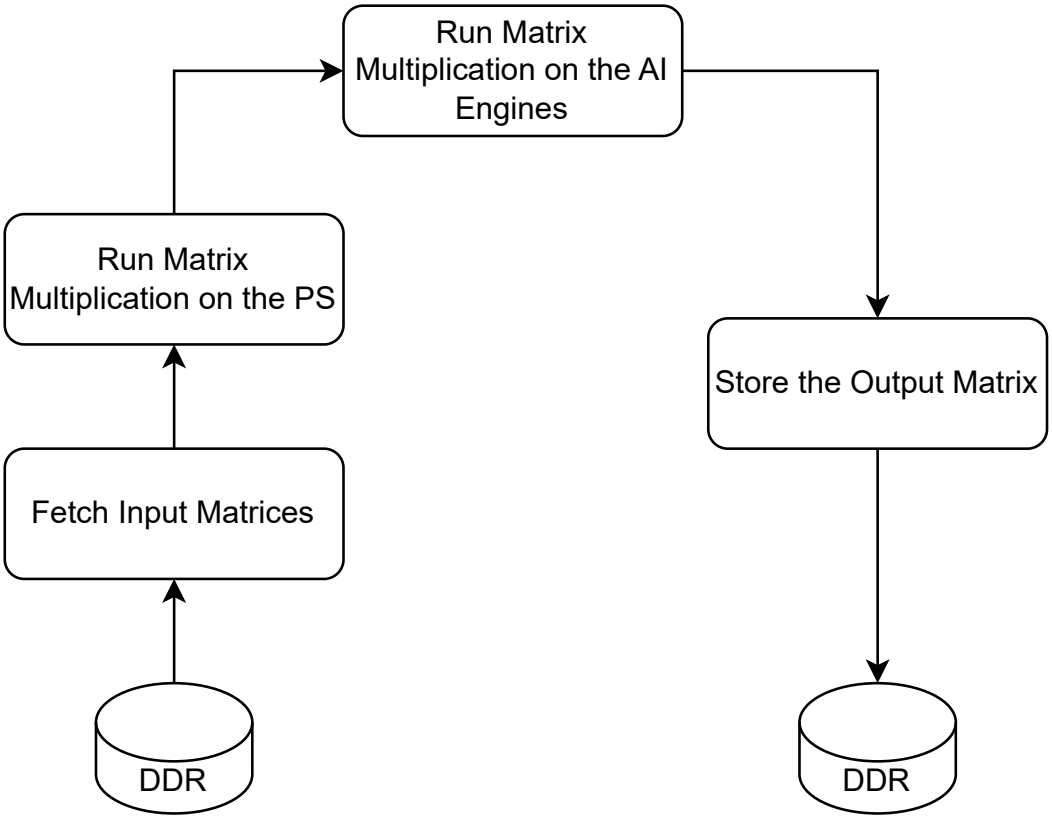


FIGURE 4.4: Matrix Multiplication Design

For the test methodology, multiple trials will be conducted, with the primary variable being the dimensions of the matrices utilized. The target matrix sizes selected for testing are 8×8, 16×16, and 32×32. Larger matrices exceeded the cache capacity of the AI Engine due to the particular implementation, necessitating additional modifications. These adjustments introduced complexity and could potentially bias the results.

To ensure the correctness of the output matrix generated by the AI Engines, traditional matrix multiplication is executed on the Arm processor and the results are compared with those produced by the AI Engines. To eliminate randomness and guarantee reliable results, each test will be run 100 times. After gathering the measurements, the throughput is calculated using the same formula (4.1) as outlined in the previous subsection.

4.2.2 Adding More Kernels

To gain a more comprehensive understanding of the NoC's performance, it is essential to increase data traffic across the network. This can be achieved by performing multiple matrix multiplications in parallel. The optimal approach would involve feeding the data through a dedicated bus and distributing it to each matrix multiplication kernel.

However, as the objective here is to intensify the data traffic, an alternative strategy will be employed. Specifically, additional kernels will be incrementally instantiated in parallel, each utilizing a dedicated data lane over the NoC. Data will be collected and analyzed as previously to obtain a deeper insight and more precise estimation of the NoC's throughput.

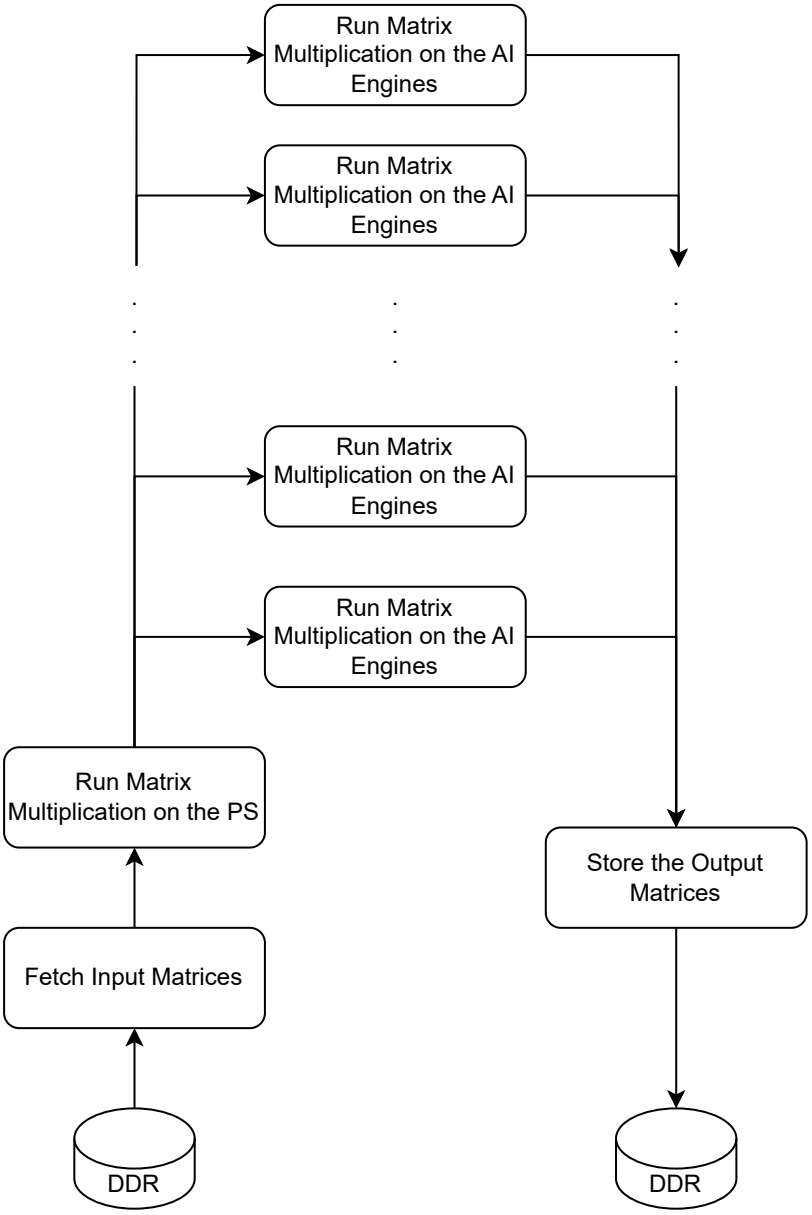


FIGURE 4.5: Multiple Matrix Multiplications Design

Chapter 5

Development and Implementation

In this chapter, the development and implementation process is outlined through a detailed examination of two key approaches: Vitis AI and Vitis. Each approach is analyzed with respect to its independently developed system components and the specific steps required for building and testing. Accompanied by illustrative photographs, this discussion provides a clear and comprehensive view of the development and modification processes for each method.

5.1 Vitis AI Approach

In Subsection 4.1, the implementation of the ResNet-20 architecture, selected as the primary CNN model for this thesis, was introduced. This section provides a more comprehensive analysis of the ResNet-20 framework, offering an in-depth exploration of its structural elements and functional mechanisms. Furthermore, it examines the integration of the max-pooling layer, emphasizing its significance within the network. The section also details the construction and implementation of the various building blocks necessary for the system to operate as intended.

5.1.1 ResNet-20

ResNet-20 is composed of 20 layers, including convolutional layers, batch normalization, ReLU activations and fully connected layers, all structured into residual blocks, as illustrated in Figure 5.1. Each residual block comprises two convolutional layers and a shortcut connection that bypasses the block, adding its input directly to the output. This architecture addresses the vanishing gradient issue, facilitating the training of deeper networks. Additionally, the residual blocks enhance gradient flow during backpropagation, improving both convergence and overall performance across diverse tasks.

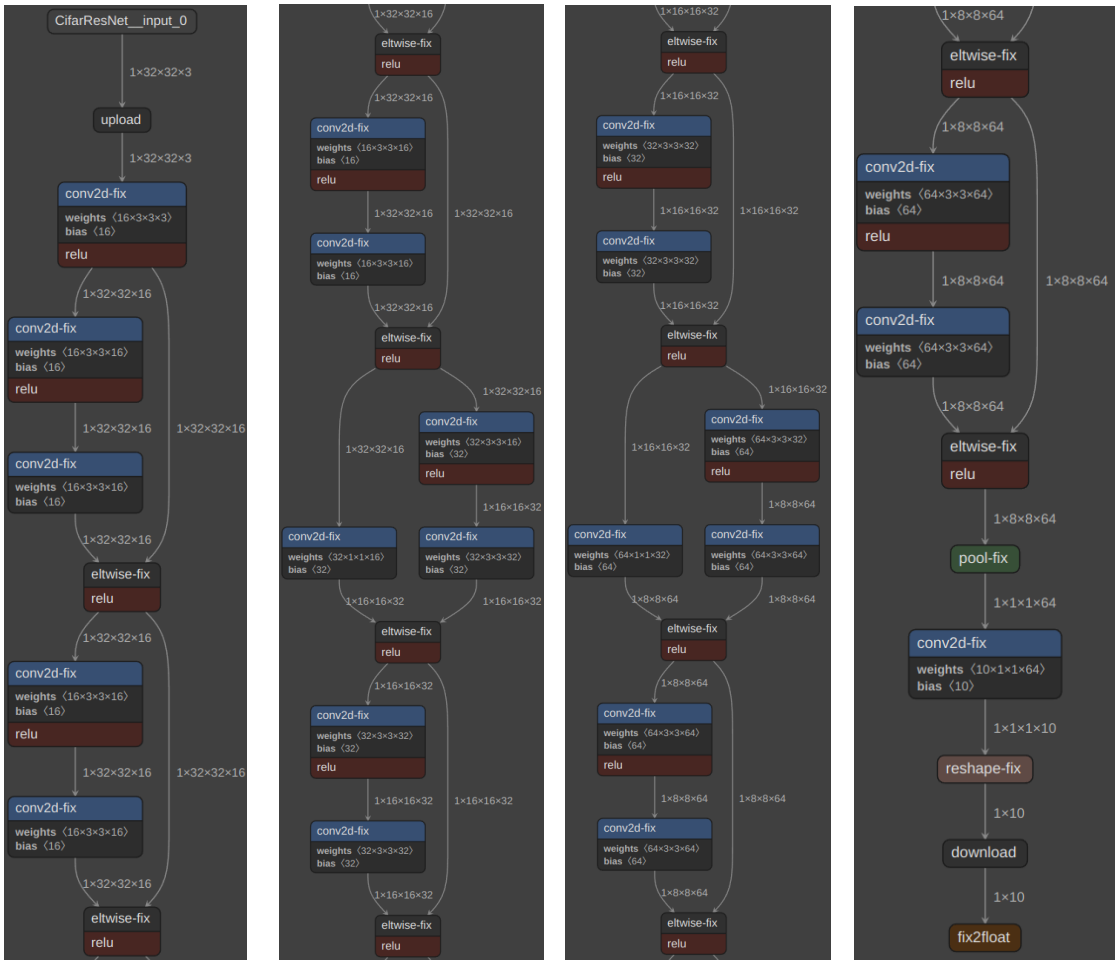
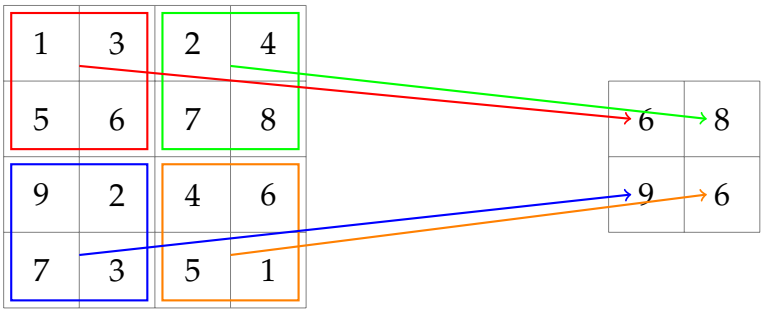


FIGURE 5.1: ResNet-20 Layer Structures

Additional Modifications - Max-pooling Layer

To accommodate varying image sizes, the ResNet-20 architecture can be modified by incorporating an additional max-pooling layer. Max-pooling operates by reducing the spatial dimensions (height and width) of the input feature maps while preserving the most important information. It achieves this by sliding a fixed-size window (commonly 2x2 or 3x3) across the input and selecting the maximum value within each window to pass to the subsequent layer. This process produces a down-sampled feature map that emphasizes the most prominent features, effectively lowering computational complexity and mitigating over-fitting.



The addition of an extra max-pooling layer enhances the CNN's ability to efficiently process larger images. In this thesis, image sizes ranging from 32×32 to 2048×2048 are employed, significantly increasing data traffic across the NoC. This max-pooling layer is seamlessly integrated into the original Python code, utilizing the CNN layers and functions supported by the PyTorch framework. Conveniently named "pool-fix," this layer is positioned after the upload block and its effectiveness is evident, particularly in down-scaling a 512×512 image to 32×32, as designed.

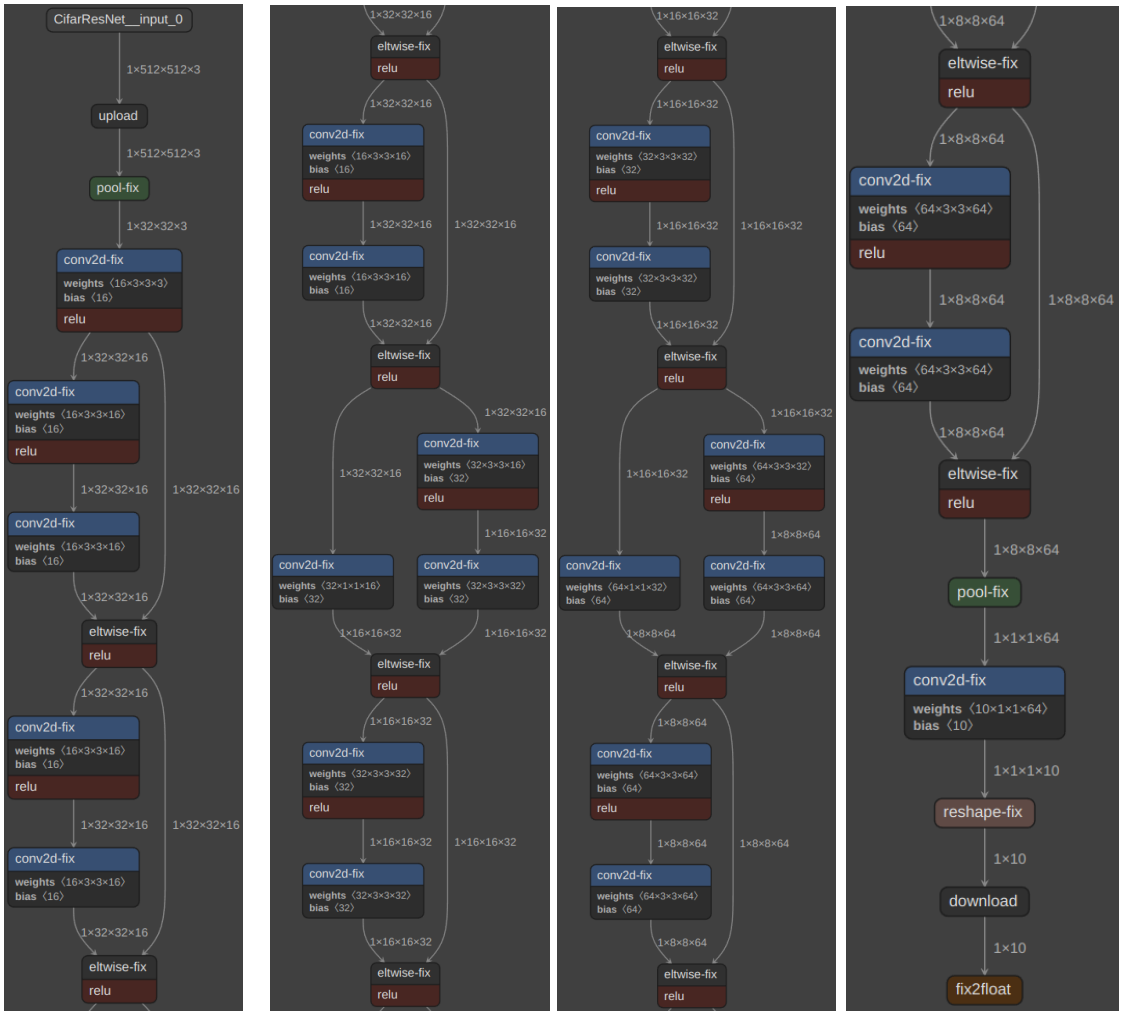


FIGURE 5.2: ResNet-20 Layer Structures Accepting a 512x512 Image after Modifications

5.1.2 System Building Blocks

To ensure the system operates as intended, several critical stages must be completed prior to final deployment, as described in the Vitis AI workflow in Subsection 2.3.2. Initially, the Python model must undergo quantization and re-calibration before the compilation phase. Following this, the host application is developed, leading to the final deployment. Each of these steps will be elaborated in the subsequent subsections.

Quantize and Calibrate

In order for the model to be deployed on the Versal's DPU, it must first undergo quantization to the INT8 format, a critical requirement due to the architecture constraints of the Versal platform's DPU. Quantization reduces the precision of the model's weights and activations from FP32 to INT8, significantly lowering computational complexity and memory usage. This process is efficiently managed by the Xilinx Runtime (XRT) API, which automates and streamlines the quantization.

Once quantization is completed, the model undergoes re-calibration to ensure that its accuracy is maintained as much as possible after being converted to INT8. Though the accuracy of the model is not the primary focus of this thesis, this step is required to validate the functionality of the model and is an integral part of the deployment pipeline. The XRT API once again provides a streamlined method for re-calibrating the model, ensuring that it continues to operate correctly despite the precision reduction.

Following re-calibration, the final step involves compiling the model. During this process, the required AI Engine graphs and sub-graphs are generated according to the model's architecture. In this study, Vitis AI consistently reports the generation of a single graph, composed of three sub-graphs, to implement the entire system across all tests, however, no further details regarding the sub-graph structure are provided. The result of this compilation is an .xmodel file, which contains all necessary information to execute the model efficiently on the Versal DPU, making it ready for deployment.

Host Application

The host application functions as the central interface for deploying and managing the execution of the CNN on the DPU. It orchestrates the entire process, beginning with the loading and pre-processing of input images, followed by their dispatch to the DPU for inference. Additionally, it captures precise execution times for each stage, enabling thorough performance evaluation and analysis of the deployed systems.

5.1.3 System Architecture

The detailed architecture of the whole system need to be deployed on the Versal DPU is depicted in the figure 5.3 below.

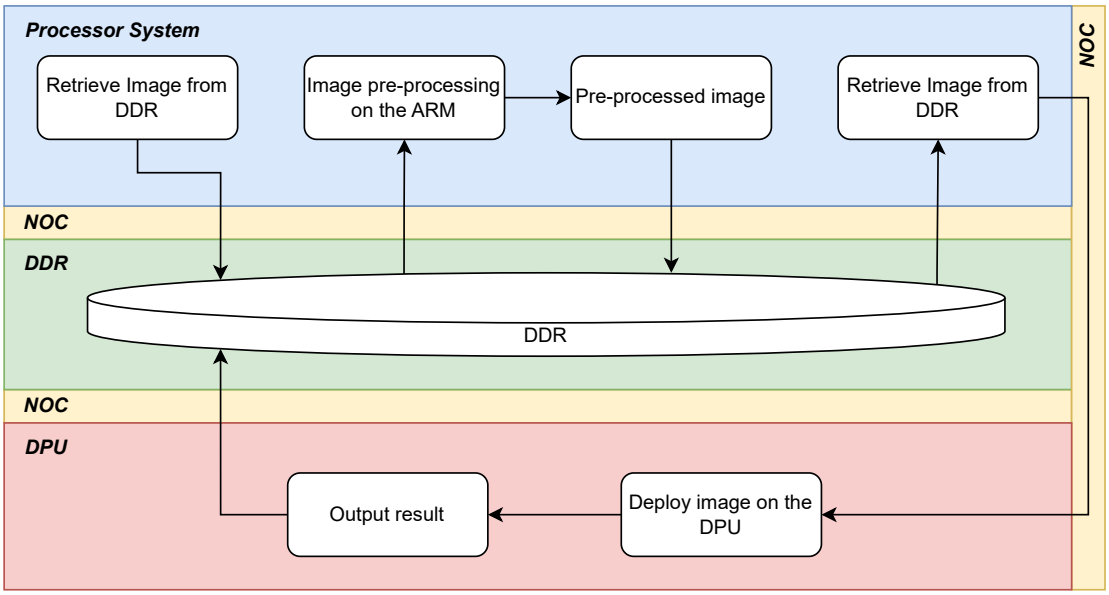


FIGURE 5.3: Vitis AI System Architecture

The test images are initially loaded into the DDR and undergo pre-processing on the PS before being passed to the DPU for inference. Originally, the pre-processing pipeline involved resizing the images to the appropriate dimensions, normalizing the pixel color values to a range between 0 and 1, scaling these values to match the input format expected by the neural network and reordering the color channels from BGR to RGB to align with the network's input requirements. However, given that the model's accuracy is not the focus of this study, only the normalization and scaling steps are performed to ensure that the pixel values are correctly formatted for the network. This pre-processing step is crucial to maintaining consistent performance during inference.

After pre-processing, the DPU is initialized and the processed image is transferred from the DDR to the DPU, with a batch size of 1, meaning images are processed individually. As illustrated in Fig. 5.3, data flows from the DDR to the DPU via the NoC. Once on the DPU, the image passes through the CNN, producing a classification result in the form of a 10-element vector, where each element represents the probability of the image belonging to a specific class. This output vector is then written back to the DDR. This process is repeated for each test image, with 100 iterations performed per test, with the timing of each step recorded.

5.1.4 Single-Layer CNN

The Single-Layer CNN, outlined in subsection 4.1.2, was specifically designed to minimize the computational complexity of the network. This was accomplished by removing all layers except the max-pooling layer, resulting in a simplified "dummy" CNN as depicted in the figure 5.4.

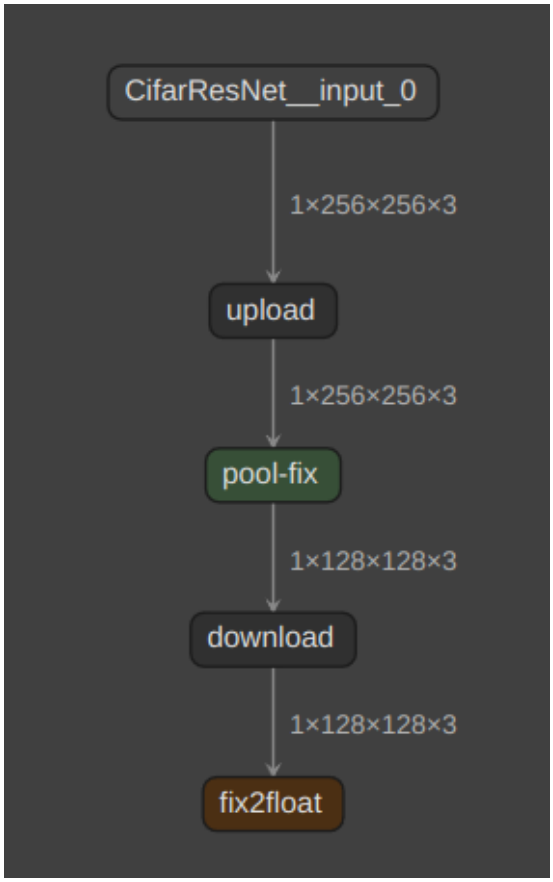


FIGURE 5.4: Single-Layer CNN - Max-pooling

The overall workflow and input data remain unchanged, but the new architecture introduces key differences both in the CNN model as well as to other important building blocks such as the host application.

The primary distinction lies in the output which in this case rather than producing a classification vector, the model now generates a complete image. This is evident and from the figure 5.4, where in this case the input image is of dimensions 256×256, passes through the max-pooling layer and the output is an image of dimensions 128×128, which is different from the classification vector that was the output in the previous model (figure 5.2). The host application needed to be changed accordingly due to the different output. Aside from that, the workflow remained the same, following the exact same steps for quantization, re-calibration and compilation of the model, following its final deployment.

5.2 Vitis Approach

As discussed in Section 4.2, the implementation of the target system for matrix multiplication involves the development of four essential components: the AI Engine graph, the PL modules, the PS application, and the System application. Each of these components plays a distinct role in the system's functionality and will be analyzed individually in the following subsections.

5.2.1 AI Engines Graph

Matrix Multiplication Kernel

The selected algorithm utilized in this thesis implements tiled matrix multiplication for matrices of INT8s, optimized for the Versal ACAP devices. Given the dimensions of the matrices, multiplication in a tiled manner is performed, processing blocks of size $M \times K$ (first matrix) and $K \times N$ (second matrix) resulting a $M \times N$ matrix. The algorithm iterates over the matrix blocks in a nested loop, loads data into vector registers and performs multiply-accumulate operations on the blocks. The results are stored back into the output matrix in a cache-friendly manner.

The use of intrinsic is extensive. Specifically, intrinsics like *aie::load_v* and *aie::store_v* were employed for efficient vector loading and storing operations, while *block_c.mul* facilitated vectorized multiplication across data blocks. The API [34] also defines the dimensions of these blocks, which in this case were chosen as $4 \times 8 \times 4$. The complete code can be found in Appendix A A.1.1.

AI Engines Hardware Build

When building the AI Engine Kernel for hardware deployment, Vitis provides an in-depth exploration of leveraging the AI Engines Array. The accompanying visuals found below illustrate the simultaneous operation of one AI Engine Kernel.

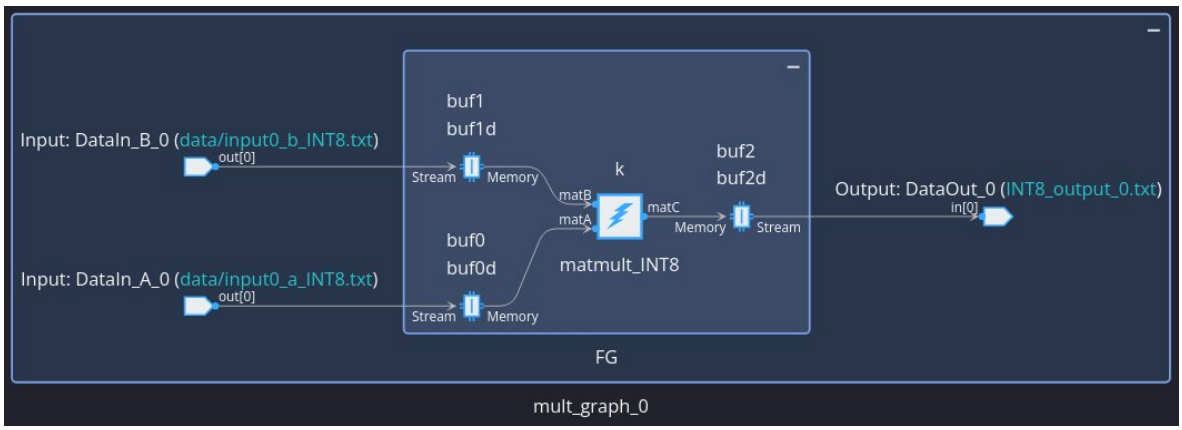


FIGURE 5.5: Matmult AI Engines kernel - Sub-graph View (1 Kernel)

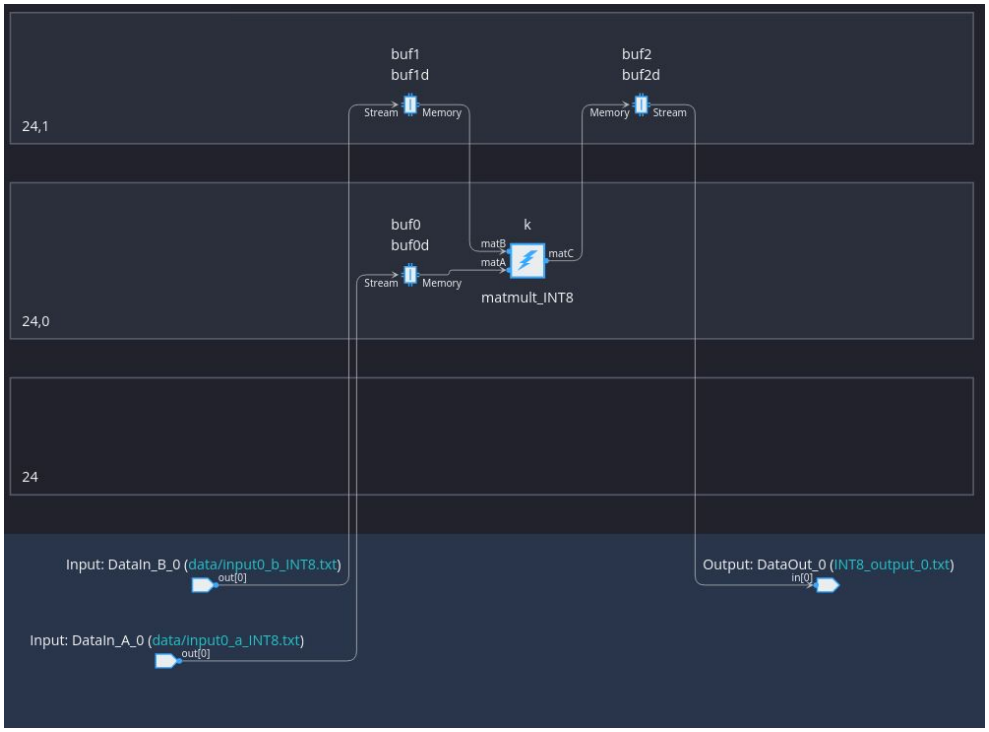


FIGURE 5.6: Matmult AI Engines kernel - Tile View (1 Kernel)

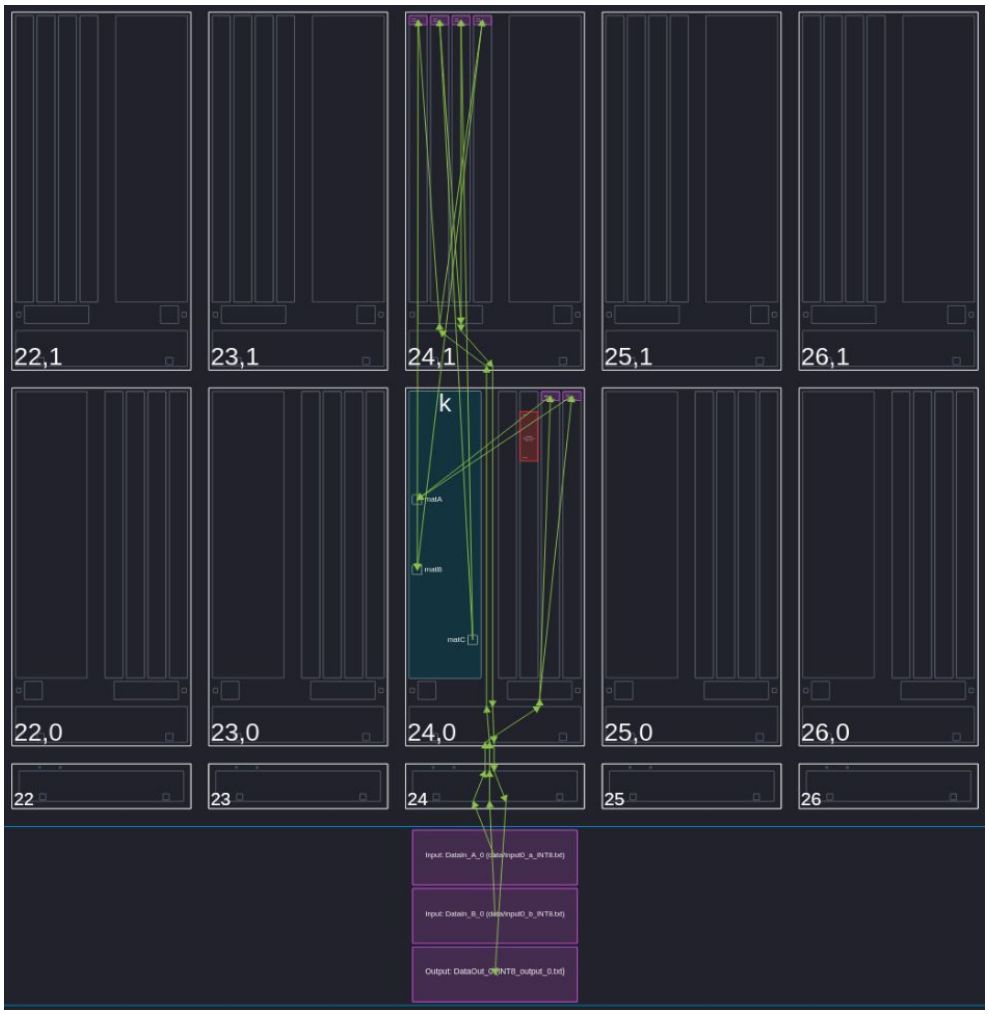


FIGURE 5.7: Matmult AI Engines kernel - Array View (1 Kernel)

The figures 5.5-5.7 illustrate a single Matrix Multiplication kernel as represented by Vitis in three distinct views: sub-graph view, tile view, and array view. The sub-graph view offers a simplified representation of the kernel, highlighting its inputs and outputs. The tile view provides a similar perspective but focuses on the AI Engines array, explicitly showing the specific tiles utilized within the array. Finally, the array view offers a more granular depiction, detailing the allocated areas of each tile and including essential information such as buffer details and graph ports. Additional images are from both Tile and Array views are provided in Appendix A (A.1.2).

Alongside the AIE kernel, input and output buffers are utilized (fig. 5.5, 5.6) operating with a ping-pong mechanism to enhance data transfer efficiency. This mechanism allows data to be fed to the AI kernel while simultaneously replenishing the buffers with new data. Additionally, the illustration shows input and output streams that are currently sourced from files; in later stages, these will be replaced with actual data movers (PL components).

5.2.2 PL components

Although the AI Engine graph serves as the core of the system, additional components are essential for ensuring full system functionality. In a real hardware deployment, matrix data must be read from and written to DDR memory, which necessitates the inclusion of specific PL components to handle these operations effectively.

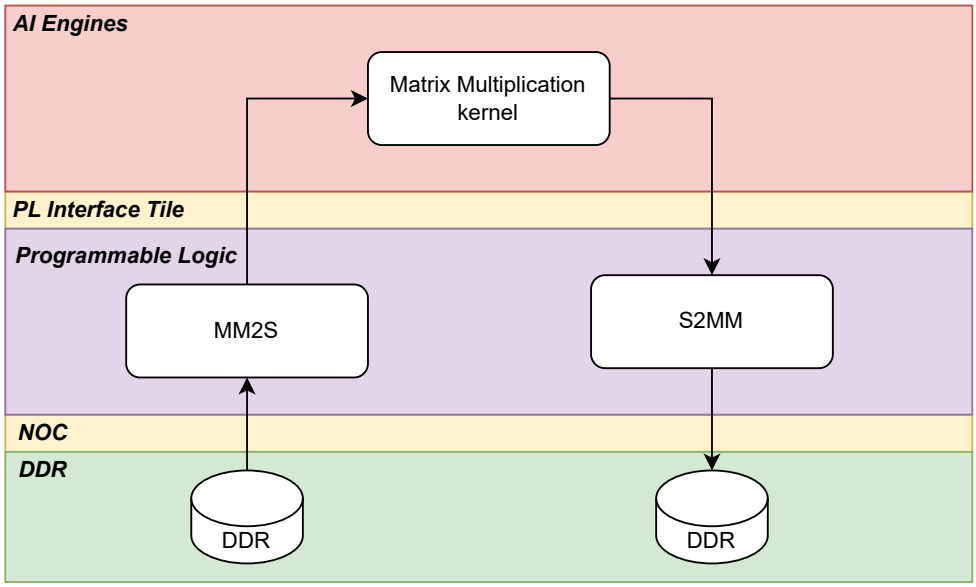


FIGURE 5.8: AI Engines - PL System

As discussed in subsection 2.1.2, the AI engines and Programmable Logic (PL) in the Versal VCK190 are directly connected via PL interface tiles using the AXI4 stream

protocol. The PL components interface with the DDR through the Network-on-Chip (NoC) using a dedicated DMA. To enable this communication, two specific PL components, mm2s and s2mm, were developed using High-Level Synthesis (HLS). These components manage data transfer between the DDR and the AI engines via dedicated AXI4 streams.

The mm2s component handles the transfer of data from specific memory locations in the DDR to the AI engines, ensuring that data is properly tiled and efficiently streamed for processing. On the other hand, the s2mm component oversees the transfer of processed data from the AI engines back to the DDR. This communication, managed by the dedicated DMA, ensures a seamless data flow through the NoC, enhancing the system's overall efficiency.

5.2.3 PS host Application

The PS host application orchestrates the entire operational workflow. Initially, the PS allocates all necessary buffers and utilizes their pointers to initialize the PL components. Once these components are successfully initialized, the AI Engine Graph is launched and executed for a single iteration. Upon completion of this iteration, both the AI Engine Graph and the PL components are terminated. The resulting matrices are then compared to verify the accuracy of the multiplication performed by the AI Engines and finally, performance metrics are computed based on total execution time.

5.2.4 System Project

The final element is the system project component, responsible for integrating all the previously mentioned building blocks to ensure the system functions as intended when deployed on the Versal device. To achieve this, a binary container is generated, which houses the indexed PL components, the AI Engine Graph and the host application. Additionally, a hardware link (HW link) file is created, defining the number of instances for each PL component and mapping the connections between the PL and AI Engines.

With these configurations established, the complete system is ready. However, before proceeding with hardware deployment, the system must be thoroughly tested via both software and hardware emulation. This step is critical for detecting any logical or linking errors across the various system components.

Software Emulation

Software emulation offers a rapid method for detecting logical errors and syntax issues within the components and their interconnections. This approach provides a more efficient and streamlined evaluation process compared to other build types, enabling

quick validation of the application's structure and logic. However, it is important to note that successful execution in software emulation does not guarantee proper functionality in hardware builds or on actual hardware. The high level of abstraction in software emulation limits its accuracy in predicting performance on physical devices, making it suitable primarily for preliminary assessments.

Hardware Emulation

Hardware emulation provides a balanced approach to simulation by combining SystemC and RTL co-simulation, offering a compromise between accuracy and speed. While it delivers a more detailed representation of hardware behavior, it does not replicate hardware behavior with complete fidelity. This can result in noticeable differences in both performance and functionality when transitioning from emulation to actual hardware. Hardware emulation effectively reproduces most interactions between the host, accelerator and memory, making it a valuable tool for debugging accelerator-related issues before deploying on physical hardware.

Similarly to the Hardware build for the AI engines graph, the Hardware emulation provides a sub-graph, a Tile and an Array view for the complete system as seen below.

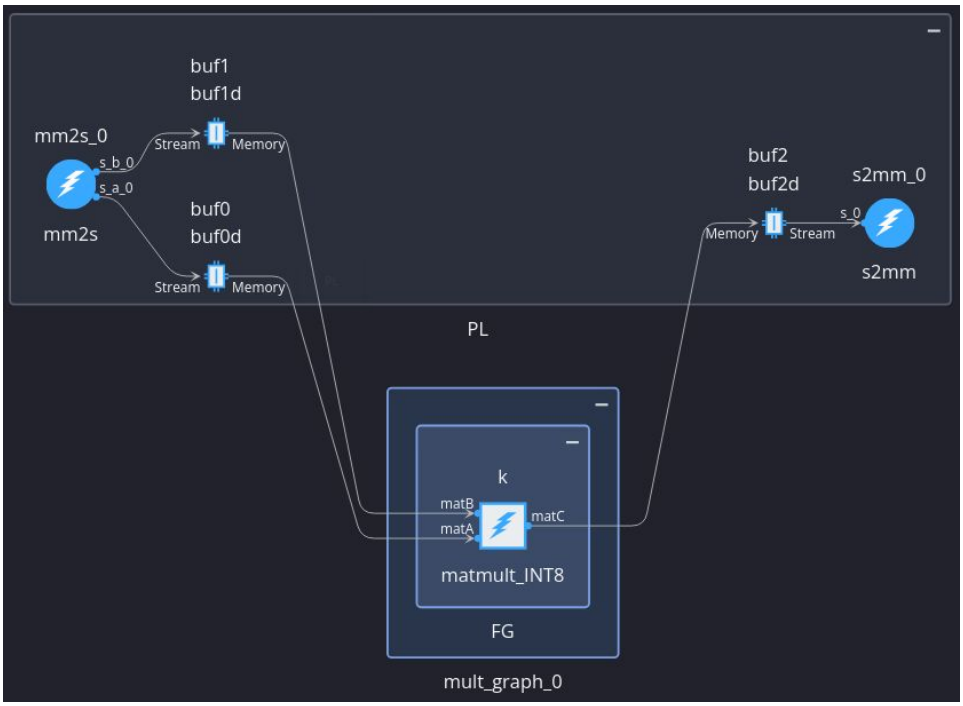


FIGURE 5.9: Matmult System - Sub-graph View

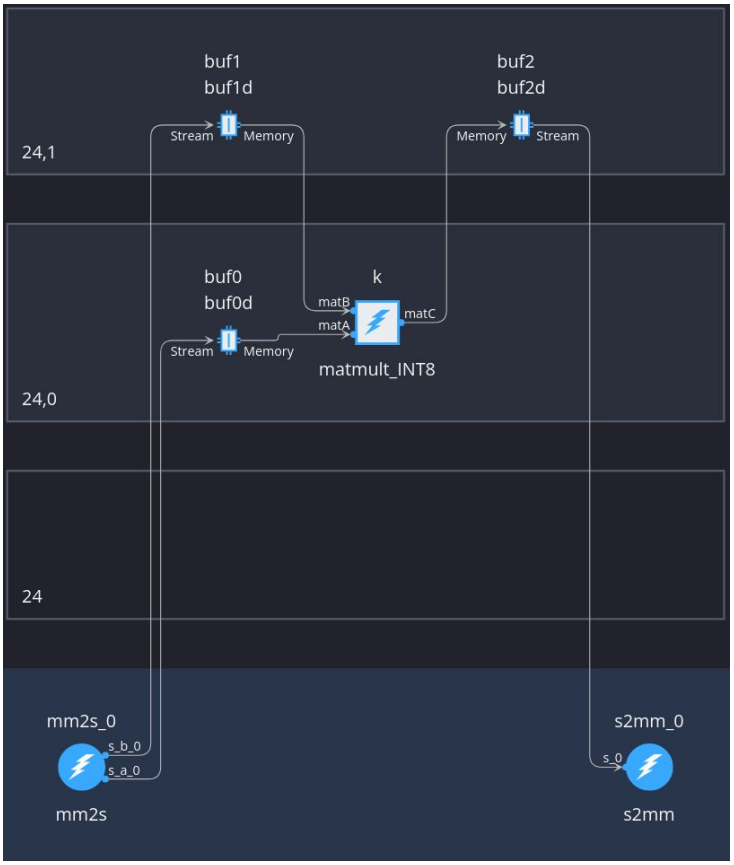


FIGURE 5.10: Matmult System - Tile View

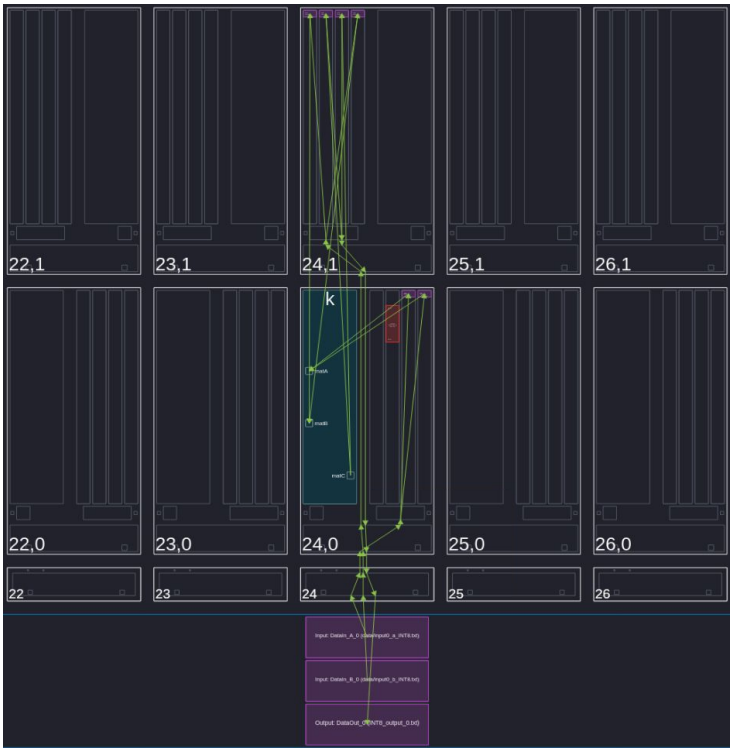


FIGURE 5.11: Matmult System - Array View

Figures 5.9-5.11 depict the matrix multiplication system as represented in Vitis. Similar to the AI Engine hardware build (Subsection 5.2.1), the matrix multiplication kernel and its associated buffers are included. However, unlike the AI Engine hardware build, the input and output streams of the AI kernel are now handled by dedicated PL components, specifically MM2S for input and S2MM for output, as previously intended. Additional images are from both Tile and Array views are provided in Appendix A (A.1.3).

Hardware Build

The hardware build is the final and most accurate stage of testing, where the design is fully implemented and executed on the physical hardware. Unlike emulation, the hardware build provides precise insight into the system's architecture as well as highlighting any unforeseen hardware-specific issues. This stage eliminates the abstraction layers present in emulations, allowing for a direct assessment of the system's functionality under actual operating conditions. Any discrepancies or failures observed here are typically due to hardware limitations, timing violations, or physical resource constraints that were not fully apparent during emulation.

Once the hardware build is complete, Vitis generates the necessary files for flashing onto the device's SD card to enable execution. Additionally, it produces a series of reports alongside a Vivado project. By opening the Vivado project, users can access these reports and view the system's block diagram.

The timing summary offers a clear assessment of the design's performance, regardless of whether the build is successful or not. It focuses on three primary timing categories: Setup, Hold, and Pulse Width. For each category, four critical metrics are reported: Worst Negative Slack (WNS), Total Negative Slack (TNS), the number of Failing Endpoints, and the Total Number of Endpoints. Below is an example of these metrics for a system using a single matrix multiplication kernel with 32×32 matrices. These figures provide valuable insights into potential timing violations and overall system performance. As no major issues were identified in these areas, the system is expected to operate as intended on the hardware.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.212 ns	Worst Hold Slack (WHS):	0.016 ns	Worst Pulse Width Slack (WPWS):	0.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	17252	Total Number of Endpoints:	17084	Total Number of Endpoints:	8131

All user specified timing constraints are met.

FIGURE 5.12: Matmult System - Timing Report

Additionally, both the block design and the NoC floor plan can be viewed to provide further details about the system. The block design outlines the connections between the needed components, while the NoC floor plan reveals data flow and communication routes.

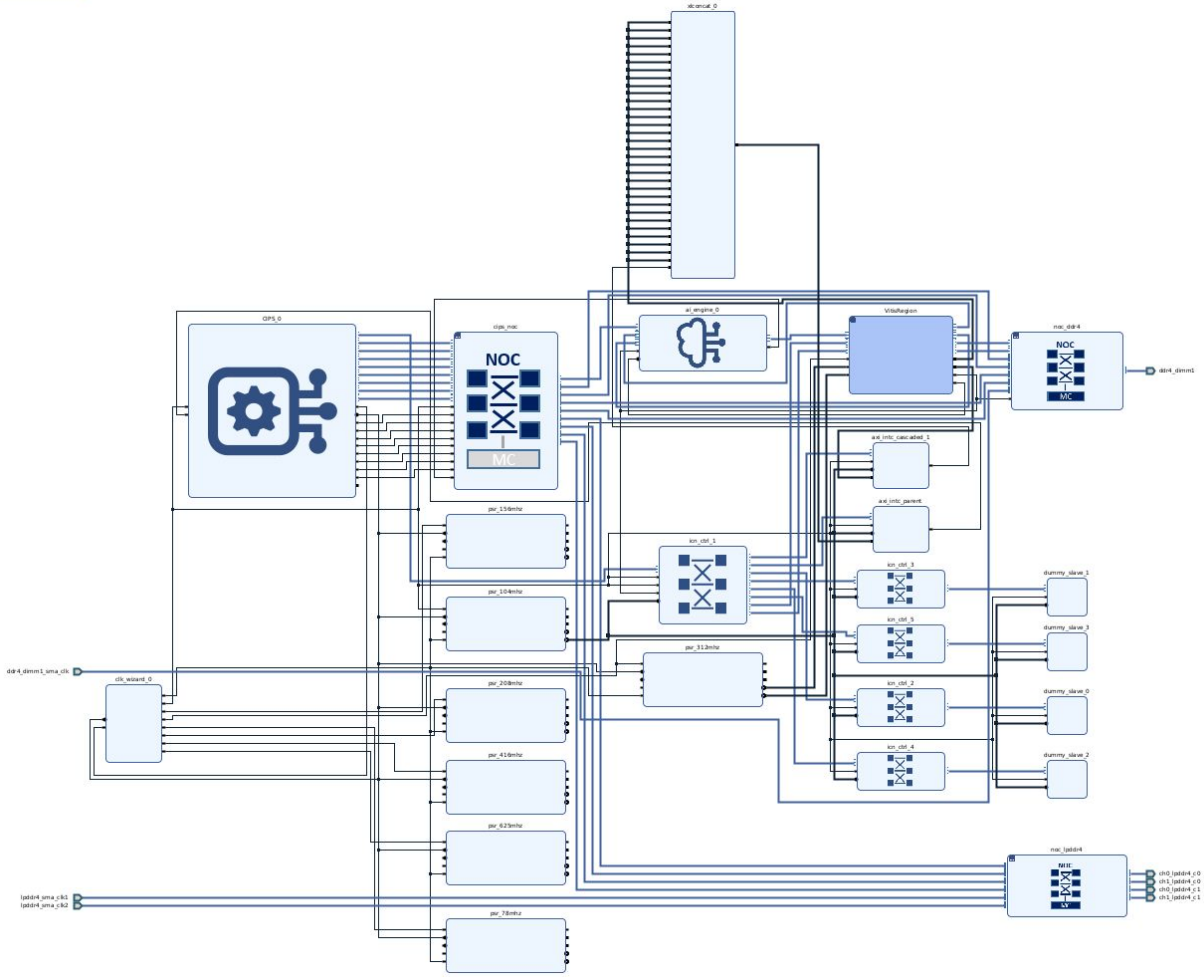


FIGURE 5.13: Matmult System - Block Design

The block design, depicted in Figure 5.13, offers a comprehensive view of the system's architecture. Upon closer inspection, it is evident that data access from the DDR is managed through the NoC, as detailed in Figure 5.8. The MM2S and S2MM modules are clocked at 312.5 MHz, although the Programmable Logic (PL) can achieve frequencies up to 500 MHz. This reduced clock speed is a direct result of the connection to the DDR and its memory controller, which imposes inherent performance constraints. Additionally, the AI Engine graph is presented, clearly illustrating all its ports and confirming the use of a single matrix multiplication kernel.

Upon further analysis, Vivado provides deeper insights into the NoC configuration and its associated connections. The diagram highlights both the NoC Master Units (NMUs) and NoC Slave Units (NSUs) involved in establishing the necessary communication pathways. As illustrated in the figure 5.14, the design utilizes three of the four available NoC channels connected to the DDR. Additionally, there are four primary NoC channels available, though only one is utilized in this particular implementation. The utilization of these channels are driven by the design's bandwidth requirements. In this case, the system's bandwidth demands do not fully saturate a single channel, rendering the activation of additional channels unnecessary.

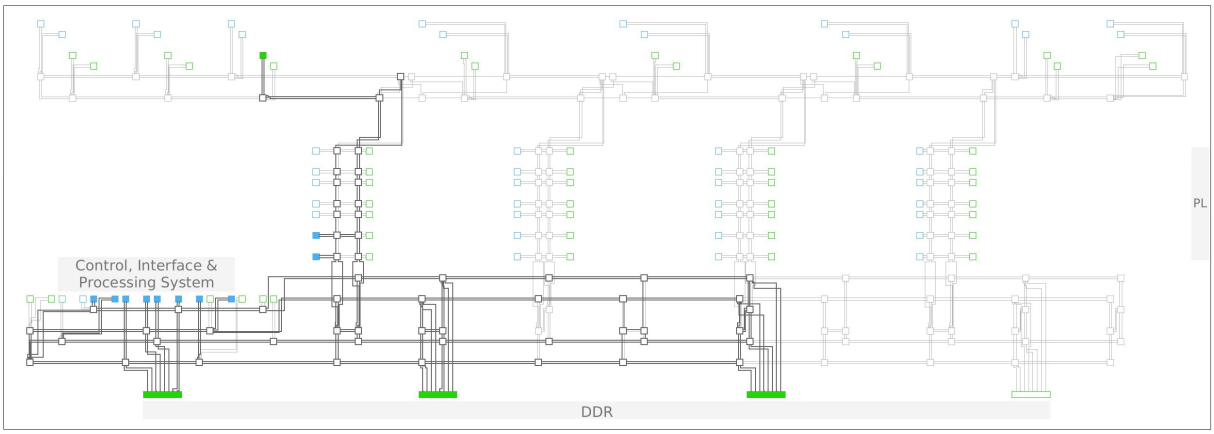


FIGURE 5.14: Matmult System - NoC

5.3 Adding More Kernels

As discussed in subsection 4.2.2, the NoC utilization and data traffic can be significantly enhanced by instantiating additional kernels in parallel. Configurations with 2, 4, 8, and 16 kernels will be employed, each progressively increasing the volume of data passing over the NoC, thereby improving overall system performance and bandwidth efficiency. System To implement these configurations, modifications are required at the top-level graph function of the AI Engine, the host application and the system application—particularly within the hardware link (Hw link). Following these adjustments, the build and emulation stages are executed as previously outlined, before proceeding to the final hardware build.

5.3.1 AI Engines Hardware Build

The hardware build process once again generates top-level diagrams, displaying the kernels along with their associated buffers. Presented below are the sub-graph, tile, and array visualizations generated by Vitis, which offer detailed representations of the system architecture for the 2-kernel and 4-kernel matrix multiplication configurations. The systems with 8 and 16 kernels were constructed following the same methodology and present a similar form

FIGURE 5.15: Matmult AI Engines Graph - Sub-graph View (2 and 4 Kernels)

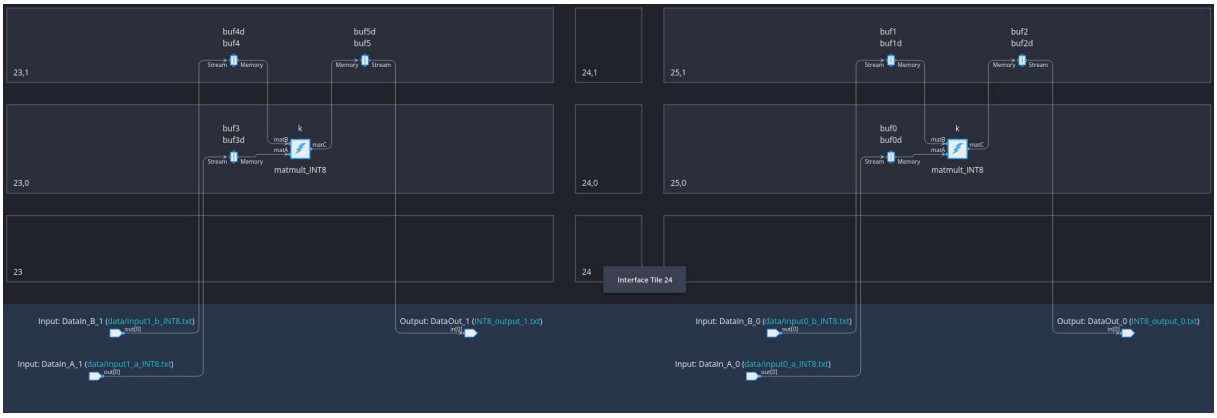


FIGURE 5.16: Matmult AI Engines Graph - Tile View (2 Kernels)

FIGURE 5.17: Matmult AI Engines Graph - Tile View (4 Kernels)

In Figures 5.15 to 5.17, the allocation of the kernels and their distribution across the AI Engine tiles are illustrated. It is important to highlight Figure 5.16, where two kernels are deployed, each occupying two tiles with no shared components between them. In contrast, Figure 5.17 shows Vitis attempting to optimize tile utilization by placing the input buffers of the third kernel on the same tile as the second kernel, demonstrating a more efficient use of resources.

5.3.2 PL Components

For the PL components, the design of each module remained consistent across all configurations. The only variation lies in the fact that each Matrix Multiplication kernel requires one MM2S and one S2MM module. Therefore, as the number of kernels increases, the number of these PL modules increases proportionally.

5.3.3 System Project

Software - Hardware Emulation

As with the previous design, each system undergoes the required emulation steps to verify the correct functionality of each kernel before proceeding to the final hardware build. Since the emulation produces diagrams similar to those from the AI Engine build, they are not repeated here. However, the results from the hardware emulation phase will be presented in the following chapter.

Hardware Build

As with the previous design, each system is built for hardware to ensure it fits within the target architecture. Given that the resulting Vivado project provides detailed insights into the system's architecture, it is important to highlight key differences between the hardware builds.

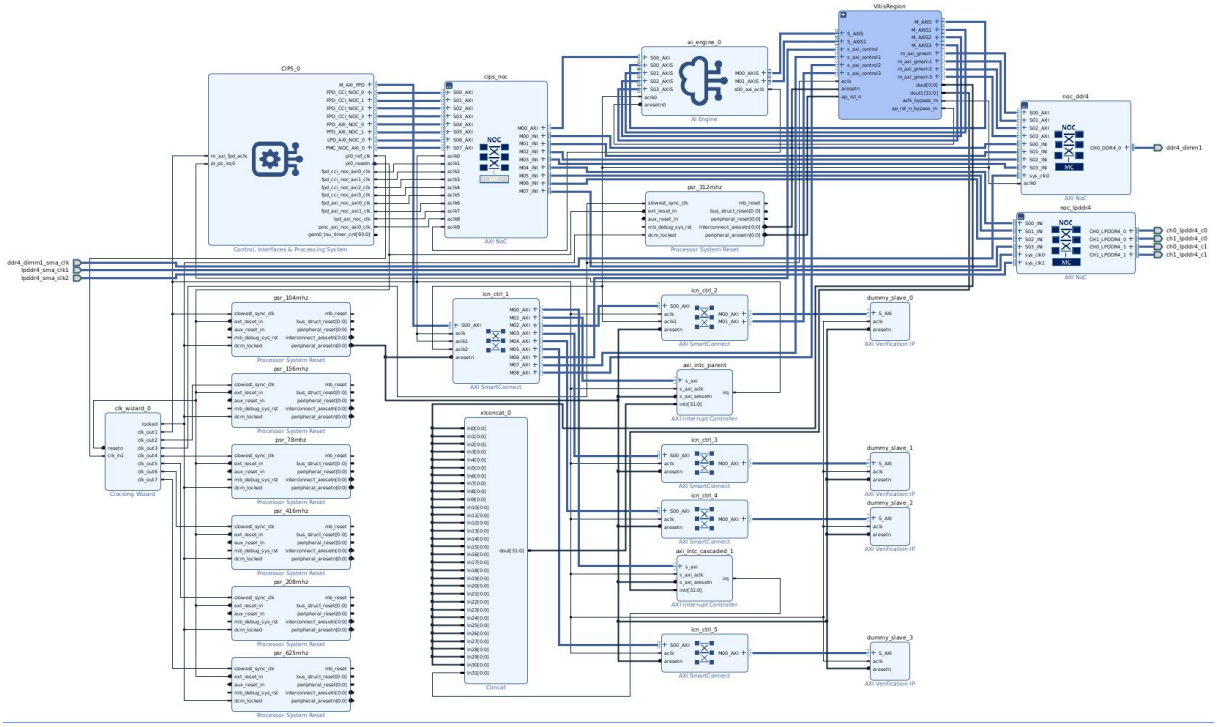


FIGURE 5.18: Matmult System - Block Design (2 Kernels)

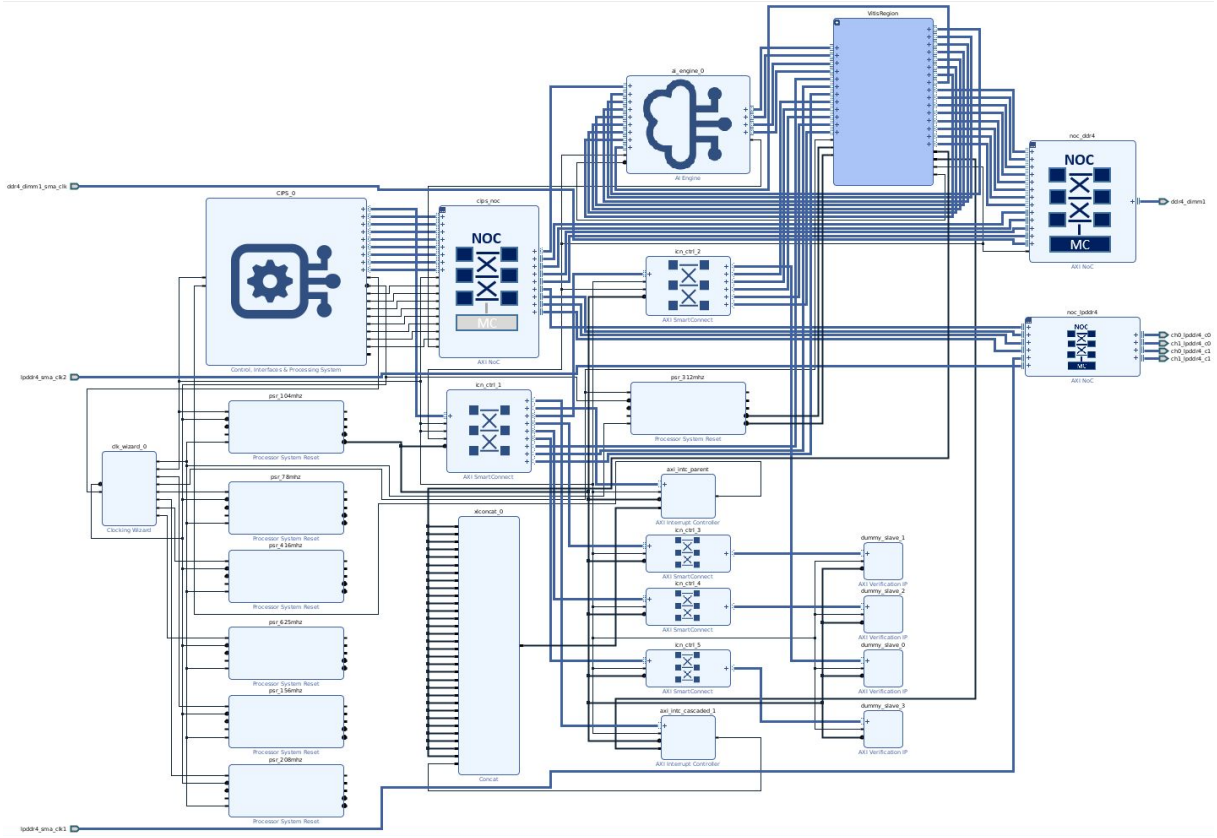
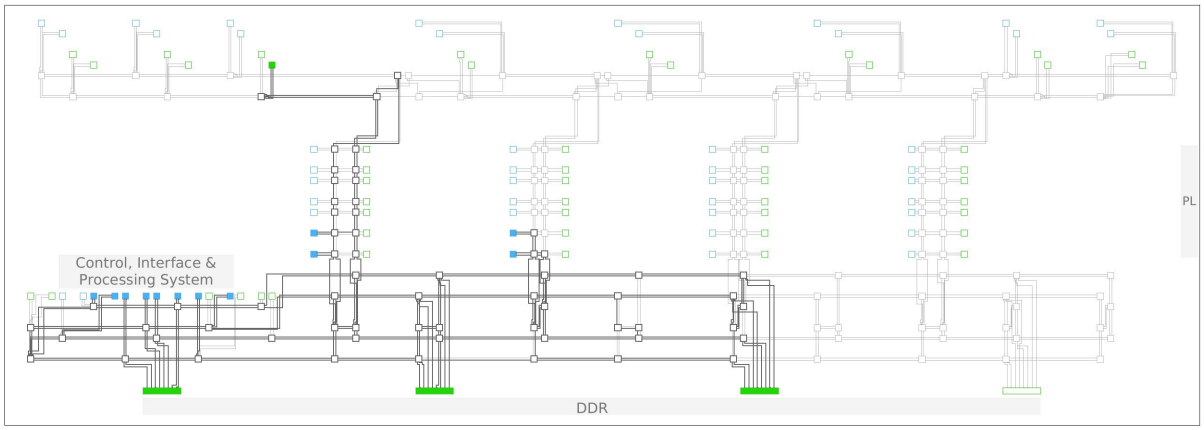
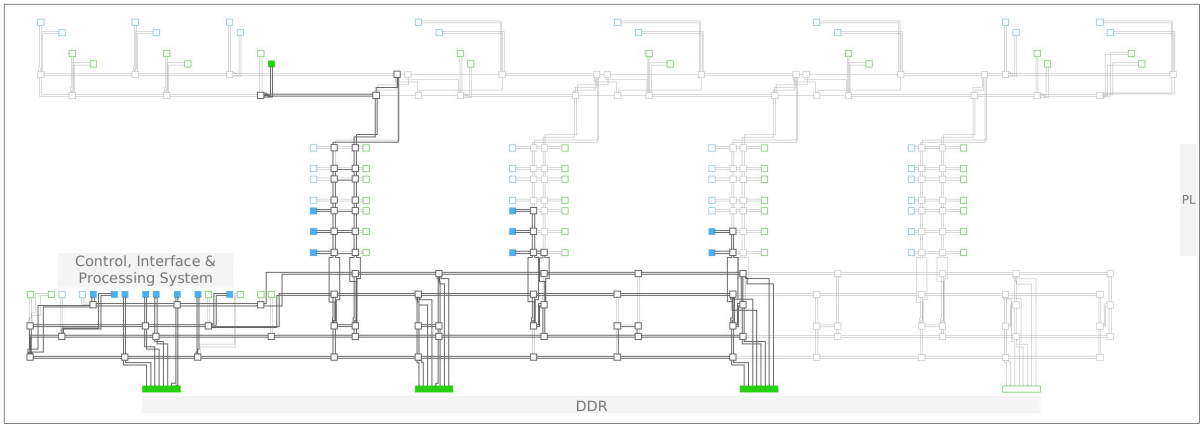


FIGURE 5.19: Matmult System - Block Design (4 Kernels)

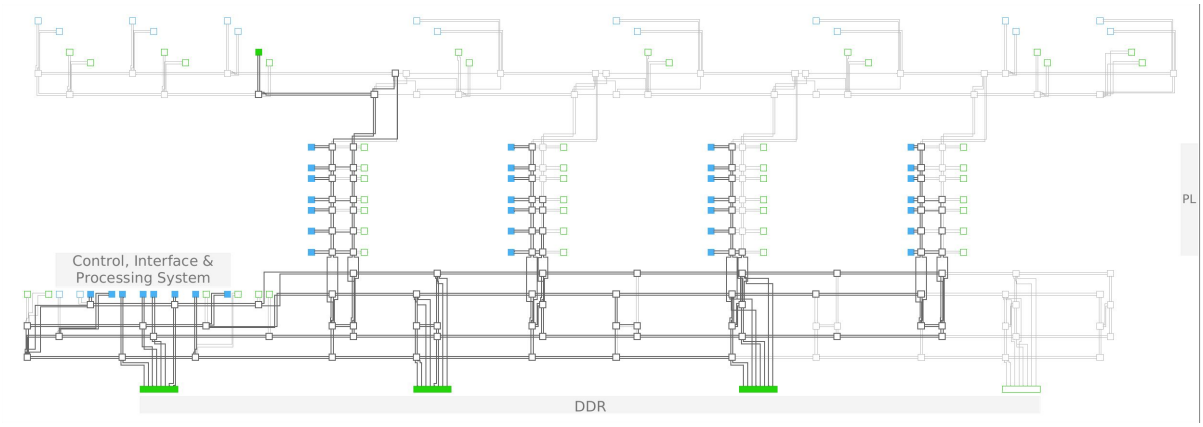
As shown in Figures 5.18-5.19, increasing the number of kernels directly impacts the system's complexity. This is reflected in the NoC, AI Engine, and Vitis Region (PL) components, which require additional ports as anticipated. However, key system parameters, such as the clock frequencies for the PL and AI Engines, remain consistent across all configurations.



(A) Matmult System - NOC (2 Kernels)



(B) Matmult System - NOC (4 Kernels)



(C) Matmult System - NOC (16 Kernels)

FIGURE 5.20: Matmult System - NOC for Different Kernel Configurations

The increased utilization of the NoC is confirmed by the NoC diagrams, which show that even with the addition of a single kernel, parts of the second NoC channel are engaged. With the addition of two more kernels, the third channel is also utilized. This pattern persists throughout the tests, and in the largest configuration with 16 kernels, multiple NoC channels are fully utilized.

Chapter 6

Evaluation

This chapter provides a thorough overview of the evaluation board and the associated test setup. It details the results obtained from each test case, offering a comparison with the anticipated values to assess performance.

6.1 Specification of the Target Platform

This thesis presents a comparative analysis of the performance between the embedded ARM processor and custom systems utilizing Versal's additional resources, including AI Engines and Programmable Logic (PL). The primary focus, however, is on evaluating the performance of the NoC (Network-on-Chip).

6.1.1 ARM Cortex-A72

The Versal VCK190 integrates an advanced ARM processing subsystem, specifically designed to enhance the flexibility and performance of FPGA-based systems. This subsystem consists of quad-core ARM Cortex-A72 processors, which serve as the primary application processors, offering high-performance compute capabilities for general-purpose applications. Additionally, it includes dual-core ARM Cortex-R5F processors dedicated to real-time operations, ideal for handling time-critical tasks such as system monitoring and control.

Feature	**Specification**
Processor Model	ARM Cortex-A72
Number of Cores	Quad-core
Architecture	64-bit
Clock Speed	Up to 1.5 GHz
Additional Processor	ARM Cortex-R5

TABLE 6.1: Embedded ARM Processor Specifications

Versal VCK190 ACAP Evaluate Board

The Versal VCK190 is a development kit designed for the Versal AI Core Series, a heterogeneous platform that integrates adaptable hardware with AI engines, DSP engines, and scalar processing elements. It is optimized for applications like AI inference, machine learning, 5G communications, and data center acceleration. The VCK190 features the VC1902 Versal AI Core device, offering powerful compute capabilities, high-speed connectivity, and rich memory interfaces. The platform supports Vitis AI development tools, allowing developers to accelerate AI workloads with ease. Its flexible architecture is designed to handle diverse tasks with enhanced efficiency and performance.

Subsystem	Specification
AI Engines	- Quantity: 400 - Clock Speed: Up to 1.2 GHz - Performance: Up to 9.2 TOPS
Programmable Logic (PL)	- Logic Cells: 1.4 million - DSP Slices: 1,056 - BRAM: 56 MB - Clock Speed: Up to 500 MHz
Deep Learning Processor Unit (DPU)	- Number: 1 or 2 DPUs - Performance: Up to 6 TOPS - Supports INT8 and FP16 precisions

TABLE 6.2: Subsystem Specifications of the Versal ACAP VCK190

6.1.2 Metrics

Throughput, or bandwidth, refers to the number of tasks a module or system can process within a given time frame, typically measured per second. In this work, throughput is calculated by dividing the total data transferred via the Network-on-Chip (NoC) —specifically, the size of the image data—by the total time required to execute the task for which each approach is designed.

6.2 Vitis AI Approach

6.2.1 System Verification

To validate the correct operation of the ResNet-20 model deployed on the Versal DPU, as generated by Vitis AI, an initial baseline test was conducted using the unmodified CNN. This involved deploying the original model and testing it with the default CIFAR-10 dataset. Since the results were consistent with expected performance in terms of both accuracy and speed, thereby confirming the correct implementation.

Subsequently, modified versions of the CNN were deployed, beginning with the altered ResNet-20 and followed by the single-layer CNN. For these custom architectures, accuracy metrics were not sufficient for validation. Instead, functionality was confirmed by inspecting the output. In the case of the modified ResNet-20, the output vector varied consistently, confirming that different images were being processed correctly, while for the single-layer CNN, the output images exhibited reduced dimensions —one-fourth of the original size—indicating that the custom architecture was functioning as intended.

6.2.2 Implemented System Characteristics

In this project, it is not possible to directly display the resource utilization metrics for the models produced by Vitis AI. This is because Vitis AI focuses on deploying machine learning models using the Deep Learning Processing Unit (DPU) as the primary IP core. The DPU, being a customizable IP core, is integrated into the FPGA design through Vivado or the Vitis platform, where its configuration—such as the number of cores and features—determines the consumption of FPGA resources like LUTs, BRAM, and URAM. However, Vitis AI itself does not provide direct insights into the FPGA resource usage for specific models.

6.2.3 Experiments that were Sought Through

The primary goal of the experiments was to evaluate the throughput of the NoC using a series of controlled tests. For the modified ResNet-20 CNN, the tests began with images sized at 64×64, progressively increasing to 128×128 and eventually reaching 2048×2048. This gradual increase allowed for a steady rise in traffic over the NoC in each test. To ensure consistency and accuracy in the results, the same methodology was applied to the single-layer CNN, using the same dataset. This consistent approach enabled more reliable evaluation of the NoC's performance under different conditions.

6.2.4 Results

Modified ResNet-20

Figure 6.1 illustrates the execution time (in milliseconds) as a function of image size (in kilobytes) for the modified version of ResNet-20 running on the Versal DPU. The figure uses a logarithmic scale for the x-axis since the data size increases exponentially with image resolution. Despite the logarithmic scale, the plot displays a straight line, signifying a steady, linear relationship between data size and execution time. This suggests that the system efficiently scales with image size.

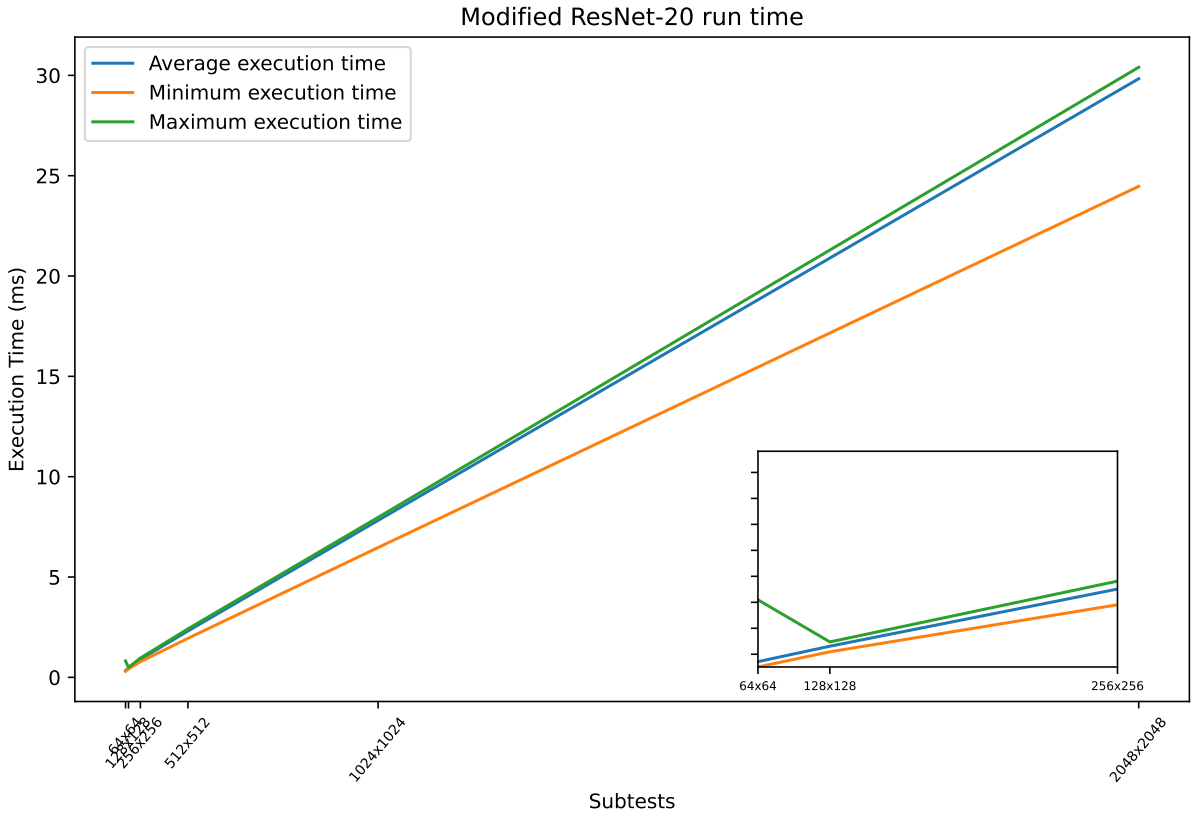


FIGURE 6.1: Modified ResNet-20 - Execution Time Over Image Resolution

By analyzing the throughput, as shown in table 6.3, it is evident that there is a significant improvement from 64x64 to 128x128, followed by a peak at 256x256. After this, the throughput stabilizes, with marginal improvements up to 2048x2048. This trend indicates that the DPU achieves near-optimal throughput, for this specific system, for larger image sizes, reaching up to 3.4 Gbps for 2048x2048 images. The initial increase in throughput followed by stabilization suggests that the DPU fully utilizes its memory and processing capabilities for larger inputs, which is expected as the system approaches its maximum data throughput limits.

Resolution	Size	Throughput
64 × 64	12 KB	1.117 Gbps
128 × 128	49 KB	1.900 Gbps
256 × 256	197 KB	2.431 Gbps
512 × 512	786 KB	3.063 Gbps
1024 × 1024	3,146 KB	3.326 Gbps
2048 × 2048	12,583 KB	3.403 Gbps

TABLE 6.3: Throughput for Various Image Resolutions (Modified ResNet-20)

Single-Layer CNN

Similarly to Figure 6.1, Figure 6.2 illustrates the execution time (in milliseconds) as a function of image size (in kilobytes) for the Single-Layer CNN running on the Versal DPU. The figure poses similar form to the previous one, again suggesting that the systems efficiently scales with image size.

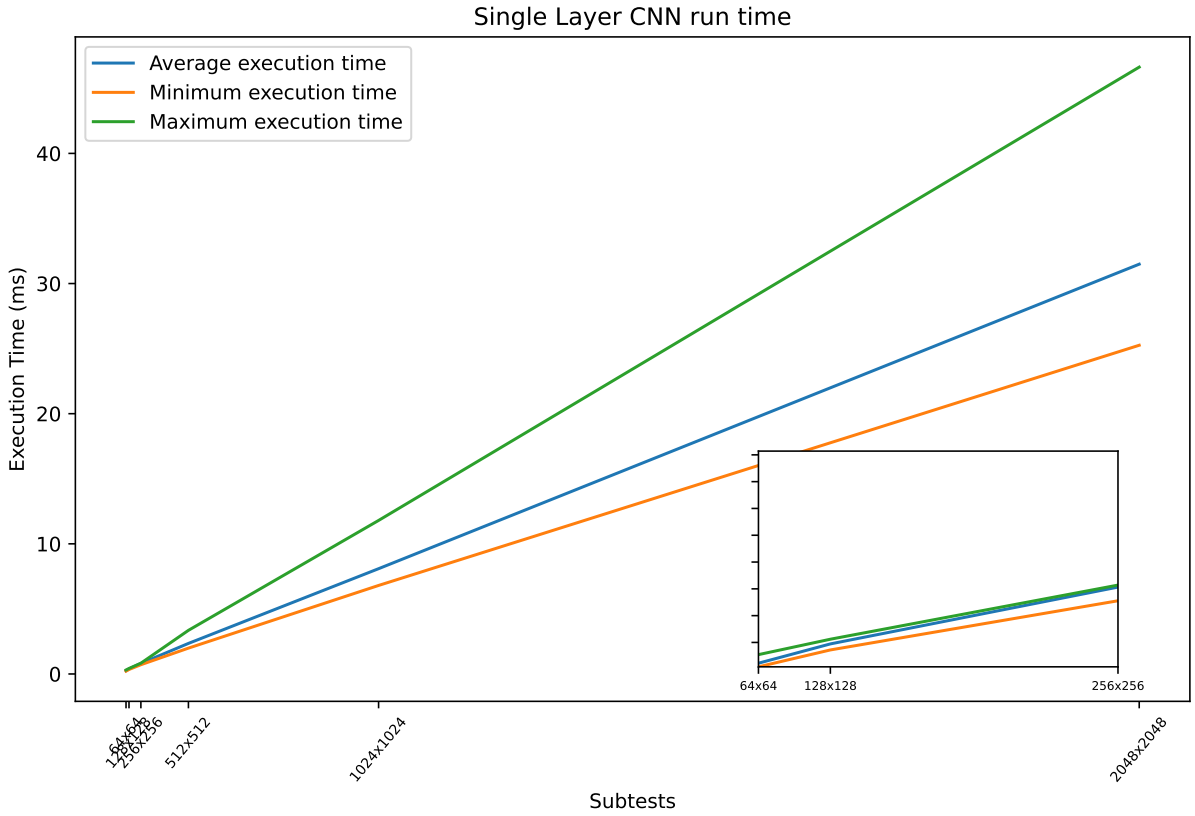


FIGURE 6.2: Single-Layer CNN - Execution Time Over Image Resolution

By analyzing the throughput results in Table 6.4, it is evident that the throughput remains relatively stable across all tests, with an average of approximately 3.8 Gbps and a maximum at 4.03 Gbps, which marks a significant improvement over previous results. However, it is also clear that a performance limit has been reached, as increasing the image resolution does not lead to a notable increase in throughput. In fact, the increase observed in the final two tests is minimal, approximately 0.014 Gbps.

Additionally, an anomaly is present in the first throughput value, which is both negative and unusually large. This discrepancy arises from the calculation process, where an "overhead" value, representing the time required to initialize the DPU, is subtracted. This overhead is determined by fitting the model and identifying the point at which the initialization time intersects the Y-axis.

Input Resolution	Input Size	Output Resolution	Output Size	Throughput
64 × 64	12 KB	32 × 32	3 KB	-12.288 Gbps
128 × 128	49 KB	64 × 64	12 KB	3.668 Gbps
256 × 256	197 KB	128 × 128	49 KB	3.523 Gbps
512 × 512	786 KB	256 × 256	197 KB	3.757 Gbps
1024 × 1024	3,146 KB	512 × 512	786 KB	4.015 Gbps
2048 × 2048	12,583 KB	1024 × 1024	3,146 KB	4.029 Gbps

TABLE 6.4: Throughput for Various Image Resolutions (Single-Layer CNN)

6.3 Vitis approach

6.3.1 System Verification

To ensure the accuracy of the matrix multiplication performed by the AI Engines, the same operation is also executed on the embedded ARM processor of the Versal evaluation board and the results from both executions are compared at the end of each test run to verify consistency. Additionally, system integrity was validated using software and hardware emulation for each individual step, as previously mentioned. Ideally, the system would be deployed on the actual hardware to confirm functionality and collect the desired performance metrics, however, even though each system was able to be built for hardware, due to deployment issues and time constraints, this step falls outside the scope of this thesis.

6.3.2 Implemented System Characteristics

This approach results in the generation of 16 distinct systems, comprising four different matrix sizes and four varying kernel counts per system. Due to the complexity involved, providing a detailed breakdown of the resources for each system individually is impractical. Instead, the primary focus is on the utilization of the NoC, as this is a key aspect of system performance. Relevant data concerning NoC utilization has already been discussed in the preceding subsection 5.3.3.

6.3.3 Experiments that were Sought Through

Keeping the same objective, the tests conducted with two adjustable variables. Initially, an 8x8 matrix was used, beginning with a single kernel and progressively increasing to 2, 4, 8, and 16 kernels. This process was repeated for larger matrices—specifically 16x16 and 32x32—to evaluate performance across different matrix sizes.

6.3.4 Results

As previously discussed, the results presented here are derived from the Hardware Emulation phase provided by the Vitis framework. As detailed in subsections 2.3.1 and 5.2.4, hardware emulation serves as a highly effective means to simulate the design, offering a balance between execution speed and accuracy. While this stage replicates most of the system interactions that would occur in actual hardware, it is important to note that the outcomes of hardware emulation differ from those of real hardware implementation in terms of of performance and thus, hardware emulation results do not provide a precise reflection of the device's real-time performance.

The following three tables present the results obtained from each respective test case.

Number of Kernels	Arm Processor Time	Hardware system Time	Throughput
1	0.425 (ms)	5.724 (s)	0.268 (Kbps)
2	0.731 (ms)	7.868 (s)	0.195 (Kbps)
4	1.945 (ms)	14.308 (s)	0.107 (Kbps)
8	2.798 (ms)	33.440 (s)	0.046 (Kbps)
16	2.968 (ms)	93.546 (s)	0.016 (Kbps)

TABLE 6.5: Matrix Multiplication with 8×8 Matrices

Number of Kernels	Arm Processor Time	Hardware system Time	Throughput
1	0.541 (ms)	14.553 (s)	0.422 (Kbps)
2	1.753 (ms)	17.195 (s)	0.357 (Kbps)
4	2.507 (ms)	26.894 (s)	0.228 (Kbps)
8	4.268 (ms)	56.907 (s)	0.108 (Kbps)
16	4.402 (ms)	273.030 (s)	0.023 (Kbps)

TABLE 6.6: Matrix Multiplication with 16×16 Matrices

Number of Kernels	Arm Processor Time	Hardware system Time	Throughput
1	2.807 (ms)	46.346 (s)	0.5303 (Kbps)
4	9.688 (ms)	81.194 (s)	0.419 (Kbps)
8	17.809 (ms)	215.746 (s)	0.114 (Kbps)
16	33.142 (ms)	1144.759 (s)	0.021 (Kbps)

TABLE 6.7: Matrix Multiplication with 32×32 Matrices

As demonstrated, in each case, the time required to complete matrix multiplication on the AI Engines exceeds that of the ARM processor. Furthermore, the execution time for the hardware system increases linearly with the number of kernels, an outcome that deviates significantly from expectations. This result appears counter-intuitive, suggesting that the system might be less efficient than anticipated.

However, it is most likely that this behavior stems from the inherent limitations of the emulation stage. As the emulation is entirely managed by the host CPU, simulating complex hardware systems places significantly higher demands on processing resources compared to running tasks natively on the ARM processor, thus accounting for the observed performance disparity. In a real-world implementation on the Versal board, the kernels would execute in parallel—a behavior not mirrored in the emulation, as indicated by the linear increase in execution time. Additionally, the notably low throughput values are a direct consequence of these suboptimal timing results.

6.4 Results Discussion

The results obtained in this study are intriguing. AMD/Xilinx reports a maximum achievable throughput of 16 Gbps, while the bandwidth observed in the Vitis AI experiments reached only a quarter of that value. Following discussions with AMD/Xilinx engineers, the most plausible explanation is the compiler's utilization of only a single NoC channel, likely due to its assessment that additional channels were unnecessary. The engineers further suggested that more aggressive compiler optimizations could be incorporated in future versions to address this limitation. It is worth noting that unimplemented advertised features are not uncommon in the industry.

Additionally, the Vitis results should be interpreted cautiously, as they are derived from emulations rather than real-world executions. Nonetheless, initial concerns regarding the system's lack of parallel execution and potentially improved performance on actual hardware were confirmed by AMD/Xilinx engineers, who affirmed that throughput on physical systems should approach the advertised values.

This finding is consistent with a previous thesis by Max Verse [48], who observed a maximum throughput of 10 to 11 Gbps during similar testing. Verse attributed this lower-than-expected value to the fact that the application was running on a Linux-based Petalinux environment, where many processes and scheduling tasks remain outside the user's direct control, likely contributing to the reduced throughput.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

In conclusion, this thesis has thoroughly examined the capabilities of the AMD/Xilinx Versal platform, focusing on the performance and workflow provided by the hard-IP Network-on-Chip (NoC), utilizing both the Vitis and Vitis AI tools. The results demonstrate that the NoC offers a high level of abstraction, simplifying system integration by streamlining connections between components. This allows designers to focus on core elements without the complexity of managing low-level interconnects.

With regard to the toolsets, Vitis AI proves to be a highly efficient and user-friendly option for rapidly implementing entire CNN architectures, requiring minimal hardware expertise. However, this convenience comes at the cost of limited customization, as it operates within a more closed-source environment, resulting in lower resource utilization and reduced control over the final design. Conversely, the Vitis toolchain offers greater flexibility and control, allowing developers to fine-tune performance, though at the expense of increased development time and effort, particularly when testing and configuring each component.

Ultimately, the Versal platform, with its integrated Hard-IP NoC, provides a robust foundation for high-performance applications. However, the selection of the appropriate development tools is critical to fully harness its potential. For rapid prototyping and deployment, particularly in scenarios where time-to-market is a priority, Vitis AI proves to be an ideal solution. On the other hand, applications with stringent performance and timing constraints demand a more tailored approach. In such cases, leveraging Vitis for custom development becomes indispensable, with the NoC playing a pivotal role in ensuring optimal system performance.

7.2 Future Work

Future research and development in this domain offer a variety of promising directions that could extend and enhance the current findings. Key areas for further exploration include:

- **Vitis AI Enhancement:** Future work could delve deeper into optimizing the Vitis AI framework:
  - Increasing the batch size in the host application to assess its impact on system performance, specifically in terms of processing speed and throughput.
  - Evaluating the scalability of the system by deploying multiple DPUs, with a particular focus on comparing performance gains with two DPUs versus one.
- **Refining the Vitis-Based Implementation:** Further improvements could be made by investigating alternative configurations in the Versal architecture:
  - Analyzing the performance of multiplication operations under standard configurations to identify any efficiency bottlenecks.
  - Transitioning the implementation to a bare-metal environment, eliminating the potential uncertainties posed by the Linux operating system, particularly regarding its scheduling and resource allocation.

Appendix A

AppendixA

A.1 AI Engine Graph Implementation

A.1.1 Matrix Multiplication Kernel Code

```
template <unsigned M, unsigned K, unsigned N, typename T>
[[gnu::always_inline]]
static void mmul_blocked_unrolled(unsigned rowA, unsigned colA, unsigned colB, const T *
    __restrict A, const T *__restrict B, T *__restrict C) {
    using MMUL = aie::mmul<M, K, N, T, T>;
    [[chess::min_loop_count(2)]]
    for (unsigned z = 0; z < rowA; z += 2) {
        T *c_ptr[2];
        c_ptr[0] = C + MMUL::size_C * (z * colB + 0);
        c_ptr[1] = C + MMUL::size_C * ((z + 1) * colB + 0);

        [[chess::min_loop_count(2)]]
        for (unsigned j = 0; j < colB; j += 2) {
            unsigned i = 0;
            const T *a_ptr = A + MMUL::size_A * (z * colA + i);
            const T *b_ptr = B + MMUL::size_B * (i * colB + j);

            aie::vector<T, MMUL::size_A> block_a[2];
            aie::vector<T, MMUL::size_B> block_b[2];

            block_a[0] = aie::load_v<MMUL::size_A>(a_ptr);
            a_ptr += MMUL::size_A;
            block_a[1] = aie::load_v<MMUL::size_A>(a_ptr);
            a_ptr += MMUL::size_A;
            block_b[0] = aie::load_v<MMUL::size_B>(b_ptr);
            b_ptr += colB * MMUL::size_B;
            block_b[1] = aie::load_v<MMUL::size_B>(b_ptr);
            b_ptr += colB * MMUL::size_B;

            MMUL block_c[2][2];
            block_c[0][0].mul(block_a[0], block_b[0]);
            block_c[0][1].mul(block_a[0], block_b[1]);
            block_c[1][0].mul(block_a[1], block_b[0]);
            block_c[1][1].mul(block_a[1], block_b[1]);

            [[chess::prepare_for_pipelining, chess::min_loop_count(3)]]
            for (i = 1; i < colA; ++i) {
                block_a[0] = aie::load_v<MMUL::size_A>(a_ptr);
                a_ptr += MMUL::size_A;
```

```
                block_a[1] = aie::load_v<MMUL::size_A>(a_ptr);
                a_ptr += MMUL::size_A;
                block_b[0] = aie::load_v<MMUL::size_B>(b_ptr);
                b_ptr += colB * MMUL::size_B;
                block_b[1] = aie::load_v<MMUL::size_B>(b_ptr);
                b_ptr += colB * MMUL::size_B;

                block_c[0][0].mac(block_a[0], block_b[0]);
                block_c[0][1].mac(block_a[0], block_b[1]);
                block_c[1][0].mac(block_a[1], block_b[0]);
                block_c[1][1].mac(block_a[1], block_b[1]);
            }
            aie::store_v(c_ptr[0], block_c[0][0].template to_vector<T>());
            c_ptr[0] += MMUL::size_C;
            aie::store_v(c_ptr[0], block_c[0][1].template to_vector<T>());
            c_ptr[0] += MMUL::size_C;
            aie::store_v(c_ptr[1], block_c[1][0].template to_vector<T>());
            c_ptr[1] += MMUL::size_C;
            aie::store_v(c_ptr[1], block_c[1][1].template to_vector<T>());
            c_ptr[1] += MMUL::size_C;
        }
    }
}
```

A.1.2 Matrix Multiplication Kernel - Views

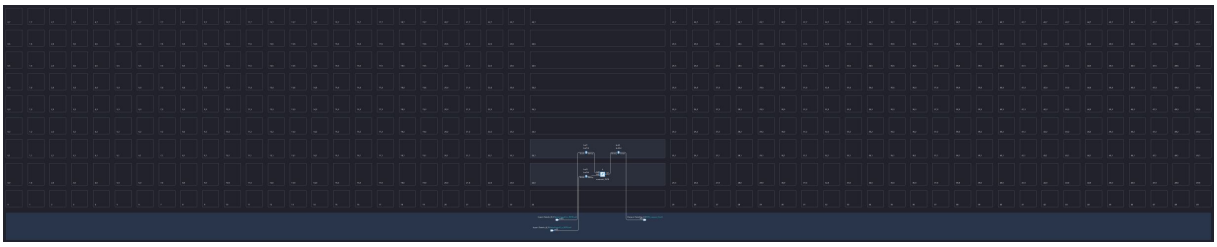


FIGURE A.1: Matmul_AIE_kernel - Tile view (Zoomed out)

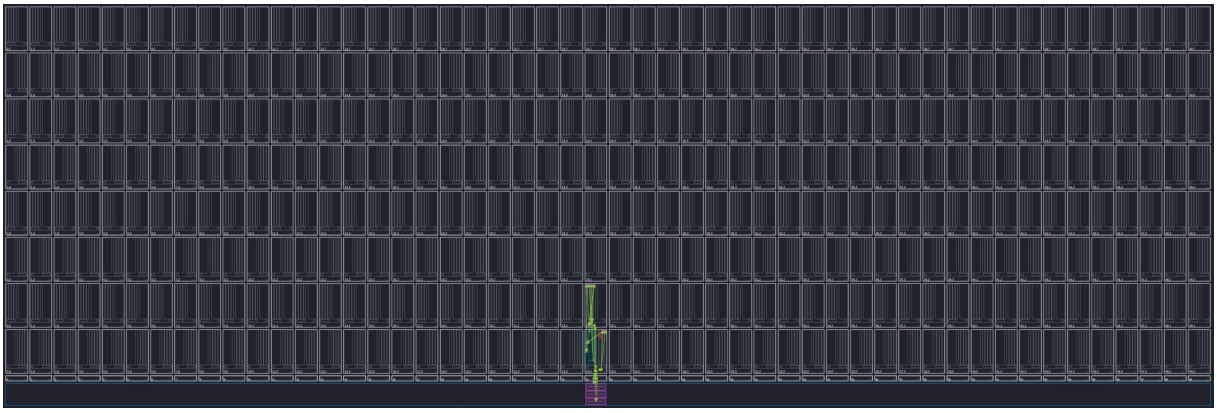


FIGURE A.2: Matmul_AIE_kernel - Array view (Zoomed out)

A.1.3 Matrix Multiplication System - Views

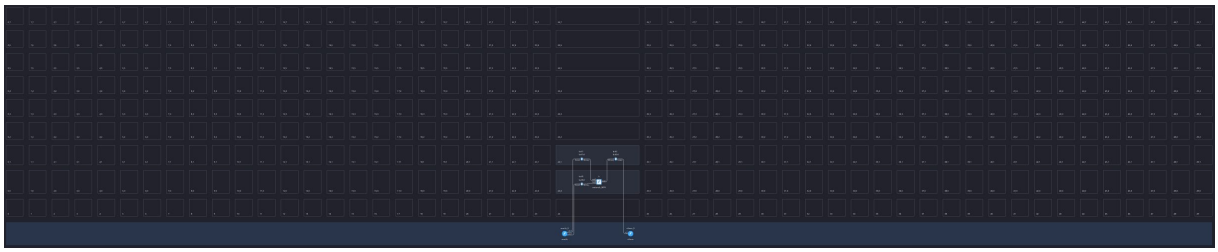


FIGURE A.3: Matmul_AIE_kernel - Tile view (Zoomed out)

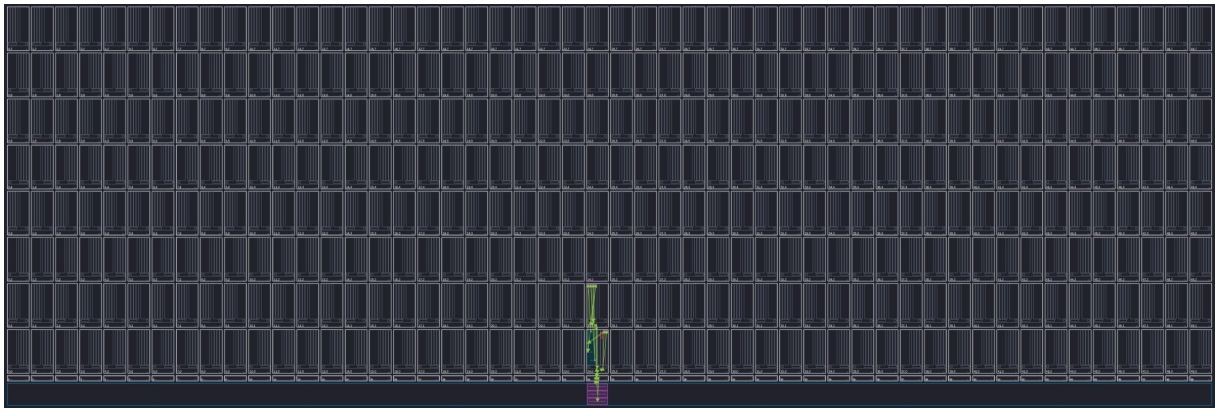


FIGURE A.4: Matmul_AIE_kernel - Array view (Zoomed out)

References

[1] AMD/Xilinx inc. “Versal: The First Adaptive Compute Acceleration Platform (ACAP)”. In: 1.11.29 (Sept. 2020), pp. 1–4. URL: https://docs.amd.com/v/u/en-US/wp505-versal-acap.

[2] AMD/Xilinx inc. “AI Engines and Their Applications (WP506)”. In: 1.2.16 (Dec. 2022), pp. 1–18. URL: https://docs.amd.com/v/u/en-US/wp506-ai-engine.

[3] Xilinx inc. “Versal Adaptive SoC AI Engine Architecture Manual (AM009)”. In: 1.6.18 (Aug. 2023), pp. 9–14, 35–36, 39–41. URL: https://docs.amd.com/r/en-US/am009-versal-ai-engine.

[4] AMD/Xilinx inc. “Versal™ Architecture and Product Data Sheet: Overview (DS950)”. In: 2.1.24 (Apr. 2024), pp. 25–30. URL: https://docs.amd.com/v/u/en-US/ds950-versal-overview.

[5] AMD/Xilinx inc. “Versal Adaptive SoC Programmable Network on Chip and Integrated Memory Controller 1.0 LogiCORE IP Product Guide (PG313)”. In: 1.0.30 (May 2024), pp. 6–8, 10, 15, 18–20, 27–28, 32–33. URL: https://docs.amd.com/r/en-US/pg313-network-on-chip.

[6] AMD/Xilinx inc. “DPUCVDX8G for Versal ACAPs Product Guide (PG389)”. In: 1.3.23 (Jan. 2023), pp. 6–9, 26–29. URL: https://docs.amd.com/r/en-US/pg389-dpucvdx8g.

[7] AMD/Xilinx inc. “Performance AXI Traffic Generator LogiCORE IP Product Guide (PG381)”. In: 1.0.190 (Oct. 2023). URL: https://docs.amd.com/r/en-US/pg381-perf-axi-tg-spec.

[8] AMD/Xilinx inc. “Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)”. In: 2.0.14 (Nov. 2023), pp. 1–8. URL: https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration.

[9] AMD/Xilinx inc. “Vitis AI User Guide (UG1414)”. In: 3.0.24 (Feb. 2023). URL: https://docs.amd.com/r/3.0-English/ug1414-vitis-ai.

[10] PyTorch Foundation. “PyTorch”. In: (2024). URL: https://pytorch.org/docs/stable/index.html.

[11] Google inc. “TensorFlow”. In: (2024). URL: https://www.tensorflow.org/about.

[12] Yann LeCun Leon Bottou Yoshua Bengio and Patrick Haner. “Gradient-Based Learning Applied to Document Recognition”. In: 46 (Nov. 1998). URL: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: 60.6 (May 2017), pp. 84–90. DOI: 10.1145/3065386.

[14] Andrew Zisserman Karen Simonyan. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (Sept. 2014). URL: https://arxiv.org/abs/1409.1556.

[15] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. "Densely Connected Convolutional Networks". In: *CoRR* abs/1608.06993 (2016). arXiv: 1608.06993. URL: http://arxiv.org/abs/1608.06993.

[16] Xiangyu Kaiming. "Deep Residual Learning for Image Recognition". In: (Dec. 2015). DOI: 10.48550/arXiv.1512.03385.

[17] Menglong Zhu Andrew G. Howard. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 2017). DOI: 10.48550/arXiv.1704.04861.

[18] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *CoRR* abs/1905.11946 (2019). arXiv: 1905.11946. URL: http://arxiv.org/abs/1905.11946.

[19] NVIDIA inc. "Achieving FP32 Accuracy for INT8 Inference Using Quantization Aware Training with NVIDIA TensorRT". In: (2021). URL: https://developer.nvidia.com/blog/achieving-fp32-accuracy-for-int8-inference-using-quantization-aware-training-with-tensorrt/.

[20] Intel inc. "INT8 Quantization for x86 CPU in PyTorch". In: (Aug. 2023). URL: https://pytorch.org/blog/int8-quantization/.

[21] Sumin Kim, Gunju Park, and Youngmin Yi. "Performance Evaluation of INT8 Quantized Inference on Mobile GPUs". In: 9 (2021), pp. 164245–164255. DOI: 10.1109/ACCESS.2021.3133100.

[22] Kang Zhao. "Distribution Adaptive INT8 Quantization for Training CNNs". In: (May 2021). DOI: 10.1609/aaai.v35i4.16462.

[23] Kai-Chiang Wu Endri Taka Aman Arora. "MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine". In: (2023). DOI: 10.48550/arXiv.2311.04980.

[24] Z. Tao - Y. Wang - H. Zhang. "I2CU: A Dedicated Im2col Hardware Unit". In: 1.0.16 (Dec. 2022), pp. 1–8. DOI: 10.1109/ICCWAMTIP56608.2022.10016515.

[25] V. STRASSEN. "Gaussian Elimination is not Optimal." In: *Numerische Mathematik* 13 (1969), pp. 354–356. URL: http://eudml.org/doc/131927.

[26] Fawzi - Balog - Huang. "Discovering faster matrix multiplication algorithms with reinforcement learning". In: 1.0 (2022), pp. 47–53. DOI: 10.1038/s41586-022-05172-4.

[27] Ralf R. Müller, Bernhard Gäde, and Ali Bereyhi. "Efficient Matrix Multiplication: The Sparse Power-of-2 Factorization". In: *CoRR* abs/2002.04002 (2020). arXiv: 2002.04002. URL: https://arxiv.org/abs/2002.04002.

[28] R. Duan, H. Wu, and R. Zhou. "Faster Matrix Multiplication via Asymmetric Hashing". In: Nov. 2023, pp. 2129–2138. DOI: 10.1109/FOCS57990.2023.00130.

[29] Senhao Shao et al. "Towards Optimal Fast Matrix Multiplication on CPU-GPU Platforms". In: Jan. 2022, pp. 223–236. DOI: 10.1007/978-3-030-96772-7_21.

[30] A. Kritikakou Vasilios Kelefouras Iosif Mporas Vasilios Kolonias. "A high performance matrix matrix multiplication methodology for CPU and GPU architectures". In: Jan. 2016, pp. 804–844. DOI: 10.1007/s11227-015-1613-7-Optimal-Fast-Matrix-Multiplication-on-CPU-GPU-Platforms.

[31] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication". In: Aug. 2004, pp. 133–137. DOI: 10.1145/1058129.1058148.

[32] Jie Lei, José Flich, and Enrique S. Quintana-Ortí. "Toward Matrix Multiplication for Deep Learning Inference on the Xilinx Versal". In: 2023, pp. 227–234. DOI: 10.1109/PDP59025.2023.00043.

[33] Jinming Zhuang et al. "CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture". In: (2023). DOI: 10.1145/3543622.3573210.

[34] AMD/Xilinx inc. "AI Engine API User Guide". In: 1.0 (Feb. 2021), p. 1. URL: https://www.xilinx.com/htmldocs/xilinx2021_2/aiengine_api/aie_api/doc/modules.html.

[35] "GPU GEMM architecture". In: (). URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[36] Oleh Misko. "Detecting Melanoma Cancer using Convolutional Neural Networks". In: (Jan. 2019). DOI: 10.13140/RG.2.2.22090.41927.

[37] "In-datacenter performance analysis of a tensor processing unit". In: (2017). URL: https://doi.org/10.48550/arXiv.1704.04760.

[38] W.J. Dally and B. Towles. "Route packets, not wires: on-chip interconnection networks". In: 2001, pp. 684–689. URL: https://ieeexplore.ieee.org/document/935594.

[39] Andrei Bartic et al. "Network-on-Chip for Reconfigurable Systems: From High-Level Design Down to Implementation". In: vol. 3203. Aug. 2004, pp. 637–647. DOI: 10.1007/978-3-540-30117-2_65.

[40] Rosemary Francis and Simon Moore. "Exploring hard and soft networks-on-chip for FPGAs". In: (2008), pp. 261–264. DOI: 10.1109/FPT.2008.4762393.

[41] Carys Hilton and B. Nelson. "PNoC: a flexible circuit-switched NoC for FPGA-based systems". In: 153 (June 2006), pp. 181–188. DOI: 10.1049/ip-cdt:20050175.

[42] R. Mullins, A. West, and S. Moore. "The design and implementation of a low-latency on-chip network". In: (2006), pp. 6–7. DOI: 10.1109/ASPDAC.2006.1594676.

[43] Mohamed S. Abdelfattah and Vaughn Betz. "AUGMENTING FPGAS WITH EMBEDDED NETWORKS-ON-CHIP". In: (2013). URL: https://api.semanticscholar.org/CorpusID:14599805.

[44] Hamed S. Kia and Cristinel Ababei. "A new fault-tolerant and congestion-aware adaptive routing algorithm for regular Networks-on-Chip". In: 2011, pp. 2465–2472. DOI: 10.1109/CEC.2011.5949923.

[45] Junshi Wang et al. "Design of Fault-Tolerant and Reliable Networks-on-Chip". In: 2015, pp. 545–550. DOI: 10.1109/ISVLSI.2015.33.

[46] Ian Swarbrick et al. "Network-on-Chip Programmable Platform in Versal ACAP Architecture". In: (Feb. 2019), pp. 212–221. DOI: 10.1145/3289602.3293908.

[47] Ian Lang, Nachiket Kapre, and Rodolfo Pellizzoni. "Worst-Case Latency Analysis for the Versal NoC Network Packet Switch". In: (2021), pp. 55–60.

[48] Max Wierse. "Evaluation of Xilinx Versal Device". In: (2023), p. 63. DOI: 10.3929/ethz-b-000635928.

[49] P. Chen et al. "Exploiting On-Chip Heterogeneity of Versal Architecture for GNN Inference Acceleration". In: Sept. 2023, pp. 219–227. DOI: 10.1109/FPL60245.2023.00038.

[50] Intel inc. "Intel Stratix 10". In: (). URL: https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html.

[51] Intel inc. "Intel Agilex". In: (). URL: https://www.intel.com/content/www/us/en/products/details/fpga/agilex.html.

[52] Nvidia inc. "NVIDIA-Jetson-AGX". In: (). URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.