CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Full System-Level Simulation of Neural Compute Architectures

Master's thesis in Computer science and engineering

Arjun Kalamkar

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025

Master's thesis 2025

Full System-Level Simulation of Neural Compute Architectures

Arjun Kalamkar

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2025

Full System-Level Simulation of Neural Compute Architectures
Arjun Kalamkar

© Arjun Kalamkar, 2025.

Supervisor: Per Stenström, Department of Computer Science and Engineering
Advisor: Joshua Klein, IMEC
Examiner: Per Stenström, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Simplified illustration of the NPU model's architecture

Typeset in LaTeX
Gothenburg, Sweden 2025

Full System-Level Simulation of Neural Compute Architectures
Arjun Kalamkar
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The proliferation of large-scale artificial intelligence models necessitates specialized hardware like Neural Processing Units (NPUs) to achieve efficient computation. However, an NPU's real-world performance is deeply influenced by system-level effects that are often overlooked. Existing simulation tools typically lack the ability to model a detailed NPU microarchitecture within a full-system context, obscuring critical performance bottlenecks arising from the operating system, device drivers, and memory contention. This thesis introduces `gem5-fsnpu`, a novel simulation framework that bridges this gap by integrating a reconfigurable, transaction-level cycle-accurate NPU model into the gem5 full-system simulator [1], [2]. The framework includes a complete, vertically-integrated software stack, featuring a custom Linux driver and a user-space library with an intelligent, hardware-aware tiling algorithm, enabling realistic hardware-software co-design studies.

We demonstrate the framework's capabilities through a comprehensive Design Space Exploration, evaluating NPU performance on benchmarks including general matrix multiplication (GEMM) and complex Transformer layers like Multi-Head Attention (MHA). Architectural parameters such as systolic array dimensions (2D vs. 3D), on-chip memory size, and dataflow are systematically varied. The results reveal that system-level overheads are frequently the dominant performance bottleneck. For instance, the framework shows how for command-intensive workloads like MHA, the software control path latency can eclipse the hardware computation time, becoming the primary performance limiter. The study also quantifies the critical relationship between on-chip memory size and software tiling efficiency, demonstrating that an undersized memory can nullify the benefits of a powerful compute core. This work validates the necessity of full-system simulation for accelerator design and provides a powerful tool for researchers, proving that a holistic, hardware-software co-design approach is paramount to achieving efficient AI acceleration.

Keywords: AI, Heterogeneous computing, Neural Processing Unit, Full-system simulation, gem5

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

2 Background **5**

2.1 NPU architectures . . . 5

2.1.1 Dataflow and On-Chip Memory Management . . . 6

2.1.2 Architectural Specialisation and Scalability . . . 7

2.1.3 System-Level Integration . . . 8

2.2 NPU simulators . . . 8

2.2.1 Stand-alone, Component-Level Simulators . . . 9

2.2.2 Full-System Simulators with Abstract Models . . . 10

2.2.3 Mapping and Dataflow Optimisers . . . 10

3 Gem5-fsnpu: Full-system NPU model **13**

3.1 NPU Hardware Architecture . . . 13

3.2 Software Stack . . . 15

3.2.1 Linux Device Driver . . . 15

3.2.2 C Library . . . 16

3.3 Tiling Algorithm . . . 16

3.3.1 Hardware-Level Tiling . . . 16

3.3.2 Software-Level Chunking . . . 18

4 Methodology **21**

4.1 Experimental Setup . . . 21

4.1.1 Baseline Simulated Hardware Platform . . . 21

4.1.2 NPU Design Space Parameters . . . 22

4.1.3 Software Benchmarks . . . 23

4.1.4 Performance Metrics and Analysis Framework . . . 23

4.2 Design Space Exploration Studies . . . 24

4.2.1 DMA Engine Characterisation . . . 24

4.2.2 Core GEMM Performance Analysis . . . 25

4.2.3 System-Level Analysis of a Multi-Layer Perceptron . . . 26

4.2.4 System-Level Analysis of a Multi-Head Attention Block . . . 26

5 Results **29**
- 5.1 DMA Engine Characterisation . . . 29
- 5.2 Core GEMM Performance Analysis . . . 31
  - 5.2.1 Impact of Systolic Array Dimension . . . 31
  - 5.2.2 Sensitivity to SPM Size . . . 34
  - 5.2.3 Comparing Dataflows and 2D vs. 3D Array Organisation . . . 35
    - 5.2.3.1 Scaling Compute . . . 35
    - 5.2.3.2 Iso-Compute Units . . . 37
  - 5.2.4 Robustness to Matrix Aspect Ratio . . . 38
- 5.3 System-Level Analysis of a Multi-Layer Perceptron . . . 40
  - 5.3.1 Workload Scaling . . . 40
  - 5.3.2 Hardware Scaling . . . 42
    - 5.3.2.1 Impact of Systolic Array Dimension . . . 42
    - 5.3.2.2 Sensitivity to SPM size . . . 43
    - 5.3.2.3 Dataflows and compute scaling . . . 45
    - 5.3.2.4 2D vs. 3D Array Organisation . . . 46
- 5.4 System-Level Analysis of a Multi-Head Attention Block . . . 48
  - 5.4.1 Workload Scaling . . . 48
  - 5.4.2 Hardware Scaling . . . 49
    - 5.4.2.1 Impact of Systolic Array Dimension . . . 50
    - 5.4.2.2 Sensitivity to SPM size . . . 51
    - 5.4.2.3 Dataflows and compute scaling . . . 52
  - 5.4.3 Heads vs. Head dimension trade-off . . . 54

6 Limitations and Future Work **57**
- 6.1 Model Validation and Reproducibility . . . 57
- 6.2 Architectural Abstractions and Performance Implications . . . 58
- 6.3 Interpretation of High Software Overhead . . . 58
- 6.4 Future Work . . . 59
  - 6.4.1 DMA optimisation . . . 59
  - 6.4.2 Enhanced NPU Memory Hierarchy . . . 59
  - 6.4.3 NPU core improvements . . . 59
  - 6.4.4 Multi-Core NPU and Multi-Tenancy Exploration . . . 60
  - 6.4.5 Hardware Validation, Power, and Area Modelling . . . 60
  - 6.4.6 ONNX Runtime Integration . . . 60

7 Conclusion **63**
- 7.1 Discussion . . . 63
- 7.2 Conclusion . . . 63

Bibliography **65**

List of Figures

3.1 NPU model in gem5 with `NPUdma` (blue) and `SysArray` (orange) Objects 14
3.2 Vertically-integrated software stack 15
3.3 Two-tiered tiling algorithm 17

4.1 Configuration parameters for NPU model 22
4.2 Timeline of simplified `npu_matmul` function (upper bar) and breakdown for latency components (lower bar) 24
4.3 Multi-layer Perceptron 26
4.4 Multi-headed Attention layer 27

5.1 Latency breakdown by component for Memory transfer 30
5.2 Scaling of SW overheads and Memory transfer time with data size . . 30
5.3 Comparison of SW overhead scaling for `dma_mvin` and `dma_mvout` . . 31
5.4 Relative latency breakdown for GEMM ($256 \times 256 \times 256$) across varying systolic array dimensions and scaling strategies 32
5.5 Scaling of performance metrics for GEMM ($256 \times 256 \times 256$) across varying systolic array dimensions and scaling strategies 33
5.6 Relative latency breakdown for GEMM ($256 \times 256 \times 256$) across varying SPM sizes 34
5.7 Scaling of performance metrics as a function of SPM size. 34
5.8 Relative latency breakdown for GEMM ($256 \times 256 \times 256$) across varying compute tile scaling factors 36
5.9 Scaling of performance metrics as a function of compute tile scaling factor 37
5.10 Absolute latency breakdown for NPU configurations with an equal number of PEs 37
5.11 Scaling of performance metrics for NPU configurations with an equal number of PEs 38
5.12 Absolute latency breakdown across different matrix aspect ratios . . . 39
5.13 Scaling of Memory Ops and Overall HW Time across different matrix aspect ratios 39
5.14 Relative latency breakdown for the MLP layer across varying workload sizes 40
5.15 Scaling of performance metrics for MLP across varying workload sizes 41
5.16 Relative latency breakdown for MLP (Batch 64, `d_model` 512) across varying systolic array dimensions 42

5.17 Scaling of performance metrics for MLP (Batch 64, d_model 512) across varying systolic array dimensions 43
5.18 Relative latency breakdown for MLP (Batch 64, d_model 512) across varying SPM sizes 44
5.19 Scaling of performance metrics for MLP (Batch 64, d_model 512) across varying SPM sizes 44
5.20 Relative latency breakdown for MLP (Batch 64, d_model 512) across varying compute tile scaling factors 45
5.21 Scaling of performance metrics for MLP (Batch 64, d_model 512) across varying compute tile scaling factors 46
5.22 Relative latency breakdown for MLP (Batch 64, d_model 512) for various systolic array organisations 47
5.23 Scaling of performance metrics for MLP (Batch 64, d_model 512) for various systolic array organisations 47
5.24 Relative latency breakdown for the MHA layer across varying workload sizes 48
5.25 Scaling of performance metrics for MHA across varying workload sizes 49
5.26 Relative latency breakdown for MHA (seq_len 128, d_model 512, num_heads 8) across varying systolic array dimensions 50
5.27 Scaling of performance metrics for MHA (seq_len 128, d_model 512, num_heads 8) across varying systolic array dimensions 51
5.28 Relative latency breakdown for MHA (seq_len 128, d_model 512, num_heads 8) across varying SPM sizes 52
5.29 Scaling of performance metrics for MHA (seq_len 128, d_model 512, num_heads 8) across varying SPM sizes 52
5.30 Relative latency breakdown for MHA (seq_len 128, d_model 512, num_heads 8) across varying compute tile scaling factors 53
5.31 Scaling of performance metrics for MHA (seq_len 128, d_model 512, num_heads 8) across varying compute tile scaling factors 53
5.32 Relative latency breakdown for MHA (seq_len 128, d_model 512) for varying number of heads 54
5.33 Scaling of performance metrics for MHA (seq_len 128, d_model 512)for varying number of heads 55

List of Tables

2.1 Features of existing NPU architecture simulators 9

1

Introduction

Computing has taken a strong hold in modern society, and the diverse set of applications has driven the need for heterogeneous computing systems that cater to a wide range of applications. In recent years, this landscape has been reshaped by the exponential growth of artificial intelligence (AI). This paradigm shift is most apparent with the advent of Large Language Models (LLMs), whose widespread adoption can be observed through the rise of services like OpenAI's ChatGPT, which amassed over 100 million weekly users within a year of its launch, signalling a new era of mainstream AI interaction [3].

Currently, the training and inference of these massive LLMs are predominantly handled by large datacenters, which employ a large number of Central Processing Units (CPUs) and Graphic Processing Units (GPUs) for computation. From a mathematical perspective, LLMs are heavily reliant on matrix-matrix and matrix-vector operations. While CPUs excel at sequential processing and GPUs offer immense parallelism, neither is fundamentally optimised for the core computations of AI workloads. The general-purpose architectures of CPUs and GPUs often lead to inefficient data movement and repeated memory accesses when executing matrix operations. This inefficiency creates a significant performance and energy bottleneck, necessitating specialised hardware solutions.

To address these challenges, many domain-specific accelerators have emerged, with Neural Processing Units (NPUs) at the forefront. Most NPUs are built around systolic arrays, an architecture that excels at performing matrix multiplication with high data reuse and minimal data movement [4]. The use of systolic arrays for matrix operations can yield significant performance and power efficiency gains over traditional CPUs and GPUs [5]. Consequently, NPUs are no longer a niche technology; they are now integral components in a wide range of computing platforms, from mobile Systems-on-Chips (SoCs) in smartphones to powerful accelerators in hyperscale datacenters [5]–[8].

However, NPU architectures are not standardised across the industry. The design space is vast, with significant variations in systolic array organisation, memory hierarchy, and dataflow strategies. Additionally, when integrated into complex heterogeneous SoCs, the performance of an NPU is subject to numerous system-level effects. Factors such as operating system (OS) scheduling, memory bandwidth contention

with other processing units, and the overhead of CPU orchestration can introduce significant inefficiencies that are not apparent when analysing the accelerator in isolation. To fully examine these effects, we need sophisticated simulation tools that can capture these effects with high fidelity.

Existing tools suitable for this purpose can be broadly categorised into three groups:

1. **Stand-alone, Component-Level Simulators:** These tools provide detailed, cycle-accurate models of the NPU architecture but lack a broader system context. They are excellent for design space exploration of the NPU's microarchitecture, but cannot model its interaction with an OS or other system components. Prominent examples include SCALE-Sim [9][10][11] and STONNE [12].

2. **Full-System Simulators with Abstract Models:** These simulators, often built on frameworks like gem5 [1], [2], can model an entire computer system, including the OS. However, when incorporating accelerators, the accelerator models are typically abstract and non-cycle-accurate, serving as simple timing or functional models rather than detailed microarchitectural representations to keep overall simulation times manageable.[13], [14].

3. **Mapping and Dataflow Optimisers :** These simulators focus on the compiler problem of efficiently mapping a workload onto a given hardware architecture. Tools like ONNXim [15] and Timeloop [16] use analytical models to rapidly explore the dataflow and scheduling design space, but they abstract away the cycle-level details of the hardware and the dynamics of the surrounding system under the assumption that the NPU design is deterministic.

From this analysis, it is evident that a framework enabling full-system simulation with a cycle-accurate, microarchitecturally detailed NPU model does not exist. The primary objective of this thesis is to address this gap.

We leverage the premier computer architecture simulator, gem5 [1][2], as our foundation, capitalising on its robust support for full-system simulation and detailed memory system modelling. Within this framework, we have designed and implemented a highly parametrizable NPU model. This model features a transaction-level cycle-accurate systolic array execution core and a modular design inspired by architectures like Google's Tensor Processing Unit (TPU)[5][8], emphasising explicit DMA-based data orchestration between system memory and a private scratchpad. The NPU is integrated as a Direct Memory Access (DMA)-capable device connected to the host CPU via a PCI (Peripheral Connect Interface) communication link. To facilitate interaction with our simulated hardware, we have developed a vertically-integrated software stack, comprising a custom Linux kernel driver and a user-space library. This stack simplifies the process of writing applications for the NPU by providing a clean API.

To demonstrate the unique research capabilities enabled by this work, we conduct a case study analysing the performance of AI-focused workloads on various NPU configurations. We perform design space exploration (DSE), sweeping hardware

parameters such as systolic array dimensions, dataflows, and on-chip memory sizes. This study provides critical insights into the system-level bottlenecks that emerge from the interaction between the software stack, the host CPU, and the NPU accelerator. Our results highlight the framework's ability to reveal how, for complex and command-intensive workloads like Multi-Headed Attention, the software control path can become the dominant performance bottleneck, showing that hardware-level optimisations alone can be rendered ineffective. We also quantify key design trade-offs, showing, for instance, that while increasing on-chip memory from 32 KiB to 128 KiB halved the execution time for a 256×256 matrix multiplication workload, any further increase in the memory size yielded negligible returns. Furthermore, the analysis uncovers non-obvious outcomes, such as how a simpler 2D systolic array can outperform a theoretically faster 3D array at the system level due to more efficient software and memory access patterns. This analysis proves that a holistic, full-system analysis, as enabled by this work, is essential to uncovering the true performance characteristics of modern NPU design.

The contributions of this work can be summarised as:

1. The design and implementation of a reconfigurable, transaction-level cycle-accurate NPU model within the gem5 full-system simulation framework, featuring a systolic array compute core and DMA-based data movement.
2. The development of a vertically integrated software stack, comprising a custom Linux kernel driver and a user-space library with a novel, two-tiered tiling algorithm that enables matrix computations to execute efficiently on the simulated NPU.
3. A comprehensive case study demonstrating the framework's capabilities by quantifying the impact of system-level effects of key NPU architectural parameters, including systolic array dimensionality (2D vs. 3D) and on-chip memory size.

The remainder of this thesis is organised as follows. Chapter 2 provides the necessary background, reviewing the architectural landscape of contemporary NPUs and surveying the existing simulation tools to contextualise the contributions of this work. Chapter 3 details the design of the `gem5-fsnpu` framework, covering the NPU hardware model, the vertically-integrated software stack, and the two-tiered tiling algorithm. Chapter 4 outlines the experimental methodology, describing the simulated platform, the software benchmarks, the performance metrics, and the specific Design Space Exploration studies conducted. Chapter 5 presents and analyses the results of these studies, deconstructing system-level performance across a range of workloads and hardware configurations. Chapter 6 discusses the limitations of the framework at present, reinterprets some of the results with those limitations in mind and presents promising avenues for future improvements. Finally, Chapter 7 concludes the thesis by discussing the key findings and summarising the primary contributions of this work.

2

Background

This chapter lays the foundational context for the work presented in this thesis. As discussed in the Introduction, NPU architectures are not standardised, and it is necessary to contextualise the vast design space. We begin in Section 2.1 by exploring the design space of NPU architectures, with a focus on the systolic array paradigm that forms the basis of our model. We then present a comprehensive survey of existing NPU simulators in Section 2.2, categorising them to identify the specific gap in full-system, cycle-accurate simulation that this thesis aims to address.

2.1 NPU architectures

The computational core of most modern artificial intelligence (AI) workloads, from convolutional neural networks (CNNs) to transformers, is heavily reliant on matrix-matrix multiplication (GEMM) operations. While extensive research has been done to optimise these operations for general-purpose CPUs through sophisticated software libraries and techniques like tiling and data packing [17][18], the inherent inefficiencies of general-purpose architectures for such tasks created a performance and energy bottleneck. This led to the development of domain-specific accelerators, such as Neural Processing Units (NPUs), designed specifically for the data patterns of matrix operations used in AI workloads.

The dominant architectural paradigm for modern NPUs is the Systolic Array, a grid of interconnected processing elements (PEs) that enables high data reuse and minimises data movement for matrix operations. CPUs can be likened to a setup with a memory feeding a singular PE, which in turn feeds back to the memory [4]. Every single operation involves a memory access for the data, followed by a computation in the PE, and then a memory access to write back the data. For a complex computation like matrix multiplication that requires numerous such operations, this process becomes slow and expensive, especially since memory accesses are the bottleneck in modern systems. To alleviate this concern, one idea that can be employed is an architecture where multiple PEs are connected in sequence to reuse the already accessed data, thereby reducing the number of memory accesses incurred. This concept, taken a bit further to use a grid of PEs connected to a local memory, is essentially a systolic array architecture. The systolic array can perform a matrix multiplication operation with just three memory accesses, instead of the numerous and repeated accesses required for CPUs.

When taking a look at contemporary NPU designs, a well-known example that popularised the systolic array approach in industry is Google's Tensor Processing Unit (TPU), initially introduced in 2017[5]. The TPU featured a large 256×256 systolic array coupled with a substantial 24 MiB on-chip scratchpad memory, termed the Unified Buffer. The scratchpad memory differs from a CPU cache in that it is an explicitly managed cache, implying that it does not take part in coherence and that the scratchpad is managed by software. It operated as a co-processor attached via a PCIe bus, executing commands from a host CPU. This fundamental design, consisting of a systolic compute core, a large private on-chip memory, and a host-managed execution model, has become a template for many subsequent NPU designs and serves as the primary inspiration for the model developed in this work. While this well-known template exists, the design space of NPUs is vast. The variety in this space can be characterised along several key dimensions that differentiate state-of-the-art architectures.

2.1.1 Dataflow and On-Chip Memory Management

A critical dimension in NPU design is the dataflow, which dictates the strategy for moving input, weight, and output data between the on-chip memory and the PEs of the systolic array. The choice of dataflow has a profound impact on performance and power, with different approaches being suited for different workloads.

- Weight Stationary (WS): In workloads that involve repeated computations with a selected set of matrices (termed as weights in the AI domain), the weights should be reused by storing them locally. In a systolic array, the PEs in an array with WS dataflow have registers to store a weight value that has to be preloaded. With the weights preloaded into the array, the input matrix is streamed in, row by row, and the corresponding values of the output of the multiplication are similarly streamed out, row by row. The output values are then stored in a separate memory called the accumulator, which has adders to accumulate the results across multiple matrix multiplications. Notably, the original Google TPU [5] employed a WS dataflow. This strategy is highly effective for reusing model weights across a batch of inputs. Additionally, the size of the accumulator acts as a limiting factor on the amount of data that can be processed in a single compute instruction, and an increase in accumulator size results in an increase in the size of a single computation.
- Output Stationary (OS): When splitting large matrix multiplications into smaller matrices (tiles) that fit the array, multiple partial sum tiles need to be accumulated to get the final output tiles. For this purpose, the OS dataflow uses the registers in the PEs to store partial sums, using the values as a bias instead of a multiplier, unlike WS. This way, while both the input and weight matrix tiles get streamed in together, the results are accumulated with the partial tile sum already stored in the array till the results become the complete output tile and can be unloaded. This dataflow simplifies the NPU architecture and management strategy at the cost of increasing the complexity of PEs.
  Computations in the OS dataflow are typically limited to the dimensions of

the systolic array. However, a novel architecture of systolic arrays organised as 3D arrays, instead of the typical 2D, is used to effectively perform multiple matrix multiplications in parallel, thereby increasing the amount of data that can be processed in a single compute instruction. Joseph et. al. [19] introduce a 3D systolic array organisation that uses the OS dataflow for computation and accumulates the results of the *layers* (z dimension) together in the bottom layer. From a computational perspective, this means that the amount of data getting processed scales with the number of layers, but the size of the output remains the same.

- Row Stationary (RS): The Eyeriss accelerator introduced the novel RS dataflow, designed to minimise the energy consumption of data movement by maximising the local reuse of all data types within the array. This work highlighted that data movement, not computation, is often the primary source of energy consumption in deep learning accelerators [20].
- Input Stationary (IS): Very similar in operation to the WS dataflow, with the difference that instead of keeping the weights stationary, this dataflow keeps the inputs stationary.

The complexity and importance of dataflow also led to the development of specialised analytical tools like MAESTRO [21] [22], Timeloop [16] and ZigZag [23] to explore the trade-offs between different dataflow strategies for a given neural network layer and hardware configuration.

2.1.2 Architectural Specialisation and Scalability

Beyond the core dataflow, NPU architectures are often specialised with features to handle the specific properties of AI workloads and to scale to meet growing computational demands.

- Sparsity: Neural network weights and activations often contain a large number of zero values. To exploit this, many modern NPUs incorporate hardware support for sparsity, allowing the accelerator to skip zero-value computations, thereby saving energy and improving throughput. The NPU in Samsung's flagship mobile SoCs is a prominent commercial example featuring sparsity-aware hardware [6].
- Scalability: As models grow larger, single-chip performance is insufficient. A key design trend is creating scalable systems by connecting multiple NPU chips. Generations of Google's TPU show a progression towards multi-chip systems that can be tightly interconnected to function as a single, massive accelerator [8]. This "scale-out" approach is an active area of research, exploring how to efficiently build larger logical systolic arrays from smaller physical modules [24].
- Alternative Paradigms: While digital systolic arrays are dominant, alternative architectures also exist. For example, analog in-memory computing approaches such as the ISAAC accelerator perform arithmetic directly within analog

memory arrays, promising significant reductions in data movement by co-locating memory and computation [25].

2.1.3 System-Level Integration

An NPU's performance is not only determined by its microarchitecture in isolation but is heavily affected by how it is integrated into the broader computer system. A common paradigm among NPUs is the use of DMA-enabled data movement between the main memory and the scratchpad or cache in the NPU.

- Memory System Integration: Efficiently feeding the NPU with data is a critical challenge. Many modern NPUs require architectural support for the system's virtual memory, including Translation Lookaside Buffers (TLBs) to cache address translations. The NeuMMU architecture is one example that provides such support to avoid the overheads of OS-managed memory for the accelerator [26].

- Resource Sharing: In datacenter environments, powerful NPUs are shared resources. This introduces the challenge of multi-tenancy, introducing the question of how the NPUs can be partitioned to serve multiple users while ensuring fairness and high utilisation. Architectures like V10 propose hardware-assisted mechanisms to manage this sharing efficiently [27].

Considering the different features, the key architectural dimensions that define the NPU design space are systolic array configuration, dataflow, specialisation for features like sparsity, and the method of system integration. The goal of the simulator developed in this thesis is to provide a flexible framework capable of exploring the performance implications of these trade-offs, particularly those arising from the interaction between the NPU and the full software and hardware system.

2.2 NPU simulators

As discussed in Section 2.1, NPU architectures have a varied design space with several dimensions that differentiate them. To evaluate these designs, sophisticated simulation tools are necessary, and several simulators have been proposed that target architectures termed as Systolic Array Accelerators, DNN Accelerators, CNN accelerators or NPUs. While highly accurate Register-Transfer Level (RTL) simulators like Gemmini [28] exist and provide deep microarchitectural insights, their simulation speed is prohibitively slow for exploring system-level interactions with a full operating system and software stack. Therefore, for this work, we focus on higher-level system simulators. As identified in the Section 1, these tools can be broadly categorised into three groups, whose features, from the perspective of holistic full-system NPU simulation, are summarised in Table 2.1.

Table 2.1: Features of existing NPU architecture simulators

Simulator	Cycle Accurate	Memory modelling	Full-System	System Software Fidelity	Mapping Exploration
SCALE-Sim [9] [10] [11]	O	O	X	X	X
STONNE [12]	O	O	X	X	X
SMAUG [29]	O	O	X	X	X
NNSim [30]	O	O	X	X	X
mNPUsim[31]	X	O	X	X	X
gem5-Aladdin [13]	X	O	O	O	X
gem5-accel [14]	X	O	O	O	X
Timeloop[16]	X	X	X	X	O
ONNXim [15]	X	X	X	X	O
ZigZag [23]	X	X	X	X	O
MAESTRO [21] [22]	X	X	X	X	O
gem5-fsnpu	O[1]	O	O	O	X[2]

[1] Transaction-level cycle-accurate model
[2] Implements an optimised, hardware-aware tiling algorithm but does not perform automated mapping space exploration

2.2.1 Stand-alone, Component-Level Simulators

These tools provide detailed, cycle-accurate models of the NPU microarchitecture but operate in isolation, without the context of a full system. They are excellent for deep dives into accelerator design. Still, they cannot model the performance effects arising from the interaction with the operating system, device drivers, or contention for shared resources like main memory. Prominent examples include SCALE-Sim [9]–[11] and STONNE [12].

SCALE-Sim [9]–[11] is a widely used, Python-based simulator that provides cycle-level performance estimates for systolic array-based accelerators. It focuses on modelling the impact of dataflow and on-chip memory hierarchy on performance and energy. While it accurately models the execution time of the systolic array itself, it does not simulate the host CPU, the OS, or the detailed process of data transfer over a system bus like PCIe. It also includes a thorough analytical description of Systolic Array operation, and we have drawn considerable inspiration from this for our own model.

Similarly, STONNE [12] is also a highly detailed, cycle-accurate simulator that can model a wide range of DNN accelerator concepts, including flexible systolic arrays and sparsity-aware hardware. Its strength lies in its detailed component-level analysis. It provides a cycle-accurate view of the accelerator's internal state but does not inherently model the full software stack overhead associated with managing the

device.

SMAUG [29] is another powerful framework in this category, offering a cycle-accurate simulation backend for a highly configurable neural network accelerator. A key feature of SMAUG is its co-design of the hardware model with a full software stack, including a compiler and runtime, which allows it to execute entire network graphs end-to-end. While comprehensive at the component level, it remains a stand-alone tool that does not capture the dynamics of a host CPU or operating system managing the accelerator.

Finally, NNSim [30] and mNPUsim [31] are simulators that, while operating at the component level, are designed with a stronger awareness of the memory system. They provide detailed models of the on-chip memory hierarchy and can be configured to model the latency and bandwidth constraints of an external DRAM system. This allows them to account for some system-level effects like memory contention, but they still abstract away the software stack, OS interactions, and CPU overhead, which are critical for a complete performance picture.

2.2.2 Full-System Simulators with Abstract Models

These tools, often built on frameworks like gem5 [1], [2], are capable of booting a full operating system and modelling an entire hardware platform. However, to manage simulation complexity and speed, the accelerator models integrated into these frameworks are often abstract timing or functional models rather than detailed microarchitectural representations.

gem5-Aladdin [13] integrates the Aladdin accelerator modelling infrastructure into gem5. It allows for the co-simulation of a host CPU with custom accelerators specified in a C-like description. While it successfully models system-level effects like memory contention, the accelerator model itself is an analytical, pre-RTL representation. It does not provide a reconfigurable, detailed microarchitectural model of a specific domain architecture like a systolic array NPU.

Other gem5-based accelerator frameworks, such as gem5-Accel [14], often treat the accelerator as a simple peripheral that consumes data and stalls for a pre-calculated number of cycles. This approach captures the accelerator's impact on system resources like memory bandwidth but abstracts away the internal pipeline, dataflow, and control overheads of the accelerator itself. This is precisely the trade-off that this thesis, titled `gem5-fsnpu`, aims to address by introducing a detailed model within the full-system context.

2.2.3 Mapping and Dataflow Optimisers

Distinct from simulators that model cycle-by-cycle hardware execution, this category of tools addresses the higher-level challenge of efficiently mapping a neural network workload onto a given hardware architecture. This challenge involves navigating a vast, combinatorial design space that is computationally infeasible to explore with slow, detailed simulators. This design space can be understood at two levels:

- Dataflow Strategy: The high-level paradigm for data movement and reuse, such as Weight Stationary (WS), Output Stationary (OS), or Row Stationary (RS).
- Mapping: The specific, low-level implementation details, including tiling factors, loop ordering, and memory allocation for a given strategy, hardware and workload.

Analytical tools were developed to rapidly explore this space by estimating the performance and energy of different approaches. However, they tackle the problem from different angles. MAESTRO [21], [22], for instance, excels at the high-level analysis of dataflow strategies. It provides a formal framework to describe and evaluate different strategies, allowing designers to compare the fundamental costs and benefits of these architectural approaches.

In contrast, tools like Timeloop [16] and ZigZag [23] focus on discovering the optimal low-level mapping. Given an abstract hardware description and a workload, they employ sophisticated search algorithms to explore the enormous space of tiling factors and loop orders. Their goal is to find the most performant or energy-efficient mapping for that specific hardware-software combination, which will inherently embody a dataflow strategy without needing it to be predefined. Operating at an even higher level, ONNXim [15] simulates entire neural networks from ONNX graphs to enable rapid, early-stage analysis of full models on configurable NPU architectures.

While indispensable for hardware-software co-design, the primary limitation of these frameworks is their abstraction of the system environment. They provide performance projections based on a deterministic and isolated view of the accelerator, without accounting for dynamic, system-level interactions such as contention for main memory bandwidth with a host CPU, the overhead of the operating system and device drivers, or the effects of virtual memory. Therefore, these tools serve a complementary role. They can identify promising dataflow strategies and mappings to be evaluated, but they cannot replace a detailed, full-system simulator for verifying performance and identifying bottlenecks in a realistic operating environment.

From this survey, it becomes clear that no existing framework provides a reconfigurable, microarchitecturally detailed NPU model integrated within a full-system simulator, while also including a complete, vertically-integrated software stack. Component-level simulators lack system context, while full-system simulators lack accelerator fidelity. The work in this thesis directly addresses this gap by combining the strengths of both approaches, enabling research into the complex interplay between NPU hardware, system software, and the host environment.

3

Gem5-fsnpu: Full-system NPU model

This chapter details the design and implementation of the `gem5-fsnpu` framework, which comprises a co-designed hardware model and software stack. Section 3.1 describes the NPU hardware architecture, detailing its implementation as two interconnected gem5 *SimObjects* and its transaction-level cycle-accurate execution model. Section 3.2 outlines the vertically-integrated software stack, including the custom Linux device driver and the user-space C library that provides the API for applications. Finally, Section 3.3 presents the novel, two-tiered tiling algorithm that is central to the framework's hardware-software co-design, enabling efficient execution of large matrix multiplications on the NPU.

3.1 NPU Hardware Architecture

The NPU device is modelled in a modular fashion using two distinct but interconnected gem5 *SimObjects*: an NPU Front-end (`NPUdma`) and a Systolic Array Compute Core (`SysArray`). The Front-end is responsible for system-level communication, including managing the PCI interface and orchestrating DMA transfers, while the Compute Core serves as the dedicated execution unit for arithmetic operations.

The template architecture simulated by these two objects is depicted in figure 3.1. The architecture features a single Systolic Array core coupled with on-chip scratchpad memory and an accumulator. A key feature of this design is its runtime reconfigurability. Parameters fundamental to the NPU's operation—such as the systolic array dimensions (`sys_array_size`, `sa_z_dim`), dataflow (`dataflow`), scratchpad size (`scratchpad_size`), and accumulator size (`accum_size`) can be modified dynamically during simulation. This is facilitated by a set of control registers within the `NPUdma` object.

These registers, defined in the `NpuCtrlRegs` structure, are mapped to a PCI Base Address Register (BAR 0). This design exposes the NPU to the simulated system as a standard PCI device, whose BAR is mapped into the system's physical address space, enabling memory-mapped I/O (MMIO). The guest operating system can therefore discover and interact with the NPU through a device driver, which writes commands (e.g., `NPU_START_MVIN_DMA`, `NPU_MATMUL`) and configuration data to these registers. The `NPUdma` object parses these commands and orchestrates the required

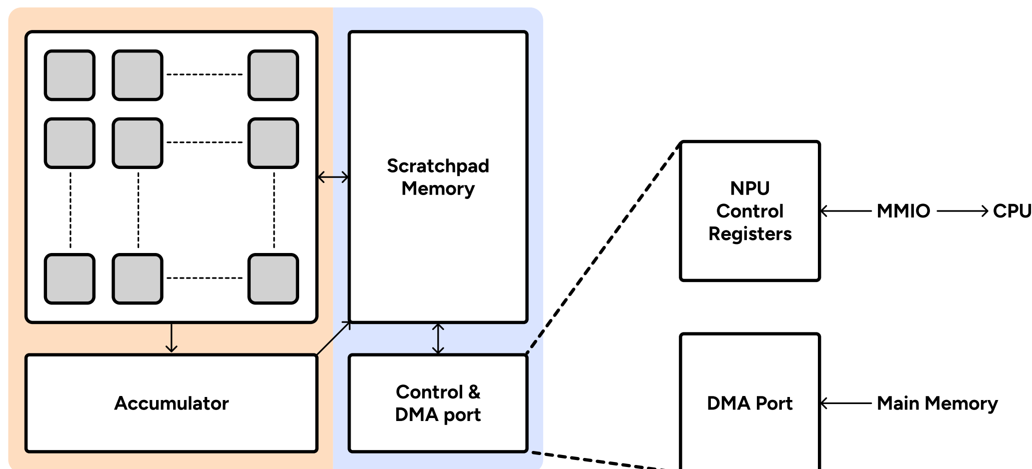


Figure 3.1: NPU model in gem5 with `NPUdma` (blue) and `SysArray` (orange) Objects

actions, either initiating a DMA transfer via the DMA engine or issuing an execution command to the `SysArray` object.

A critical design choice was the simulation fidelity of the Systolic Array. A purely cycle-accurate model, while precise, imposes a large simulation time overhead within a full-system context. Conversely, a purely analytical model fails to capture essential hardware performance characteristics. Instead, this framework adopts a balanced, *transaction-level cycle-accurate* approach. The latency of a matrix multiplication tile is modelled deterministically based on its dimensions and the systolic array's configuration. This is based on the insight that systolic array execution is highly regular, and stalls primarily arise from data starvation, a factor which the framework's tiling algorithm (Section 3.3) is designed to prevent. The compute latency is calculated in the function `SysArray::getCompLatency()` as:

$$\text{Latency (cycles)} = N_{tile} + M_{tile} + \frac{K_{tile}}{Z_{dim}} + (Z_{dim} - 1) - 2 \tag{3.1}$$

This function is a modified version of the equation presented in SCALE-Sim [9][10], with an additional inclusion of the effect of a systolic array organised in a 3D structure as described by Joseph et. al. [19]. This function accurately captures the execution time for a single compute transaction without the overhead of simulating every single clock cycle, striking a practical balance between accuracy and simulation speed. While the core computation latency is deterministic, the model still captures all system-level overheads, as the initiation of each operation is managed by the NPU's event-driven state machine, which is in turn controlled by the software stack. The interaction between the NPU and Systolic Array is event-driven, following a state machine based on commands like `NPU_MATMUL_PRELOAD`, `NPU_MATMUL_COMPUTE_PRELOADED`, and `NPU_ACCUM_UNLOAD`, ensuring that data movement and computation are correctly sequenced.

3.2 Software Stack

The software stack bridges the user-space applications and the NPU hardware. It is designed to be lightweight and efficient to ensure that performance bottlenecks are dominated by the hardware execution, not the software overhead. Additionally, the software stack is also designed to be easily extendable to allow for future development. The stack consists of a Linux device driver and a user-space C library.

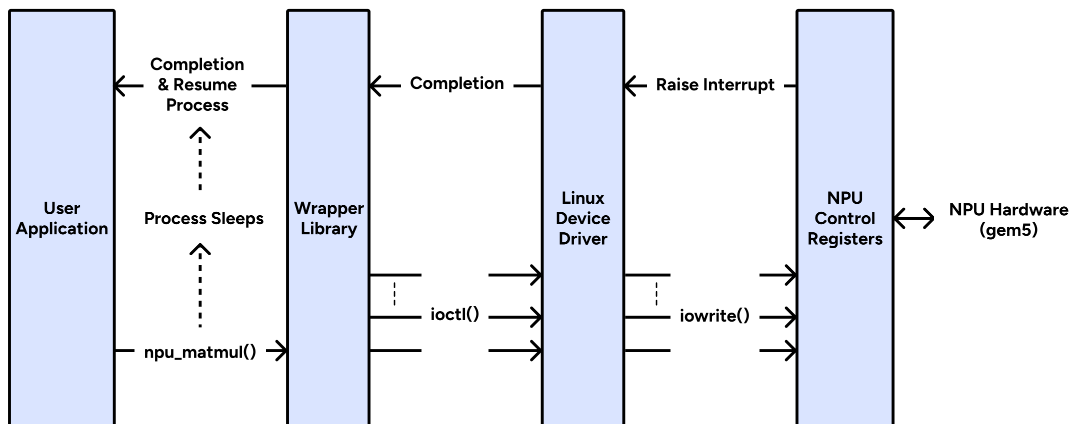


Figure 3.2: Vertically-integrated software stack

3.2.1 Linux Device Driver

The NPU is managed by a custom Linux PCI device driver. During driver initialisation, the driver's `pci_probe` function identifies the NPU using its vendor and device IDs. Upon successful probing, the driver performs several important setup tasks:

1. Enables the PCI device and requests its memory regions.
2. Maps the memory region for BAR0 into the kernel's virtual address space using `pci_iomap`, allowing for memory-mapped I/O (MMIO) to the NPU's control registers.
3. Allocates a coherent DMA buffer using `dma_alloc_coherent` to serve as a bounce buffer for transfers between user space and the NPU's DMA engine.
4. Initialises an interrupt handler (`npu_isr`) to receive completion notifications from the NPU.
5. Creates a character device (`/dev/npu`) which acts as the primary interface for user-space applications.

Communication from user space is handled via the `ioctl` system call. Rather than exposing raw register writes, the driver defines a set of high-level `ioctl` commands such as `NPU_IOCTL_DMA_XFER` and `NPU_IOCTL_EXE_CMD`. These commands accept C

structures (npu_dma_op, npu_exec_op) containing all necessary parameters for an operation. For instance, to execute a matrix multiplication chunk, the C library makes a single ioctl call with the npu_exec_op struct. The driver then performs the sequence of individual iowrite calls to the control registers and issues the final command. This approach is more robust and efficient than performing multiple read and write system calls from user space for a single hardware operation. Upon issuing a command, the driver puts the calling process to sleep using wait_event_interruptible, and the interrupt service routine wakes it upon completion.

3.2.2 C Library

The user-space C library provides a simplified, high-level API that abstracts the complexities of the driver and hardware. Its cornerstone is the npu_matmul function, which allows a programmer to perform large-scale matrix multiplication on any two arbitrarily sized matrices. This function contains a sophisticated, two-level tiling and data movement engine designed to maximise hardware utilisation. It orchestrates the entire workflow:

1. Opening the file descriptor /dev/npu with npu_open
2. Executing the matrix multiplication via npu_matmul, which internally handles all tiling, memory allocation, DMA transfers, and computation commands.
3. Releasing the device handle with npu_close

The intelligence of the library lies in its ability to automatically tile and schedule operations based on the NPU's hardware configuration, which it queries from the driver at initialisation. This tiling strategy is a central element of the framework's hardware-software co-design and is detailed in the following section.

3.3 Tiling Algorithm

To efficiently execute large matrix multiplications on a hardware accelerator with finite on-chip memory, a robust tiling strategy is paramount. This framework employs a two-tiered tiling algorithm that is a key aspect of its hardware-software co-design. The algorithm distinguishes between hardware-level tiles, which are sized to fit the Systolic Array's dimensions, and software-level chunks, which are sized to optimally utilise the available scratchpad memory.

3.3.1 Hardware-Level Tiling

The systolic array operates on fixed-size data tiles. The dimensions of these tiles (N_{tile}, M_{tile}, K_{tile}) are derived directly from the hardware parameters. For the Weight Stationary (WS) dataflow, the primary constraint is saturating the accumulator. M_{tile} and K_{tile} are constrained by the systolic array dimension, while N_{tile} is determined

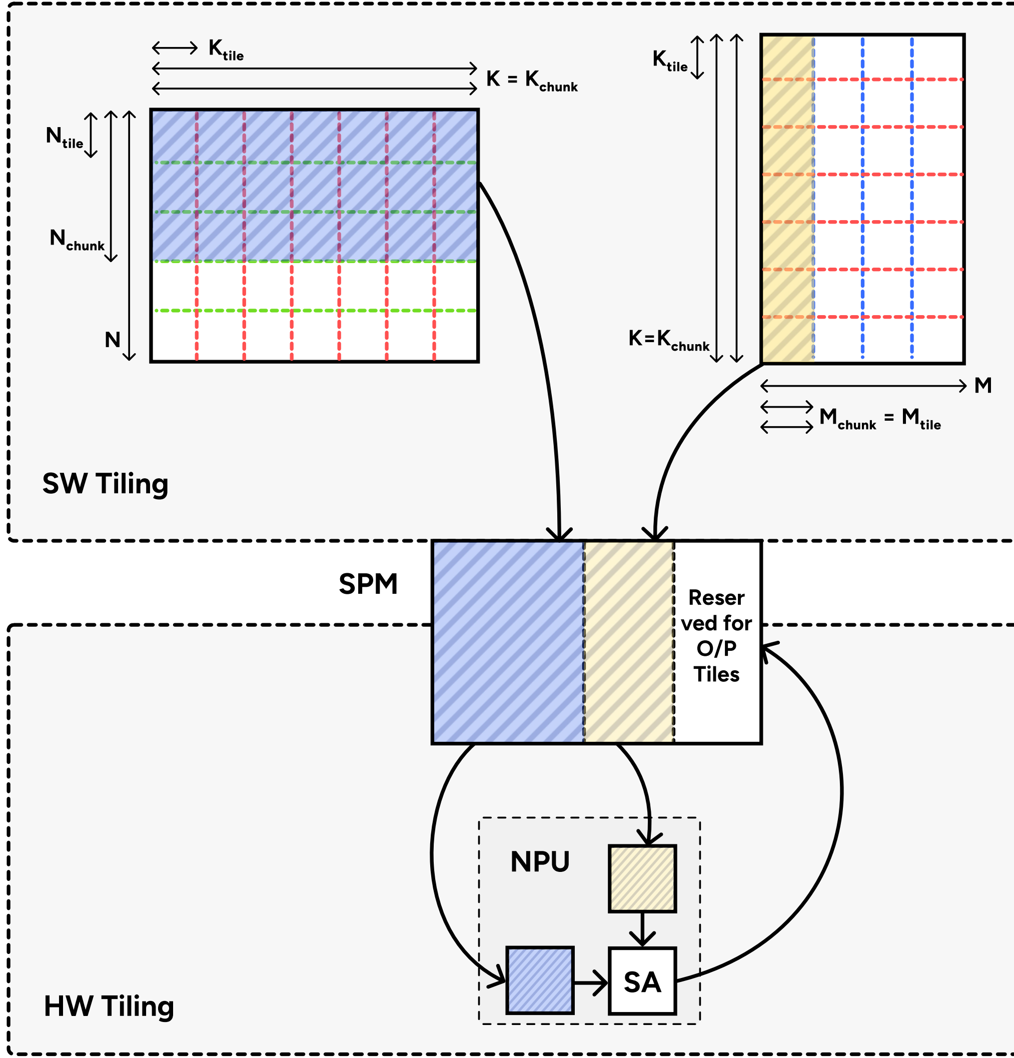


Figure 3.3: Two-tiered tiling algorithm

by the accumulator size.

$$N_{tile} = \min\left(N, \left\lfloor \frac{\text{Accumulator size}}{\text{Systolic Array dimension}} \right\rfloor\right) \tag{3.2}$$
$$M_{tile} = K_{tile} = \text{Systolic Array dimension}$$

For Output Stationary (OS) dataflow, the tile dimensions are constrained by the physical dimensions of the systolic array, with the additional consideration for the 3D systolic array, which results in a larger K_{tile} value

$$N_{tile} = M_{tile} = \text{Systolic Array dimension} \tag{3.3}$$
$$K_{tile} = \text{Systolic Array dimension} \times \text{Layers of Systolic Array (Z dimension)}$$

The Systolic Array is designed to process these specific tile sizes, which are passed by the NPU control logic during a compute command.

3.3.2 Software-Level Chunking

The software library function, `npu_matmul`, is responsible for partitioning the large input matrices into chunks that can fit into the NPU's scratchpad memory (SPM). The goal is to minimise DMA transfers between main memory and the scratchpad. The algorithm first calculates the space required for one output tile and one pair of input and weight tiles and then determines the chunking strategy based on scratchpad capacity.

$$\text{Tile Pairs} = \left\lfloor \frac{\text{Scratchpad size} - (N_{tile} \times M_{tile})}{(N_{tile} \times K_{tile}) + (K_{tile} \times M_{tile})} \right\rfloor \tag{3.4}$$

1. Memory-Constrained. If the scratchpad cannot hold the full set of tile-pairs needed to compute one output tile, i.e., Tile pairs $< \left\lceil \frac{K}{K_{tile}} \right\rceil$, the algorithm partitions the problem along the `K` dimension. It loads chunks of the matrices corresponding to a single `N` and `M` tile but a partial `K` dimension, and accumulates the results for the output tile in the NPU's accumulator over multiple compute calls. The chunk sizes for this case are:

$$\begin{aligned} N_{chunk} &= N_{tile} \\ M_{chunk} &= M_{tile} \\ K_{chunk} &= \text{Tile Pairs} \times K_{tile} \end{aligned} \tag{3.5}$$

2. Memory-Sufficient. If the scratchpad can hold a full tile-row or tile-column, the problem becomes one of optimisation, and we want to maximise the reuse of data stored in the scratchpad. The algorithm aims to fit p tile-rows of the activation matrix and q tile-columns of the weight matrix, along with space for the resulting $p \times q$ output tiles. While this can be formulated as a complex integer non-linear programming problem, we employ a greedy heuristic that provides a near-optimal solution with much lower computational overhead:

$$\text{Scratchpad size} \geq p \cdot (N_{tile} \times K) + q \cdot (K \times M_{tile}) + p \cdot q \cdot (N_{tile} \times M_{tile}) \tag{3.6}$$

While improving reuse, the difference in sizes of the two matrices, the inputs and weights, becomes a determining factor. The algorithm therefore prioritises keeping the larger of the two input matrices (Inputs or Weights) more static in the scratchpad, as it would result in smaller memory transfers. Thus, the values for p and q are determined accordingly. For example, if $N \geq M$, we first find the maximum possible value of p. Then the inequality (Equation 3.6) is rearranged in terms of q, and we iterate downward on the value of p from the maximum, until the resulting value for q is ≥ 1. This greedy approach finds a

near-optimal solution for p and q, from which the chunk sizes are derived.

$$\begin{gathered}
p_{max} = \left\lfloor \frac{\text{Scratchpad size}}{N_{tile} \times K} \right\rfloor \\
q_p = \left\lfloor \frac{\text{Scratchpad size} - p \cdot (N_{tile} \times K)}{(K \times M_{tile}) + p \cdot (N_{tile} \times M_{tile})} \right\rfloor \\
N_{chunk} = p \cdot N_{tile} \\
M_{chunk} = q \cdot M_{tile} \\
K_{chunk} = K
\end{gathered} \tag{3.7}$$

On the other hand, if $N < M$, we start with a maximal q value and find a solution where $p \geq 1$.

This two-level strategy ensures that the hardware is always fed with optimally sized tiles for execution. At the same time, the software layer intelligently manages the scratchpad to minimise data movement, thereby achieving high simulation performance and modelling a realistic, optimised software stack.

4

Methodology

This chapter details the three core pillars of the experimental methodology used to evaluate the gem5-fsnpu framework. Section 4.1 describes the experimental setup, including the baseline simulated hardware platform, the configurable NPU design space parameters, and the suite of software benchmarks used to stress the system. Section 4.1.4 defines the performance metrics and the analysis framework used to deconstruct latency into its constituent components. Finally, Section 4.2 outlines the specific Design Space Exploration (DSE) studies conducted, from characterising the DMA engine to analysing full system-level performance on complex AI workloads like MLP and MHA.

4.1 Experimental Setup

All experiments are conducted within a consistent full-system simulation environment to ensure that performance variations can be attributed directly to the NPU parameters under investigation.

4.1.1 Baseline Simulated Hardware Platform

The host system is modelled on the gem5 `ArmBoard` using the `VExpress_GEM5_Foundation` platform. Its key components are:

- Processor: A single-core, 3 GHz ARM CPU. The simulation boots using the functional and fast `ATOMIC` CPU model and transitions to the detailed `MINOR` timing model for workload execution to balance simulation speed and accuracy.
- Memory Hierarchy: The system includes a predefined cache hierarchy featuring discrete 16 KiB L1 instruction and data caches and a 256 KiB L2 cache. Main memory is a 2 GiB DRAM modelled using the `DualChannelDDR4_2400` model in gem5.
- NPU Integration: The custom NPU model is integrated as a PCI device. This configuration models a realistic SoC where accelerators are peripherals managed by the host OS via a system bus.
- Operating System: The environment runs a guest Linux OS based on kernel version `5.4.49`, using a compatible `arm64` bootloader and disk image provided by the gem5-resources repository.

4.1.2 NPU Design Space Parameters

The DSE revolves around systematically varying the NPU's core architectural parameters. These are configured at runtime and directly influence both the hardware model's behaviour and the decisions made by the software stack's tiling algorithm.

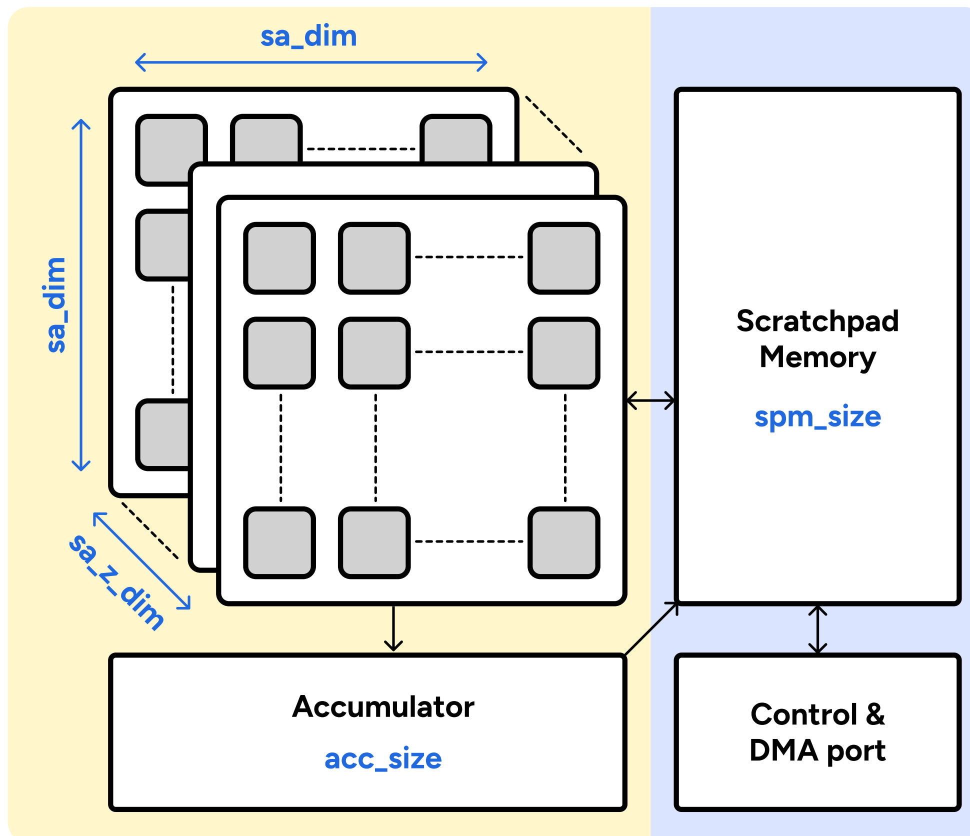


Figure 4.1: Configuration parameters for NPU model

- Scratchpad Memory (SPM) Size (`spm_size`): Defines the on-chip SRAM capacity. A small SPM necessitates "memory-constrained" software tiling, increasing DMA traffic and control overhead (Refer to Section 3.3). A large SPM enables "memory-sufficient" tiling, maximising data reuse and minimising main memory accesses. This parameter is swept from **32 KiB to 4 MiB** to analyse this critical trade-off.
- Systolic Array Dimensions (`sa_dim`, `sa_z_dim`): These parameters set the physical size of the array. `sa_dim` defines the 2D ($Rows \times Columns$) dimensions of a single layer, quadratically increasing the number of PEs and peak throughput. `sa_z_dim` adds a third dimension ($Layers$), enabling 3D array configurations. This provides an alternative path to parallelism, primarily exploited by the Output Stationary dataflow[19]. Dimensions **16, 32, and 64** are explored for

sa_dim, and layers from **1 to 5** for sa_z_dim.

- Accumulator Size (acc_size): This buffer stores partial sums and directly dictates the hardware tile dimension (N_{tile}) for the Weight Stationary (WS) dataflow. A larger accumulator allows for larger hardware tiles, increasing computational intensity and efficiency.
- Dataflow (dataflow): The experiments compare two canonical dataflows: Weight Stationary (WS) and Output Stationary (OS). WS is optimised for weight reuse, while OS is efficient for accumulating partial products in general matrix multiplications.

4.1.3 Software Benchmarks

A curated suite of benchmarks is used to evaluate the NPU across a spectrum of computational patterns, from primitive operations to complex neural network layers.

- DMA Microbenchmark: Isolates the DMA engine's performance by executing only memory transfers (dma_mvin, dma_mvout) of varying sizes. This establishes a baseline for I/O latency and characterises the data movement in the system.
- General Matrix Multiplication (GEMM): As the foundational operation for most AI workloads, GEMM is the primary benchmark for characterising core NPU performance. Experiments vary matrix dimensions and NPU configurations to stress the tiling algorithm and dataflow efficiency.
- Multi-Layer Perceptron (MLP): Represents a common feed-forward network, consisting of a sequence of GEMM operations interleaved with CPU-based bias and activation functions. This benchmark highlights the system-level overhead of frequent CPU-NPU interaction.
- Multi-Head Attention (MHA): The most complex workload, representing a key component of Transformer models. It involves multiple, dependent GEMM operations of varying sizes, stressing the entire hardware-software stack, including driver call overhead and OS scheduling.

4.1.4 Performance Metrics and Analysis Framework

A post-processing pipeline correlates extracted software timestamps with gem5 hardware statistics to deconstruct total latency into its constituent components, enabling precise bottleneck analysis. Figure 4.2 illustrates the timeline of a npu_matmul function, starting from the ioctl calls made by the user application by invoking the wrapper library, and finishing at the point where the interrupt is raised to signal completion. In the following bar in the figure, the components of the overall execution time as perceived by the software are shown.

The primary metrics are:

- Total SW-Perceived Latency: The end-to-end time measured in the user application, from invoking an NPU function to its return. This represents the

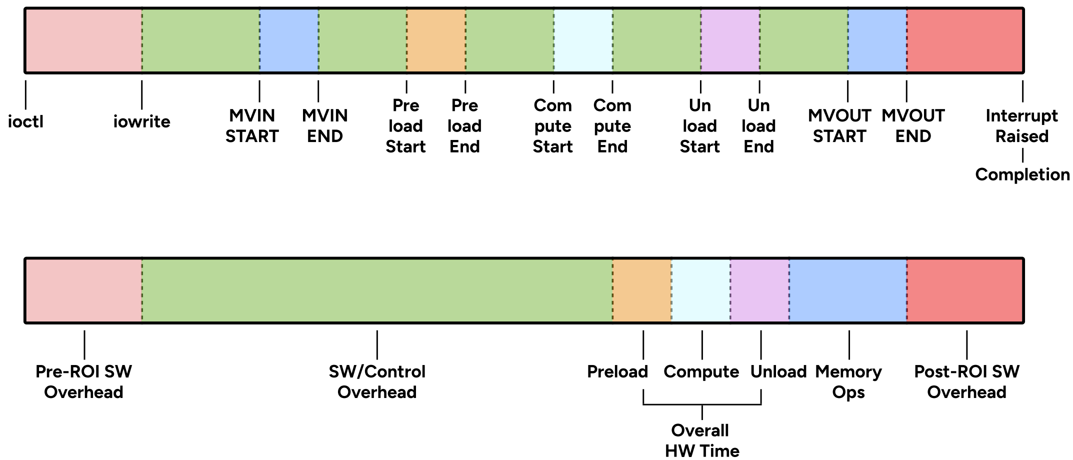


Figure 4.2: Timeline of simplified `npu_matmul` function (upper bar) and breakdown for latency components (lower bar)

effective performance for the end-user.

- True HW Execution Latency: The time the NPU hardware is active, from the first MMIO command write to the interrupt raised for completion of the NPU function. This is further broken down into:
  - Compute Ticks: Time spent in the systolic array performing computations.
  - Pre/Un-loading Ticks: Time spent in Pre/Un-loading the systolic array (or Accumulator) before and after computation.
  - DMA Ticks: Time spent transferring data between main memory and SPM.
  - SW/Control Overhead Ticks: The latency between successive hardware commands. This critical metric captures the fine-grained software overhead that analytical models miss, especially during tiled execution, where computations and memory transfers get interleaved for SW-level tiling.
- Pre/Post-ROI SW Overhead: The software latency before the first hardware command and after the final hardware event, respectively. This includes system call transitions, driver processing, and process wakeup delays.

4.2 Design Space Exploration Studies

The DSE is structured as a series of targeted studies, each designed to answer specific questions about the NPU architecture and its interaction with the system.

4.2.1 DMA Engine Characterisation

This initial study isolates the DMA subsystem to establish a baseline for control overhead and to quantify the latency of memory transfers as well as the associated

software overhead as a function of data size. Simple `dma_mvin` and `dma_mvout` operations are executed for data sizes of 256, 512, 1024, and 2048 KiB.

We hypothesise that core memory transfer time will scale linearly with data size. The software overhead is expected to scale asymmetrically: pre-ROI overhead should scale with `dma_mvin` (due to `copy_from_user`) and post-ROI overhead should scale with `dma_mvout` (due to `copy_to_user`). Additionally, the software overheads are not expected to scale as strongly as the hardware execution time, so the hardware transfer time, as a percentage of total execution time, should grow with the data transfer size.

4.2.2 Core GEMM Performance Analysis

This extensive study uses GEMM to explore the primary trade-offs in NPU hardware design. All experiments are run under two memory conditions: HW-tiling only (4 MiB SPM, preventing software tiling) and HW+SW tiling (64 KiB SPM, forcing software tiling).

1. Impact of Systolic Array Dimension: This multi-angle study analyses how scaling the number of PEs affects performance on a fixed $256 \times 256 \times 256$ GEMM. We sweep `sa_dim` through **16, 32, 64** in three scenarios:
   (a) Iso-Unit (WS): Accumulator size is scaled proportionally (`acc_size` = `sa_dim`^{2}) to maintain a constant tile shape relative to the array size.
   (b) Iso-Cost (WS): Accumulator size is fixed at 4096 elements to model a constant on-chip memory cost.
   (c) OS Dataflow: The OS dataflow is tested, which is not dependent on accumulator size.
2. Sensitivity to SPM Size: To quantify the overhead of software tiling, GEMM ($256 \times 256 \times 256$) is run on a fixed NPU configuration (`sa_dim` = 32) while sweeping SPM size from 32 KiB to 512 KiB. We expect performance to improve and then plateau as the SPM becomes large enough to reduce the number of software-managed chunks to the extent that the entire input, weight and output matrices can be stored in the SPM.
3. Comparing Dataflows and 2D vs. 3D Array organisation: This study investigates two paths to increasing parallelism: increasing `acc_size` to enable larger N_{tile} in a 2D WS dataflow, versus increasing `sa_z_dim` to the array for the OS dataflow, thereby increasing K_{tile}. We compare:
   (a) Scaling Compute: For a 32×32 array, we scale the WS dataflow by increasing `acc_size` by factors of 1-5, and scale the OS dataflow by increasing `sa_z_dim` from 1 to 5.
   (b) Iso-Compute Units: We compare configurations with an equal number of PEs, such as a 2D WS $32 \times 32 \times 1$ array against a 3D OS $16 \times 16 \times 4$ array, and a 2D WS $64 \times 64 \times 1$ array against a 3D OS $32 \times 32 \times 4$ array to analyse the efficiency of different physical arrangements.

4. Robustness to Matrix Aspect Ratio: To test the tiling algorithm's intelligence, we execute GEMMs with a near-constant number of MACs ($\approx$ 16.7M) but different shapes: Square $((256 \times 256) \times (256 \times 256))$, Tall $((1024 \times 128) \times (128 \times 128))$, and Wide $((128 \times 1024) \times (1024 \times 128))$. This tests the algorithm's ability to minimise DMA traffic and to maintain similar performance under different conditions.

4.2.3 System-Level Analysis of a Multi-Layer Perceptron

This study moves beyond single operations to a representative Machine Layer Perceptron (MLP) workload, introducing CPU-NPU interaction overheads. Figure 4.3 depicts the flow of operations in an MLP. Note that `batch` refers to Batch size and `d_model` refers to Model dimension.

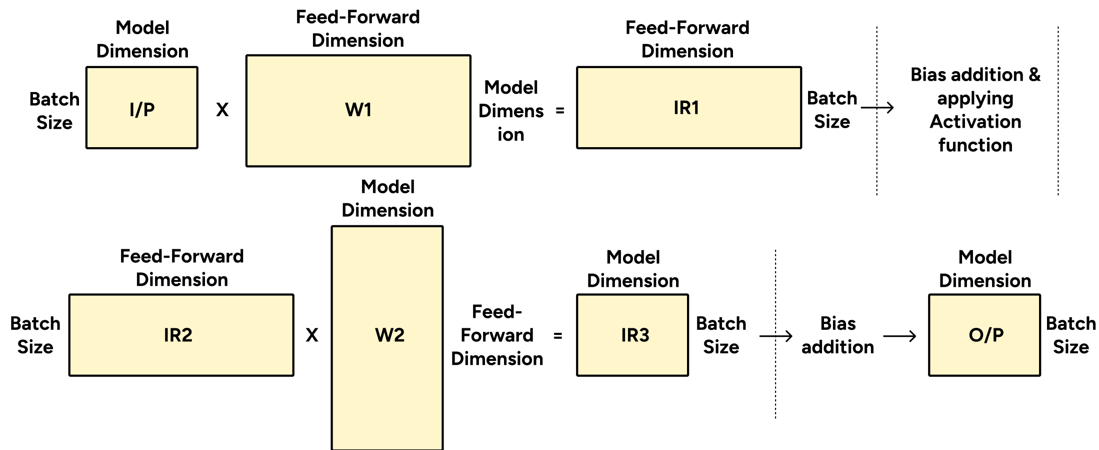


Figure 4.3: Multi-layer Perceptron

1. Workload Scaling: With a fixed NPU configuration (`sa_dim` = 32, 4 MiB SPM), we vary MLP parameters, sweeping batch sizes (32, 128, 256) and model dimensions (256, 512, 768), to understand how performance scales with problem size.
2. Hardware Scaling: For a fixed MLP workload (`batch` = 64, `d_model` = 512), we sweep key hardware parameters (`sa_dim`, SPM size, and dataflow/compute scaling) similar to the GEMM studies to see how these choices impact a multi-stage workload.

4.2.4 System-Level Analysis of a Multi-Head Attention Block

The final study uses the complex MHA workload to place maximum stress on the full system, particularly the control path. Note that `seq_len` refers to sequence length, `d_model` refers to Model dimension, `d_k` refers to head dimension.

- Workload Scaling: We scale MHA parameters, sweeping sequence lengths (64, 128, 256) and model dimensions (256, 512, 768), keeping the head dimension

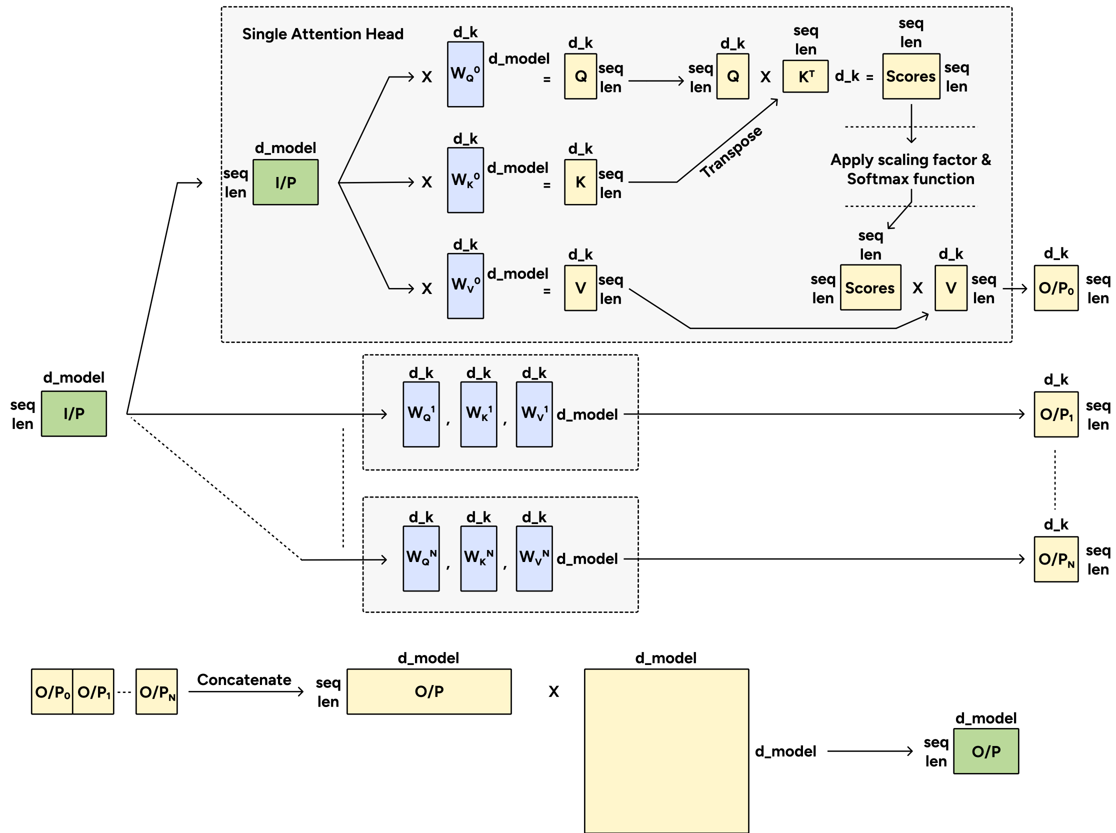


Figure 4.4: Multi-headed Attention layer

constant at 64.

- Hardware Scaling: For a fixed MHA workload (`seq_len` = 128, `d_model` = 512), we repeat the hardware scaling experiments from the MLP study. The SPM sweep is adjusted to 64 KiB to 4 MiB to account for MHA's larger memory footprint.
- Heads vs. Head Dimension Trade-off: With a fixed model dimension (512), we vary the number of heads (4, 8, 16, 32, 64). This creates a trade-off as more heads lead to smaller, more numerous matrix multiplications, testing the system's ability to handle frequent, low-latency dispatch versus fewer, larger computations.

5

Results

This chapter presents and analyses the results obtained from the Design Space Exploration (DSE) studies detailed in Chapter 4. We begin in Section 5.1 with a characterisation of the DMA engine to establish a baseline for memory transfer latency and software overhead. Section 5.2 provides a deep dive into core GEMM performance, exploring the impact of systolic array dimensions, SPM size, and dataflow. Following this, Section 5.3 evaluates the system under a Multi-Layer Perceptron (MLP) workload to analyse CPU-NPU interaction. Finally, Section 5.4 places maximum stress on the framework with a Multi-Head Attention (MHA) block, revealing the performance-limiting role of the software control path in complex, command-intensive workloads.

5.1 DMA Engine Characterisation

This initial study isolates the DMA subsystem to establish a baseline for the latency of memory transfers and to quantify the software overheads inherent in managing the NPU as a function of data size. As detailed in the methodology (4.2.1), simple `dma_mvin` (main memory to SPM) and `dma_mvout` (SPM to main memory) workloads were executed across a range of data sizes. The results, visualised in Figures 5.1, 5.2, and 5.3, provide valuable insights into the system-level costs that affect all NPU operations.

Based on figure 5.1, the hypothesis that the proportion of HW transfer time would grow as the data size grew is confirmed for both memory operations. One surprising observation is how small the proportion of time spent in the DMA engine is, when compared to the software overheads, proving that the system-level or SW-level effects indeed dominate the overall execution time, rather than the actual hardware execution. Additionally, the figure reveals that for smaller transfers, the collective software overheads constitute the vast majority of the total latency.

Examining the absolute scaling of latency components in Figure 5.2 further clarifies these dynamics. The plot shows that the `HW DMA Latency` scales in a near-perfect linear fashion with the data size, validating the hypothesis that the hardware's performance is proportional to the workload. The software overheads also exhibit a clear, albeit less pronounced, scaling with data size. This is an important nuance, as it indicates that the overhead is not a fixed constant but contains components that are dependent on the amount of data being handled by the driver and OS kernel.

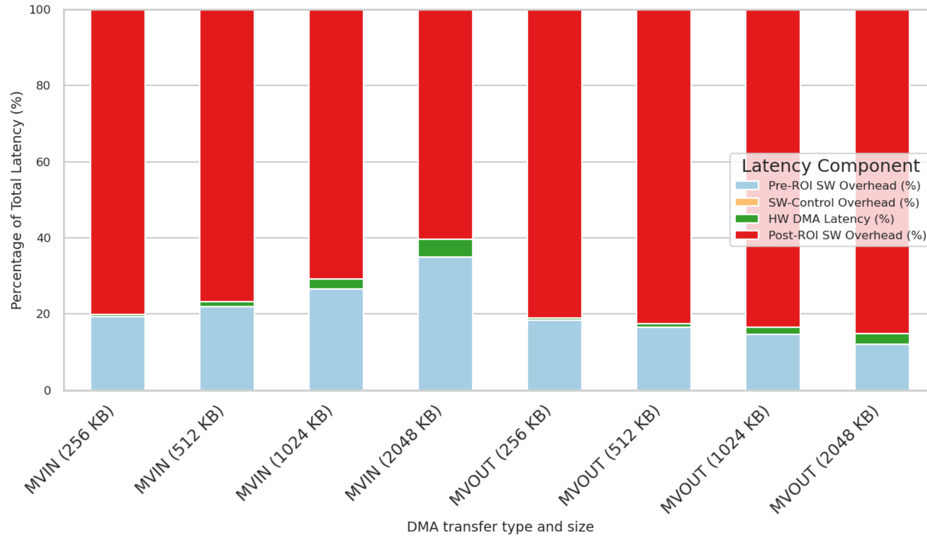


Figure 5.1: Latency breakdown by component for Memory transfer

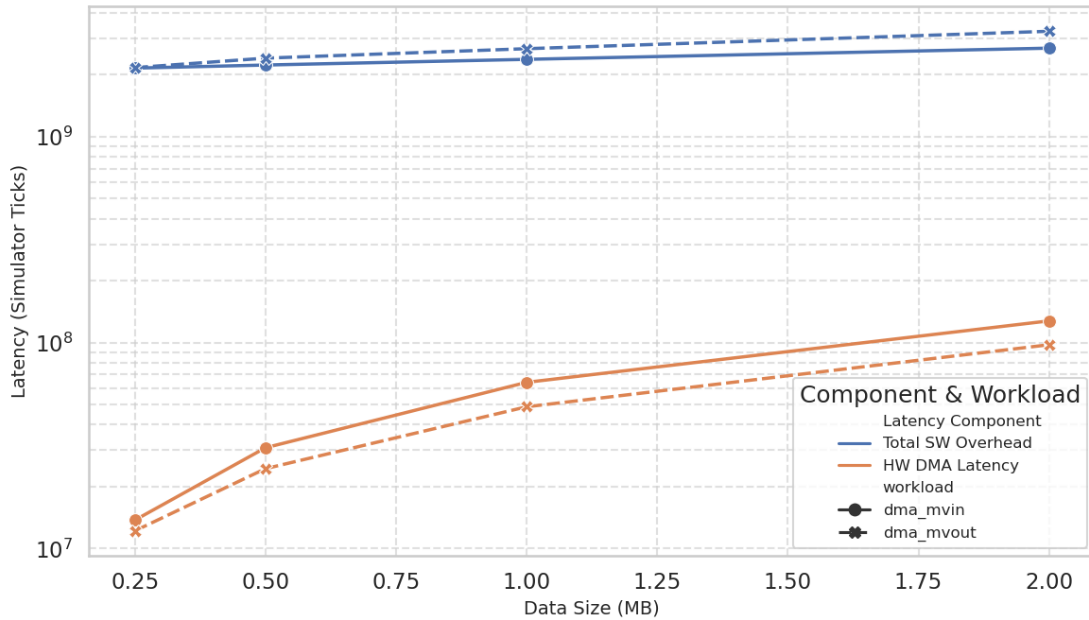


Figure 5.2: Scaling of SW overheads and Memory transfer time with data size

Finally, Figure 5.3 provides a detailed view of the Pre- and Post-ROI software overheads, confirming the hypothesis regarding their asymmetric scaling. For the `dma_mvin` operation, the Pre-ROI overhead scales with data size, which is directly attributable to the `copy_from_user` call required to move data into the kernel's bounce buffer before initiating the transfer. Conversely, for `dma_mvout`, the Post-ROI overhead scales with data size, reflecting the cost of the `copy_to_user` call after the DMA completes. Furthermore, the Post-ROI overhead is consistently larger than its Pre-ROI counterpart across all tests. We attribute this additional latency to the OS-level operations required to wake the user process, which was put to

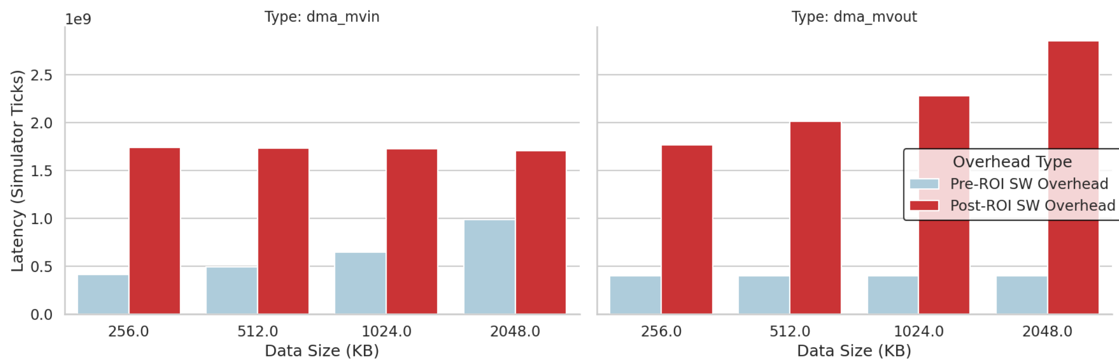


Figure 5.3: Comparison of SW overhead scaling for `dma_mvin` and `dma_mvout`

sleep by a `wait_event_interruptible` call in the driver, and perform a context switch back to it after the hardware interrupt signals completion. These results illustrate the dominant system-level bottlenecks that are present in a heterogeneous system, and are a direct validation of this thesis's core premise that system-level and software-stack effects can easily dominate hardware execution time, and their impact can only be accurately measured within a full-system simulation framework such as the one being proposed here.

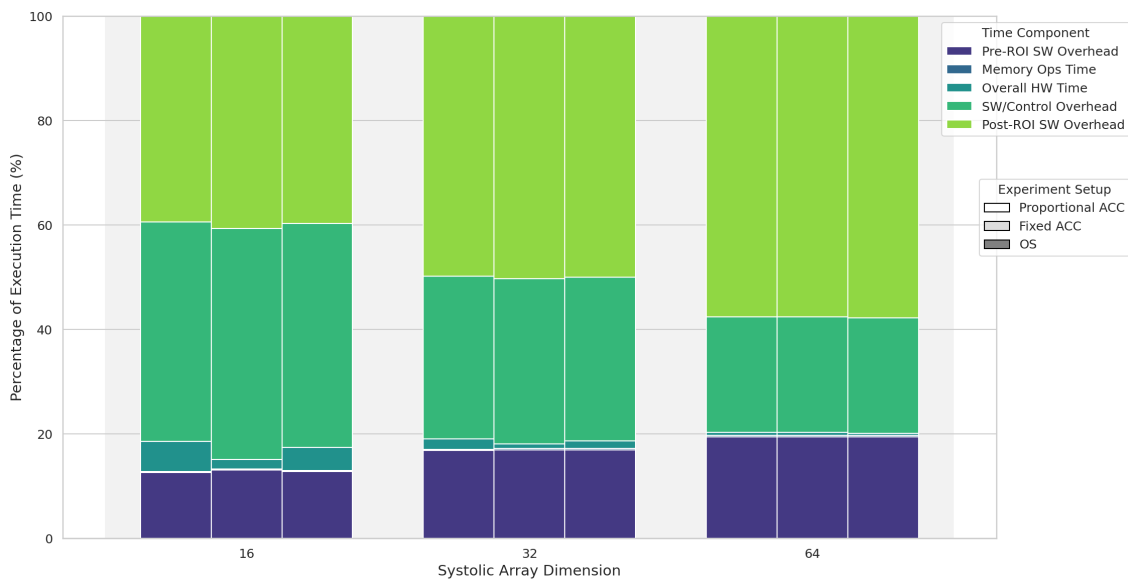
5.2 Core GEMM Performance Analysis

This study explores the trade-offs in the NPU hardware parameters using the foundational GEMM benchmark. The analysis is structured around the DSE studies from Section 4.2, systematically evaluating the impact of systolic array dimensions, scratchpad memory size, dataflow, as well as matrix shape on the overall performance. All experiments are presented under two distinct scenarios: a memory-sufficient case with a 4 MiB SPM, which forces hardware-only tiling, and a memory-constrained case with a 64 KiB SPM, which necessitates the use of the two-level software and hardware tiling algorithm.

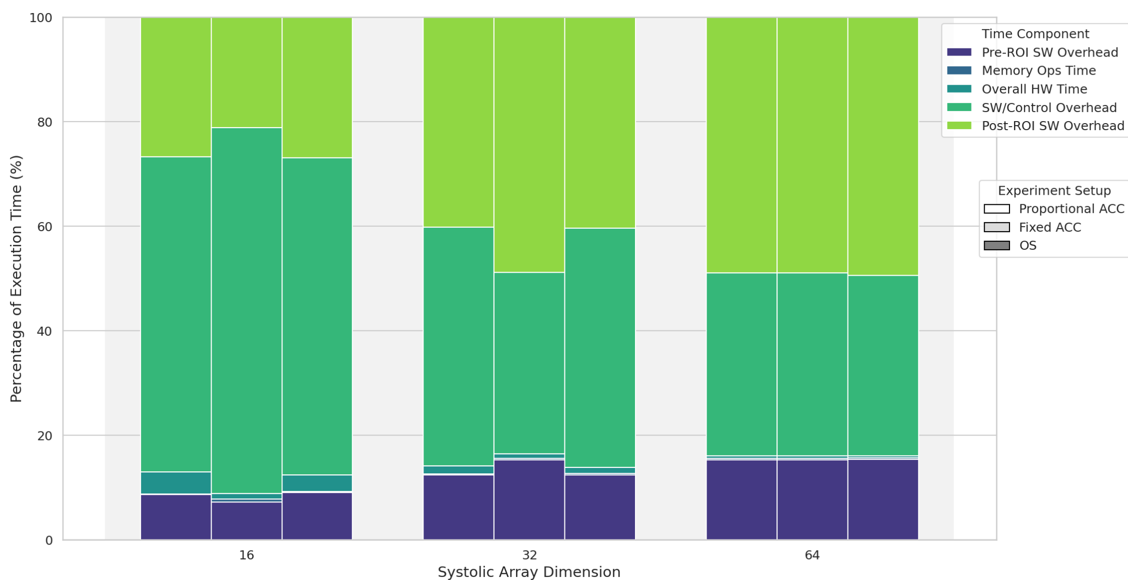
5.2.1 Impact of Systolic Array Dimension

This study analyses how performance scales with an increasing number of Processing Elements (PEs) for a fixed $256 \times 256 \times 256$ GEMM operation. We evaluate three distinct architectural scaling philosophies as described in the methodology: a Weight Stationary (WS) dataflow with an accumulator size proportional to the PE count (Proportional ACC), a WS dataflow with a fixed accumulator size (Fixed ACC), and an Output Stationary (OS) dataflow.

Figure 5.4 illustrates the relative time spent in each component of the execution. In the memory-sufficient case (Figure 5.4a), we observe that as the systolic array dimension increases, the Overall HW Time (blue-green) constitutes a progressively smaller fraction of the total latency for all configurations. This is significant as it shows that as the hardware becomes faster, fixed software overheads become the



(a) Memory-Sufficient (4 MiB SPM)

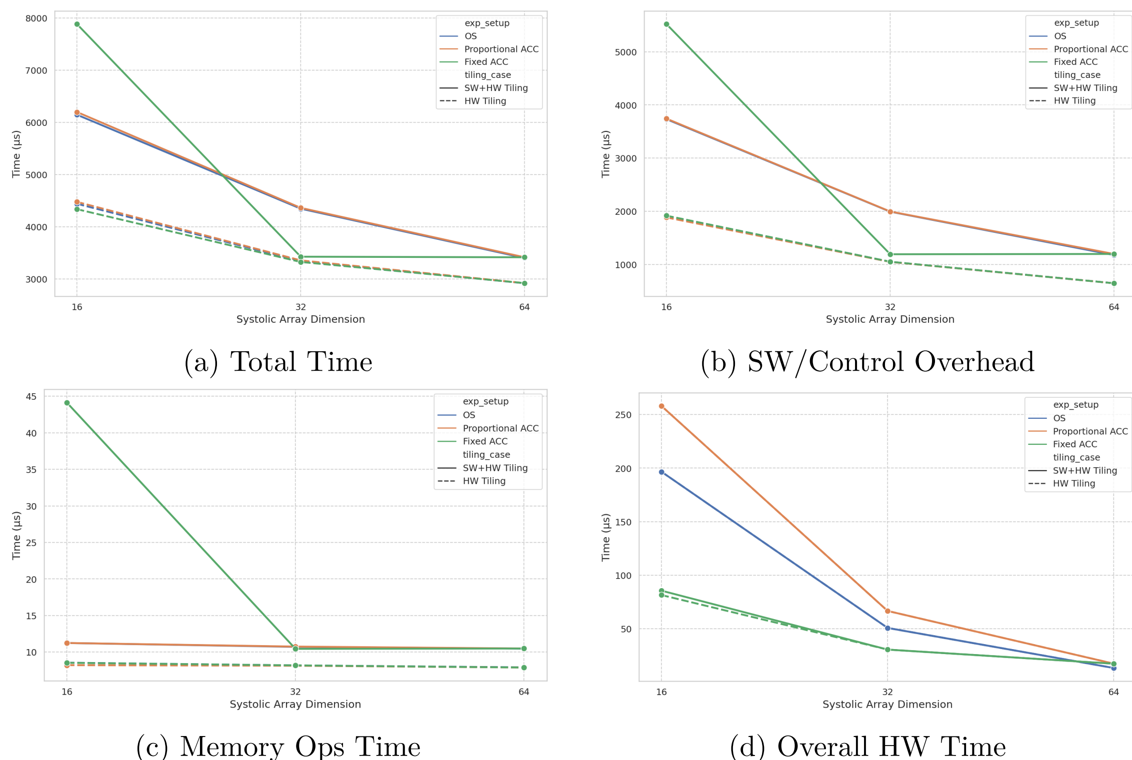


(b) Memory-Constrained (64 KiB SPM)

Figure 5.4: Relative latency breakdown for GEMM ($256 \times 256 \times 256$) across varying systolic array dimensions and scaling strategies

dominant bottleneck, an effect consistent with Amdahl's Law.

When the system is memory-constrained (Figure 5.4b), the performance profile changes dramatically. The introduction of software tiling leads to a significant increase in the proportion of SW/Control Overhead (purple) across all configurations. While scaling the array dimension still reduces total latency, the relative gains are diminished by the overhead of managing data movement between main memory and the smaller SPM. This demonstrates the trade-off that investing in more compute resources without sufficiently large on-chip memory can yield diminishing returns.



(a) Total Time (b) SW/Control Overhead

(c) Memory Ops Time (d) Overall HW Time

Figure 5.5: Scaling of performance metrics for GEMM ($256 \times 256 \times 256$) across varying systolic array dimensions and scaling strategies

The absolute scaling trends in Figure 5.5 provide a deeper understanding of these dynamics. As expected, Figure 5.5d shows that the Overall HW Time decreases substantially as the systolic array dimension grows, confirming that the core computational throughput scales with the number of PEs. For all configurations, quadrupling the PEs (e.g., from 16×16 to 32×32) results in a nearly four-fold reduction in hardware execution time.

However, the total software-perceived latency (Figure 5.5a) does not scale as ideally, particularly in the memory-constrained (SW+HW Tiling) case. The Fixed ACC configuration is a stark outlier, exhibiting poor performance with a 16×16 array, which then improves dramatically at 32×32 before plateauing. The reason for this behaviour is revealed in Figures 5.5b and 5.5c. At the 16×16 dimension, the combination of a fixed-size accumulator and a small array forces the software tiling algorithm into a highly inefficient mode, generating a large number of data chunks. This results in excessive DMA operations and control commands, leading to similarly excessive SW/Control Overhead and Memory Ops Time that completely dominate the execution.

In contrast, the Proportional ACC and OS cases scale more gracefully. By maintaining a hardware tile size that is better matched to the array's compute capacity, they enable the software to generate fewer, larger chunks, thus mitigating the control and memory overheads. This highlights a crucial hardware-software co-design principle: the raw compute power of an accelerator is only effective if its internal buffering and

data-path capabilities are scaled appropriately to allow the software stack to feed it efficiently. Interestingly, as the array size increases to 64×64, the performance of all three strategies begins to converge in the memory-constrained scenario. At this scale, the hardware becomes so fast that the total execution time is once again dominated by the fixed software overheads and the memory bandwidth, reducing the impact of the specific hardware dataflow configuration.

5.2.2 Sensitivity to SPM Size

To quantify the overhead of software tiling, a $256 \times 256 \times 256$ GEMM was executed on a fixed NPU configuration (`sa_dim` $= 32$) while sweeping the SPM size from 32 KiB to 512 KiB. This range ensures the SPM is always smaller than the total data footprint, forcing the software tiling algorithm to activate.

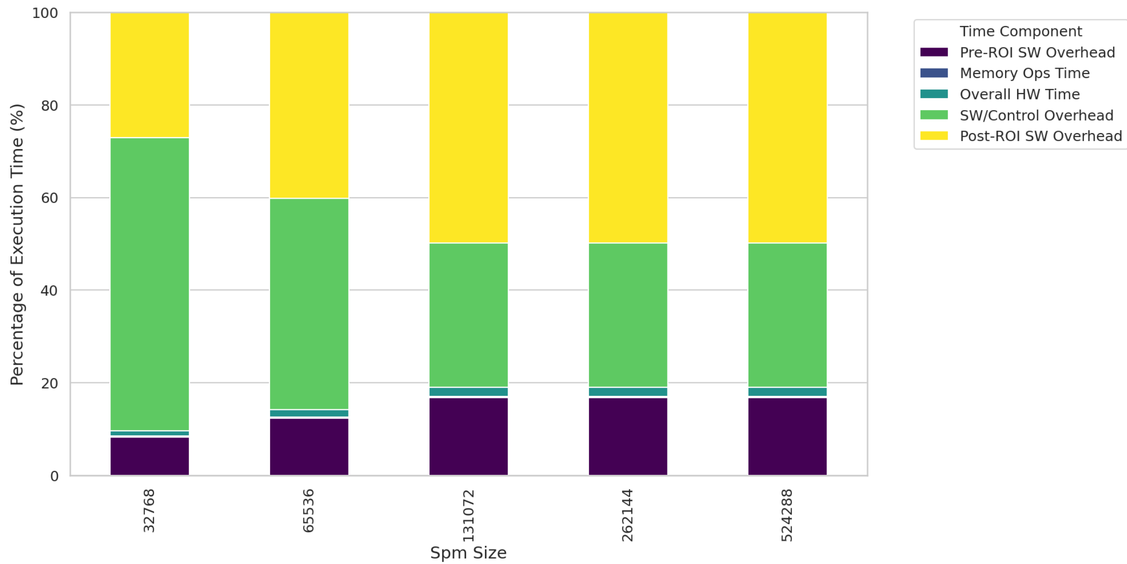
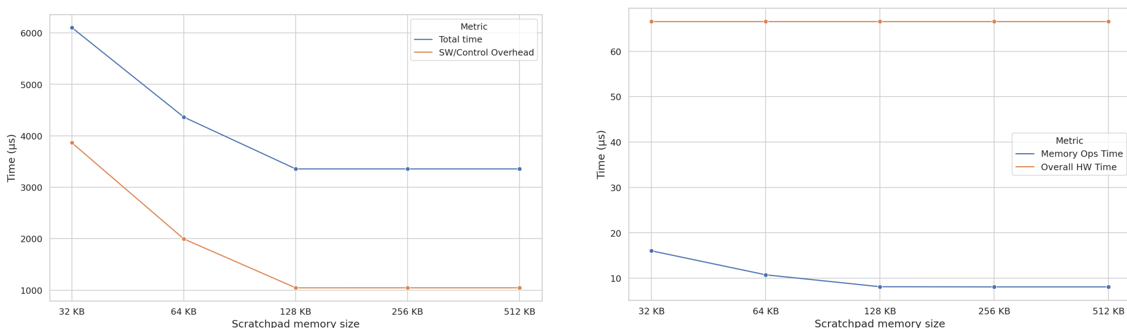


Figure 5.6: Relative latency breakdown for GEMM ($256 \times 256 \times 256$) across varying SPM sizes



(a) Total latency and SW/Control overhead

(b) Memory Ops and Overall HW Time

Figure 5.7: Scaling of performance metrics as a function of SPM size.

The results, shown in Figures 5.6 and 5.7, reveal the impact of on-chip memory capacity on system-level performance. As the SPM size increases, the total software-

perceived latency drops sharply and then begins to plateau. This proves that once the SPM is large enough, the overheads associated with frequent data movement and control commands are drastically reduced as they require minimal memory accesses, and further increases in memory yield diminishing returns.

Figure 5.7a shows that the Total time falls by nearly 50% as the SPM size is increased from 32 KiB to 128 KiB. The SW/Control Overhead scales in near-perfect lockstep, confirming that this overhead is directly correlated with the number of software-managed chunks required to complete the operation. The underlying cause is explained by Figure 5.7b. The Overall HW Time remains constant, as the total number of computations is unchanged. However, the Memory Ops Time decreases significantly. A larger SPM allows the tiling algorithm to load bigger chunks of the input matrices, reducing repeated transfers of the same matrix chunks as well as the total number of DMA transfers required. Fewer transfers mean less time spent on memory operations and, just as importantly, fewer `ioctl` calls from the user library to the driver, which is a significant source of the SW/Control Overhead.

Once the SPM size reaches 128 KiB, the performance plateaus. At this point, the SPM is large enough for the software tiling algorithm to find a near-optimal solution for chunk sizes, maximising data reuse within the scratchpad. Any additional memory beyond this point goes unused for this particular problem size, providing no further performance benefit. This study validates the effectiveness of the tiling algorithm's memory-sufficient strategy and highlights that balancing on-chip memory capacity with compute resources is essential for efficient accelerator design, and simply over-provisioning memory is not a cost-effective solution.

5.2.3 Comparing Dataflows and 2D vs. 3D Array Organisation

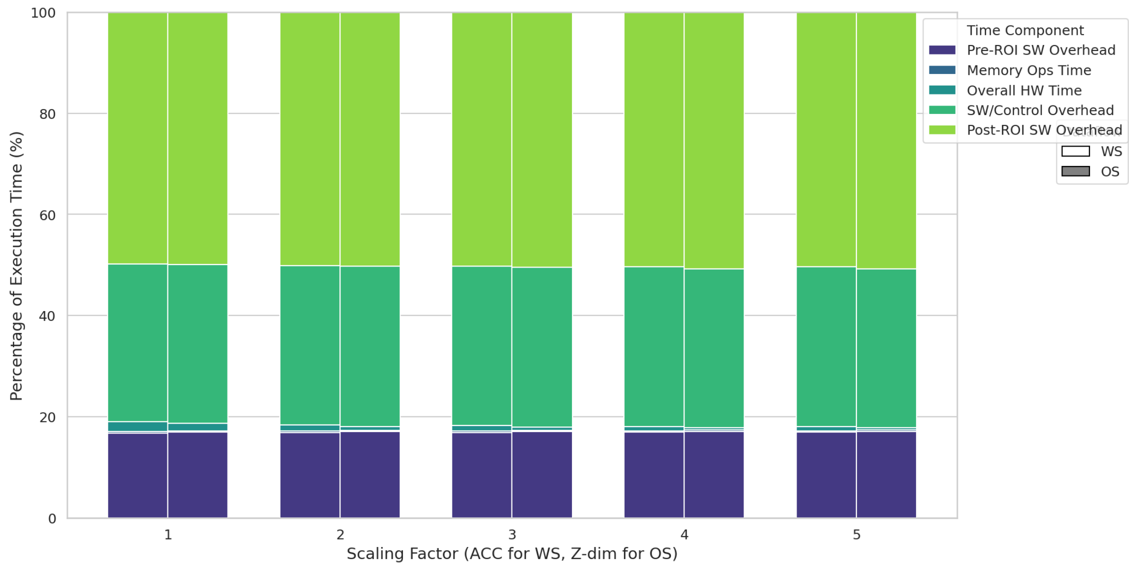
This study investigates two dataflows and their respective paths to increasing parallelism. For a WS dataflow, we increase computational intensity by scaling the accumulator size, while for an OS dataflow, we add layers to the systolic array in the third dimension (`sa_z_dim`).

5.2.3.1 Scaling Compute

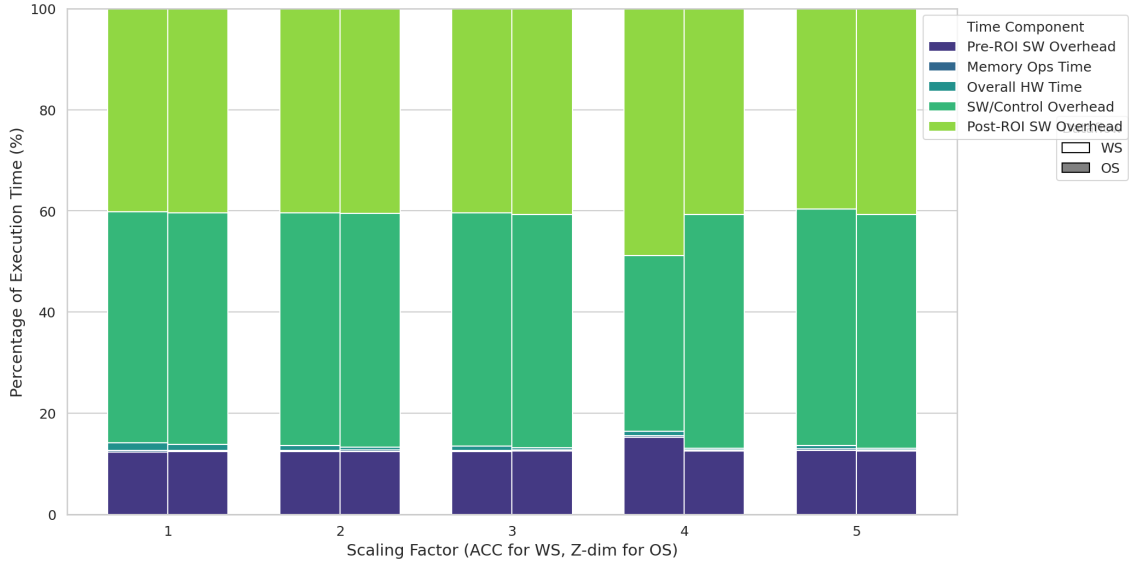
In this experiment, we start with a baseline 32×32 array and scale the accumulator size (for WS) and the number of Z-layers (for OS) by a factor from 1 to 5.

The absolute scaling plots in Figure 5.9 reveal the limitations of naively scaling compute. In all cases, increasing the scaling factor reduces the Overall HW Time. In contrast, the total time shows no improvement, as the hardware tile size is already optimal and the system is not compute-bound.

The reason behind the flat total time curve is revealed when we observe Figure 5.8. The Overall HW time is a small sliver of the total time, and the performance gains from faster hardware (seen in the decreasing Overall HW Time for OS) are completely nullified by the dominant SW/Control Overhead. This overhead is a



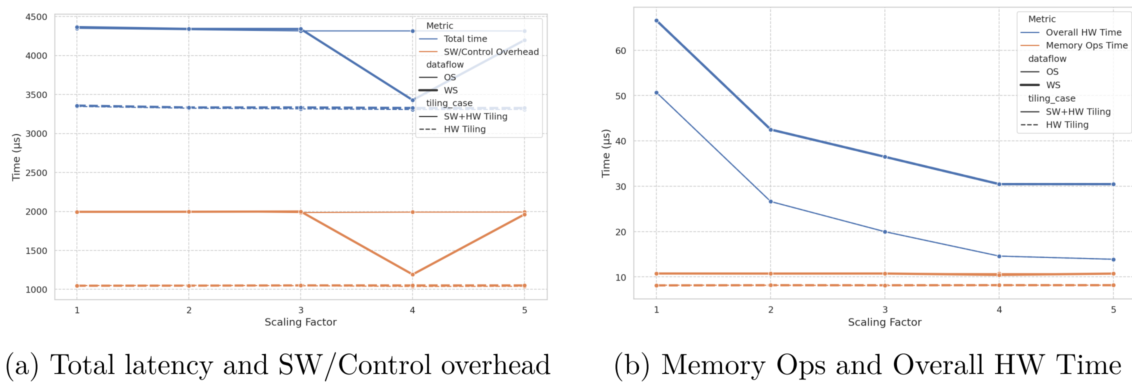
(a) Memory-Sufficient (4 MiB SPM)



(b) Memory-Constrained (64 KiB SPM)

Figure 5.8: Relative latency breakdown for GEMM ($256 \times 256 \times 256$) across varying compute tile scaling factors

direct result of the fine-grained control and DMA commands required to perform the computation. The system is bottlenecked by the software's ability to feed the accelerator, not by the accelerator's raw compute power. This reinforces the conclusion that increasing computational resources is ineffective if the on-chip memory and control path cannot keep pace. The insight that improving hardware resulted in negligible overall performance is uniquely enabled by this framework, as it requires a combination of dynamic system-level characteristics as well as the NPU's core execution.



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.9: Scaling of performance metrics as a function of compute tile scaling factor

5.2.3.2 Iso-Compute Units

Here, we compare configurations with an equal number of PEs but different physical arrangements to analyse the efficiency of 2D versus 3D array organisations. Figure 5.10 compares a 2D $32 \times 32 \times 1$ WS array against a 3D $16 \times 16 \times 4$ OS array (both 1024 PEs), and a 2D $64 \times 64 \times 1$ WS array against a 3D $32 \times 32 \times 4$ OS array (both 4096 PEs).

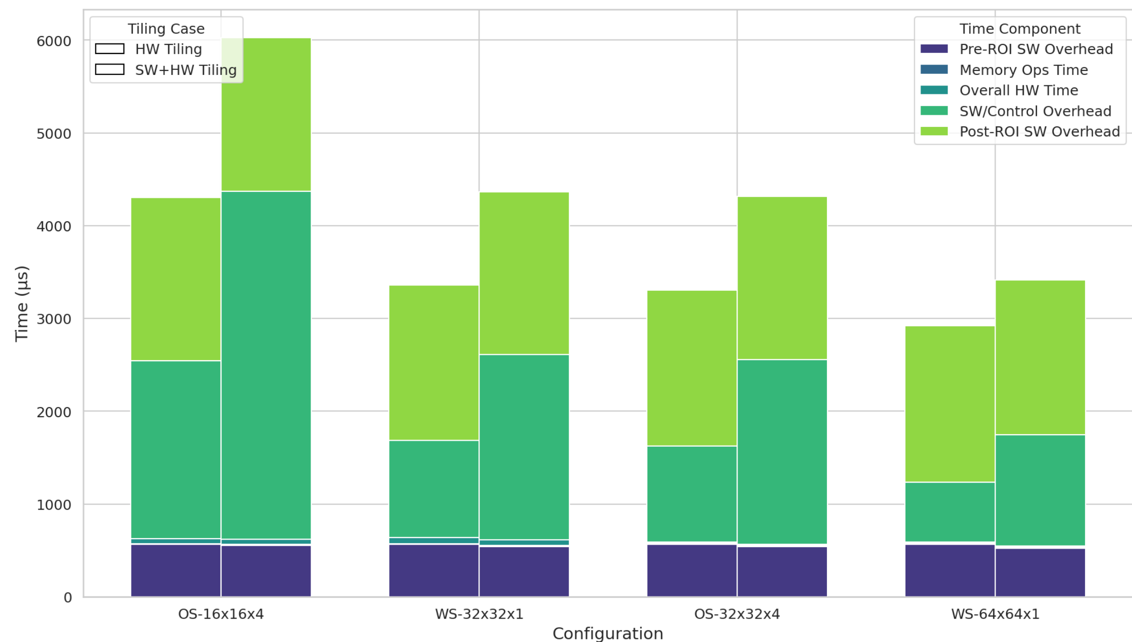
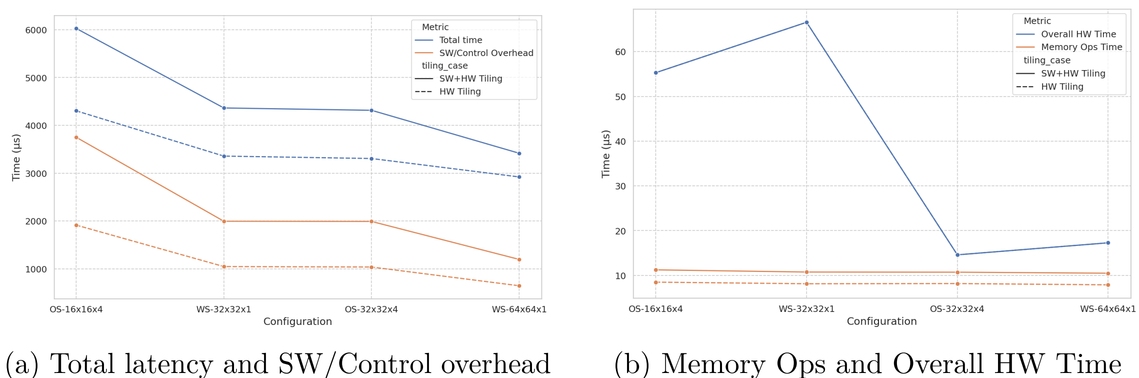


Figure 5.10: Absolute latency breakdown for NPU configurations with an equal number of PEs

The results in Figures 5.10 and 5.11 show that for the same PE count, the 2D WS configurations consistently outperform their 3D OS counterparts in terms of total software-perceived latency. The breakdown reveals a fascinating trade-off. The 3D OS configurations exhibit a lower Overall HW Time due to their architectural design, which processes the K dimension more efficiently. However, this hardware advantage



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.11: Scaling of performance metrics for NPU configurations with an equal number of PEs

is more than negated by significantly higher SW/Control Overhead and Memory Ops Time.

This suggests that while the 3D OS array is theoretically faster at computation, its data requirements lead to a less efficient memory access pattern when managed by the software tiling algorithm. The smaller tile footprint in the N and M dimensions may force the software to generate a larger number of smaller, less contiguous data chunks, thereby increasing the frequency of DMA operations and the associated driver interaction overhead. This insight is critical as it demonstrates that the physical arrangement of PEs and its synergy with the dataflow and software stack are as critical as the raw PE count itself. This counterintuitive result, where a simpler architecture proves more efficient, is a key finding uniquely enabled by this framework. Analytical models or isolated simulators would likely favour the 3D array's higher peak throughput, completely missing the software and memory access overheads that dictate real-world performance.

5.2.4 Robustness to Matrix Aspect Ratio

The final GEMM study tests the intelligence of the tiling algorithm by executing multiplications with a near-constant number of MACs but three different matrix shapes: Square ($256 \times 256 \times 256$), Tall ($1024 \times 128 \times 128$), and Wide ($128 \times 1024 \times 128$). A robust algorithm should achieve similar performance across all three by adapting its data chunking strategy.

Figure 5.12 confirms the effectiveness of the tiling algorithm. The total software-perceived execution time is remarkably consistent across all three aspect ratios for both WS and OS dataflows. This proves that the algorithm successfully adapts its chunking strategy to minimise DMA traffic and maintain high hardware utilisation regardless of the input data shape.

The minor variations in performance can be understood by examining the underlying hardware and memory operations in Figure 5.13. While the total number of computations is nearly identical, the Tall and Wide matrix shapes lead to a slight increase in both Overall HW Time and Memory Ops Time compared to the Square case. This

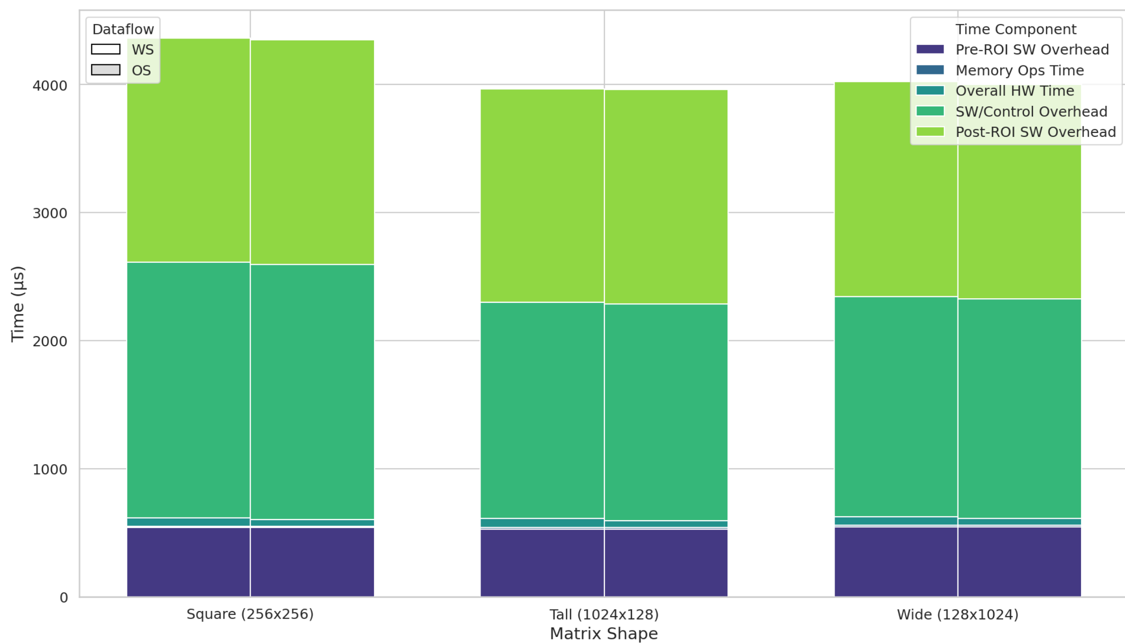


Figure 5.12: Absolute latency breakdown across different matrix aspect ratios

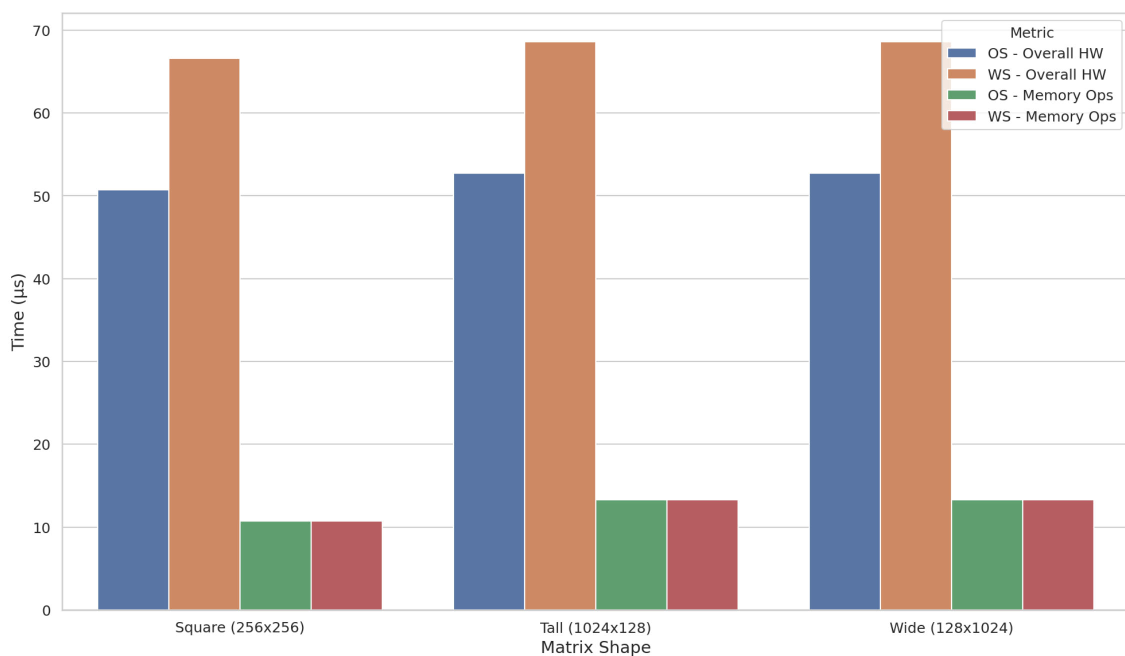


Figure 5.13: Scaling of Memory Ops and Overall HW Time across different matrix aspect ratios

is an expected outcome of the tiling process; non-square shapes can result in edge tiles that are not perfectly sized for the systolic array, leading to slight inefficiencies in both computation and data packing for DMA. Furthermore, the small difference between the Tall and Wide results reflects the greedy heuristic in the software, which prioritises keeping the larger of the two input matrices resident in the SPM to reduce data movement. The consistent performance across these diverse workloads validates

the hardware-software co-design approach, demonstrating that an intelligent software stack can effectively abstract away the complexities of the underlying hardware for different input shapes.

5.3 System-Level Analysis of a Multi-Layer Perceptron

This study moves beyond primitive operations to analyse a representative neural network layer, the Multi-Layer Perceptron (MLP). An MLP workload consists of a sequence of GEMMs interleaved with CPU-handled bias addition and activation functions. This structure makes it a benchmark for quantifying the system-level overheads that arise from frequent interaction between the host CPU and the NPU.

5.3.1 Workload Scaling

This initial set of experiments evaluates the system's performance on a fixed NPU configuration (`sa_dim`=32, 4 MiB SPM) while scaling the MLP workload's batch size and model dimension.

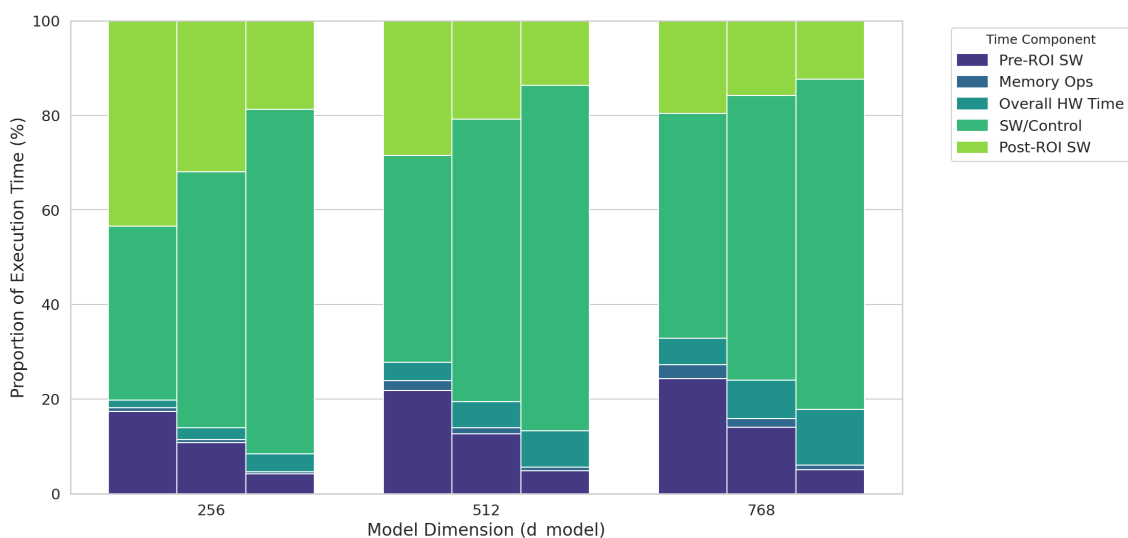
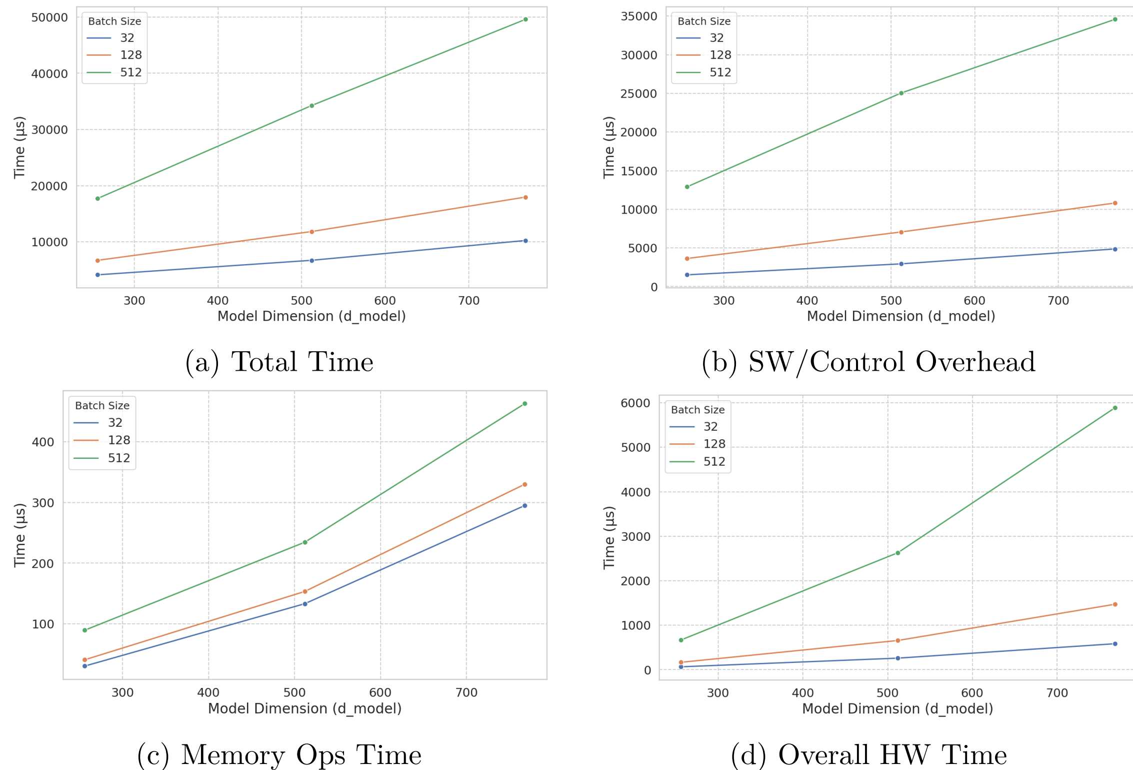


Figure 5.14: Relative latency breakdown for the MLP layer across varying workload sizes

Figure 5.14 illustrates the relative time spent in each component of the execution. As the workload size increases (moving from smaller batch/model dimensions to larger ones), the Overall HW Time constitutes a progressively larger fraction of the total latency. This is significant because it demonstrates the amortisation of fixed software overheads. For the smallest workload (Batch 32, `d_model` 256), the combined software overheads (SW/Control, as well as Pre & Post ROI) and memory operations are substantial. However, for the largest workload (Batch 512, `d_model` 768), the execution profile indicates that the proportion of NPU computation time is

considerably large. This finding validates a core principle of accelerator offloading that the efficiency is maximised when the offloaded task is large enough to make the fixed costs of software interaction negligible.



(a) Total Time

(b) SW/Control Overhead

(c) Memory Ops Time

(d) Overall HW Time

Figure 5.15: Scaling of performance metrics for MLP across varying workload sizes

The absolute scaling trends in Figure 5.15 provide a deeper understanding of these dynamics. As expected, Figure 5.15d shows that the Overall HW Time increases substantially with both batch size and model dimension, confirming that the NPU is correctly handling the increasing computational load of the GEMM operations. This hardware time is a significant factor behind the increase in Total time seen in Figure 5.15a.

However, the software and memory components scale very differently. The SW/Control Overhead (Figure 5.15b) and Memory Ops Time (Figure 5.15c) also grow with the workload size, but their slopes are far less steep than that of the hardware execution time. This shows that the cost of managing the MLP's sequence of operations (issuing commands via the driver, handling bias/activation on the CPU, and managing DMA transfers) does not grow as quickly as the core computation itself. It is this disparity in scaling that causes the software overheads to shrink into a small fraction of the total time for large problems, as was shown in Figure 5.14. These results demonstrate that in a realistic, multi-stage workload like an MLP, system-level efficiency is fundamentally linked to the problem size.

5.3.2 Hardware Scaling

In this section, the MLP workload is fixed (Batch 64, `d_model` 512), and the NPU's hardware parameters are varied to identify how architectural choices impact the performance of this multi-stage workload.

5.3.2.1 Impact of Systolic Array Dimension

This study analyses how performance scales with an increasing number of Processing Elements (PEs) for the fixed MLP workload. By scaling the systolic array dimension, we can observe the point at which the NPU's computational speed is no longer the primary system bottleneck.

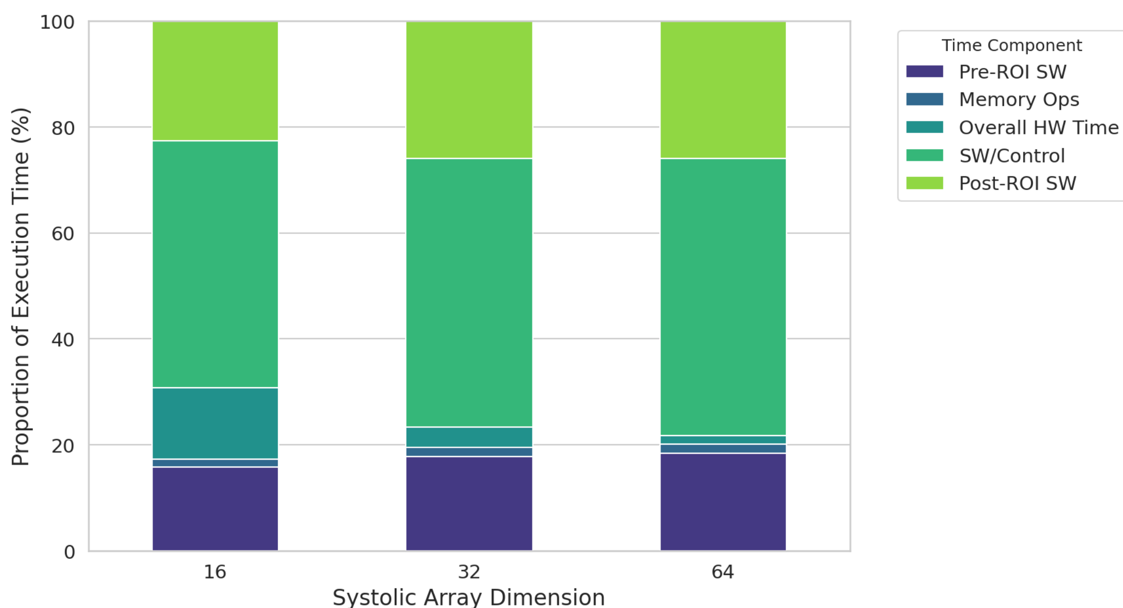
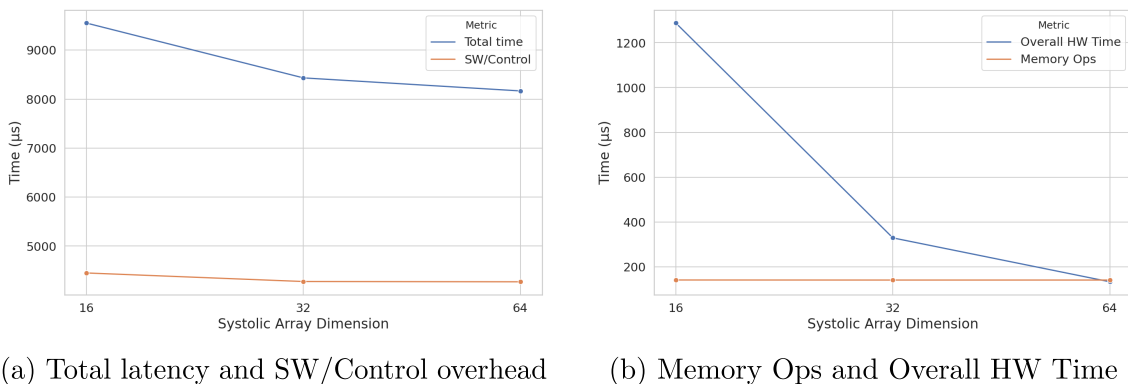


Figure 5.16: Relative latency breakdown for MLP (Batch 64, `d_model` 512) across varying systolic array dimensions

Figure 5.16 illustrates the relative time spent in each component of the execution as the array size increases. The most striking trend is the dramatic reduction in the proportion of Overall HW Time. For the 16×16 array, NPU computation accounts for a significant portion of the execution time. However, by the time we scale to a 64×64 array, the hardware execution is a much smaller fraction of the total latency. On the other hand, as we scale the systolic array, the software overheads, especially the SW/Control Overhead, become the dominant component. This is a clear demonstration of Amdahl's Law within a realistic workload. As the parallelizable portion (GEMM computation) is accelerated, the serial portions (software setup, CPU-side processing) begin to dictate overall performance.

The absolute scaling trends in Figure 5.17 further clarify these dynamics. Figure 5.17b shows that the Overall HW Time decreases substantially as the systolic array dimension grows, with a nearly four-fold reduction when moving from a 16×16 to a 32×32 array, confirming that the NPU's throughput scales with the number of PEs



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.17: Scaling of performance metrics for MLP (Batch 64, `d_model` 512) across varying systolic array dimensions

as expected. The Memory Ops time remains almost constant, as the total amount of data that needs to be moved for the MLP layer does not change with array size.

However, the Total time, shown in Figure 5.17a, does not scale nearly as ideally. While there is a significant performance improvement from the 16×16 to the 32×32 array, the benefit of scaling further to 64×64 is much smaller, demonstrating diminishing returns. The reason is evident from the other components: the SW/Control overhead remains largely flat, and as we know from Figure 5.16, the software overheads are a dominant component. Since the MLP workload involves multiple stages (two GEMMs, plus CPU-based bias/activation), the system incurs fixed software costs for each stage. Once the NPU becomes fast enough to make the hardware computation time smaller than these fixed overheads, further acceleration of the NPU yields little benefit to the total latency. This reinforces a key insight of this thesis that for multi-stage, interactive workloads, system performance is often limited by software and communication overhead rather than the raw compute power of the NPU.

5.3.2.2 Sensitivity to SPM size

To quantify the overhead of software tiling on a multi-stage workload, the fixed MLP workload was executed while sweeping the SPM size from 32 KiB to 512 KiB. This range shows the system transition from a memory-constrained to a memory-sufficient state, revealing the performance impact of on-chip memory capacity.

The results, shown in Figures 5.18 and 5.19, reveal the critical impact of on-chip memory on system-level performance. Figure 5.18 shows that for small SPM sizes (< 128 KiB), the execution is overwhelmingly dominated by SW/Control overhead. This overhead, which represents the latency of repeated driver calls needed to manage numerous small data chunks, is a direct consequence of the software tiling algorithm being constrained by limited memory. As the SPM size increases to 128 KiB and beyond, this overhead is drastically reduced, and the Overall HW Time becomes a much larger proportion of the total latency.

Examining the absolute scaling in Figure 5.19 clarifies these dynamics. As the SPM size increases, the Total time drops sharply, but after 128 KiB, it scales less

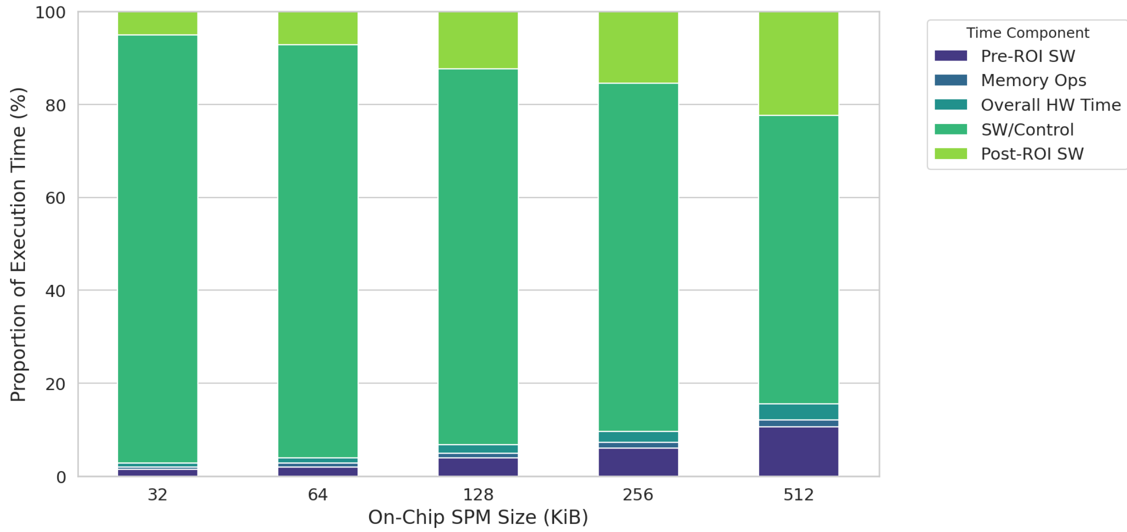
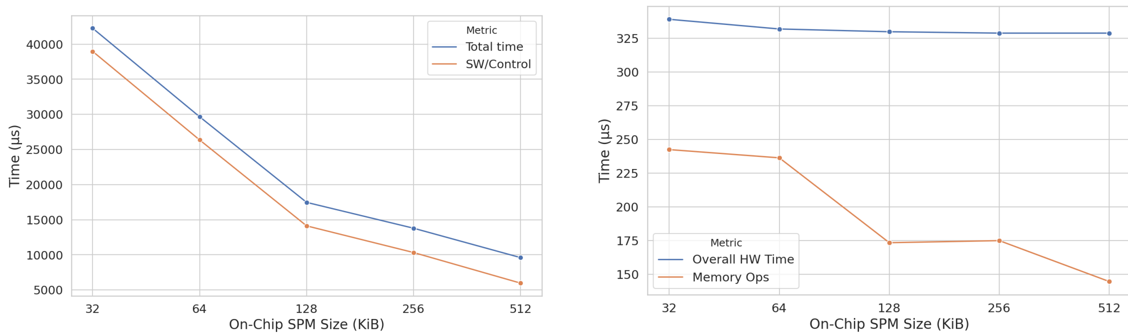


Figure 5.18: Relative latency breakdown for MLP (Batch 64, `d_model` 512) across varying SPM sizes



(a) Total latency and SW/Control overhead

(b) Memory Ops and Overall HW Time

Figure 5.19: Scaling of performance metrics for MLP (Batch 64, `d_model` 512) across varying SPM sizes

strongly. Figure 5.19a shows that the SW/Control Overhead scales in near-perfect lockstep with the total time, confirming that this overhead is the primary driver of the performance improvement. The underlying cause is explained by Figure 5.19b. The Overall HW Time remains nearly constant, as the total number of computations in the MLP layer is unchanged regardless of SPM size. However, the Memory Ops Time decreases significantly as the SPM grows. A larger SPM allows the tiling algorithm to load bigger chunks of the input and intermediate matrices, reducing the total number of DMA transfers required. Fewer DMA transfers imply less time spent on memory operations and, more importantly, fewer `ioctl` calls from the user library. This reduction in software interaction is the direct source of the dramatic drop in SW/Control Overhead.

Once the SPM size reaches 128 KiB, the performance doesn't improve as rapidly. At this point, the SPM is large enough for the software tiling algorithm to find a solution that is more optimal for the chunk sizes required by the MLP workload,

maximising data reuse within the scratchpad. Any additional memory beyond this point can be utilised, but only up to the plateau point (Similar to Section 5.2.2), after which additional memory will not improve performance. This study validates that balancing on-chip memory capacity with compute resources is essential and that an undersized SPM can create a severe software-level bottleneck that nullifies the benefits of a powerful accelerator core.

5.3.2.3 Dataflows and compute scaling

This study investigates the effect of the two dataflows and their distinct paths to increasing NPU parallelism for the fixed MLP workload. This comparison aims to determine if either dataflow can effectively reduce latency in a multi-stage, software-managed workload.

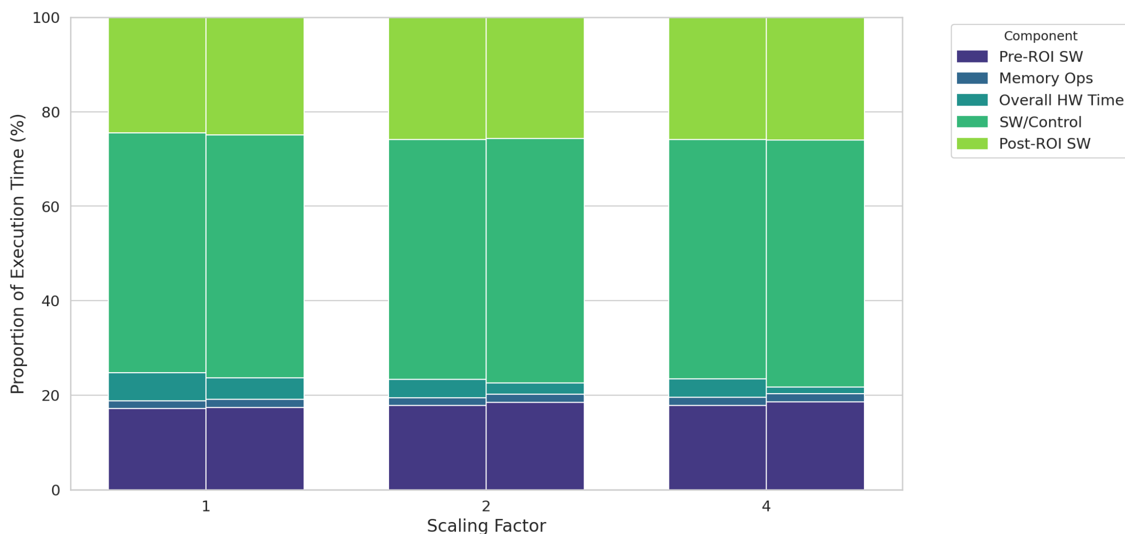
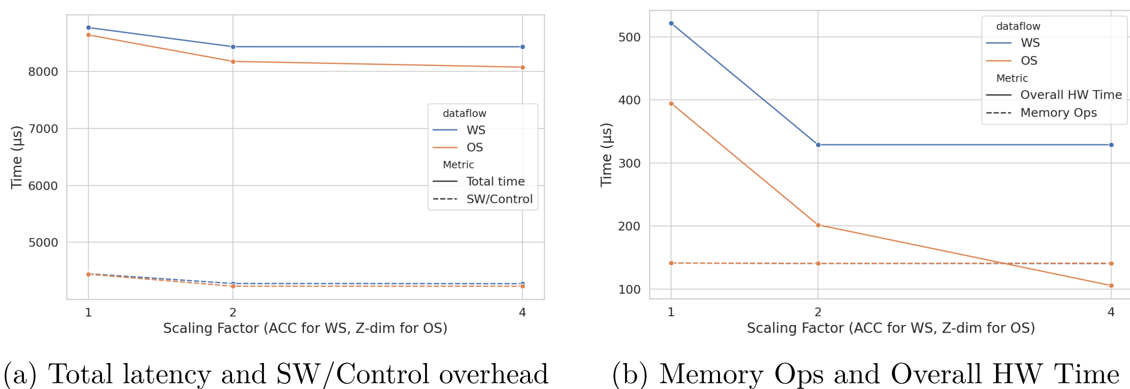


Figure 5.20: Relative latency breakdown for MLP (Batch 64, `d_model` 512) across varying compute tile scaling factors

The proportional breakdown in Figure 5.20 immediately reveals that the system is not compute-bound. Across all scaling factors, the execution profile is overwhelmingly dominated by software overhead components, particularly SW/Control Overhead. The actual Overall HW Time constitutes a very small fraction of the total execution time, suggesting that improvements in hardware speed may have little impact on the final performance.

The absolute scaling plots in Figure 5.21 confirm this hypothesis. Figure 5.21b shows that increasing the scaling factor does make the hardware faster. For the OS dataflow, the Overall HW Time decreases effectively as more layers are added. The WS dataflow also sees an initial hardware speed-up. This demonstrates that the architectural enhancements are functioning as intended at the hardware level.

However, Figure 5.21a reveals that these hardware gains do not translate into meaningful system-level performance improvements. The Total time for both dataflows is significantly higher and remains largely flat regardless of the scaling factor. The



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.21: Scaling of performance metrics for MLP (Batch 64, `d_model` 512) across varying compute tile scaling factors

performance gains from faster hardware are completely nullified by the dominant and nearly constant software overheads, particularly the SW/Control overhead. The system is fundamentally bottlenecked by the software's ability to orchestrate the multiple steps of the MLP layer, not by the NPU's raw compute power. This underscores a critical conclusion that for workloads with frequent software interaction, increasing computational resources is an ineffective optimisation strategy as the control path overhead remains the primary performance limiter. This conclusion also highlights the need for a holistic, full-system evaluation strategy that can explore the characteristics of both hardware and software effects.

5.3.2.4 2D vs. 3D Array Organisation

Finally, we compare configurations with an equal number of PEs but different array organisations to analyse the efficiency of 2D versus 3D array organisations for the fixed MLP workload. The study compares a 2D $32 \times 32 \times 1$ WS array against a 3D $16 \times 16 \times 4$ OS array (both 1024 PEs), and a 2D $64 \times 64 \times 1$ WS array against a 3D $32 \times 32 \times 4$ OS array (both 4096 PEs).

Figure 5.22 shows that, proportionally, the execution profiles are remarkably similar across all configurations. Regardless of whether the PEs are arranged in a 2D or 3D structure, the total latency is dominated by software overheads. This indicates that for this heavily software-bottlenecked workload, the physical organisation of the PEs does not fundamentally alter the distribution of system-level bottlenecks.

However, the absolute performance metrics in Figure 5.23 reveal a clear, but somewhat modest, trend. Figure 5.23a shows that the 2D WS configurations consistently achieve a lower total software-perceived latency than their 3D OS counterparts with the same PE count. The reason for this is multifaceted and best understood by examining the underlying hardware performance in Figure 5.23b. These results follow the pure GEMM results, as the Overall HW Time for the MLP workload is lower on the 3D OS arrays than on the 2D WS arrays. While the 3D organisation is architecturally superior for processing a large K dimension efficiently, the specific sequence and dimensions of the matrix multiplications within the MLP layer may not align with

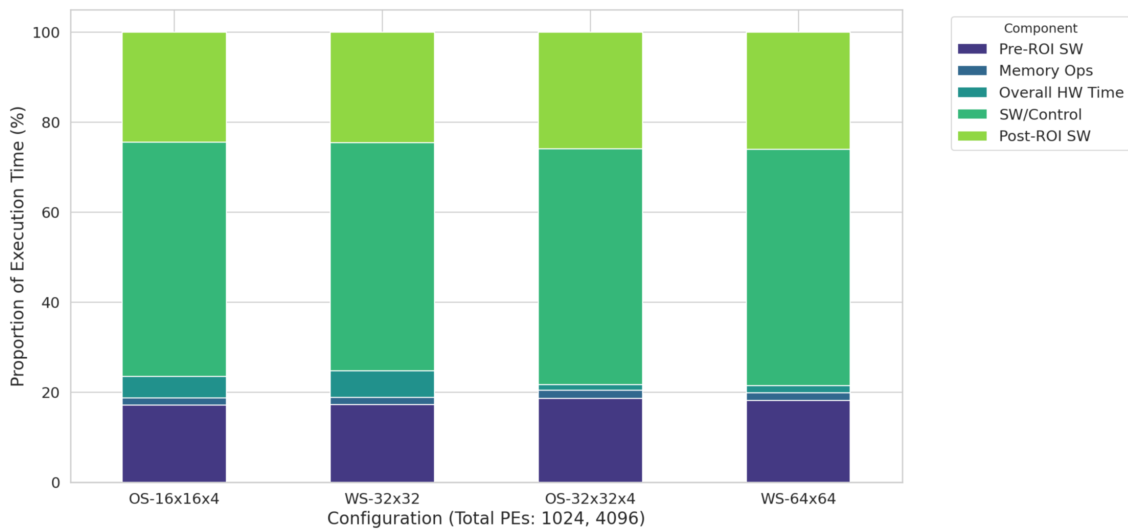
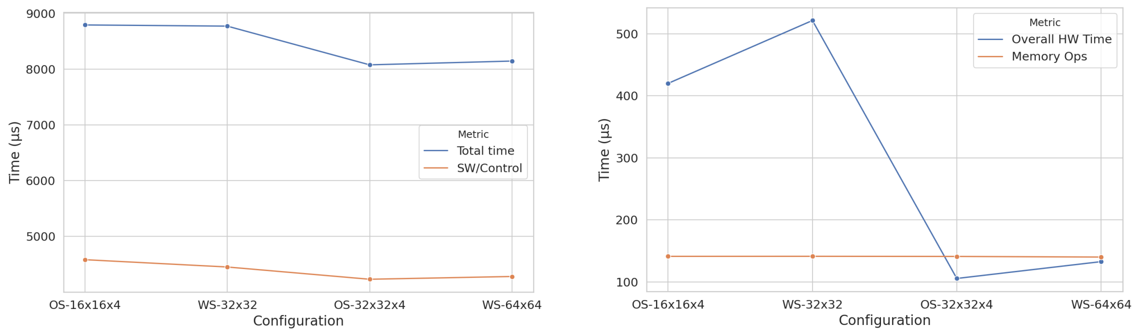


Figure 5.22: Relative latency breakdown for MLP (Batch 64, `d_model` 512) for various systolic array organisations



(a) Total latency and SW/Control overhead

(b) Memory Ops and Overall HW Time

Figure 5.23: Scaling of performance metrics for MLP (Batch 64, `d_model` 512) for various systolic array organisations

this strength. The simpler, more regular memory access patterns required by the 2D WS dataflow appear to be more efficient for the specific computational pattern of this MLP workload. This slight hardware advantage, combined with a correspondingly lower SW/Control overhead, results in a better overall system performance for the 2D configurations.

This insight is critical, as it demonstrates that the optimal physical arrangement of PEs is not universal but is instead highly dependent on the target workload. For a complex, multi-stage workload like an MLP, the simpler and more regular structure of a 2D array can prove more efficient from a holistic, system-level perspective.

5.4 System-Level Analysis of a Multi-Head Attention Block

This final study evaluates the framework using the Multi-Head Attention (MHA) block, the most complex workload in our benchmark suite. As a cornerstone of modern Transformer architectures, MHA's performance is critical, yet its computational pattern presents a unique challenge for NPU-based systems. Unlike the sequential and relatively uniform operations within an MLP, an MHA layer is composed of numerous smaller, interdependent matrix multiplications. This structure is designed to place maximum stress on the entire system, moving the performance bottleneck away from raw computation and towards the efficiency of the memory subsystem and the software control path.

5.4.1 Workload Scaling

This initial set of experiments evaluates the system's performance on a fixed NPU configuration (`sa_dim`=32, 4 MiB SPM) while scaling the MHA workload's parameters: sequence length and model dimension. The goal is to understand how the system's various overheads respond to the increasing complexity and size of a realistic Transformer layer.

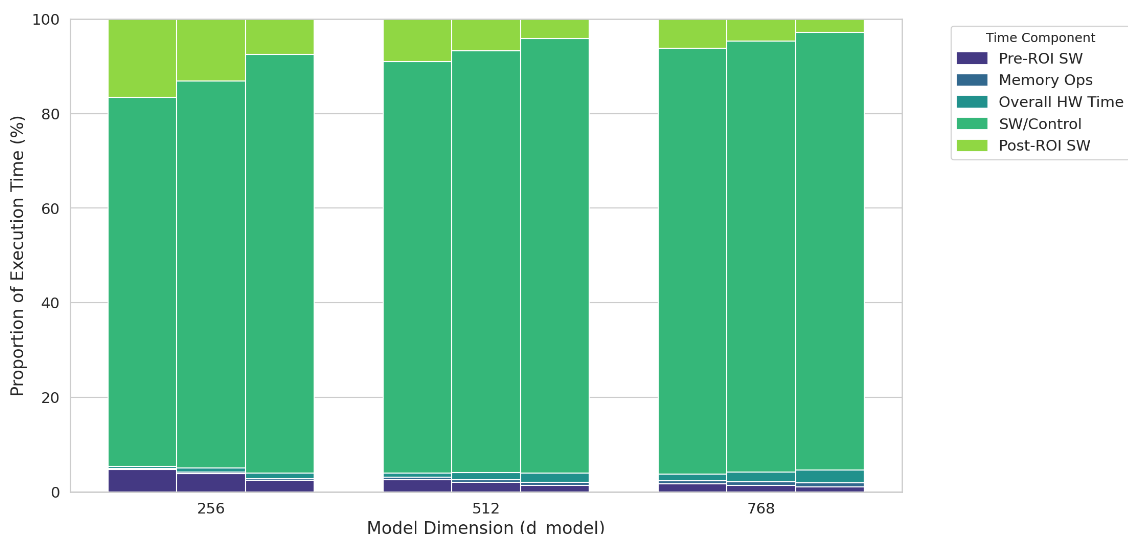
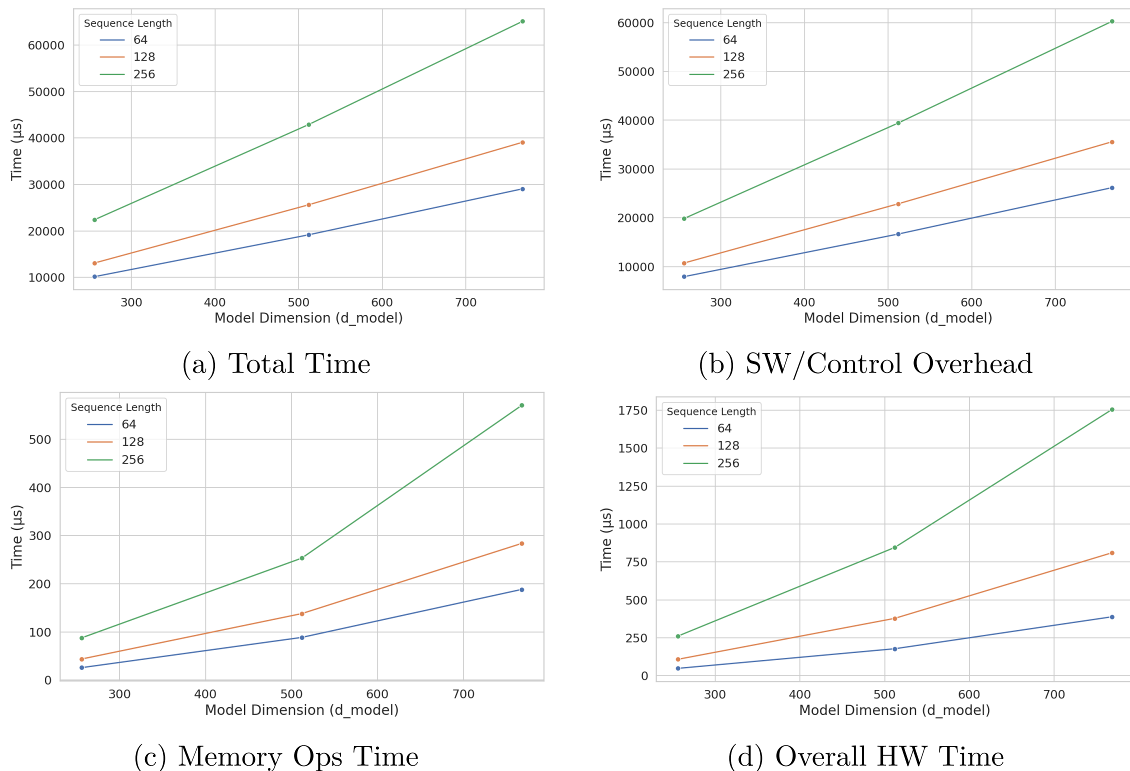


Figure 5.24: Relative latency breakdown for the MHA layer across varying workload sizes

The proportional breakdown in Figure 5.24 immediately reveals a starkly different performance profile compared to the MLP and GEMM benchmarks. The execution is overwhelmingly dominated by the SW/Control overhead across all workload sizes. This is a direct consequence of the MHA layer's structure, which involves a large number of smaller, interdependent matrix multiplications. This pattern creates a high frequency of `ioctl` calls to the driver to manage each step, making the software control path, rather than the NPU hardware, the primary system bottleneck. While

increasing the model dimension does lead to a slight increase in the proportion of time spent on Overall HW Time, the software overheads remain the largest contributing factor by a significant margin.



(a) Total Time

(b) SW/Control Overhead

(c) Memory Ops Time

(d) Overall HW Time

Figure 5.25: Scaling of performance metrics for MHA across varying workload sizes

The absolute scaling trends, shown in Figure 5.25, reinforce this conclusion. The Total time (Figure 5.25a) scales almost in perfect lockstep with the SW/Control Overhead (Figure 5.25b), which is by far the largest component of latency in absolute terms. In contrast, the Overall HW Time (Figure 5.25d) and Memory Ops Time (Figure 5.25c), while scaling as expected with the workload size, are orders of magnitude smaller. Their contribution to the total latency is almost negligible in comparison to the time spent managing the operations in software.

This powerfully demonstrates that for complex, fragmented workloads like MHA, system performance is not limited by the NPU's raw computational power but by the efficiency of the software stack in handling a high frequency of control commands. The results underscore a critical challenge in NPU design: as hardware becomes faster, the overhead of the OS, driver, and control logic becomes the dominant factor. Accurately modelling these system-level effects is paramount to understanding real-world performance. And is a core feature of the work being presented here.

5.4.2 Hardware Scaling

In this section, the MHA workload is fixed (`seq_len`=128, `d_model`=512, `num_heads`=8), and the NPU's hardware parameters are varied. This allows us to identify how ar-

chitectural choices impact the performance of this complex, multi-stage workload, which has already been shown to be heavily influenced by software overheads.

5.4.2.1 Impact of Systolic Array Dimension

This study analyses how system performance scales with an increasing number of PEs for the fixed MHA workload. Given that the workload is already bottlenecked by the software control path, this experiment is designed to quantify the degree of diminishing returns when accelerating only the compute portion of the execution.

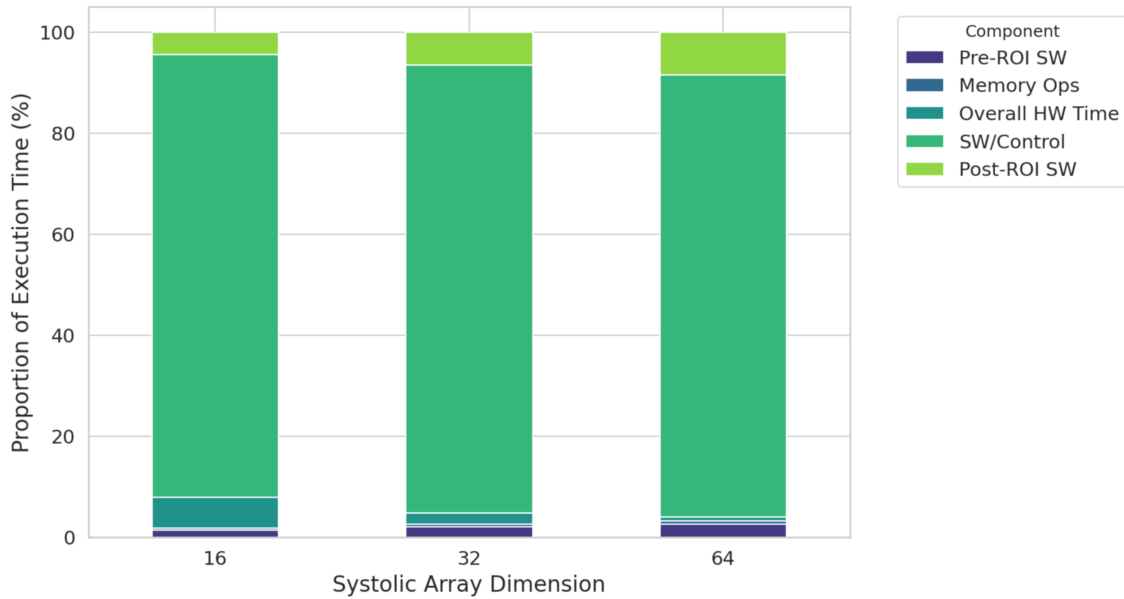
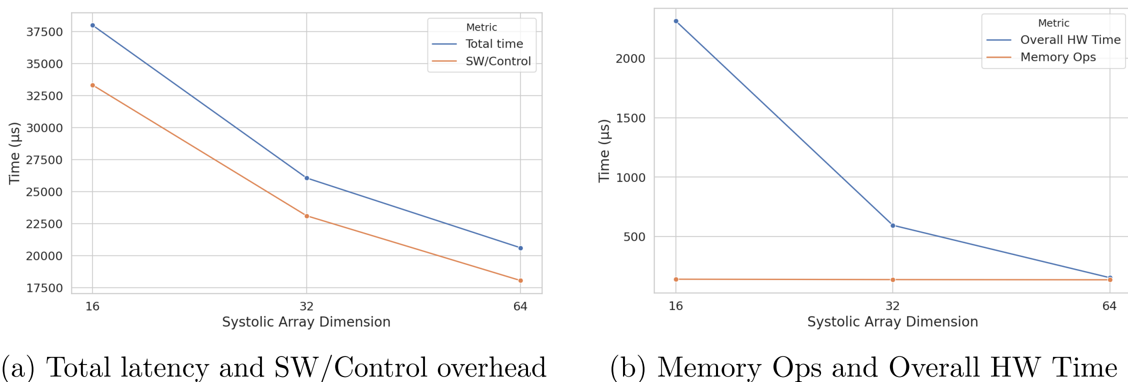


Figure 5.26: Relative latency breakdown for MHA (`seq_len` 128, `d_model` 512, `num_heads` 8) across varying systolic array dimensions

Figure 5.26 confirms the system's software-bound nature in the starkest terms yet. Even with the smallest 16×16 array, the Overall HW Time accounts for less than 10% of the total latency. As the array size is scaled up to 64×64, the hardware's contribution shrinks to a negligible fraction, while the SW/Control overhead continues to dominate the execution profile, making up close to 90% of the total time. This is a classic demonstration of Amdahl's Law, where optimising a small fraction of the total execution time yields minimal overall improvement.

The absolute scaling trends in Figure 5.27 provide a quantitative view of these diminishing returns. As expected, Figure 5.27b shows that the Overall HW Time decreases substantially as the systolic array grows, confirming that the NPU's raw throughput scales with the number of PEs. The Memory Ops time remains constant, as the total data moved is independent of the array size.

However, the impact on total performance is muted. Figure 5.27a shows that while the Total time does decrease, the improvement is modest. The reason is that the massive SW/Control overhead, while also decreasing, still constitutes the bulk of the latency. Interestingly, the SW/Control overhead itself scales down with a larger



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.27: Scaling of performance metrics for MHA (`seq_len` 128, `d_model` 512, `num_heads` 8) across varying systolic array dimensions

systolic array. This is because a larger array can process larger hardware tiles (N_{tile}, M_{tile}, K_{tile}), which in turn allows the software to process the overall workload in fewer, larger chunks. This reduces the total number of `ioctl` calls and MMIO commands required, thus lowering the software control overhead. Despite this software-level benefit, the fundamental latency of the control path remains the hard limit on performance. This result powerfully reinforces the central argument of this thesis that for complex, command-intensive workloads, investing in compute resources provides a twofold benefit of faster computation and reduced software overhead, but the gains are ultimately capped by the irreducible latency of the software control path.

5.4.2.2 Sensitivity to SPM size

This study quantifies the impact of on-chip memory capacity on the performance of the software-bottlenecked MHA workload. By sweeping the SPM size from 64 KiB to 4 MiB, we can observe how memory constraints interact with the already high software overheads inherent to this benchmark.

The proportional breakdown in Figure 5.28 shows that, regardless of memory size, the execution is overwhelmingly dominated by the SW/Control overhead. However, there is a noticeable, albeit small, reduction in this component's share as the SPM size increases. This suggests that while software overhead is the primary bottleneck, memory constraints can exacerbate the issue.

The absolute scaling trends in Figure 5.29 reveal the nature of this interaction. As the SPM size is increased from 64 KiB to 256 KiB, both the Total time and the SW/Control overhead drop significantly (Figure 5.29a). The underlying cause is revealed in Figure 5.29b, where it is seen that the Memory Ops Time is reduced. A larger SPM allows the software tiling algorithm to create larger data chunks. This reduces the total number of DMA transfers needed to service the MHA workload. Crucially, fewer DMA operations translate directly into fewer `ioctl` calls to the driver, which in turn reduces the primary source of the SW/Control overhead. The Overall HW Time remains constant, as the total computation is unchanged.

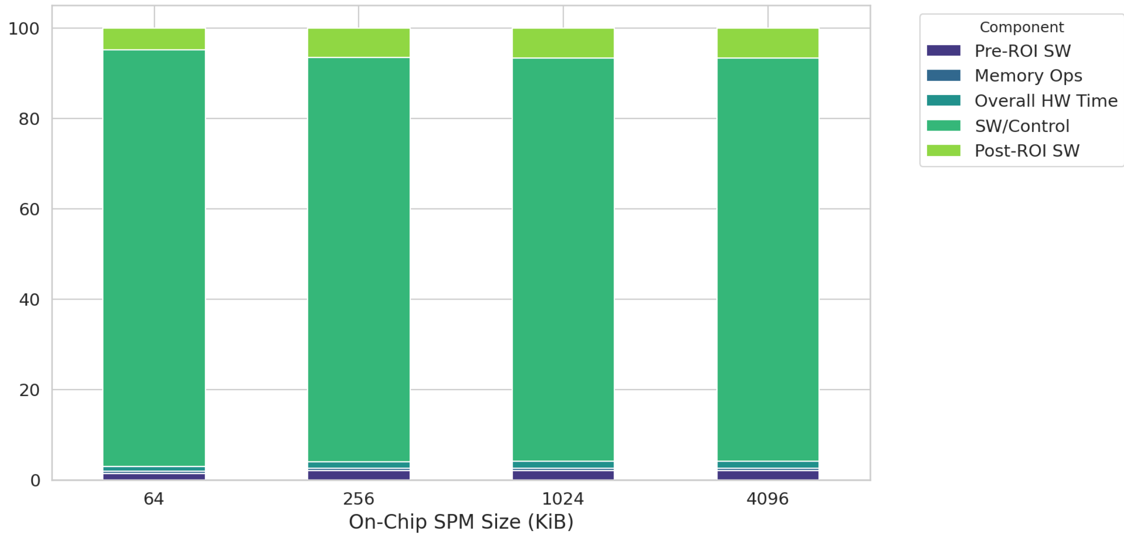
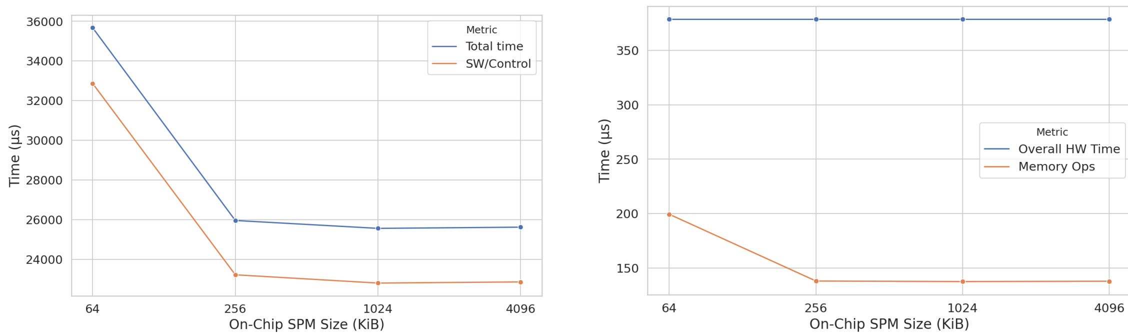


Figure 5.28: Relative latency breakdown for MHA (`seq_len` 128, `d_model` 512, `num_heads` 8) across varying SPM sizes



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.29: Scaling of performance metrics for MHA (`seq_len` 128, `d_model` 512, `num_heads` 8) across varying SPM sizes

After 256 KiB, the performance plateaus, and increasing the SPM to 4 MiB provides no further benefit. At this point, the SPM is large enough for the software to find a near-optimal chunking strategy, and the performance becomes entirely limited by the irreducible software overhead of managing the large number of sequential operations in the MHA block. This study highlights a critical co-design principle: while a sufficiently large SPM is essential to enable an efficient software tiling strategy, for command-intensive workloads, simply overprovisioning on-chip memory cannot overcome a fundamental bottleneck in the software control path.

5.4.2.3 Dataflows and compute scaling

This study investigates the effect of the two dataflows and their distinct paths to increasing NPU parallelism for the fixed MHA workload. This comparison aims to determine if either dataflow can effectively reduce latency in a workload that is already heavily bottlenecked by software.

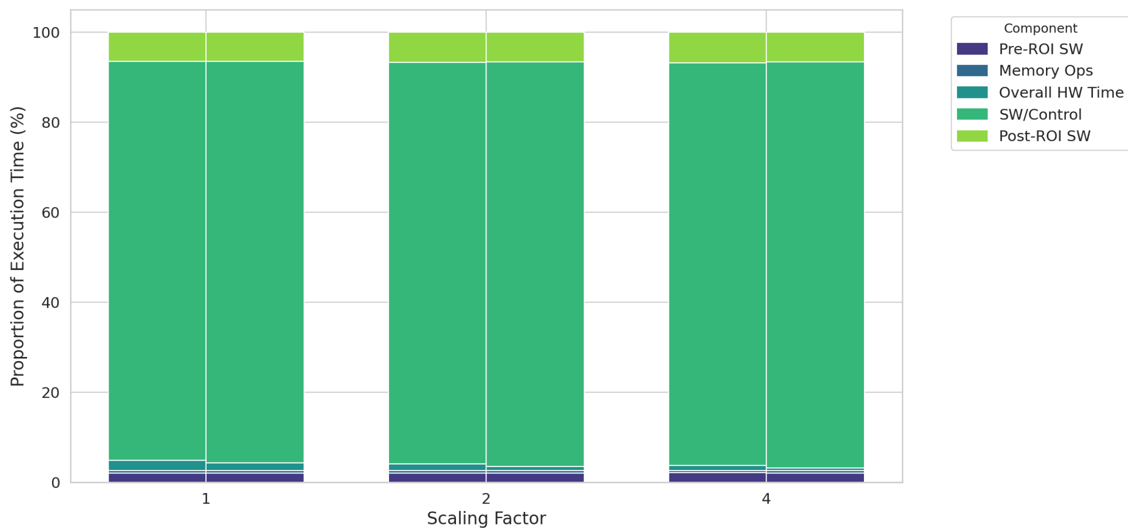
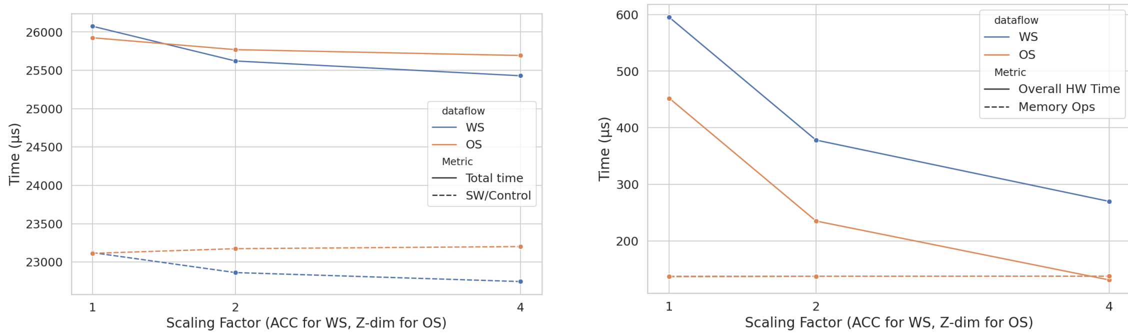


Figure 5.30: Relative latency breakdown for MHA (`seq_len` 128, `d_model` 512, `num_heads` 8) across varying compute tile scaling factors

The proportional breakdown in Figure 5.30 immediately reveals that the system is not compute-bound. Across all scaling factors and for both dataflows, the execution profile is still overwhelmingly dominated by the SW/Control overhead component. The Overall HW Time constitutes a very small fraction of the total execution time, suggesting that improvements in hardware speed will have minimal impact on the final performance.



(a) Total latency and SW/Control overhead (b) Memory Ops and Overall HW Time

Figure 5.31: Scaling of performance metrics for MHA (`seq_len` 128, `d_model` 512, `num_heads` 8) across varying compute tile scaling factors

The absolute scaling plots in Figure 5.31 confirm this hypothesis. Figure 5.31b shows that increasing the scaling factor does make the hardware faster. For both the WS and OS dataflows, the Overall HW Time decreases as compute resources are added, with the OS dataflow performing slightly better. This demonstrates that the architectural enhancements are functioning as intended at the hardware level.

However, Figure 5.31a reveals that these hardware gains fail to translate into meaningful system-level performance improvements. The Total time for both dataflows

is significantly higher and remains largely flat regardless of the scaling factor. The performance gains from faster hardware are completely nullified by the dominant and nearly constant software overheads, particularly the SW/Control overhead. An interesting minor trend is that for the WS dataflow, the SW/Control overhead actually decreases slightly with a larger accumulator, as this allows for larger hardware tiles and thus fewer software commands. In contrast, the OS dataflow's overhead remains flat. Nevertheless, this effect is minor. The system is fundamentally bottlenecked by the software's ability to orchestrate the many steps of the MHA layer, not by the NPU's raw compute power. This brings forth the conclusion that for workloads with frequent software interaction, increasing computational resources is an ineffective optimisation strategy if the control path overhead remains the primary performance limiter.

5.4.3 Heads vs. Head dimension trade-off

The final study for the MHA workload investigates a critical architectural trade-off in Transformer models: For a fixed model dimension (`d_model`=512), is it more efficient to have fewer attention heads with a large head dimension (`d_k`), or more heads with a smaller head dimension? This fundamentally changes the nature of the workload, transitioning from a few large matrix multiplications to a multitude of small ones, thereby placing maximum stress on the software control path.

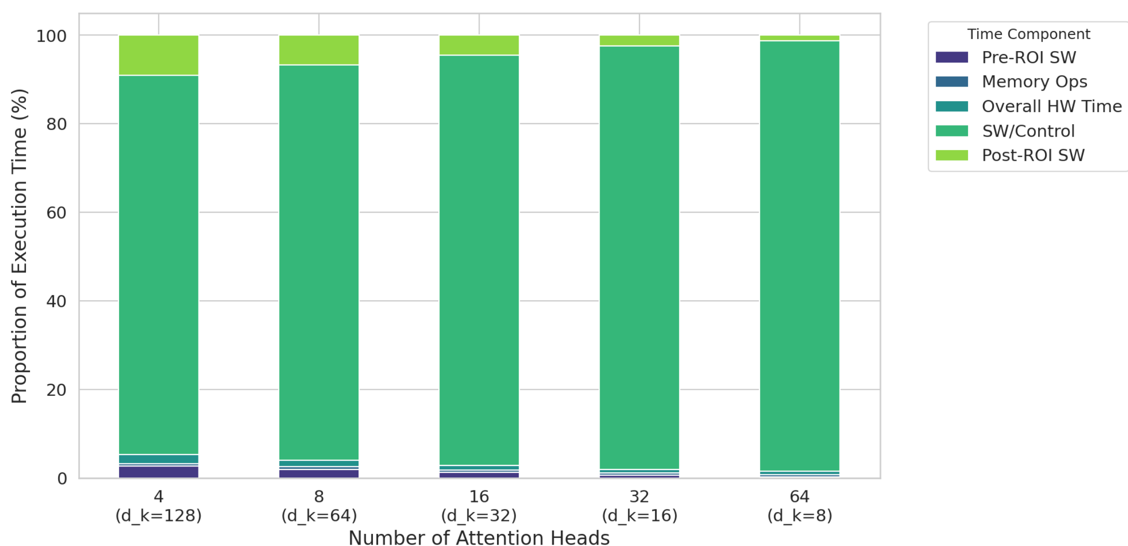
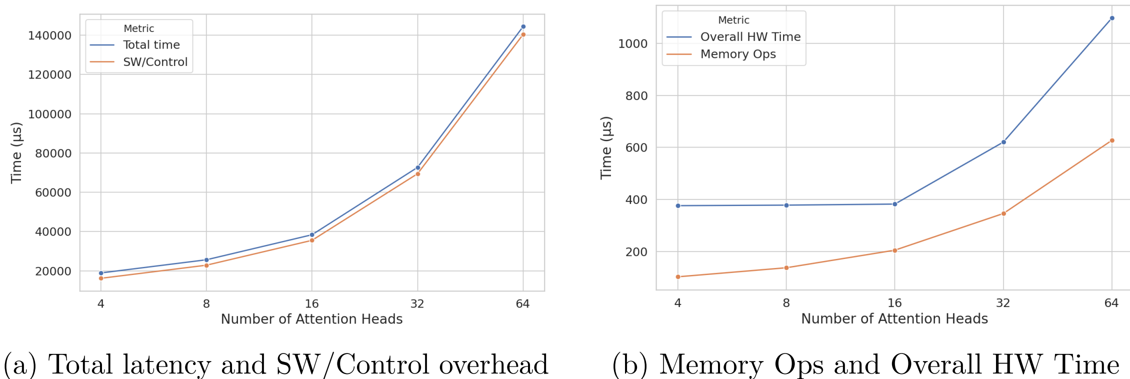


Figure 5.32: Relative latency breakdown for MHA (`seq_len` 128, `d_model` 512) for varying number of heads

The proportional breakdown in Figure 5.32 shows that across all configurations, the execution is still completely dominated by the SW/Control overhead. The actual hardware execution and memory operations constitute a tiny fraction of the total time. This immediately indicates that the primary performance driver in this trade-off is not the NPU's computational efficiency but rather the software stack's ability to manage the workload.



(a) Total latency and SW/Control overhead

(b) Memory Ops and Overall HW Time

Figure 5.33: Scaling of performance metrics for MHA (`seq_len` 128, `d_model` 512)for varying number of heads

The absolute scaling results in Figure 5.33 provide a clear and dramatic conclusion. As the number of heads increases, the Total time explodes, rising by around 7× when moving from 4 to 64 heads (Figure 5.33a). The SW/Control overhead scales in almost perfect lockstep, confirming that this performance degradation is almost entirely due to the software cost of managing the workload. Increasing the number of heads means the software must issue a proportionally larger number of `ioctl` calls and NPU commands to handle the multitude of smaller, independent GEMM operations, and this overhead becomes the overwhelming bottleneck.

More surprisingly, Figure 5.33b reveals that performance degrades at the hardware level as well. Both the Overall HW Time and Memory Ops Time increase with a larger number of heads. The increase in hardware time suggests that the NPU is less efficient at processing many small matrices compared to a few large ones, likely due to the overhead of filling the systolic array pipeline for each small operation. The increase in memory time is also significant, as a larger number of heads results in more intermediate matrices (Q, K, V, and attention scores) that must be written to and read from memory.

The conclusion from this study is unambiguous: for this system architecture, there is a severe performance penalty for increasing the number of attention heads. The trade-off is heavily skewed in favour of using fewer, larger heads, as this minimises the number of software interactions, improves hardware utilisation, and reduces memory traffic. This finding provides a powerful, quantitative insight into how a model's architectural choices can have a profound impact on system-level performance, an effect that could only be captured through a full-system simulation approach like ours.

6

Limitations and Future Work

The `gem5-fsnpu` framework provides a robust foundation for full-system NPU simulation, enabling detailed analysis of hardware-software interactions. However, the scope of this thesis necessitated several simplifying abstractions. This chapter discusses the current limitations of the model, which provide essential context for interpreting the results presented, and outlines promising avenues for future research.

6.1 Model Validation and Reproducibility

A key challenge in academic simulation work is validation against real-world hardware and ensuring reproducibility.

- Lack of Hardware Validation: The NPU model implemented in this work has not been validated against a physical hardware counterpart. Such validation is a non-trivial undertaking, presenting both practical and technical challenges. From a practical standpoint, the proprietary nature of commercial NPU designs severely limits the public availability of detailed architectural data needed for a direct comparison. From a technical perspective, accurately validating an accelerator with a tightly integrated memory and compute subsystem, like the NPU model presented, is inherently complex. The model's performance is deeply intertwined with the behaviour of its SPM, DMA engine, and the system's main memory. Accurately modelling these memory components and their interaction with the compute core is a significant challenge, as the memory subsystem is often a primary source of inaccuracy in full-system simulation [32].
  Consequently, while the performance trends and system-level interactions observed are qualitatively insightful, the absolute latency values may differ from those of a real-world system. The results should be interpreted as characterising the behaviour of the modelled architecture, rather than predicting the performance of a specific commercial product.
- Reproducibility: The custom software developed for this thesis, including the Linux driver and user-space library, is not publicly available at this time. This makes direct reproduction of the experimental results presented in this work challenging. However, it is important to note that the entire software stack was developed without the use of any proprietary data, ensuring the methodology itself is transparent and based on the principles described in this work

6.2 Architectural Abstractions and Performance Implications

The current NPU model represents a simplified accelerator architecture to maintain manageable simulation times and development scope. These abstractions have a direct impact on the performance results, particularly regarding software overhead.

- Simplified DMA and Memory Model: The DMA engine relies on a kernel-space "bounce buffer" for data transfers. This approach, while functional, requires `copy_from_user` and `copy_to_user` operations, which contribute to the measured software overhead. A more advanced "zero-copy" DMA mechanism, common in tightly-integrated accelerators, would eliminate this overhead.
- Lack of Latency Hiding Techniques: The model does not implement latency-hiding techniques such as double buffering. In a double-buffered system, data transfers for the next computation can be overlapped with the current computation, effectively hiding much of the DMA latency and reducing control path stalls. The absence of this feature in the model means that all data transfer and software management latencies are exposed sequentially, contributing significantly to the total execution time, especially in memory-constrained scenarios.
- Single-Core Architecture: The framework currently models a single NPU core. Modern systems often employ multi-core NPUs, allowing for parallel execution of tasks (e.g., processing different attention heads simultaneously). The sequential processing in our single-core model naturally amplifies the total software overhead for workloads like MHA, which are composed of many independent sub-tasks which incur repeated software overheads.

6.3 Interpretation of High Software Overhead

The DSE studies, particularly for the MHA workload, revealed cases where software and control overhead dominated the total execution time, reaching up to 90% of the total latency. It is crucial to contextualise this finding. This result is not a suggestion that NPUs are inherently inefficient, but rather a direct consequence of the aforementioned architectural limitations.

For a complex workload like MHA with many small, sequential operations, a simple architecture without latency-hiding or parallel execution capabilities will inevitably be bottlenecked by control path and data movement costs. Therefore, this finding should not be seen as a flaw in the model, but rather as a successful demonstration of the framework's primary goal, which was to provide a tool capable of revealing and quantifying the system-level bottlenecks that emerge from the interplay between a specific workload and a given hardware architecture. The results strongly suggest that for workloads like MHA, architectural features like double buffering and multi-core parallelism are not just optimisations, but essential requirements for achieving high performance.

6.4 Future Work

As discussed in the chapter up to this point, several simplifying abstractions were made in the NPU model, and numerous avenues for future research and development remain.

6.4.1 DMA optimisation

The current DMA engine implementation relies on a kernel-space "bounce buffer", a single contiguous 4 MiB memory region allocated by the driver. While functional, this approach introduces two limitations. First, it necessitates `copy_from_user` and `copy_to_user` operations, adding software overhead that would not exist in an ideal system. Second, it limits the maximum size of a single DMA transfer since a contiguous buffer of a larger size would be difficult to allocate. A significant future improvement would be to implement a true "zero-copy" DMA mechanism. This would involve enabling the NPU model to handle user-space virtual addresses directly. The NPU would need to traverse page tables to translate virtual to physical addresses, handle non-contiguous memory by breaking large transfers into page-sized chunks, and manage the associated Translation Lookaside Buffer (TLB) lookups. While this represents a substantial engineering effort, it would provide a more accurate model of modern, tightly-integrated accelerators.

6.4.2 Enhanced NPU Memory Hierarchy

The current NPU model features a single-level on-chip memory, a software-managed scratchpad, analogous to an L1 data cache. Modern NPUs often feature deeper memory hierarchies, including a larger L2 cache, to further reduce traffic to main memory. Future work could involve designing and integrating an L2 cache into the NPU. This would introduce new concerns regarding its management as it would bring up the question of whether it should be explicitly managed by software, similar to the scratchpad, or if it should be a traditional, coherent hardware cache.

Furthermore, a common technique for hiding memory latency, double buffering, was not implemented. Adding support for double buffering would allow the NPU to compute on one set of data while the DMA engine simultaneously fetches the next set, improving hardware utilisation, especially in memory-constrained scenarios.

6.4.3 NPU core improvements

At present, the systolic array is modelled using a transaction-level cycle-accurate approach, which assumes a regular, homogenous array structure and provides a deterministic latency for any given tile computation. While this strikes a balance between simulation speed and accuracy, a more granular, fully cycle-accurate model could be developed. Such a model, while slower, would enable the study of more flexible NPU architectures, such as those with non-square dimensions or heterogeneous Processing Elements. Additionally, the trade-off introduced between the improved accuracy and increased simulation time would also be an interesting study.

6.4.4 Multi-Core NPU and Multi-Tenancy Exploration

The framework currently models a single NPU core. A natural and highly relevant extension is to support multi-core NPU systems. This could be explored in two primary ways:

1. Multiple Independent NPUs: Instantiating multiple NPU devices (`NPUdma`+`SysArray`) on the system bus, each managed independently.
2. A Unified Multi-Core NPU: Modifying the `NPUdma` object to manage multiple `SysArray` compute cores, potentially sharing a unified on-chip memory hierarchy.

Both approaches would open up critical research questions in workload scheduling, data distribution, and inter-core communication. This extension would also enable the study of multi-tenancy, where the NPU resources are partitioned and shared among multiple applications, a key challenge in datacenter environments.

6.4.5 Hardware Validation, Power, and Area Modelling

This work focuses primarily on performance modelling. A comprehensive evaluation of an accelerator also requires analysis of its power consumption and physical area. Future work could integrate the `gem5-fsnpu` framework with established power and area estimation tools (such as McPAT or CACTI). The configurable parameters of the NPU could be used as inputs to these tools, providing holistic performance-per-watt and performance-per-area metrics.

Finally, validating the accuracy of the simulation model against real hardware is a significant challenge, primarily due to the proprietary nature of commercial NPU designs and the limited public data available. As more hardware details become accessible, validating and calibrating the model against a physical reference would be a crucial step in enhancing its fidelity.

6.4.6 ONNX Runtime Integration

To broaden the framework's applicability and enable the execution of a wide range of standard neural network models, a crucial future step is integration with the ONNX (Open Neural Network Exchange) Runtime. The standard mechanism for linking custom hardware to this runtime is through a dedicated Execution Provider (EP). However, this path was not pursued in the current work due to the significant trade-offs between implementation complexity and meaningful results. There are two primary strategies for creating such an EP:

1. Direct Operator Mapping: The most straightforward approach is a one-to-one mapping where the EP intercepts each supported operator in an ONNX graph (e.g., MatMul, Add) and delegates it to a corresponding function call in the `gem5-fsnpu` user-space library. While this is simpler to implement, each operator call would incur the full system-level overhead, and more importantly, it would fail to exploit crucial hardware optimisations, making it highly inefficient.

The performance would be dominated by software overhead rather than the NPU's capabilities.

2. Graph Fusion: A far more effective, but significantly more complex, solution involves implementing graph fusion. In this advanced approach, the EP analyses the model's entire dataflow graph to find specific patterns or sub-graphs that the NPU can execute as a single, optimised unit. For example, a sequence of GEMM and activation functions could be "fused" into one kernel call, allowing the software to intelligently schedule all necessary data into the scratchpad at once, maximising data reuse and minimising DMA traffic.

Given these options, the direct mapping approach was deemed a large engineering effort that would yield suboptimal performance results. The more powerful graph fusion approach requires building a compiler-like layer for pattern matching and optimisation, which was a substantial project in its own right and thus fell beyond the scope of this thesis.

7
Conclusion

This final chapter synthesises the findings and concludes the thesis. The first section provides a discussion that consolidates the key insights from the diverse DSE studies, reinforcing the core argument that NPU performance is an emergent property of the entire system. The second section presents the conclusion, summarising the primary contributions of this work and contextualising its importance in the field of AI accelerator research.

7.1 Discussion

Across the diverse set of DSE studies, a clear and consistent narrative emerges: NPU performance is not a simple function of its hardware specifications but an emergent property of the deep interplay between the hardware architecture, the software stack, and the workload's intrinsic characteristics. The results demonstrate that hardware parameters cannot be optimised in isolation. For instance, the performance gains from a larger systolic array are significantly blunted by an inadequately sized SPM, which forces the software into an inefficient, memory-bound tiling strategy. Conversely, for workloads already bottlenecked by software, like MHA, even a generously provisioned SPM cannot overcome the fundamental latency of the control path. This highlights the dual nature of the software stack: its intelligent tiling algorithm is critical for maximising hardware utilisation, yet its inherent overhead in managing operations can become the primary performance limiter. This effect was most powerfully illustrated by the MHA studies, where the workload's structure, comprising many small, sequential operations, amplified the SW/Control overhead to the point of dominating the entire execution time. The "Heads vs. Head dimension" trade-off, in particular, showed how a pure software-level choice in the AI model's architecture can dramatically alter performance on identical hardware, proving that the path to efficient computation lies in a holistic co-design methodology that treats the hardware, software, and workload as inseparable components of a single system.

7.2 Conclusion

The primary objective of this thesis was to address a critical gap in AI accelerator research: the lack of a tool capable of simulating a detailed NPU architecture within a full-system context. We successfully addressed this by developing `gem5-fsnpu`,

a framework integrating a configurable, transaction-level cycle-accurate NPU into the gem5 simulator, complete with a vertically-integrated software stack to model realistic system overheads.

Through a series of comprehensive Design Space Exploration studies, we demonstrated the framework's ability to quantify system-level effects which are often missed by component-level simulators, proving that these effects are not minor considerations but are frequently the primary determinants of real-world performance. A key finding was that the interplay between the software stack and the hardware architecture can lead to situations where software management overhead, rather than hardware execution, becomes the primary performance bottleneck. For command-intensive workloads like Multi-Headed Attention (MHA), the model revealed that the latency of the control path could dominate the total execution time.

This insight underscores that for certain workloads, hardware-level optimisations alone are insufficient, highlighting the critical need for a holistic, full-system approach to performance analysis. Furthermore, our analysis quantified critical hardware-software trade-offs. For instance, the MHA layer's architectural choice between fewer, larger attention heads versus more, smaller heads resulted in a 7-fold increase in total latency for the latter, a penalty driven almost entirely by software management costs. Similarly, we identified the existence of a "sweet spot" for on-chip memory size, as increasing the SPM size for a GEMM workload from 32 KiB to 128 KiB yielded a 50% performance improvement, after which returns diminished sharply.

Ultimately, this work concludes that NPU performance cannot be understood, let alone optimised, by analysing the accelerator in isolation. Our findings show that a component-level analysis is not merely incomplete but can be actively misleading, as it ignores the software overheads that are often the primary performance limiters. The complex interplay between the hardware's capabilities, the software's efficiency, and the workload's structure dictates the final performance. The `gem5-fsnpu` framework provides a robust platform for modelling these interactions, reinforcing the principle that a holistic, hardware-software co-design methodology is essential for engineering the next generation of efficient AI systems.

Bibliography

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: https://doi.org/10.1145/2024716.2024718.

[2] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, *et al.*, *The gem5 simulator: Version 20.0+*, 2020. arXiv: 2007.03152 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2007.03152.

[3] K. Hu and A. Tong, *Openai unveils personalized ai apps as it seeks to expand its chatgpt consumer business*, [Online; accessed 26-July-2025], 2023. [Online]. Available: https://www.reuters.com/technology/openai-enables-customized-gpt-bots-offers-cheaper-more-powerful-models-2023-11-06/.

[4] Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, 1982. DOI: 10.1109/MC.1982.1653825.

[5] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017, ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. [Online]. Available: https://doi.org/10.1145/3140659.3080246.

[6] J.-W. Jang, S. Lee, D. Kim, H. Park, *et al.*, "Sparsity-aware and re-configurable npu architecture for samsung flagship mobile soc," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 15–28. DOI: 10.1109/ISCA52012.2021.00011.

[7] A. Frumusanu, *Apple announces m1 pro & m1 max: Giant new arm socs with all-out performance*, [Online; accessed 26-July-2025], 2021. [Online]. Available: https://www.anandtech.com/show/17019/apple-announced-m1-pro-m1-max-giant-new-socs-with-allout-performance.

[8] N. Jouppi, G. Kurian, S. Li, *et al.*, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 9798400700958. DOI: 10.1145/3579371.3589350. [Online]. Available: https://doi.org/10.1145/3579371.3589350.

[9] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, *SCALE-Sim: Systolic CNN Accelerator Simulator*, arXiv:1811.02883 [cs], Feb. 2019. DOI: 10.48550/arXiv.1811.02883. [Online]. Available: http://arxiv.org/abs/1811.02883 (visited on 05/08/2025).

[10] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Aug. 2020, pp. 58–68. DOI: 10.1109/ISPASS48437.2020.00016. [Online]. Available: https://ieeexplore.ieee.org/document/9238602 (visited on 05/11/2025).

[11] R. Raj, S. Banerjee, N. Chandra, *et al.*, *SCALE-Sim v3: A modular cycle-accurate systolic accelerator simulator for end-to-end system analysis*, arXiv:2504.15377 [cs], May 2025. DOI: 10.48550/arXiv.2504.15377. [Online]. Available: http://arxiv.org/abs/2504.15377 (visited on 06/02/2025).

[12] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio, and T. Krishna, "STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, Nov. 2021, pp. 201–213. DOI: 10.1109/IISWC53511.2021.00028. [Online]. Available: https://ieeexplore.ieee.org/document/9668279 (visited on 05/08/2025).

[13] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783751. [Online]. Available: https://ieeexplore.ieee.org/document/7783751 (visited on 05/15/2025).

[14] J. Vieira, N. Roma, G. Falcao, and P. Tomás, "Gem5-accel: A Pre-RTL Simulation Toolchain for Accelerator Architecture Validation," *IEEE Computer Architecture Letters*, vol. 23, no. 1, pp. 1–4, Jan. 2024, ISSN: 1556-6064. DOI: 10.1109/LCA.2023.3329443. [Online]. Available: https://ieeexplore.ieee.org/document/10304264 (visited on 05/11/2025).

[15] H. Ham, W. Yang, Y. Shin, *et al.*, *ONNXim: A Fast, Cycle-level Multi-core NPU Simulator*, arXiv:2406.08051 [cs], Jun. 2024. DOI: 10.48550/arXiv.2406.08051. [Online]. Available: http://arxiv.org/abs/2406.08051 (visited on 05/08/2025).

[16] A. Parashar, P. Raina, Y. S. Shao, *et al.*, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2019, pp. 304–315. DOI: 10.1109/ISPASS.2019.00042. [Online]. Available: https://ieeexplore.ieee.org/document/8695666 (visited on 05/08/2025).

[17] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, May 2008, ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. [Online]. Available: https://doi.org/10.1145/1356052.1356053.

[18] T. M. Smith, R. v. d. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1049–1059. DOI: 10.1109/IPDPS.2014.110.

[19] J. M. Joseph, A. Samajdar, L. Zhu, *et al.*, *Architecture, dataflow and physical design implications of 3d-ics for dnn-accelerators*, 2021. arXiv: 2012.12563 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2012.12563.

[20] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017. DOI: 10.1109/JSSC.2016.2616357.

[21] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, *Understanding reuse, performance, and hardware cost of dnn dataflows: A data-centric approach using maestro*, 2020. arXiv: 1805.02566 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1805.02566.

[22] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, “Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings,” *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020. DOI: 10.1109/MM.2020.2985963.

[23] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, “Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators,” *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021. DOI: 10.1109/TC.2021.3059962.

[24] A. C. Yüzügüler, C. Sönmez, M. Drumond, Y. Oh, B. Falsafi, and P. Frossard, “Scale-out systolic arrays,” *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, Mar. 2023, ISSN: 1544-3566. DOI: 10.1145/3572917. [Online]. Available: https://doi.org/10.1145/3572917.

[25] A. Shafiee, A. Nag, N. Muralimanohar, *et al.*, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12.

[26] B. Hyun, Y. Kwon, Y. Choi, J. Kim, *et al.*, “Neummu: Architectural support for efficient address translations in neural processing units,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1109–1124, ISBN: 9781450371025. DOI: 10.1145/3373376.3378494. [Online]. Available: https://doi.org/10.1145/3373376.3378494.

[27] Y. Xue, Y. Liu, L. Nai, and J. Huang, “V10: Hardware-assisted npu multi-tenancy for improved resource utilization and fairness,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA ’23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 9798400700958. DOI: 10.1145/3579371.3589059. [Online]. Available: https://doi.org/10.1145/3579371.3589059.

[28] H. Genc, S. Kim, A. Amid, *et al.*, “Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration,” in *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference*, ser. DAC ’21, San Francisco, California, United States: IEEE Press, May 2022, pp. 769–774, ISBN: 978-1-66543-274-0. DOI: 10.1109/DAC18074.2021.9586216. [Online]. Available: https://dl.acm.org/doi/10.1109/DAC18074.2021.9586216 (visited on 05/08/2025).

[29] S. L. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, “SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning

Workloads," en, *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 1–26, Dec. 2020, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3424669. [Online]. Available: https://dl.acm.org/doi/10.1145/3424669 (visited on 05/08/2025).

[30] Y.-C. Lee, T.-S. Hsu, C.-T. Chen, J.-J. Liou, and J.-M. Lu, "NNSim: A Fast and Accurate SystemC/TLM Simulator for Deep Convolutional Neural Network Accelerators," in *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, ISSN: 2472-9124, Apr. 2019, pp. 1–4. DOI: 10.1109/VLSI-DAT.2019.8741950. [Online]. Available: https://ieeexplore.ieee.org/document/8741950 (visited on 05/15/2025).

[31] S. Hwang, S. Lee, J. Kim, H. Kim, and J. Huh, "mNPUsim: Evaluating the Effect of Sharing Resources in Multi-core NPUs," in *2023 IEEE International Symposium on Workload Characterization (IISWC)*, ISSN: 2835-2238, Oct. 2023, pp. 167–179. DOI: 10.1109/IISWC59245.2023.00018. [Online]. Available: https://ieeexplore.ieee.org/document/10289501 (visited on 05/08/2025).

[32] K. Pathak, J. Klein, G. Ansaloni, *et al.*, "Towards accurate risc-v full system simulation via component-level calibration," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 4, Jul. 2025, ISSN: 1539-9087. DOI: 10.1145/3737876. [Online]. Available: https://doi.org/10.1145/3737876.