Final Report

Project #43: Reconfigurable Neural Processing Unit (NPU) for Energy-Efficient AI at the Edge

Kristelle Sampang

Project Report

Project Partner:	Pratham Chhabra
Supervisor:	Dr Morteza Biglari-Abhari
Co-Supervisor:	Dr Maryam Hemmati

18th October 2025

Waipapa Taumata Rau University of Auckland

COMPSYS043-2025

RECONFIGURABLE NEURAL PROCESSING UNIT (NPU) FOR ENERGY-EFFICIENT AI AT THE EDGE

Kristelle Sampang

ABSTRACT

Abstract goes here.

DECLARATION

Student I hereby declare that:

1. This report is the result of the final year project work carried out by my project partner (see cover page) and I under the guidance of our supervisor (see cover page) in the 2025 academic year at the Department of Electrical, Computer and Software Engineering, Faculty of Engineering, University of Auckland.
2. This report is not the outcome of work done previously.
3. This report is not the outcome of work done in collaboration, except that with a potential project sponsor (if any) as stated in the text.
4. This report is not the same as any report, thesis, conference article or journal paper, or any other publication or unpublished work in any format.

newc In the case of a continuing project, please state clearly what has been developed during the project and what was available from previous year(s):

Signature: [illegible]

Date: 19/10/2025

Table of Contents

Acknowledgements

Thank important people here.

Glossary of Terms

Term	Definition

Abbreviations

AOA	**Angle of attack**

1. Introduction

1.1 Motivation

- As the demand for artificial intelligence grows to become prominent in today's society, its energy consumption has become a key issue.
- The processing required to run machine learning models is large, with millions of computations needed to be executed.
- This means that low-power processing devices at edge are limited to its use.
- The processing power to run machine learning models is large, limiting its use for low-power processing devices at the edge.
- With high computational power required, energy-efficiency is a key concern, especially nowadays when energy is an essential resource to not waste.
- In the past decade, common types of processors for running AI are CPU, GPU, and FPGA.
- However, as technology continues to improve, Neural Processing Units (NPU) are developed. These are processors that specialise in processing AI computations.
- By integrating NPU alongside other processors, the performance and energy-efficiency are improved compared to a standalone processor.

1.2 Problem Statement

1.3 Report Structure

The remainder of this report goes as follows: Section X covers Y....

2. Background

The increasingly growth of Artificial Intelligence (AI) in the past decade has driven significant advancements across numerous field. Machine Learning (ML) models, particularly Deep Neural Networks (DNNs), are popular for its applications from image classification to autonomous driving [1]. There has been significant shift from AI inferencing occurring at a cloud-level to resource-constrained edge devices. This move motivates the *AI at the Edge* portion in the research, where lower latency, enhanced data privacy, and real-time processing capabilities must are requirements that must be met without relying on constant network connection [2].

However, this shift presents an alarming issue where DNNs are computationally intensive and power-hungry, whilst edge devices operate under strict power and resource limitations. To bridge this gap, hardware accelerators, such as Neural Processing Units (NPUs), are introduced to execute AI algorithms at faster rates than general-purpose CPUs [3]. This section provides necessary background on the core technologies that support this project, starting with the most popular model for image-based tasks: the Convolutional Neural Network.

2.1 Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are a prominent type of DNN model, mainly utilised for image processing tasks like object recognition, classification, and detection. The network is composed of four main layers: convolutional, activation, pooling, and fully connected layers; this research focuses on the convolutional and activation layers. The convolutional layer is where the majority of the computations occur [4], as it extracts features from an image and converts them into numerical values. In a convolutional layer, there are several filters that slide through the image, searching for a specific pattern. These filters are typically of size 3×3 or 5×5, and are applied to the image by multiplying the filter by the 2D pixel representation of the image. Mathematically, this operation can be represented as:

$$O[h][w][c] = \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} \sum_{k=1}^{i_c} I[h + i \cdot s_h][w + j \cdot s_w][k] \times F[i][j][k][c]$$

where I, O, F are input activation, output activation, and filter weights respectively [4]. This can be represented as an enormous number of Multiply-Accumulate (MAC) operations, making CNNs computationally intensive.

An activation layer determines whether a neuron should be activated based on its input. Its primary role is to introduce non-linearity into the network. Without this, a neural network can only learn simple, linear patterns. Non-linearity allows the network to execute complex tasks, such as learning complicated patterns. A commonly used function is the Rectified Linear Unit (ReLU). The main functionality is to allow for positive inputs to remain unchanged whist setting any negative input to zero [1]. Mathematically, this can be represented as:

$$\text{ReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

A critical consequence of this is that it introduces significant sparsity as approximately half of the elements are zero [5]. This sparsity is a key property that this project exploits to improve energy-efficiency, and will be discussed further in Section 3.1.1.

Although not the core focus of this project, the pooling layers are used to reduce the spatial dimensions of the input whilst retaining the most relevant features. This helps to decrease computational load and control overfitting. Furthermore, a fully connected layer is typically used at the end of the CNN to perform high-level reasoning based on the features extracted by previous layers[6].

2.2 Hardware Acceleration for Machine Learning

General-purpose CPUs alone are not powerful enough to handle the computational requirements of modern deep learning models [7]. This puts additionally strain on edge devices, which are typically resource-constrained and have limited power budgets. This means that specialised hardware accelerators are needed to efficiently execute AI algorithms [3]. Typically, multiple processes exist in parallel, such as Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Neural Processing Units (NPUs), and are integrated with the CPU as a heterogeneous system to enhance the performance of AI applications [3]. Several studies have compared these hardware accelerators for AI applications, summarised as follows:

- A study discovered that **GPUs** highly accelerate computationally intensive tasks with parallelism at ease but with a trade-off of high power consumption [8]. Furthermore, GPUs are not typically found in edge devices, such as smartphones and IoT devices, due to its high power consumption and thermal output.
- **NPUs** are a type of specialised hardware accelerator, best at executing AI-based algorithms. Due to its dedicated use, it has extreme performance and low-energy efficiency, however, it is limited in its flexibility.
- **FPGAs** can achieve energy-efficiency by consuming half the power of a GPU [9] by tailoring the hardware architecture to a specific task.

The customisability of FPGAs allows for optimised dataflow and memory access patterns, crucial for improving the performance of deep learning models. This provides a platform to design and implement specialised NPUs that can efficiently handle the unique demands of AI workloads. Therefore, this project chooses to implement the NPU on an FPGA to create a reconfigurable hardware accelerator.

2.3 Systolic Array Architecture

A systolic array is a structured network of interconnected processors known as Processing Elements (PE) that perform computations in a rhythmic and synchronised manner [10]. Each PE receives data, processes it, and passes it to the next PE in a pipelined fashion [5], allowing for high levels of parallelism and efficient data flow. This structure allows processes to be completed at higher speeds, with computations timed by a global clock for predictable data movement and faster memory access times. Additionally, systolic arrays are highly compact together and allow simple data control flow. As its architecture is an array, it can easily be scaled for larger data sets, perfect for ML, and the predictable structure makes it easier to design and optimise algorithms. Furthermore, since PEs are densely packed together, data locality is improved, reducing data transmission and lowering energy consumption. Energy-efficiency is a critical consideration when designing hardware accelerators for edge devices, where power resources are limited. Efficient energy usage not only extends battery life but produces less heat, important for maintaining device performance and lifespan. .

Although it offers many advantages, there are some disadvantages to systolic arrays. These include being difficult and costly to build, as well as being specialized and inflexible in the problems it can solve, limiting its versatility. Because systolic arrays offer high throughput and efficiency, it is commonly used in NPUs to accelerate matrix multiplication [11], the core operation in a CNN's convolutional layer. However, its rigid and structured dataflow is inefficient for sparse matrices [5], [12], leading to imbalanced workloads and low PE utilisation. This limitation motivates the need for a reconfigurable systolic array that can adapt to the sparsity in neural networks, becoming the focus of this project.

3. Literature Review

This section delves deep into the novelties and gaps in the current research on handling sparsity in neural networks, particularly in the context of systolic arrays. It explores the challenges posed by sparsity and reviews existing hardware architectures designed to mitigate these issues, leading to the motivation for this project.

3.1 Sparsity in Neural Networks

As discussed in Section 2.1, sparsity is a property of matrices that arise from many sources, such as activation functions and model compression techniques. This section explores these in detail to find the root cause of sparsity in neural networks.

3.1.1 Rectified Linear Unit (ReLU)

To optimise the performance of a neural network, gradients are calculated to update the network's weights. However, this introduces the Vanishing Gradient Problem (VGP) where gradients become increasingly miniscule when backpropagated from the output to earlier layers [13]. The consequence of VGP presents slow convergence and impaired learning. This issue becomes even more problematic when structures have many layers and a solution to this is applying ReLU. As previously mentioned in Section 2.1, the outcome of ReLU is that all negative values are set to zero, introducing significant sparsity in the activation matrices. However, since ReLU is most commonly applied [5], it is difficult to avoid using a model that does not produce sparse matrices due to ReLU in real-case scenarios.

3.1.2 Pruning

Model Pruning is a technique used to remove redundant data that may not significantly contribute to the final prediction. Kim et al. [14] found that more than half of weights in convolutional layers can be set to zero without impacting the overall accuracy. To decide what weights to prune, the sensitivity of the weight is considered and how it influences the output. Gorvadiya et al. [15] discovered that their Weight Pruning technique resulted in minimal accuracy loss of 1-2% whilst saving 25% in power consumption. These results suggests that pruning is an effective method to reduce model size and computation without sacrificing performance.

Whilst both ReLU and pruning work together to help accelerate the efficiency of the neural network by introducing sparsity in the system, this presents a challenge for systolic array based hardware architecture. As the architecture of systolic arrays is rigid and structured, it is advantageous for dense matrices with regular data access patterns. However, with the introduction of irregularity due to sparsity, this leads to inefficient memory access patterns and imbalanced workloads, causing low PE utilisation [12]. Additionally, with the high volume of computations required to run a CNN, the impact of the inefficiency is magnified. This means that handling sparsity is a critical issue that must be solved to fully exploit the benefits of sparsity in neural networks. Balancing the benefits and drawbacks of how sparsity is handled in systolic arrays is explored in the following section.

3.2 Hardware Architectures for Sparsity

This section explores various hardware techniques that have been discovered to handle sparsity in neural networks. The main methods are power-gating and zero-skipping, compressed data formats, and specialised dataflows and architectures. Each method has its own advantages and disadvantages, discussed in detail below.

3.2.1 Power Gating and Zero-Skipping

As multiplying and accumulating with zero does not change the overall result, it is redundant to execute operations with zero. Thus, a method like power-gating is introduced to skip these unnecessary operations. The most straight forward approach to handling sparsity in hardware is power-gating. This method skips the MAC operation if one of the operands is zero, dynamically saving power [1]. However, this method requires additional hardware to check if an operand is zero, adding complexity to the design. Additionally, this does not reduce the latency as the PE has to wait for the data to be fetched from memory for the systolic pipeline to advance at the same rate.

A more advanced approach that reduces latency is zero-skipping. This technique involves only processing non-zero values and skipping over the zeros entirely. On a smaller scale, this can be applied to individiual weights to reduce the number of clock cycles [16]. On a larger scale, in the architecture by Kim et al. [14], the accelerator was able to skip 128x128 square blocks if structured sparsity existed, leading to a significant reduction in latency. However, this requires high levels of structured sparsity, that may not always occur in real-case scenarios.

3.2.2 Compressed Data Formats

To further improve performance, many architectures use compressed data formats to reduce the significant energy and latency costs of memory access. This is often implemented as a software-hardware co-design where a pre-processing stage reorganises the sparse matrix before it is sent to the accelerator. For example, He et al. [12] proposes a packing technique to confense the matrix offline then loads it into the systolic array. Similarly, Seo and Kong [11] uses a pre-processing column-wise condensation of the weights to remove zero values in advanced. Additionally, Kim et al. [14] and Palacios et al. [17] found that with structured pruning, where entire rows or columns of weights are pruned, the hardware can be optimised to skip entire rows or columns of computations. This is more efficient than unstructured pruning, where individual weights are pruned, as it maintains the regularity of the data access patterns.

Although these methods significantly reduce memory access costs and improve performance, there are trade-offs. As these methods change the structure of the input matrices, it requires complex hardware and control logic to manage the new dataflow. Additionally, post-processing is required to rearrange the result matrix is required to ensure that the indices match up the original input Seo and Kong [11]. This added complexity may lead to increased power consumption and use of hardware resources, potentially nullifying the benefits of handling sparsity. Therefore, whilst compressed data formats are effective, there are significant design challenges that must be carefully considered to fully exploit its advantages.

3.2.3 Specialised Dataflows and Architectures

Advanced architectures redesign the entire dataflow and control logic to efficiently handle sparsity in systolic array based hardware. These architecture offer higher performance but at the cost of significant hardware complexity. This section focuses on two key architectures, Sparse-TPU [12] and SCNN [1], to illustrate the current state-of-the-art and identify the research gap that this project aims to fill.

- **Sparse-TPU**: This architecture implements a column-packing algorithm to reorganise the sparse weight matrix into a denser format. It does this by examining each column and pushing non-zero elements to the left, creating a new row if a collision occurs. A collision is considered when two non-zero elements exist in the same row after pushing. This effectively reduces the number of columns, especially with non-structured sparsity, leading to lower memory access costs. However, this method requires complex pre-processing and alters the input data structures, complicating the hardware dataflow. It is critical that the rearrangement is tracked to ensure that the MAC operations are mathematically correct [12].
- **SCNN**: This architecture uses a different approach to handle sparsity. It operates on a compressed-sparse dataflow, where only non-zero weights and activations are fetched from memory and delivered to the multiplier array. This dataflow tracks the output coordinate for each multiplication and sends the resulting product to a specialised scatter accumulator, ensuring the value is summed at the correct location in the output feature map. This approach maximises data reuse but diverges significantly from standard systolic array designs [1].

The examples of Sparse-TPU and SCNN demonstrate a complex yet powerful solutions for unstructured, fine-grained sparsity. However, they require significant redesign of the core dataflow and control logic, leading to increased hardware complexity. As Palacios et al. [17] suggests, fine-grained sparsity requires a fair amount of control logic overhead. In contrast, structured sparsity, where entire rows or columns of weights are zero, is easier to exploit in hardware. This project aims to fill the gap for a simpler that can exploit structured sparsity without overhauling the classic systolic array pipeline.

4. Design and Methodology

4.1 System Architecture Overview

4.2 Data Generation and Pre-processing

This is section covers the software side of the co-design, where Figure X demonstrates a graphical overview. It utilises AlexNet to generate and extract realistic weight and activation matcies, then pre-processes it into tiles suitable for hardware testing. The processed tiles are then saved as a mif file and stored into the FPGA's ROMs for static testing. This section is crucial as it ensures that the hardware is tested with real-world data, validating its effectiveness in handling sparsity. Additionally, this section was implemented in a straight forward manner, meaning there were no major iterations or challenges faced.

4.2.1 AlexNet Model and Data Extraction

The AlexNet model was chosen for its popularity in image classification task and balance between complexity and size. It was commonly found in literature as the model of choice for edge devices due to its efficiency [1], [5]. A pre-trained model from the PyTorch library was used, as training is out of the scope of this project. The weights and activations were extracted from the first convolutional layer, as it has the largest matrices and highest sparsity due to ReLU.

This was quantised to INT8 as edge devices are resource-constrained and require low memory usage. In comparison to other data types such as FP32 and INT16, INT8 provides a good balance between accuracy, memory usage, and energy-efficieny [15]. The model was tested with a sample image to extract the activation matrix. The image used for testing is a photo of a cat (shown in Figure 1), as it is a common test image for image classification tasks and has a good variety of features for the convolutional layer to extract. Additionally, the image has been resized to 256x256 pixels to match the input size of the AlexNet model.

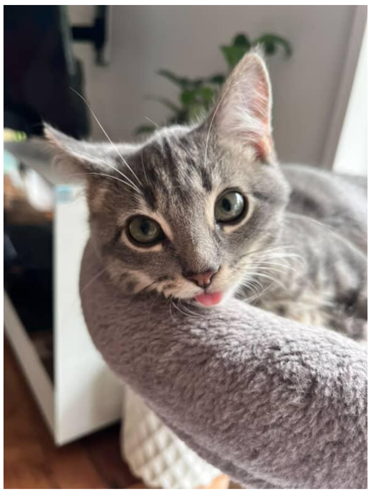


Figure 1 Image of a cat used for testing the AlexNet model

4.2.2 Tiling and File Generation

Once the weight and activation matrices were extracted from the model, it was found that its size were (64, 363) and (3025, 576) respectively. These sizes are much larger than what the systolic array can handle, making it necessary to tile the matrices into smaller sub-matrices. Literature from Palacios et al. [17] and Parashar et al. [1] support this approach, as it improves data locality and reduces memory access costs. Additionally, sizes were not perfectly divisible by 8, leading to incomplete tiles. To address this, zero-padding was applied to the matrices to ensure that they could be evenly divided into 8x8 tiles. This involved adding rows and columns of zeros to the bottom and right of the matrices until their dimensions were multiples of 8. This ensured that all tiles were complete and could be processed by the systolic array without any issues. Furthermore, by tiling the matrices, it allows for easier identification of features that look like structured sparsity that may not be apparent when looking at the entire matrix. This means that when looking at the layer as a whole, it may not be obvious that there are entire rows or columns of zeros. However, when the layer is broken down into smaller tiles, it becomes easier to see these patterns. This is important as it allows for more effective pre-processing and optimisation of the data for the hardware design.

These are saved as .mif files, which can be directly loaded into the FPGA's ROMs for static testing. This allows for easy verification of the hardware design with real-world data. It is important to note that only one tile is currently loaded into the ROMs at a time, meaning that the testing is static and not real-time. However, a master data and master weight file does exist to load different tiles into the ROMs, but this is not currently implemented in the hardware design.

Overall, by pre-processing the data in this manner, it ensures that the hardware tested is relevant and compatible with the hardware design detailed in the following sections. This validates the effectiveness of the approach in handling sparsity present in CNN models. The code for this software component can be found in the compendium with the filename `pipelined_v2.py`.

4.3 Baseline Systolic Array Architecture

The core of the hardware accelerator is an 8x8 systolic array, designed to perform matrix multiplication for convolutional layers. The fundamental building block of this array is the Processing Element (PE), a custom-designed Multiply-Accumulate (MAC) unit. The entire hardware accelerator was designed in VHDL.

4.3.1 Processing Elements and Systolic Array Formation

The fundamental process that the PE executes is the MAC operation, the core computation in convolutional layers. This can be represented in a Register-Transfer Level (RTL) diagram in Figure 2. The PE is designed to handle INT8 data types, as discussed in Section 4.2.1, to balance accuracy and resource usage. The PE takes two inputs: a weight from the filter matrix and an activation from the input feature map, both as INT8 data types. These inputs are multiplied together and adds the result to a 32-bit accumulated register that holds the partial sum for the output feature map. As each PE is independent from one another and operates concurrently, it allows for parallel processing of multiple MAC operations. Together, 64 PEs are arranged in an 8x8 grid to form the systolic array, as shown in Figure 3.

Each PE communicates with its neighbours to pass data along the array, enabling a continuous flow of data through the system. The dataflow chosen for this design is output-stationary, where the output partial sums remain in the PE until the final result is computed [5]. This approach minimises data movement and maximises data locality, reducing memory access costs. The PE is designed to operate in a pipelined manner, allowing it to accept new inputs every clock cycle while still processing previous inputs. This pipelining is crucial for maintaining high throughput in the systolic array.

The bus communication between each PE uses the `generate` statement in VHDL to create a scalable and modular design. This allows for easy expansion or modification of the array size in the future. The systolic array is integrated into a top-level module alongside the control unit (detailed in Section 4.3.2). For the purpose of this project, the systolic array is designed to handle up to 8x8 tiles, as discussed in Section 4.2.2. This size was chosen as it provides a good balance between hardware resource usage and computational capability, making it suitable for edge devices with limited resources.

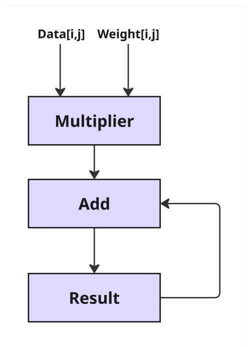


Figure 2 Image of the MAC unit in the Processing Element (PE)

4.3.2 Control Unit

The control unit is the mastermind of the systolic array operation, orchestrating the flow of data and ensures that each PE is synchronous to one another. It creates the necessary control signals to manage the timing and coordination of data movement through the arrya. The control

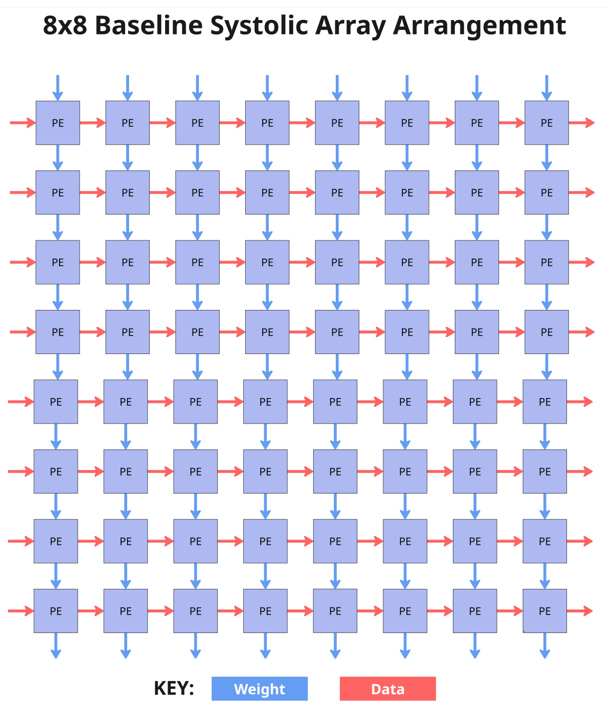


Figure 3 Image of the Baseline Systolic Array Arrangement

unit is responsible for loading the weights and activations into the array, as well as signalling when to start and stop the computation.

The dataflow inputted to the systolic array is designed to be highly efficient. Weights are fed into the array from the left, moving horizontally across each row of PEs. Activations are fed in from the top, moving vertically down each column. As weights and activations move through the array, each PE performs its MAC operation and passes the activation to the PE below it and the weight to the PE to its right. This creates a wave-like motion of data flowing through the array, ensuring that all PEs are utilised effectively. The output partial sums are stored in local registers within each PE until the entire matrix multiplication is complete. Once all inputs have been processed, the final output feature map is read out from the bottom-right corner of the array.

The dataflow of the systolic array is designed to be highly efficient. Weights are fed into the array from the left, moving horizontally across each row of PEs. Activations are fed in from the top, moving vertically down each column. As weights and activations move through the array, each PE performs its MAC operation and passes the activation to the PE below it and the weight to the PE to its right. This creates a wave-like motion of data flowing through the array, ensuring that all PEs are utilised effectively. This staggered input of weights and activations allows for continuous operation of the array, maximising throughput. Figure 4 illustrates the dataflow for a 3x3 systolic array. The output partial sums are stored in local registers within each PE until the entire matrix multiplication is complete. Once all inputs have been processed, the final output feature map is read out.

As stated previously, the hardware accelerator is designed to take in 8x8 tiles of weights and activation matrices. However, the systolic array is capable of handling different sized inputs up to 8x8 (theoretically it can do NxN but sizes larger than 8x8 were not test). This is achieved by the control unit checking the input matrix sizes, assuming both are uniform in size, and adjusting the operation accordingly. For example, if the input matrix is the size of 6x6, the control unit will ensure that the systolic array will synthesise the correct amount of PEs and busses required.

This effectively downsizes the systolic array to only use the necessary resources, improving efficiency. This feature is crucial for handling sparsity, as it allows the array to adapt to the effective size of the input matrices after pre-processing, which will be discussed in Section 4.4.

The performance and behaviour of the Baseline Systolic Array was verified through simulation only without hardware synthesis. It is important to note that the control unit and the systolic array together are considered to be the NPU. More details on the verification process can be found in Section 5.2.

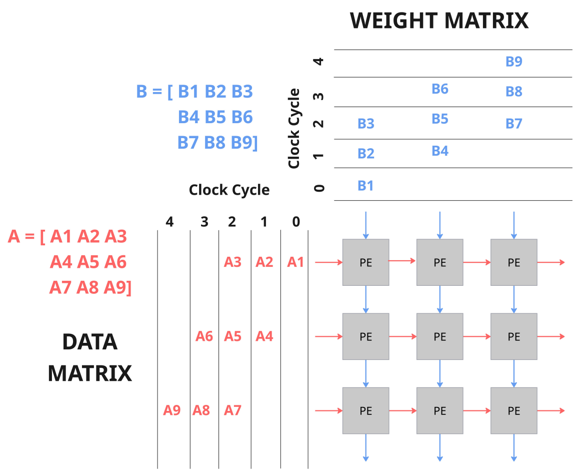


Figure 4 Image of the Baseline Systolic Array's Dataflow

4.4 Optimised Systolic Array for Sparse Matrices Architecture

As the research gap identified in Section 3.2.3 suggests, there is a need for a simpler solution that can exploit structured sparsity without changing the classic pipeline of the systolic array. To improve upon the Baseline Systolic Array architecture, a software-hardware co-design approach is integrated. This involves an extra step in the pre-processing stage of the data, where a Python-based algorithm function called `coordinated_row_removal` is implemented to remove rows of zeros in the activation and weight matrix. This simple addition significantly improves the performance of the systolic array when handling sparse matrices.

4.4.1 Hardware System Overview

Figure X illustrates the high-level block diagram of the final hardware system. The top-level module, named `de1_soc_top`, interfaces the hardware accelerator with the FPGA board. The `NPU_Wrapper` module encapsulates the entire NPU design, including the systolic array, control unit, and the ROMs. The ROMs are utilised to store the pre-processed weight and activation tiles in mif format, allowing for static testing of the hardware design. The control unit is responsible for managing the dataflow and synchronisation of the systolic array, as well as adapting to the effective size of the input matrices after pre-processing. The systolic array itself remains largely unchanged from the Baseline Systolic Array, retaining its 8x8 PE grid and output-stationary dataflow. However, it now benefits from the optimised input data, leading to improved efficiency and performance when handling sparse matrices.

4.4.2 Sparse Handling Algorithm

The `coordinated_row_removal` function is an algorithm that safely removes rows and columns of zeros from the weight and activation matrices. To ensure mathematical correctness, it cannot directly remove a row or column without checking the other matrix. The logic of the algorithm is as follows:

- Calculate the active rows for both the weight and activation matrices.
- Determine the inner dimension (also known as the shared dimension) by finding the intersection of the active activation columns and the active weight rows. This is denoted as the K value.
- Create new dense matrices by stripping all the inactive rows and columns. The new activation matrix will be of size (active_rows, K) and the new weight matrix will be of size (K, active_cols).

Qin et al. [18]'s paper supports this idea of maintaining the shared dimension, where the inner dimension is cruicial in maintaining mathematical correctness in its General Matrix Multiplication (GEMM) operations. As the MAC operation is simply a series of matrix multiplications, it is critical that the inner dimension is the same for both matrices to ensure that the operations are mathematically valid. The function returns the M value (active rows of the activation matrix), N value (active columns of the weight matrix), K value (shared inner dimension), and the new dense matrices. This is a crucial step in the pre-processing stage, as it ensures that the data fed into the systolic array is optimised for performance. It is especially effective for structured sparsity, where entire rows or columns of weights are zero, as it allows the systolic array to adapt to the effective size of the input matrices. The `coordinated_row_removal` algorithm is implemented in the same Python script as Section 4.2.1 and Section 4.2.2. The M, N, and K values are now stored in the mif file, meaning that each file can have up to 67 values. This is important as the control unit needs to know these values to reconfigure the systolic array.

4.4.3 ROMs

In the software pre-processing stage, the mif files are generated and stored in a folder. Currently, there is no mechanism to dynamically change the mif file being read by the ROMs, so it has to be manually updated to test different tiles. However, with the ROMs instantiated in the system, it is possible to run the Python script to generate new mif files and directly load them into the ROMs without modifying the hardware design. This flexibility is crucial for testing different tiles and sparsity patterns, allowing for efficient validation of the hardware design against various scenarios. A ROM is instantiated for both the weight and activation matrices in the NPU Wrapper, which serves as the top-level module. Each ROM is configured to read from its respective mif file, storing the pre-processed tiles for static testing. The ROMs are designed to output data in a format compatible with the systolic array, ensuring seamless integration with the control unit and PEs. This setup allows for efficient data retrieval during operation, enabling the systolic array to process the input matrices effectively.

4.4.4 NPU Wrapper

The purpose of the NPU Wrapper is to act as the interface with the top-level module and the NPU core. It is responsible for instantiating all the components of the NPU, consisting of the systolic array, control unit, and ROMs. With the addition of the sparsity handling algorithm, modification of the MIF files, and introduction of the ROMs, the NPU Wrapper was created to handle the new changes. The NPU Wrapper is designed to read data from the ROMs correctly. It utilises a Finite State Machine (FSM) to manage the data flow and control signals. The FSM operates as follows:

- **S_IDLE**: The system starts in the idle state, waiting for a start signal to initiate the operation.
- **S_LOAD_PARAMS**: Upon receiving the start signal, the FSM transitions to this state to read the first three values from the ROMs, which are M, N, and K. These values determine how many reads are required, as M*K values will be read from the activation ROM and K*N values from the weight ROM.
- **S_LOAD_MATRICES**: Once the parameters are known, the FSM moves to this state to read the matrices from the ROMs. Since the data is stored in a 1D array format, it is converted into a 2D array before being sent to the control unit.
- **S_EXECUTE**: After loading the matrices, the FSM enters this state and waits for a signal from the control unit indicating that it has sent the values to the systolic array.
- **S_DONE**: Once execution is complete, the FSM transitions to this state and waits for the maximum theoretical number of clock cycles required to process the data, calculated as (M + N + K - 2) clock cycles. After this period, it sends a done signal to the top-level module, which then notifies the control unit that the operation is complete, returning to the S_IDLE state.

4.4.5 Control Unit and Systolic Array

The biggest change in the control unit is influenced by the introduction of the NPU Wrapper. The NPU wrapper sends the control unit the weight and activation matrices alongisde the new M, N, and K values as separate inputs. This means that the control unit's matrix handling logic remains largely unchanged, requiring only minor modifications to incorporate the M, N, and K values. These values are crucial for verifying the shape of the matrices and ensuring that the systolic array operates correctly based on the effective size of the input matrices after pre-processing. Additionally, the control unit must now account for when the NPU is ready to send signals, ensuring proper synchronization between components. Fundamentally, the systolic array and its processing elements have not changed due to the modular design of the system. The control unit continues to send the same data format to the systolic array, allowing it to function without any modifications. As the Baseline Systolic Array considered modularity from the start, the incorporation of sparsity handling features was achieved with minimal changes to the core components. This modularity is a key advantage of the design, as it enables easy integration of new features, such as sparsity handling, without requiring extensive changes to the core components.

Overall, the integration of the sparsity handling algorithm into the hardware design through the NPU Wrapper and minor modifications to the control unit has significantly enhanced the efficiency of the systolic array when processing sparse matrices. This software-hardware co-design approach effectively addresses the research gap identified in Section 3.2.3, providing a simpler solution for exploiting structured sparsity without changing the classic systolic array pipeline.

5. Verification and Results

This section presents the verification process and results of the software-hardware co-design approach iteration. The performance of the NPU is evaluated using both dense and sparse matrices, demonstrating the flexibility and reconfigurability of the systolic array. The results are compared between the Baseline Systolic Array and the Optimised Systolic Array to highlight the improvements achieved through the sparsity handling algorithm. The verification process includes simulation and synthesis on an FPGA platform, providing a comprehensive assessment of the design's effectiveness in handling sparsity.

5.1 Testbench and Simulation Environment

The VHDL testbench was designed to simulate the operation of the NPU, including both the Baseline Systolic Array and the Optimised Systolic Array. The testbench is executed on ModelSim-Altera 10.1d, a widely used simulation tool for VHDL designs. Furthermore, the Optimised Systolic Array is synthesised on the Intel DE1-SoC FPGA board using Quartus Prime 18.1 software to evaluate its performance on real hardware.
The testbench operates by providing input matrices to the NPU and monitoring the output results. Both systolic array designs are tested under similar conditions to ensure a fair comparison. The key differences between the two designs are as follows:

- **Baseline Systolic Array**: The input matrices are hardcoded into the testbench for static testing. The system starts operation immediately upon, without waiting for a start signal, and the matrices are directly sent to the control unit.
- **Optimised Systolic Array**: The input matrices are pre-loaded and read from the ROMs using mif files generated from the Python pre-processing stage. The system waits for a start signal to begin operation to check if the control unit has received the inputs correctly.

The testbench counts the latency by counting the number of clock cycles taken for the NPU to complete the matrix multiplication operation. This measurement provides insight into the NPU performance, allowing for a direct comparison between the Baseline and Optimised Systolic Array. Furthermore, the testbench monitors the output of the NPU to verify the correct of the matrix multiplication results. This is cross-verified against expected results calculated in both Python and MATLAB to ensure the NPU operates correctly. Another crucial behaviour that the testbench verifies is the staggering behaviour of the control unit inputs, ensuring that data is fed into the systolic array in the expected manner.

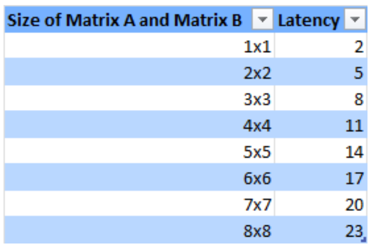
The modularity of the testbench allows for easy modification and extension, essential for testing different test cases. This includes varying size of input matrices and different sparsity patterns. Overall, the VHDL testbench provides a comprehensive environment for verifying the functionality and performance of the NPU design.

Appendix X includes screenshots of all the ModelSim results alongside the expected results from Python and MATLAB for reference. For the purpose of this report, three types of matrices were tested: a dense 8x8 matrix, a sparse 1x8 matrix, and a 4x8 matrix. These examples were chosen to compare the performance of the Baseline and Optimised Systolic Arrays under different conditions.

5.2 Baseline Systolic Array Performance

This section presents the performance results of the Baseline Systolic Array when processing various input matrices. The focus is on evaluating the latency in clock cycles for different matrix sizes and sparsity patterns. The Baseline Systolic Array is a reference design, primarily used to validate the functionality of the systolic array architecture without optimisation for sparsity. Therefore, all verification was conducted on simulation without synthesis, as its primary goal is to establish a baseline comparison point for the Optimised Systolic Array. Additionally, as this version does not implement sparsity handling, synthesising would not yield meaningful insights into its performance in sparse scenarios.

Figure 5 summarises the latency results when NxN matrices are processed through the Baseline Systolic Array. The results indicate that the latency increases with the size of the input matrices, as expected due to the increased number of computations required. This proves that the systolic array easily adjusts to different input sizes up to its maximum capacity of 8x8, indicating that it is capable of saving power when smaller matrices are used. The latency values are consistent with theoretical expectations for an 8x8 systolic array, confirming the correct operation of the design.

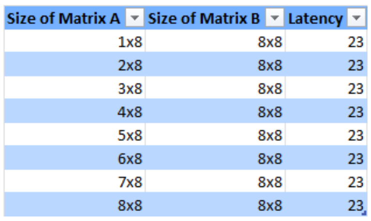


Size of Matrix A and Matrix B	Latency
1x1	2
2x2	5
3x3	8
4x4	11
5x5	14
6x6	17
7x7	20
8x8	23

Figure 5 Latency of Baseline Systolic Array on Different NxN sizes

Three crucial test cases were executed to assess the performance of the Baseline Systolic Array:

- **Dense 8x8 Matrix**: This test case confirms the correct operation of the systolic array. The expected output matches the Python calculations, validating the functionality of the design. The measured latency for this case was 23 clock cycles.
- **4x8 Matrix**: This test case demonstrates the adaptability of the systolic array to different input sizes. The systolic array successfully processed the 4x8 matrix. The measured latency for this case was 23 clock cyclces.
- **1x8 Sparse Matrix**: This test case highlights the inefficiency of the Baseline Systolic Array when handling sparsity. Despite the high sparsity of the input matrix, the systolic array processed the inputs as 8x8, resulting in unnecessary computations. The expected output matched the simulation results, confirming correct operation. The measured latency for this case was 23 clock cycles.



Size of Matrix A	Size of Matrix B	Latency
1x8	8x8	23
2x8	8x8	23
3x8	8x8	23
4x8	8x8	23
5x8	8x8	23
6x8	8x8	23
7x8	8x8	23
8x8	8x8	23

Figure 6 Latency of Baseline Systolic Array on Varying Input Matrix Size

Figure 6 summarises the latency results for each test case, with some additional test cases included for completeness. The results demonstrated that the Baseline Systolic Array consistently processed all input matrices with a latency of 23 clock cycles, regardless of the matrix size or sparsity. This highlights the inefficiency of the Baseline Systolic Array in handling sparse matrices, as it does not adapt to the effective size of the input matrices after pre-processing.

The verification process involved cross-referencing the output results from ModelSim with expected results calculated in Python and MATLAB. The successful matching of outputs across all test cases confirms that the Baseline Systolic Array operates as intended. The latency measurements provide a benchmark for comparison with the Optimised Systolic Array. Furthermore, the simulation results validated the correctness of the systolic array's dataflow and control unit operations mentioned in Section 4.3.2.

5.3 Optimised Systolic Array Performance

This section presents the performance results of the Optimised Systolic Array when processing various input matrices. The focus is on evaluating the latency in clock cycles for different matrix sizes and sparsity patterns. The Optimised Systolic Array incorporates the software-hardware co-design approach, utilising the `coordinated_row_removal` algorithm to pre-process input matrices and remove rows of zeros, thereby improving efficiency when handling sparse matrices.
THe verification of the software co-design approach involved testing on a singular Python script that executes the entire software design, from loading the pre-trained ML model to tiling and generating MIF files. The script was tested with different images to observe how sparsity patterns change and to gather a diverse range of values for the MIF files. The expected performance of the systolic array was calculated within the script, including latency and matrix multiplication results. Verification of the Python script was achieved through three methods:

1. Matching the expected output with ModelSim results, confirming that the ROMs read from the correct MIF files.
2. Manually checking the MIF file format and values to ensure correctness.
3. Comparing the matrix multiplication output from Python with the simulation results from ModelSim.

To verify the effectiveness of the Sparse Handling Algorithm, the latency results of the Optimised Systolic Array were summarised based on the M values obtained after pre-processing. The results demonstrated a linear decrease in latency as more rows were stripped from the activation matrix, even when the weight matrix remained dense (shown in Figure 7). This confirmed that the `coordinated_row_removal` algorithm effectively improves the performance of the systolic array when handling sparse matrices.

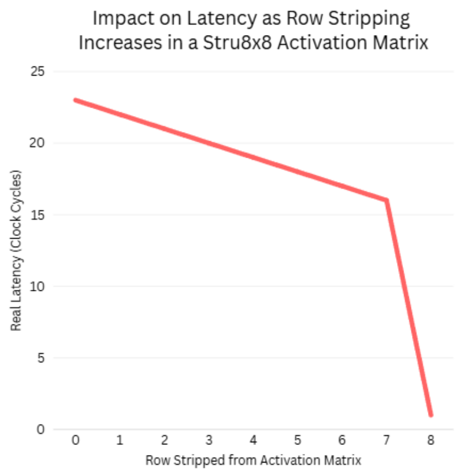
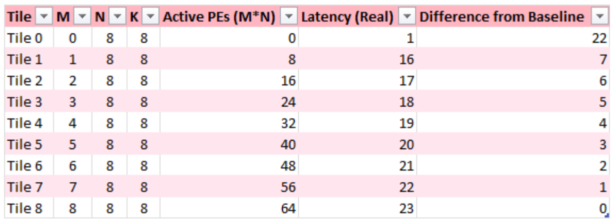


Figure 7 Impact on Latency as Row Stripping Increases in a 8x8 Activation

As a direct comparison to the Baseline Systolic Array, the 8x8 dense matrix corresponds to an M value of 8, the 4x8 matrix corresponds to an M value of 4, and the 1x8 sparse matrix corresponds to an M value of 1. The latency results of these is shown in Figure 8, demonstrating the performance improvements achieved through the sparsity handling algorithm.



Tile	M	N	K	Active PEs (M*N)	Latency (Real)	Difference from Baseline
Tile 0	0	8	8	0	1	22
Tile 1	1	8	8	8	16	7
Tile 2	2	8	8	16	17	6
Tile 3	3	8	8	24	18	5
Tile 4	4	8	8	32	19	4
Tile 5	5	8	8	40	20	3
Tile 6	6	8	8	48	21	2
Tile 7	7	8	8	56	22	1
Tile 8	8	8	8	64	23	0

Figure 8 Effect of M value on the number of Active PEs in the Systolic Array and Latency

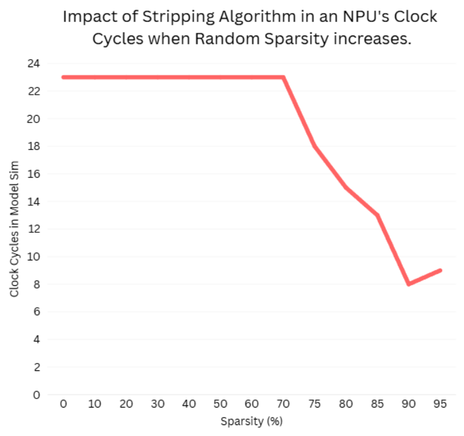


Figure 9 IImpact of Stripping Algorithm on Clock Cycles when Applied on Random Sparsity

5.4 Performance Analysis

The successful synthesis on the DE1-SoC FPGA board confirms that the hardware implementation aligns with the simulated model. Furthermore, the model's accuracy in reflecting the Python script's behavior reinforces confidence to do theoretical speedup calculations derived from high-level testing. This multi-tiered validation process ensures that the performance claims are robust and justified. Thus, the following test case was conducted in Python.
The testcase found that AlexNet has approximately 710 million MAC operations, closely aligning with Tiwari et al. [19]'s discovery of 724 million MAC operations. A variety of 156 images, including of xrays, cats, and nature scenes, were pre-processed using the same Python script mentioned in Section 4.2. This generated 8x8 tiles with different sparsity patterns, allowing for a comprehensive evaluation of the systolic array's performance across diverse scenarios.
Figure 10 illustrates the average latency for the Baseline Systolic Array, which was determined to be 23.00 clock cycles per tile, as it does not adapt to the effective size of the input matrices after pre-processing. In contrast, the average latency for the Optimised Systolic Array was found to be 9.07 clock cycles per tile, significantly benefitting from the sparsity in the input matrices. This results in an average reduction of approximately 60.87% in latency when using the Optimised Systolic Array compared to the Baseline Systolic Array.
Relating this to the total number of MAC operations in AlexNet, the Optimised Systolic Array has the potential to reduce approaximately 432 million clock cycles on average when processing AlexNet images compared to the Baseline Systolic Array. This is a considerable improvement, demonstrating the efficiency of the sparse handling algorithm in a software-hardware co-design context.

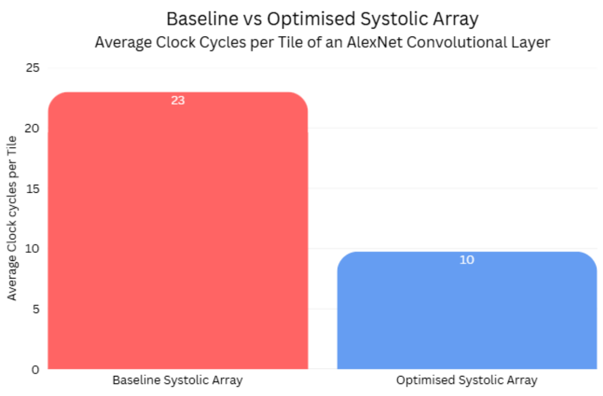


Figure 10 Performance Comparison: Average Clock Cycles per Tile of an AlexNet Convolutional Layer

6. Discussion

6.1 Analysis of Results

- the linear decrease in latency as more rows were stripped from the activation matrix demonstrates that the `coordinated_row_removal` algorithm exploits systems with structured sparsity effectively - relevant in systems where entire rows or columns or weights are zero. - although the random sparsity patterns did not show significant improvements, this was expected as the probability of having an entire row of zeros in a random sparsity pattern is low - this highlights the importance of understanding the sparsity patterns in the target application when designing hardware

- !! link back to a reference

- the theorectical latency calculation was 3n-2 or M + N + K -2 clock cycles - the actual latency observed in the simulation results closely matched these theoretical values, confirming the accuracy of the design and its implementation - any one clock cycle delay contributed to hardware delays such as signal propagation. could also be from zero-value inputs, where the results do not look like they changed but did take up clock cycles to process

- link it back to the research gap and scope - although this is specifically for structured sparsity, the project still successfully addresses the identified research gap in section 3.2.3 by providing a simpler solution that can exploit structured sparsity without changing the classic pipeline of the systolic array - add some scope stuff

- key takeawakays - the software-hardware co-design approach is effective in handling sparsity in systolic arrays - this kept us in-scope as well since we did not have to deploy ML models on the hardware itself - there was dynamic reconfiguration based on the pre-processed sparsity patterns, that lead to significant performance improvements - the modular design of the systolic array allowed the integration of teh sparsity handling algorithm with minimal changes to the core components

- the implication of 60.87% latency reduction is huge for resource constrainted edge devices - having a 9 clock cycle on an average suggests that there were lots of input matricers that were 100% sparse, bringing the average down significantly. - by minimising the number of clock cycles, we reduce the computations, leading to lower power consumption and longer battery life - this makes it applicable to real world situations where edge devices need to perform ML inference on sparse data efficiently !! link back to the problem !! link back to research

6.2 Design Trade-offs

pros: - by performing the stripping algo in software, the hardware design was able to remain simple - this reduces hardware complexity, making it easier to verify and synthesise on FPGA. it was very similar to the original systolic array design, meaning that the modularity of the design was preserved - software design is more flexible and easier to modify. if we wanted to change the stripping algorithm, we could do so in python without having to modify the hardware design. pyhon being a high levle language also makes it significantly easier and faster to develop - the pre-processing occurs offline, meaning that the hardware does not have to handle computation loads of the stripping algorithm. this is especially important for resource-constrained edge devices, where hardware resources are limited - the algorithm maintianed mathematical correctness of the matrix multiplication operation, ensuring that the output remains accurate despite the removal of zero rows

cons: - not real-time dynamic reconfiguration, reliance on MIF means that we can't get live data streams. however, this was out of scope for this project - from the testing that we've done, we saw that the weights were always densely packed. there were no zero rows to strip. this could be a limitation of the pre-trained model we used, meaning that the stripping algorithm's effectiveness is limited by the sparsity patterns in the input data. even if activation was extremely sparse, if the weights are dense, the overall speedup is limited. as we saw, when M = 1, N =8, K = 8, it still took 16 clock cycles. from the 23 clock cycles, this is only a 30.43% reduction, which is not as significant as we had hoped - this limits us to only stripping from the activation matrix, meaning that when we consier M + N + K -2, the K value will always be at its maximum, limiting the overall speedup we can achieve - having to pass on the M N K values in the MIF with the comrpessed matrix. this added extra logic that we had to account for. however, this was minimal due to the modularity of the design - we do not have a post processing stage where we take the output of the hardware and send it back to the software to process to revert it back to its original shape. this could be an area of future work

6.3 Limitations

- the current design relies on static testing using MIF files, which limits its ability to handle live data streams. this means that the system cannot adapt to changing sparsity patterns in real-time, which is a significant limitation for applications requiring dynamic data processing - pre-processing is separate, its not technically a full pipeline from end to end. ther ei slaos no post-processing stage to revert the output back to its original shape - the algorithm does not look into fine-grained sparsity, not taking advantage of the potential exploitation of unstructured sparsity patterns that are common in real-world data - fixed array size of 8x8, whilst this is technically scalable or can be downgraded, it is limited to 8x8. this means that larger matrices have to be tiled, which could introduce overhead and complexity in managing multiple tiles, esp with the current design of using mifs - the high level python test didn't directly test real-world applications (since we only did 156 rather than thousands of images). this could limit the generalisability of the results to real-world scenarios - time constraints limited the depth of exploration into more advanced sparsity handling techniques, such as dynamic reconfiguration or support for unstructured sparsity. the scope of this project was only one year, and the fact that we had to build our knowledge from the ground definitely slowed us down, as we had to learn heaps of unfamiliar concepts rather than going straight into critquing current designs. further research and development are needed to fully realise the potential of sparsity handling in systolic arrays.

7. Future Work

short term: - as we are utilising the Pico as the middle man of our UART transmission, this actually take get the pico to do the pre-processing stage itself. this means that we can have a live data stream where the pico takes in the data, processes it, and sends it to the DE1-SoC board to process. this would make the system more real-time and dynamic. - then, it can do some post-processing as well, where it takes the output from the DE1-SoC board and reverts it back to its original shape. this would complete the entire pipeline from end to end. - this would also mean we aren't relying on the MIF files, which would simplify the hardware design even further as we don't have to account for reading from ROMs. instead, the data can be sent directly to the control unit from the pico via UART. - Investigate the different sparsity patterns in different layers of AlexNet or CNN models. If a significant structure sparsity is found, the algorithm can be further optimized to exploit these patterns. - investigate the resource utilisation and find a way to optimise It

long term: - full hardware implemetnation of saprsity handling algorithnm. a whole hardware block can be impolemented for just the algorthm iretly on the fpga. this would enable real-time dynamic reconfiguration of the systolic array based on live data streams. - integrate the NPU with a softcore processor like NIOS II to create a complete heterogenous system. we can let the NPU handle all the matrix multiplications and offlaod the other tasks to the NIOS II processor. this would create a more versatile system capable of handling a wider range of applications. !! there is a paper that talks about idling time - DPR is something that was explored earlier in the project. due to time/skill constraint, we did not delve deeply into it. this invovlves reconfiguration of the fpga fabric at runtime to instantiate the neceessary PEs. this would allow for greater power savings and resource flexibility - support for unstructured sparsity, as this is more common, this can be investigated - end to end cnn inferences. we do the whole processing between layers untl the final output. this would involve integrating with other hardware blocks for convolution, pooling, activation functions etc.

8. Conclusion

References

[1] A. Parashar et al., *SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks*, en, arXiv:1708.04485 [cs], May 2017. DOI: 10.48550/arXiv.1708.04485. Accessed: Sep. 18, 2025. [Online]. Available: http://arxiv.org/abs/1708.04485.

[2] B. Kim, S. Lee, A. R. Trivedi, and W. J. Song, "Energy-Efficient Acceleration of Deep Neural Networks on Realtime-Constrained Embedded Edge Devices," en, *IEEE Access*, vol. 8, pp. 216 259–216 270, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3038908. Accessed: Mar. 31, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9262933/.

[3] E. Manor and S. Greenberg, "Custom Hardware Inference Accelerator for TensorFlow Lite for Microcontrollers," en, *IEEE Access*, vol. 10, pp. 73 484–73 493, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3189776. Accessed: Mar. 31, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9825651/.

[4] J. Choi et al., "Enabling Fine-Grained Spatial Multitasking on Systolic-Array NPUs Using Dataflow Mirroring," en, *IEEE Transactions on Computers*, vol. 72, no. 12, pp. 3383–3398, Dec. 2023, ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: 10.1109/TC.2023.3299030. Accessed: Apr. 2, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10198513/.

[5] W. Sun, D. Liu, Z. Zou, W. Sun, S. Chen, and Y. Kang, "Sense: Model-Hardware Codesign for Accelerating Sparse CNNs on Systolic Arrays," en, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 4, pp. 470–483, Apr. 2023, Publisher: Institute of Electrical and Electronics Engineers (IEEE), ISSN: 1063-8210, 1557-9999. DOI: 10.1109/tvlsi.2023.3241933. Accessed: Jul. 22, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10043636/.

[6] D. Cain, O. Eldash, K. Khalil, and M. Bayoumi, "Convolution Processing Unit Featuring Adaptive Precision using Dynamic Reconfiguration," en, in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, New Orleans, LA, USA: IEEE, Jun. 2021, pp. 592–597, ISBN: 978-1-6654-4431-6. DOI: 10.1109/WF-IoT51360.2021.9595539. Accessed: Oct. 17, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9595539/.

[7] C. Zhang, X. Wang, S. Yong, Y. Zhang, Q. Li, and C. Wang, "An Energy-Efficient Convolutional Neural Network Processor Architecture Based on a Systolic Array," en, *Applied Sciences*, vol. 12, no. 24, p. 12 633, Dec. 2022, Publisher: MDPI AG, ISSN: 2076-3417. DOI: 10.3390/app122412633. Accessed: Jul. 14, 2025. [Online]. Available: https://www.mdpi.com/2076-3417/12/24/12633.

[8] S. Oh, M. Kim, D. Kim, M. Jeong, and M. Lee, "Investigation on performance and energy efficiency of CNN-based object detection on embedded device," en, in *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, Kuta Bali: IEEE, Aug. 2017, pp. 1–4, ISBN: 978-1-5386-0600-1. DOI: 10.1109/CAIPT.2017.8320657. Accessed: Mar. 31, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/8320657/.

[9] X. Liu, W. Xu, Q. Wang, and M. Zhang, "Energy-Efficient Computing Acceleration of Unmanned Aerial Vehicles Based on a CPU/FPGA/NPU Heterogeneous System," en, *IEEE Internet of Things Journal*, vol. 11, no. 16, pp. 27 126–27 138, Aug. 2024, ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2024.3397649. Accessed: Mar. 31, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10525057/.

[10] H. Kung, C. Leiserson, C.-M. U. P. P. D. o. C. SCIENCE, and C. M. U. C. S. Department, *Systolic Arrays for (VLSI)* (CMU-CS). Carnegie-Mellon University, Department of Computer Science, 1978. [Online]. Available: https://books.google.co.nz/books?id=pAKfHAAACAAJ.

[11] J. Seo and J. Kong, "VerSA: Versatile Systolic Array Architecture for Sparse and Dense Matrix Multiplications," en, *Electronics*, vol. 13, no. 8, p. 1500, Apr. 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13081500. Accessed: Oct. 15, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/13/8/1500.

[12] X. He et al., "Sparse-TPU: Adapting systolic arrays for sparse matrices," en, in *Proceedings of the 34th ACM International Conference on Supercomputing*, Barcelona Spain: ACM, Jun. 2020, pp. 1–12, ISBN: 978-1-4503-7983-0. DOI: 10.1145/3392717.3392751. Accessed: Apr. 2, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3392717.3392751.

[13] H. H. Tan and K. H. Lim, "Vanishing Gradient Mitigation with Deep Learning Neural Network Optimization," en, 2019.

[14] S. Kim, S. Cho, E. Park, and S. Yoo, "FPGA Prototyping of Systolic Array-based Accelerator for Low-Precision Inference of Deep Neural Networks," en, in *2021 IEEE International Workshop on Rapid System Prototyping (RSP)*, Paris, France: IEEE, Oct. 2021, pp. 1–7, ISBN: 978-1-6654-6956-2. DOI: 10.1109/RSP53691.2021.9806200. Accessed: Mar. 30, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9806200/.

[15] J. Gorvadiya, A. Chagela, and M. Roy, "Energy Efficient Pruning and Quantization Methods for Deep Learning Models," en, in *2025 International Conference on Sustainable Energy Technologies and Computational Intelligence (SETCOM)*, Gandhinagar, India: IEEE, Feb. 2025, pp. 1–6, ISBN: 979-8-3315-2054-0. DOI: 10.1109/SETCOM64758.2025.10932458. Accessed: Mar. 28, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10932458/.

[16] Y. Duk Kim et al., "2.4 A 7nm High-Performance and Energy-Efficient Mobile Application Processor with Tri-Cluster CPUs and a Sparsity-Aware NPU," en, in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, San Francisco, CA, USA: IEEE, Feb. 2020, pp. 48–50, ISBN: 978-1-7281-3205-1. DOI: 10.1109/ISSCC19947.2020.9062907. Accessed: Oct. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9062907/.

[17] P. Palacios, R. Medina, J.-L. Rouas, G. Ansaloni, and D. Atienza, "Systolic Arrays and Structured Pruning Co-design for Efficient Transformers in Edge Systems," en, arXiv:2411.10285 [cs], Jun. 2025, pp. 320–327. DOI: 10.1145/3716368.3735158. Accessed: Jul. 16, 2025. [Online]. Available: http://arxiv.org/abs/2411.10285.

[18] E. Qin et al., "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," en, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, San Diego, CA, USA: IEEE, Feb. 2020, pp. 58–70, ISBN: 978-1-7281-6149-5. DOI: 10.1109/HPCA47549.2020.00015. Accessed: Oct. 16, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9065523/.

[19] B. Tiwari, M. Yang, X. Wang, and Y. Jiang, "Data Streaming and Traffic Gathering in Mesh-based NoC for Deep Neural Network Acceleration," en, *Journal of Systems Architecture*, vol. 126, p. 102 466, May 2022, arXiv:2108.02569 [cs], ISSN: 13837621. DOI: 10.1016/j.sysarc.2022.102466. Accessed: Oct. 18, 2025. [Online]. Available: http://arxiv.org/abs/2108.02569.

Appendix A List of Figures

This appendix contains a comprehensive list of all figures used throughout this document.

List of Figures