A General Purpose Near Data Processing Architecture Optimized for Data-intensive Applications

XINGDA LI
HAIDI HU
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

A General Purpose Near Data Processing Architecture Optimized for Data-intensive Applications

Xingda Li
xi6341li-s@student.lu.se
Haidi Hu
ha3077hu-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors:
Joachim Rodrigues
Arturo Prieto
Masoud Nouripayam

Examiner: Erik Larsson

August 8, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In recent years, as Internet of Things (IoT) and machine learning technologies have advanced, there has been increasing interest in the study of energy-efficient and flexible architectures for embedded systems. To bridge the performance gap between microprocessors and memory systems, Near-Data Processing (NDP) was introduced. Although some works have implemented NDP, few of them utilize the microprocessor's cache memory.

In this thesis, we present an NDP architecture that integrates static random access memory (SRAM), which is regarded as the L2 cache of a microcontroller unit (MCU). The proposed NDP is tailored for data-intensive applications and seeks to address multiple problems. A coarse-grained reconfigurable array (CGRA)-based strategy is utilized to maximize flexibility while decreasing power consumption. Additionally, numerous approaches, such as convolution-and-pooling-integrated computation, two-level clock gating, etc., are implemented to improve energy efficiency even more.

The design was constructed utilizing STMicroelectronics (STM) 65 nm Low Power Low VT (LPLVT) technology with a maximum clock rate of 167 MHz. Two popular algorithms, the convolutional neural network (CNN) and K-means, were mapped onto the hardware to evaluate it. As a result, the power efficiency of CNN and K-means algorithms can be boosted by 12x and 26x relative to field-programmable gate array (FPGA) and MCU implementations, respectively, and by several orders of magnitude relative to other K-Means accelerators.

Popular Science Summary

In the past two decades, Machine Learning (ML) has rapidly advanced and been widely utilized in research, technology, and commerce [1]. In a variety of domains, including automated driving, computer vision, and speech recognition, ML has proven to be an effective technique for producing useful applications. However, existing ML methods and computer systems are currently confronted with a huge obstacle provided by the exponential growth of data.

The most prevalent method for accelerating data-intensive ML applications is to increase the speed of existing central processing units (CPUs) and graphics processing units (GPUs). However, given to the flexibility of these two processors, they can be somewhat power-hungry when executing particular ML approaches. Certain candidates, such as FPGA/ASIC accelerators, can achieve a decent balance between performance and power consumption. Unfortunately, due to their high cost and comparable huge power consumption, they may not be suitable for use in various rapidly expanding consumer electronics, such as smart watches, sweeping robots, smart speakers, etc.

In order to address these issues, the proposed solution must be as accommodating as possible and capable of supporting a wide variety of applications in the ML domain. Additionally, novel techniques should be employed to enhance the power efficiency so that it can be integrated into IoT devices.

Acronyms

AI Artificial Intelligence.

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuit.

ASSOC Associativity.

CGRA Coarse-Grained Reconfigurable Array.

CIFAR Canadian Institute For Advanced Research.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DNN Deep Neural Network.

DPEU Dual Processing Element Unit.

DRAM Dynamic Random Access Memory.

ED Euclidean Distance.

FC Fully Connected.

FPGA Field-Programmable Gate Array.

FSM Finite State Machine.

FU Functional Unit.

GAN Generative Adversarial Network.

GP General Purpose.

GPU Graphics Processing Unit.

IC Integrated Circuit.

LPLVT Low Power Low Vt.

MAC Multiply–Accumulate.

MCU Microcontroller Unit.

ML Machine Learning.

MLP Multilayer Perceptron.

MNIST Modified National Institute of Standards and Technology.

MUL Multiply.

NDP Near Data Processing.

NDPU Near Data Processing Unit.

PE Processing Element.

PSUM Partial Sum.

RC Reconfigurable Array.

ReLU Rectified Linear Unit.

SED Squared Euclidean Distance.

SIMD Single Instruction Multiple Data.

SRAM Static Random Access Memory.

VLIW Very Long Instruction Word.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The memory wall was first discussed by Wulf and McKee in 1994, predicting that the performance gap between microprocessors and memory could become a significant concern in the future [6]. For many years, various forecasts have been made regarding the impending memory wall, which would result in severe processor performance limitations. Such concerns are not baseless, given the yearly gain in central processing unit (CPU) performance from 1980 to 2000 was over 60%, whilst the annual improvement in dynamic random-access memory (DRAM) access time has been less than 10% [7]. There is a mismatch between the rise in memory capacity and the corresponding decrease in memory latency as the main memory size grows continually [8].

Although memory-wall refers specifically more to main memory (DRAM), it can be extended somehow to cover the mismatch between processor speed (multi-core multi-threading) and cache (SRAM) responsiveness. With the emergence of memory-intensive workloads, such as MapReduce, graph processing and deep neural networks [9], the lack of temporal locality exacerbates the problem because the processor must wait when there is a cache miss. An obvious way to tackle this issue is to move the computation closer to the data. To this end, near-data processing (NDP) was proposed and became a promising solution.

The memory hierarchy of today typically includes multiple cache levels, main memory, and storage. The conventional method involves moving data from storage to caches before processing it. NDP refers to the creation of dedicated hardware positioned near the data. This data-centric strategy aims to avoid costly data transfers [10]. A key decision for NDP systems is the processing element type. In the past, NDP systems have employed a range of approaches, including general-purpose programmable cores with Single Instruction Multiple Data (SIMD) or multi-threading capabilities, throughput-oriented GPUs, reconfigurable arrays, and application-specific integrated circuits (ASICs). Each alternative represents a unique compromise between performance, area and energy efficiency, and adaptability. Programmable cores and Graphics Processing Units (GPUs) offer the greatest degree of flexibility; but, their increased area and power requirements may limit the number of elements per chip, thereby underutilizing the high memory bandwidth offered by NDP [11]. ASICs are highly efficient, but can only serve a limited number of applications; hence, they lack the adaptability to accommodate a variety of workloads [11].

1.1 Thesis Scope

Our Master's thesis work aims to design and implement a general-purpose NDP that helps reduce excessive power consumption related to abundance of data accesses and has the potential to fit on a Microcontroller Unit (MCU) platform.

To accomplish this objective, the thesis is organized into the following tasks:

- Identifying applications for optimization
- Implementing NDP hardware architecture
- Evaluating design with applications mapping
- Comparing design with other hardware

1.2 Related Work

Research on computing near memory started in 1990s [12–16]. Recently, NDP has been brought back to researchers' attention, and this resurgence is largely attribute to the following three reasons: the advanced technology, the memory-package pin-count limitations and the advent of modern data-intensive applications [17].

Processing near memory can be coupled with different processing elements [17]. Reconfigurable architecture such as coarse-grained reconfigurable arrays (CGRA) and field programmable gate array (FPGA) is more flexible while dedicated hardware is highly efficient. Farmahini-Farahani et al. [18] implemented an acceleration engine with multiple CGRAs into the NDP logic layer to accelerate a large-scale loop body of big data applications and indicated that CGRAs have higher performance and less power consumption than FPGAs. Gao et al. [11] took advantages of both CGRA and FPGA to improve power and area efficiency and utilized specialized units to handle branch operation. Lim et al. [19] proposed a Triple Engine Processor (TEP), a heterogeneous near-memory processor with three types of computing engines, which are in-order core, CGRA, and dedicated hardware, to accelerate different types of kernel operations.

This work proposes an NDP architecture that utilizes CGRAs to achieve flexibility and energy efficiency and dedicated hardware to facilitate various data-intensive kernel operations.

1.3 Thesis Outlines

Chapter 1 Introduction This chapter highlights the challenges faced by current computer architecture and defines the scope and objectives of this thesis.

Chapter 2 Background This Chapter gives a brief introduction of the architecture, target applications, and target MCU platform of the design.

Chapter 3 Hardware Architecture and Configuration This Chapter describes the details of the architecture and the techniques used to optimize for the target applications.

Chapter 4 Algorithm Mapping Examples This Chapter shows three mapping strategies for better demonstration.

Chapter 5 Results and Analysis This Chapter presents the implementation results of the examples in Chapter 4 and compares them with other works.

Chapter 6 Conclusion This Chapter is a discussion regarding the results and future work that can be done.

Chapter 2
Background

This Chapter introduces the background and the motivation behind the design.

2.1 Machine Learning

Machine Learning (ML) refers to statistical models that may be taught to a system by presenting it with examples of desirable input-output behavior as opposed to being directly programmed [1]. It has grown significantly over the past several decades and developed into a number of algorithms to address a wide range of ML challenges. Although the majority of machine learning algorithms comprise relatively simple operations, the dataset size is expanding. For instance, a state-of-the-art deep neural network (DNN) known as Alex-Net has over 60 million parameters [20], resulting in lengthy CPU processing times.

2.1.1 Convolutional Neural Network

Convolutional Neural Network (CNN) is commonly employed in computer vision applications. As images have a two-dimensional structure, it is logical to explore correlations in adjacent pixels. CNNs receive as inputs a series of nonlinear functions from geographically adjacent regions of outputs from the prior layer, which are subsequently multiplied by the weights, reusing the weights several times [2].

The concept behind CNNs is that each layer increases the level of picture abstraction. For instance, the initial layer may just distinguish horizontal and vertical lines. They may be combined in the second layer to locate corners. The following phase may involve rectangles and circles. This input might be used by the subsequent layer to detect dog body parts such as the eyes and ears. The upper layers would attempt to identify traits of various dog breeds [2].

Each neural layer generates a collection of two-dimensional feature maps, where each cell attempts to detect a single feature in the corresponding area of the input [2].

subsectionConvolutional Neural Network Convolutional Neural Network (CNN) is commonly employed in computer vision applications. As images have a two-dimensional structure, it is logical to explore correlations in adjacent pixels. CNNs receive as inputs a series of nonlinear functions from geographically adjacent regions of outputs from the prior layer, which are subsequently multiplied by the

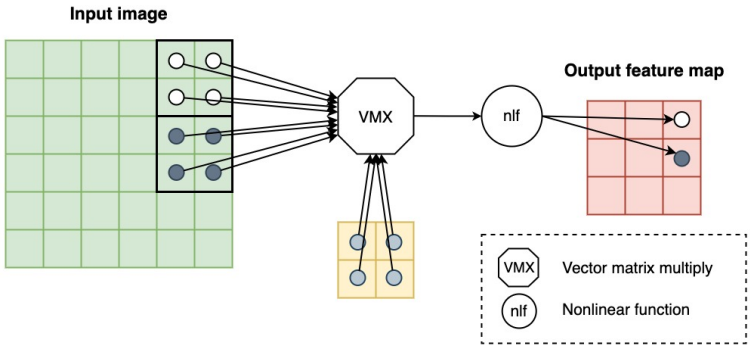


Figure 2.1: Convolution layer [2]

weights, reusing the weights several times [2].

The concept behind CNNs is that each layer increases the level of picture abstraction. For instance, the initial layer may just distinguish horizontal and vertical lines. They may be combined in the second layer to locate corners. The following phase may involve rectangles and circles. This input might be used by the subsequent layer to detect dog body parts such as the eyes and ears. The upper layers would attempt to identify traits of various dog breeds [2].

Each neural layer generates a collection of two-dimensional feature maps, where each cell attempts to detect a single feature in the corresponding area of the input [2].

The input data for an image-processing DNN would be a pixel of a image, with the pixel values multiplied by the weights. Numerous nonlinear functions have been attempted, but a prominent one today is $f(x) = max(x, 0)$, which returns 0 if x is negative or the original value if positive or zero. (This simple function is known by the complex acronym ReLU, or rectified linear unit.) An activation is the output of a nonlinear function since it is the output of the artificial neuron that has been "activated" [2].

Pooling Layer

Pooling is also known as downsampling. Commonly, a convolutional layer is accompanied by a max pooling layer, which computes a local maximum across the filtered response. This permits a little degree of translation invariance. It also minimizes the size of the higher layers, which significantly accelerates computing [2].

Fully Connected Layer

This layer will be explained in the following section-Section 2.1.2.

2.1.2 Multilayer Perception

Multilayer Perception (MLP), which is also known as Fully Connected (FC), is characterized by the dependence of each output neuron outcome on all input neurons of the prior layer. Each new layer is a set of nonlinear functions of weighted sum of all outputs from a prior one. The weighted sum is computed by multiplying the output vector matrices by the weights [2].

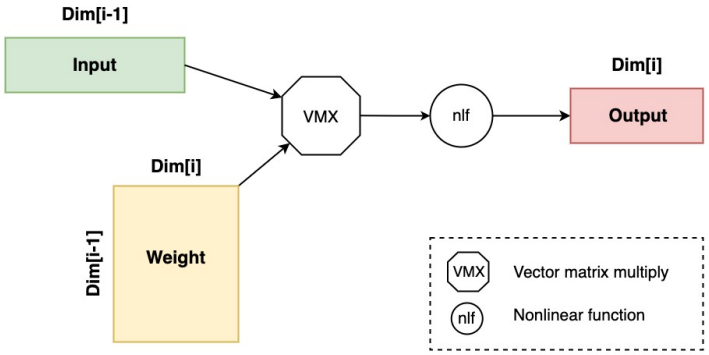


Figure 2.2: Multilayer Perception [2]

Figure 2.2 illustrates an FC layer that resembles a vector-matrix multiplication of an input vector by the array of weights. A Dim[i-1] input array and a $Dim[i-1] * Dim[i]$ weight matrix yields $Dim[i]$ output array [2].

2.1.3 K-means

The objective of the K-means clustering algorithm is to divide M points in N dimensions into K clusters so as to minimize the within-cluster sum of squares [21]. First, it allocates each instance to the cluster with the nearest current centroid (mean). Second, it computes the new mean of each cluster using the current cluster members, which becomes the new cluster centroid. The most likely cluster for x_i can be determined by locating its closest prototype [22] as

$$z_i^* = \underset{k}{argmin} ||x_i - \mu_k||_2^2 \,, \tag{2.1}$$

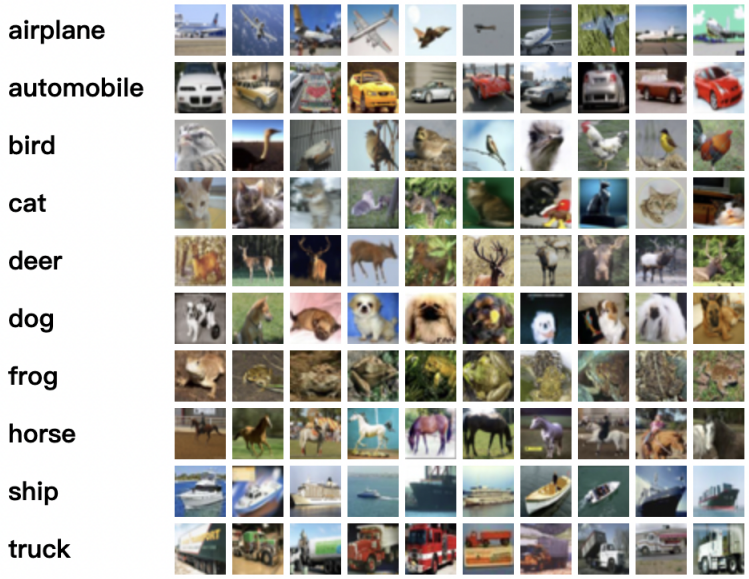
Given the hard cluster assignments, the M step revises each cluster center by calculating the mean of all assigned points [22] as

$$\mu = \frac{1}{N_k} \sum_{i:z_i=k} x_i \,. \tag{2.2}$$

2.1.4 CIFAR-10 Dataset

The CIFAR-10 dataset comprises 600,000 32x32 color images organized into 10 classes, with 6,000 images per class. There are 50,000 images for training and 10,000 for testing [3].

Each of the five training batches and the test batch contains 10,000 images. The test batch consists of exactly 1,000 images randomly selected from each category. The training batches contain the remaining images in a random order; however, some training batches may contain a disproportionate number of images from one class. Each training batch contains exactly 5000 images from its respective class [3]. Figure 2.3 shows the classes and some examples in the dataset.



.

Figure 2.3: Classes and their Examples in CIFAR-10 [3]

2.2 Coarse-Grained Reconfigurable Array

In the past few decades, a variety of hardware, such as GPU, FPGA, and ASIC, have emerged to address various issues; nevertheless, each of these hardware types has disadvantages when it comes to DNNs and other data-intensive applications.

To bridge the enormous power/performance gap between ASICs and general-purpose processors, a novel architecture platform that can be effectively adapted to a wide variety of applications in a domain or set of domains is urgently needed. CGRA, which is generically described as reconfigurable architectures that leverage hardware flexibility to adapt the data-path at runtime to the application, was proposed for this purpose [23].

A very popular CGRA architecture named `ADRES` [4], is shown in Figure 2.4. The CGRA shares resources with a tightly coupled very long instruction word (VLIW) processor, including the register file and VLIW functional units (FUs), and may thus operate in either VLIW or CGRA mode. In CGRA mode, the RA has access to the VLIW's register file, hence decreasing the cost of transfer [23].

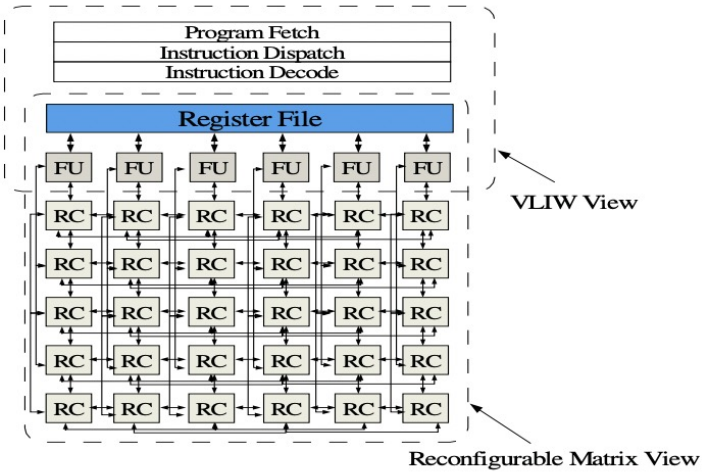


Figure 2.4: ADRES core [4]

Another widely studied CGRA architecture is `MorphSys`. Unlike `ADRES` which leverages instruction-level parallelism, `MorphSys` exploits data-level parallelism to accelerate applications [24].

2.3 PULPissimo

`PULPissimo` is the microcontroller architecture of the most recent `PULP` chips, developed by `ETH Zurich` and the `University of Bologna` as part of their continuous "PULP platform" partnership [5]. It is a single-core platform that used as the main System-on-Chip controller for all recent multi-core PULP chips, handling autonomous I/O, advanced data pre-processing, external interrupts, etc. The block diagram of `PULPissimo` is shown in Figure 2.5 [5].

Since it is an open-source project, it was chosen as the target platform for our design. Consequently, the design should be capable of coupling with the `PULPissimo` platform.

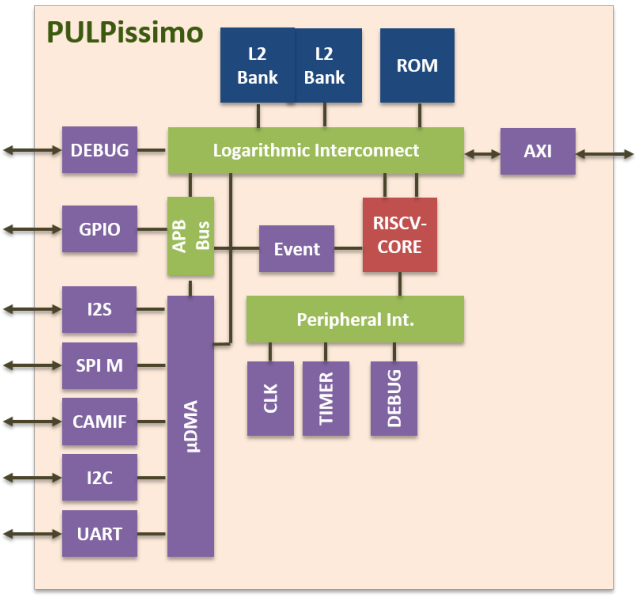


Figure 2.5: Block Diagram of PULPissimo [5]

Chapter 3

Hardware Architecture and Configuration

This Chapter elaborates on the NDP hardware architecture. In addition, it describes the design's setup and how this configurable component helps in the mapping of DNN algorithms.

3.1 Hardware Architecture Overview

The architecture was designed to be a general purpose near data processor and has the potential to be integrated within `PULPissimo`. The block diagram of the design is shown in Figure 3.1. The architecture of the design consists of four major components, which are `SRAM`, `Crossbar`, `Controller` and `NDP Unit`.

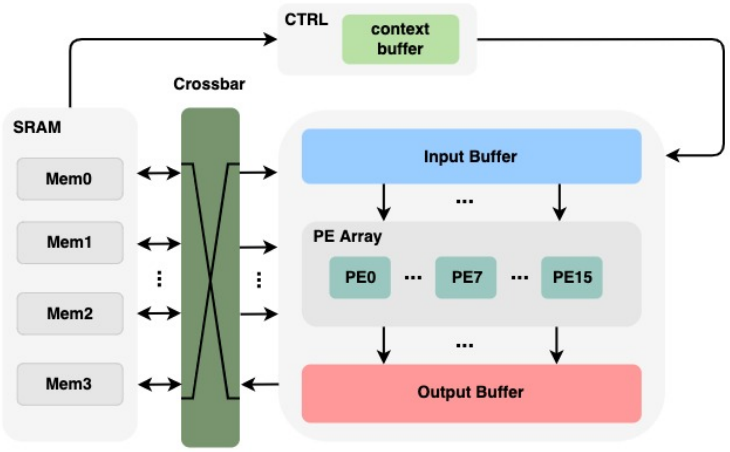


Figure 3.1: Block Diagram

As the architecture was designed for the `PULPissimo` platform, the storage element should correspond to that depicted in Figure 2.5. In other words, the `SRAM` in Figure 3.1 must have the same width as the 32-bit L2 bank in `PULPissimo` from MCU perspective.

`Crossbar` is the connection between `SRAM` and `NDP Unit (NDPU)`, and it can be configured to accommodate various input data schemes. It retrieves data from

SRAM and assigns them to the NDPU according to the Crossbar configuration.

Controller contains a context buffer that stores configuration data and distributes them to configurable components. In addition, it generates the address signals, write-enable signals, valid signals, etc., to help with loading data and computation inside processing units (PEs). This functionality is achieved through the use of a finite state machine (FSM).

Finally, the core of the design, NDP Unit (NDPU), is a computation unit that is made up by Input Buffer, PE Array and Output Buffer. Input Buffer retrieves data from Crossbar and transmits them to the corresponding processing element, whereas Output Buffer does the opposite. As for the PE Array, it is comprised of 16 PEs that are capable of performing multiple arithmetic and logical operations like multiply–accumulate (MAC), multiply (MUL), ADD, XOR, etc.

3.2 Memory Organization

Figure 3.2illustrates the memory architecture from the viewpoint of the MCU. The SRAM is divided into four submodules since each PE has 4 inout ports, which are **InputA**, **InputB**, **Psum** and **Result**. The correspondence between memories and inout ports is not fixed, for example, Mem0 does not necessarily need to be assigned to **inputA**. In addition, each submodule is subdivided into 16 cuts to match the bandwidth of the PE array. Thus, these 16 PEs can simultaneously load and store data. Although there are four memories in the design, each with 16 cuts, the MCU can only access one entry per clock cycle. Therefore, it can be interpreted as a 32-bit L2 cache, as depicted in Figure 2.5.

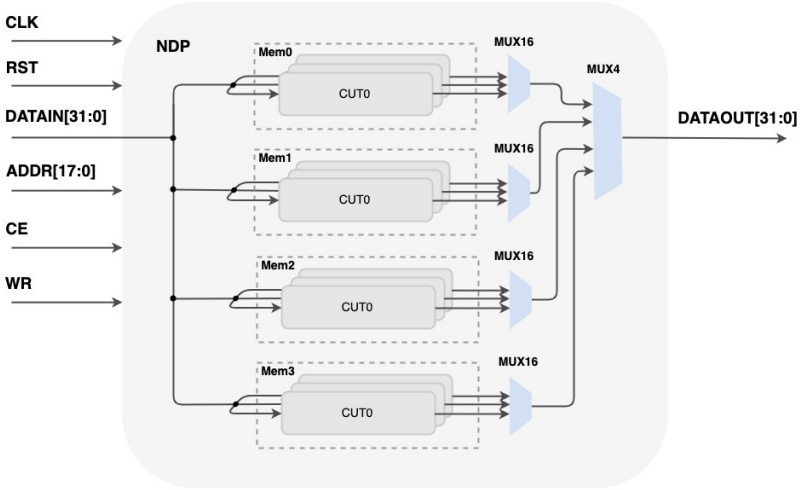


Figure 3.2: Memory architecture from MCU perspective

However, from the perspective of the NDPU, the memory access scheme is quite different. Figure 3.3 illustrates the memory architecture as seen by the NDPU. The memory access scheme can be roughly divided into 3 types: idle, setup and

`operation`. When the system is in `idle` mode and if the `NDPU` is deactivated, only the MCU can access the memory, which is depicted in Figure 3.3. Once the **is_config** signal is asserted, the system enters the `setup` state, during which configuration data are read from the `SRAM` and stored in the `Context buffer`. In theory, the `NDPU` can simultaneously access all sixteen cuts in each memory, sixty-four cuts in total, to retrieve 2048-bit configuration data in this state. However, 512 bits are sufficient for system configuration in our case. Thus, only one memory (`MEM3` in Figure 3.3) is allocated with these data. This state lasts for one clock cycle and is followed by an `operation` state in which the `Crossbar` connects memories to a bidirectional, multichannel data bus.

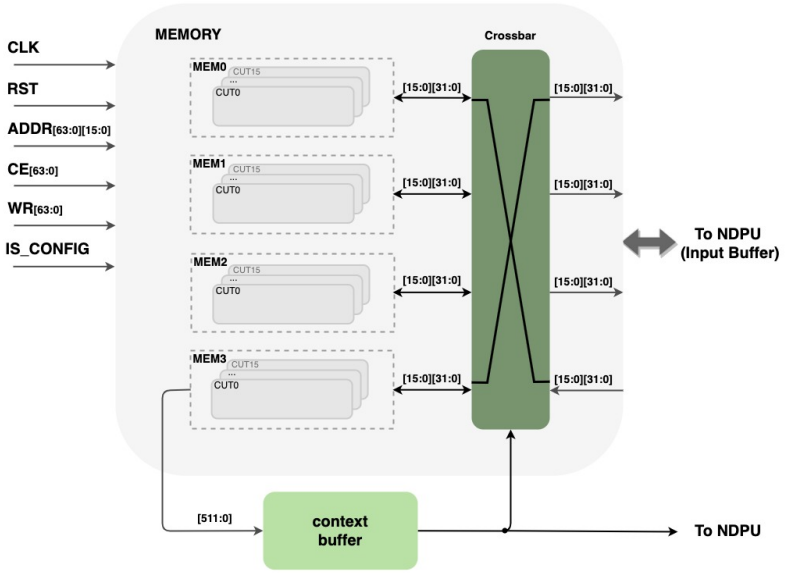


Figure 3.3: Memory architecture from NDPU perspective

3.3 Crossbar

As the design is intended to support multiple algorithms with distinct data allocations, multiple data allocations must be supported. In order to adopt different memory layouts, a bidirectional interconnection between `SRAMs` and `NDPU` is required. As illustrated in Figure 3.4, the `Crossbar` consists primarily of some mod-16 multiplexers with the select signal from `Context buffer`. With dedicated configuration, the target memory banks can be connected to the `NDPU`, allowing all `PEs` access to each memory bank.

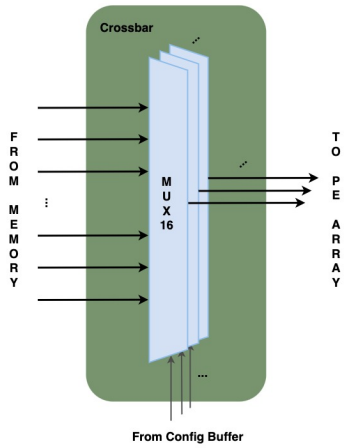


Figure 3.4: Simplified architecture of Crossbar

3.4 Controller

`Controller` is used to configure the design at three levels, as shown in Table 3.1: PE array level, mode level, and PE level. The layout of all the configuration parameters is depicted in Figure 3.5.

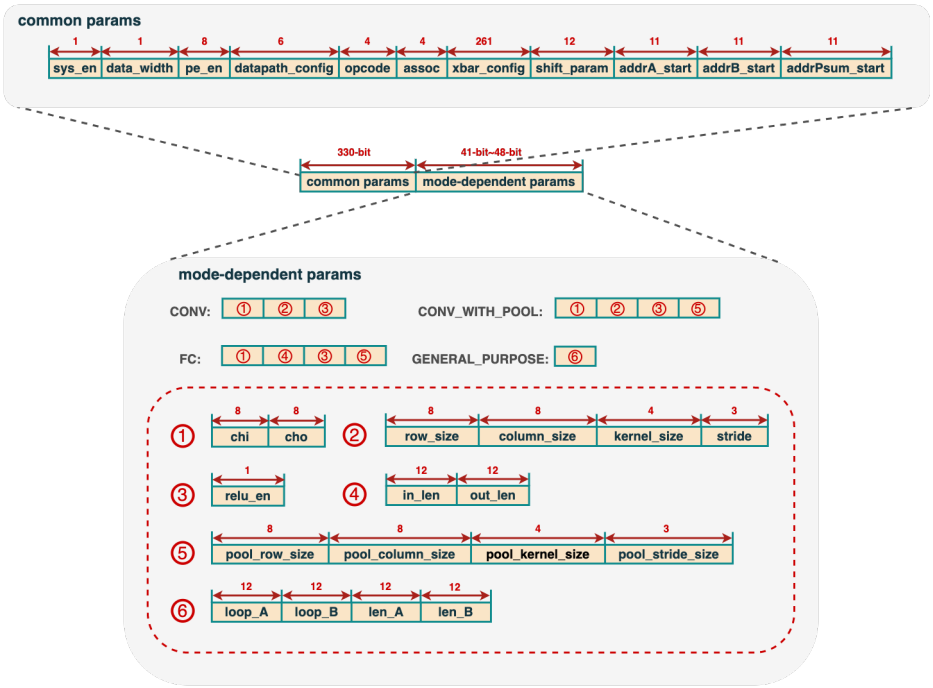


Figure 3.5: Configuration parameters layout

Table 3.1: Three level configuration hierarchy: PE array parameters, mode parameters, PE parameters

Configuration Level	Parameter	Description
PE Array	*sys_en*	Systolic enable
	pe_en	PE enable
	assoc	PE grouping
	xbar_config	Crossbar configuration
	addrA_start	Start address of inputA
	addrB_start	Start address of inputB
	addrPsum_start	Start address of Psum
Mode_CONV	*chi*	# of input channel
	cho	# of output channel
	row_size	Horizontal size of ifmap
	column_size	Vertical size of ifmap
	kernel_size	Filter plane size
	stride	Convolution stride
	relu_en	ReLu enable
Mode_POOL	*Pool_row_size*	Horizontal size of pooling
	Pool_column_size	Vertical size of pooling
	Pool_kernel_size	Size of pooling kernel
	Pool_stride	Pooling stride
Mode_FC	*in_len*	# of input neurons
	out_len	# of output neurons
	relu_en	ReLu enable
Mode_GP	*loop_A*	Loop cycles for input A
	loop_B	Loop cycles for input B
	len_A	Length of input A
	len_B	Length of input B
PE	*data_width*	Bit-width select
	datapath_config	Computing pattern
	opcode	ALU operation code
	shift_param	Parameters for shifter

3.4.1 PE Array Level

At the PE array level, there are primarily seven configuration parameters: *sys_en*, *pe_en*, *assoc*, *xbar_config*, *addrA_start*, *addrB_start*, and *addrPsum_start*. The *pe_en* is used to control power-saving clock gating at the `PE` level The *assoc* and *sys_en* define the grouping pattern of `PEs` and the data flow fashion within `Input Buffer` and `Output Buffer`, respectively. As described in Section 3.3, the *xbar_config* determines the source of inputs and destination of outputs for

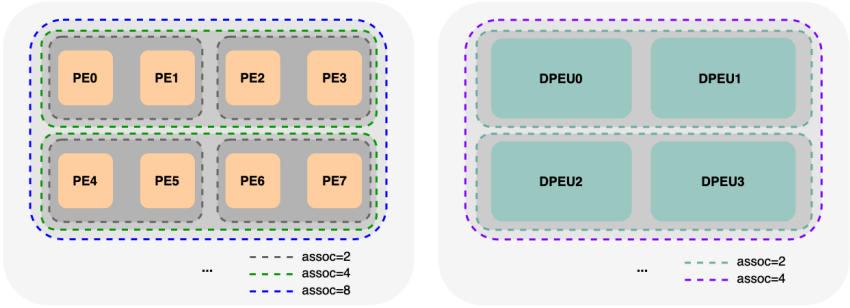
`PE Array`. *addrA_start*, *addrB_start*, and *addrPsum_start* are required as base addresses since data for different layers may be saved in the same memory block.

Associativity

To further increase the flexibility of the design, *associativity (assoc)* is introduced to the design. Due to the limited data parallelism in some tasks, it may not be possible for all `PEs` to collaborate on a single task. *Assoc* is then proposed to aid in the grouping and assignment of `PE` tasks.

By configuring *assoc*, we can decide the number of members in each `PE` set or `DPEU` set. However, due to the hardware constraints, our *assoc* only supports numbers that are the powers of two and range from 1 to 16 for `PE` set or 1 to 8 for `DPEU` set. The detailed grouping pattern is presented in Figure 3.6.

To avoid read-and-write conflicts when accessing the same memory blocks, the `PEs` or `DPEUs` in the same set are activated sequentially in a systolic manner. Section 4.1 will expand on the usage of *assoc*.



(a) 8-bit data mode **(b)** 16-bit data mode

Figure 3.6: Example of PE grouping and DPEU grouping

3.4.2 Mode Level

Although the hardware is to be general purpose, several optional modes are provided to optimize DNN applications. There is a total of four modes of operation: `Convolution` mode, `Convolution with Pooling` mode, `Fully Connected` mode and `General Purpose` mode.

The mode level parameters are used to control the processing of one specific mode, achieved by using finite state machine(FSM), as seen in Figure 3.7. The default state of the `Controller` is `IDLE` state, in which `NDP` is disabled. Once the configuration data has been written to `MEM3` and **ndp_en** signal has been pulled high, the `Controller` will transition to `CONFIG` state, where three-level configuration parameters will be loaded and decoded. The `Controller` then enters into `EXE` state to generate addresses based on the mode level parameters, and will not shift to the subsequent state until **exe_done** is set to '1'. To better illustrate mode level configuration, the address generation patterns of three representative modes are expressed by the pseudo code.

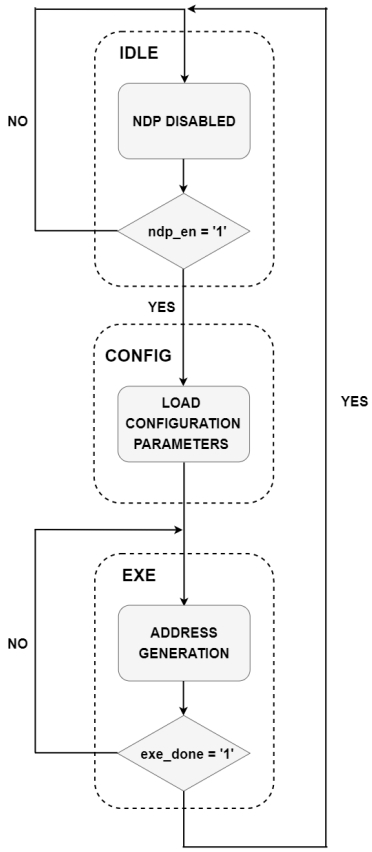


Figure 3.7: The `Controller` FSM chart

A. Convolution with Pooling Mode

`Convolution with Pooling` (`CONV_POOL`) mode is proposed to accelerate convolution layers in CNN algorithms. Since the shape of the layer can vary significantly between CNNs, all the shape parameters in our design are configurable to better support general purpose acceleration.

As seen in Algorithm 1, the addresses of the data needed for computation in `CONV_POOL` mode are generated based on the layer shape parameters like chi, $kernel\_size$, $stride$, $pool\_kernel\_size$, $pool\_stride$, etc., which are listed in Table 3.1. For the sake of brevity, only the address generation pattern of constructing a single output feature map (ofmap) channel is illustrated in the algorithm.

B. Fully Connected Mode

In `Fully Connected` (`FC`) mode, large matrix multiplication is divided into smaller ones to accelerate the computation.

The address generation pattern is made up of two loops controlled by mainly three parameters: in_len, out_len and $assoc$, as can be seen in Algorithm 2. The

inner loop repeats in_len times to complete the vector-matrix multiplication of one output neuron. As for the outer loop, rather than executing out_len times, it executes out $out\_len/assoc$ times to compute all the output neurons. This is accomplished by utilizing assoc `PEs` or `DPEUs` concurrently performing one task.

C. General Purpose Mode

In addition to CNN layers, our `NDP` also supports acceleration for some other operations, for example, distance calculation, bit operations, etc. To achieve this, we propose a `General Purpose` (`GP`) mode with four parameters: $loop\_A$, $loop\_B$, len_A, and len_B. The usage of these parameters is presented in Algorithm 3.

$loop\_A$ and $loop\_B$ refer to the update frequency of **inputA** and **inputB**, which means **inputA** and **inputB** will be updated every $loop\_A$ cycles and $loop\_B$ cylces, while len_A and len_B represent the number of **inputA** and **inputB**, respectively. For instance, suppose we wish to calculate all distances between 1024 data points and four centroids using eight `DPEUs`. In that case, we can set $loop\_A, loop\_B, len\_A, len\_B$ to 1, 1024/8, 1024/8, and 4, respectively, as demonstrated in the mapping example in Section 4.2.

3.4.3 PE Level

The PE level configuration determines the input data width, `PE`'s functionality, the precision of outputs, etc. By configuring $data\_width$, we can choose between 8-bit and 16-bit input data width. The combination of $datapath\_config$ and op_code specifies `PE`'s functionality, including multiply-and-accumulate (MAC), squared euclidean distance (SED), and some other bit operations. The detail of `PE`'s functionality will be illustrated in Chapter 4. The configurable precision of fixed-point outputs is achieved by shift operation controlled by $shift\_param$.

3.5 Near Data Processing Unit

As mentioned before, the `NDPU` is the computational unit of the design. It can be divided into three parts, `PE array`, `input buffer` and `output buffer`. The `PE array` contains 16 PEs that do arithmetic or logic operations independently, while `input buffer` and `output buffer` are responsible for the data streaming.

3.5.1 PE Array

The simplified architecture of the `PE array` is shown in Figure 3.8. In order to leverage power and area efficiency, an 8-bit fixed-point computation is implemented in `PE array`. However, for some applications like DNNs, 8-bit computation is not sufficient. For example, quantization errors can easily get accumulated if the results go through a couple of layers in DNNs. Therefore, as shown in Figure 3.8, two `PEs` are organised as a `Dual Processing Element Unit (DPEU)` to realize the mix precision fixed-point computation. The details in the mixed precision fixed-point implementation will be introduced in Section 3.5.3.

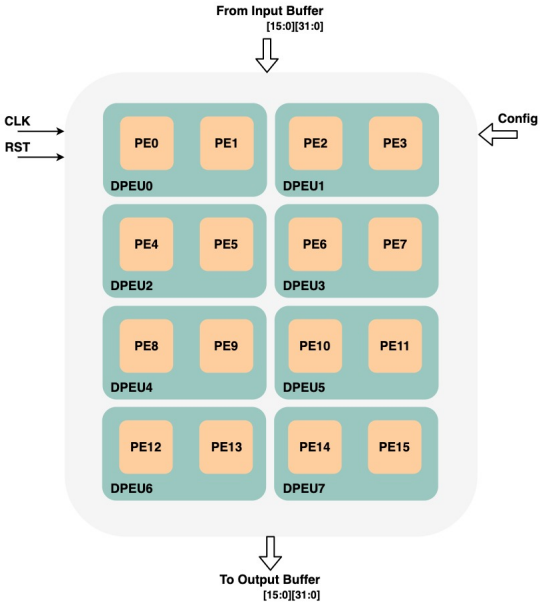


Figure 3.8: PE array

3.5.2 Processing Element

The architecture of `PE` is illustrated in Figure 3.9a. There is a total of three `flip-flops`, one 32-bit for multiplication result buffering, one 32-bit and one 8-bit for intermediate value storage. The component outlined in purple and symbolized by 'x' represents a multiplier. `Multiplexers` and `MUL-ALU unit` can be configured to execute `MAC` operations, bit operations, or other types of computation. A `shifter` is placed before the result is delivered to the output for mix precision fixed-point implementation. It can also be configured to shift the result if it is necessary.

3.5.3 Dual Processing Element Unit

The implementation of bit-adaptive fixed-point is based on the distributive law of multiplication which is given by

$$A(16) * B(8) = 2^8 * A_{MSB}(8) * B(8) + A_{LSB}(8) * B(8) \,, \tag{3.1}$$

where A is the input, B is the weight, A_{MSB} and A_{LSB} are the most significant bits and least significant bits of **inputA** respectively, and following numbers in parentheses represent the number of bits. For the implementation of an `MAC` operation, Equation 3.1 can be revised to

$$\sum A(16) * B(8) = 2^8 * \sum A_{MSB}(8) * B(8) + \sum A_{LSB}(8) * B(8) \,. \tag{3.2}$$

The architecture of `DPEU` is illustrated in Figure 3.9b, which is the implementation of Equation 3.2. Each `DPEU` has two Flip-Flops, two `PEs` and one extra adder

(depicted as the ALU shape with the 'ADD' symbol in Figure 3.9b). When the DPEU is in 16-bit configuration, PE0 and PE1 are responsible for the first and second terms of Equation 3.2 when the DPEU is in 16-bit configuration, and the result of PE0 is shifted left for 8-bits before it is available. In the end, the adder combines these two terms into a single result. Otherwise, DPEU is capable of performing two 8-bit operations in parallel.

3.5.4 Input/Output Buffer

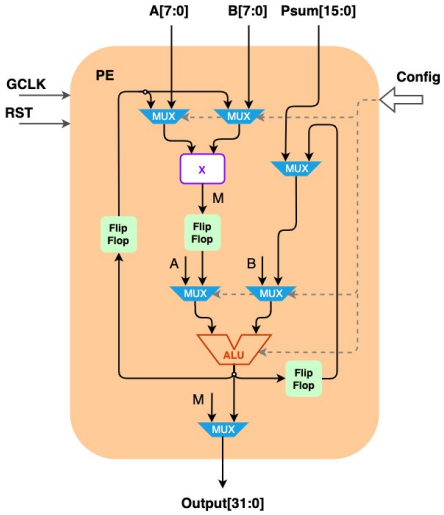
Figure 3.10 shows the block diagram of the input buffer. It can be configured to be systolic or independent depending on the applications. The systolic approach may be preferable when it comes to DNNs, as the memory bandwidth is limited.

In contrast, Output buffer is a simple module that is used to concatenate the results to match the width of SRAM and then store them.

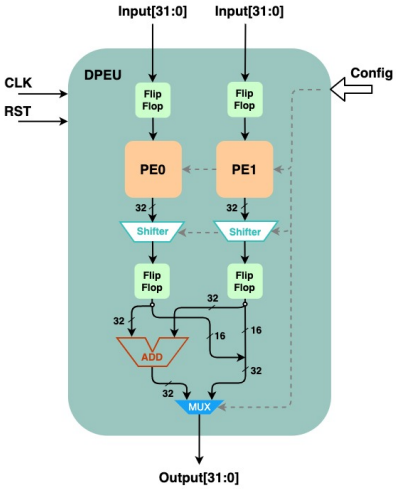
3.6 Clock Gating

Sparse activations arise spontaneously in DNNs for a variety of reasons. One reason is that many DNNs use the rectified linear unit (ReLU) as the activation function, which translates negative values to zero. In deeper layers, this sparsity tends to increase and can even surpass 90% [25]. The fact that many popular DNNs are autoencoders or generative adversarial networks (GAN) with decoder layers that use zero insertion to up-sample the input feature maps, resulting in over 75% zeros, is an additional factor that is becoming increasingly significant [25].

To reduce the power overhead caused by sparse matrix, clock gating is introduced to the system. As DNNs are the applications that are most likely to benefit from it, clock gating should be activated only when the PEs are doing MAC operation which is the dominant operation in DNNs. As shown in Figure 3.9a and Figure 3.11, when the zero detect logic encounters a zero from either **InputA** or **InputB** and the system is in MAC configuration, the gated clock signal GCLK fed to PEs is set to low to avoid unnecessary switching inside PEs, which in turn reduces power consumption.



(a) Processing Element Unit



(b) Dual Processing Element Unit

Figure 3.9: Architecture of PE and DPEU

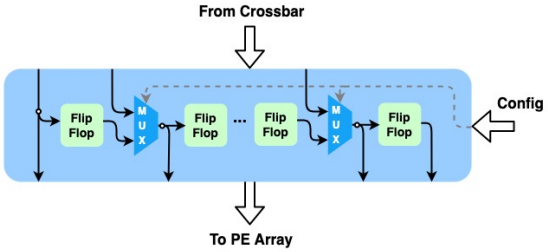


Figure 3.10: Input Buffer

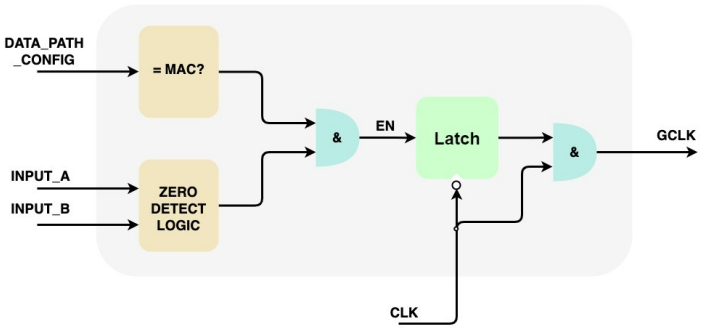


Figure 3.11: Clock gating circuit

Chapter 4
Algorithm Mapping Examples

This chapter demonstrates several strategies to map computing primitives in different algorithms to the proposed NDP. Furthermore, the data flow within the PE array and PE is also depicted to better illustrate the mapping strategy.

4.1 Mapping Strategy for CNN Algorithm

In a typical CNN architecture, there are several layers including the convolution layer, pooling layer, and fully connected layer. Each of these layers presents its own computational challenges. Consequently, strategies for effectively mapping these layers onto the architecture are of importance in order to achieve optimal performance.

4.1.1 Convolution Layer and Pooling Layer

Considering many CNN architectures, for example, LeNet-5 [26], AlexNet [20], VGG-16 [27], max-pooling layer directly downsamples the ofmaps from convolution layer, we propose an operation called CONV_POOL on the basis of CONV operation to combine the mapping for these two layers. Thus, redundant memory accesses and data transfer can be avoided. When executing a CONV_POOL operation, each PE or DPEU is configured to perform MAC operation (shown in Figure 4.1) to accommodate the computation of one ofmap channel, and will not move to another until the current one has been obtained. Figure 4.2 illustrates the mapping strategy by using an example.

Suppose a convolution layer with a kernel size of 3 x 3, stride of 1 x 1, and four ofmap channels, followed by a pooling layer with max-pooling window size of 2 x 2 and a step size of 2 x 2. The data width of the input fmap for the layer is 16 bits. In this instance, we can divide the DPEUs into two sets by setting *assoc* to 4. Each set is responsible for the construction of a single ofmap, so the batch size of the input feature map (ifmap) can be increased to two in order to maximize the use of computing resources.

The ifmaps, kernels, and biases are stored in different memory banks to enable parallel processing. In order to minimize the movement of input data, the ifmap pixel flows in the DPEU set in a systolic fashion so it can be reused across all the filters in the same set. Each DPEU can begin processing the moment any ifmap

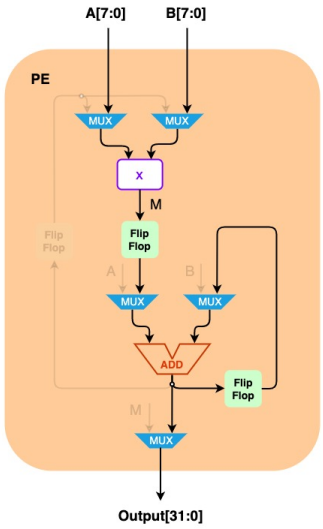


Figure 4.1: Dataflow of `MAC` operation in PE with Psum omitted

pixel or kernel weight arrives. In the interest of simplicity, we will only describe the first four steps of one `DPEU` set's computation flow.

`Step #0`: At the beginning, the ifmap pixels encompassed by the blue rectangle are flattened and sequentially loaded into the `DPEU` set. Meanwhile, corresponding kernel weights and biases are also read from the associated memory banks and written to the `Input Buffer` for use in generating the first ofmap pixel for each channel. Once the computation is completed, the ofmap pixel data will be sent to the output registers within `Controller` unit for further processing, such as ReLU and max-pooling. The filters will then shift to the right by one column and cover the red rectangle-encircled area.

`Step #1`: In this step, the `DPEU` set first loads the input data, kernel weights, and biases again to compute new ofmap pixels. Second, when the new ofmap pixels are produced and forwarded to the ReLU and max-pooling unit, the values stored in the output registers will be updated if the incoming values are greater. Finally, since the ofmap pixels prepared for max-pooling must be consecutive, the filters will move down by one row and left by one column to the area outlined in purple rather than continuing to move to the right as is customary for a convolution kernel.

`Step #2,3`: These two steps involve the generation of the third and fourth ofmap pixels for each channel. Once the values stored in the output registers have been updated to the greatest, they will be written to the `Output Buffer`. The associated registers will then be cleared for storing the intermediate results of the subsequent four-step computation.

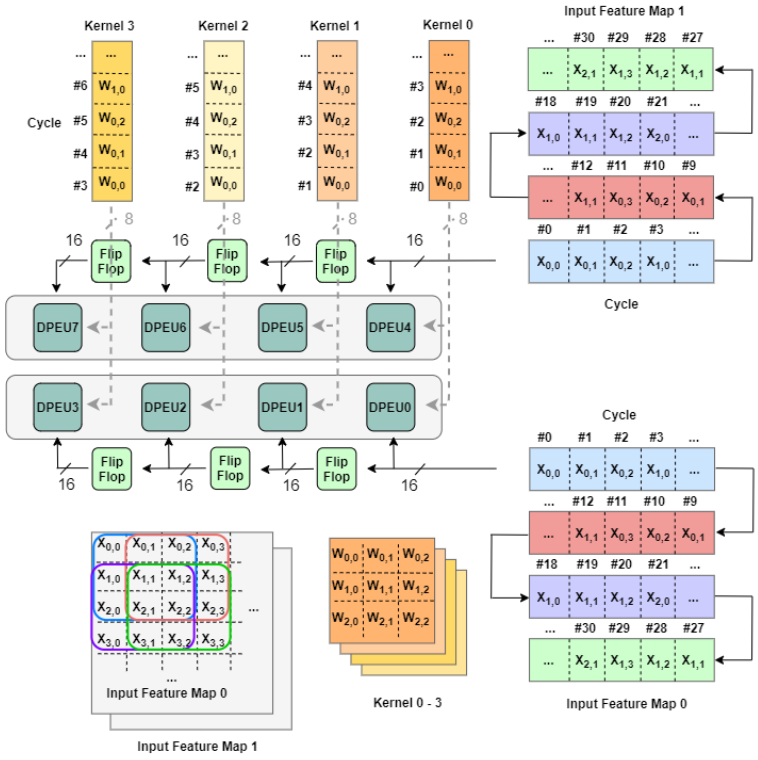


Figure 4.2: Computing flow of `CONV_POOL` operation

4.1.2 Fully connected Layer

The predominant computation pattern of `FC` operations is a matrix multiplication. In order to maximize data parallelism, the input vector, weight matrix, and biases are partitioned and stored in separate memory banks. By configuring the `Crossbar` and `Input Buffer`, each `PE` or `DPEU` can access the input data in 8-bit or 16-bit format from the same memory bank sequentially or from different banks simultaneously. During computation, each `PE` or `DPEU` takes care of one output neuron by performing MAC operation, and will not switch to another until the current output has been obtained. For instance, the operations in the first fully connected layer of the LeNet-5 architecture [26] can be expressed as:

$$O_{120\times1} = W_{120\times400} \times I_{400\times1} + B_{120\times1} \ . \tag{4.1}$$

The weight matrix W and biases B are partitioned into eight submatrices to satisfy the memory bank's capacity constraints and maximize the use of computing resources. Thus, the operations can be completed in 15 iterations, with eight output neurons are computed per iteration. Figure 4.3 depicts the computing flow of the first loop of `FC` operation. First off, at `cycle #0`, `DPEU0` performs multiplication between the first element of the input vector I and the first element

of the weight matrix W Row 0, while adding the first element of the biases. At `cycle #1`, `DPEU0` performs partial sum (Psum) accumulation by adding the result of multiplying I_1 by $W_{0,1}$ to the psum stored at its local register. Meanwhile, `DPEU1` loads I_0, $W_{1,0}$, and B_1 to do the multiplication and addition. In the next few cycles, all `DPEUs` will be activated and do the similar operation as in `cycle #0` and `cycle #1`. After a certain number of cycles, when the vector-matrix multiplication associated with the current eight output neurons has been computed, the results will be written to the `Output Buffer` for further processing, and all the `DPEUs` will move to next eight neurons.

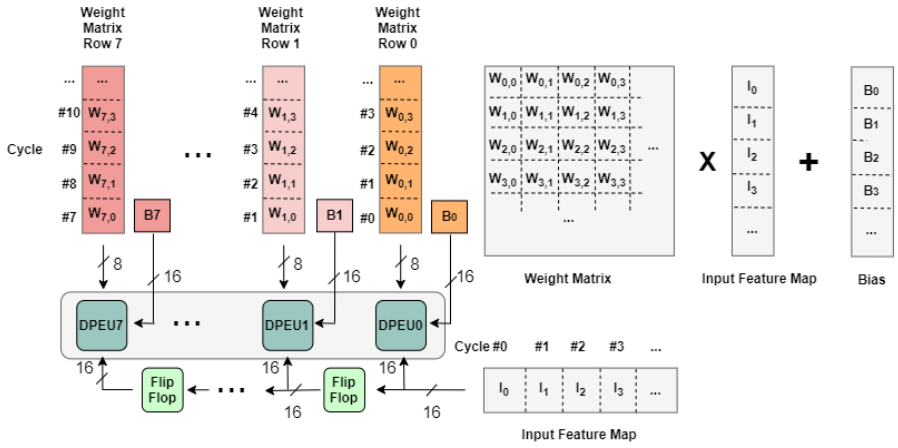


Figure 4.3: Computing flow of `FC` operation

To reduce memory access, the input vector is broadcast to the `DPEU` set systolically during the process. Additionally, the weights can be reused by adopting a batch strategy, which can be achieved by rerouting the input vector source and output vector destination.

4.2 Mapping Strategy for K-Means Algorithm

The most computation-heavy part of the K-Means algorithm is distance calculation. A common way to define the distance between a data point and a centroid is Euclidean Distance (ED) [28]. Here we adopt the Squared Euclidean Distance (SED) method, as compared with ED, it is more hardware-friendly while the output is not affected.

As shown in Figure 4.4, the input data points are partitioned into eight groups and stored separately in different memory banks, allowing all the `PEs` to participate concurrently in performing distance calculations.

Due to the hardware resource limitations, our `NDP` currently only supports two-dimensional data points. Each data point is 16 bits wide and consists of two 8-bit one-dimensional features. Given a fixed number of input data points that need to be partitioned into four clusters, Figure 4.5 depicts the computation flow of

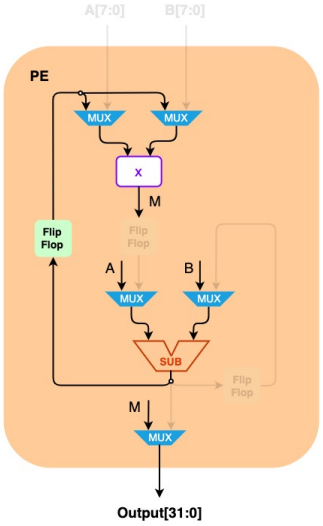


Figure 4.4: Dataflow of distance calculation operation in PE

distance calculation. It is clear that different groups of data points are assigned to different `DPEUs`, while the same centroids are shared among all the `DPEUs`. First, one data point and one centroid are input into the `DPEU`; second, each `PE` uses one-dimensional features of the data point and centroid to compute the squared difference using the ALU and multiplier; and third, the adder sums the squared differences of each dimension to obtain the squared distance. Each cycle, the squared distances between eight data points and one centroid are obtained and forwarded to the `Output Buffer`. The operation will continue until the distances between all the data points and centroids have been computed.

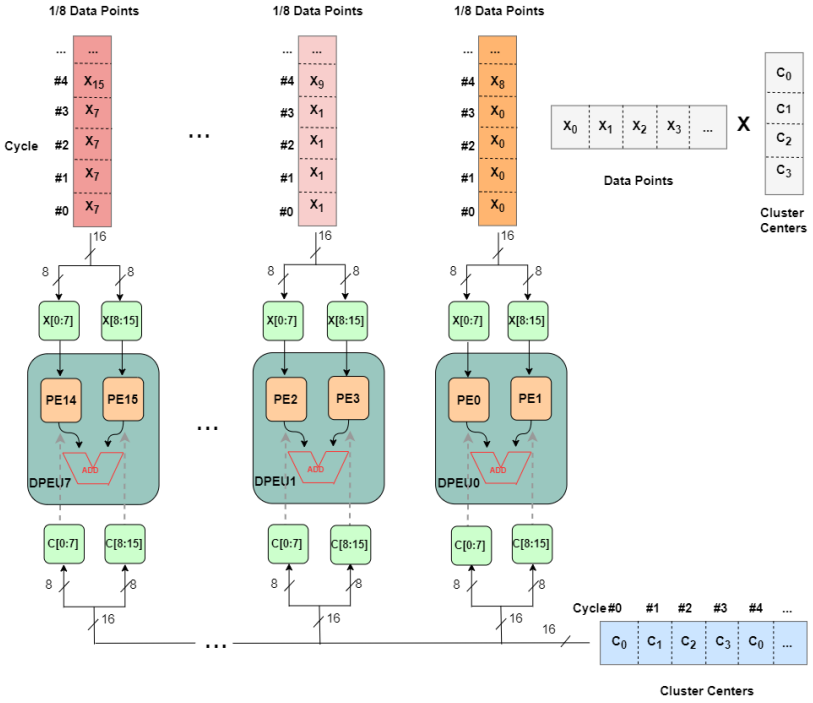


Figure 4.5: Computing flow of distance calculation

Chapter 5
Results and Analysis

This chapter presents the results of synthesis, place and route (PNR), and post-layout power analysis. CNN and K-Means Algorithms are selected as benchmarks to analyze the performance and energy efficiency of the proposed design.

5.1 Synthesis Results

Synthesis relates the conceptual description of the design's desired logic functions to their actual physical architecture elements. This section specifies the process technology and constraints needed for synthesis and gives the result of synthesizing the `NDP` with L2 Cache.

To achieve a good tradeoff between power consumption and performance, STM65nm LPLVT technologies are applied to the design. Compared with HVT, LVT technology introduces shorter switching delays which can lead to higher operating frequencies. The design was synthesized with the constraints listed in Table 5.1. Constraints for external input and output delay are defined as 200 ps. The synthesis timing report indicates a timing slack of 1391 ps, which satisfies the setup timing constraint.

Table 5.1: Synthesis constraints

Supply Voltage	1.0 V
Temperature	25°C
Clock Rate	167 MHz
Clock Uncertainty	100 ps
Clock Rise Time	50 ps
Clock Fall Time	55 ps
Clock Source Latency	100 ps
Clock Network Latency	100 ps

Table 5.2 reveals that L2 Cache occupies nearly 86% of the area. The PAD ranks second, accounting for 7.8% of the total area. Compared to memories, the

Table 5.2: Synthesis area report

Library	**Instances**	**Area[um^2]**	**Instances%**
CLOCK65LPLVT	865	1517.880	1.1
CORE65LPLVT	77825	319193.160	98.7
PAD	86	385280.000	0.1
SPHS_151013	64	4198157.000	0.1
TOTAL	78840	4904148.000	100.0

area of the NDP core is only 0.32 mm2 (6.5% of the total area), which is relatively small.

5.2 Implementation Results and Evaluation

The proposed NDP architecture shown in Figure 5.1 was placed and routed using Cadence SoC Encounter and STM 65nm LPLVT technology. Table 5.3 provides an overview of the NDP architecture specifications.

Table 5.3: NDP architecture specifications

Technology	STM 65 LPLVT
Chip Size	3.1 mm x 3.0 mm
Core Size	3.0 mm x 2.8 mm
SRAM	512 KB
Num. of PEs	16
Supply Voltage	1.0 - 1.1 V
Clock Rate	100 - 167 Mhz
Peak Performance	6.67 GOPS
Operation Mode	8/16 bit
Supported Convolution Layer Shapes	Kernel Width: 1 - 16 Kernel Height: 1 - 16 Num. of Filters: 1 - 256 Num. of Input Channel: 1 - 256 Num. of Output Channel: 1 - 256 Stride: 1 - 8
Supported MaxPooling Layer Shapes	Kernel Width: 1 - 16 Kernel Height: 1 - 16 Stride: 1 - 8
Supported FC Layer Shapes	Input length: 1 - 4096 Output length: 1 - 4096
Other Operations	Distance Calculation, Relu, Bit Operations

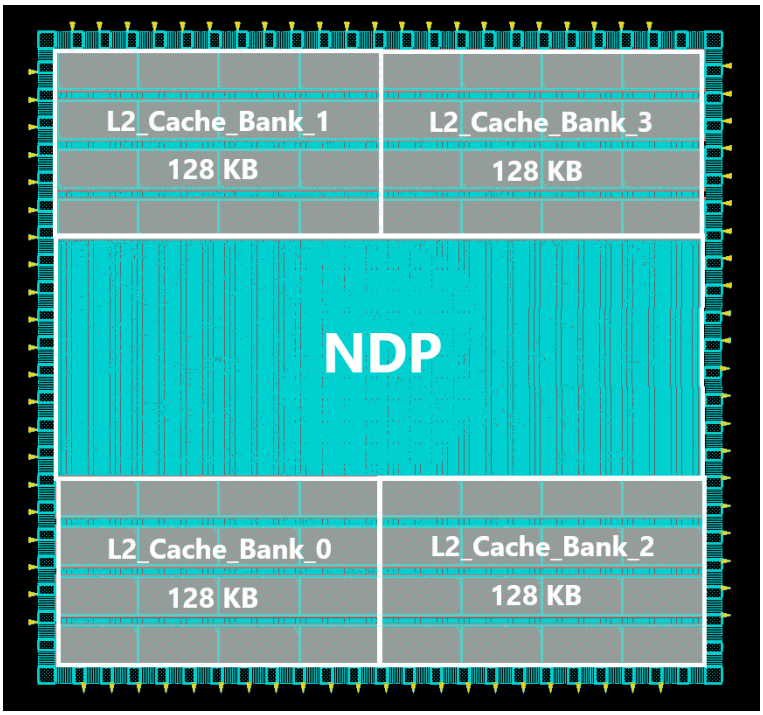


Figure 5.1: Floorplan of NDP with L2 cache

To evaluate the NDP architecture performance and power efficiency, the LeNet-5 [26] CNN architecture (Figure 5.2) and K-Means clustering algorithm are selected as benchmarks. The input frames are from CIFAR-10 Dataset for LeNet-5 CNN architecture.

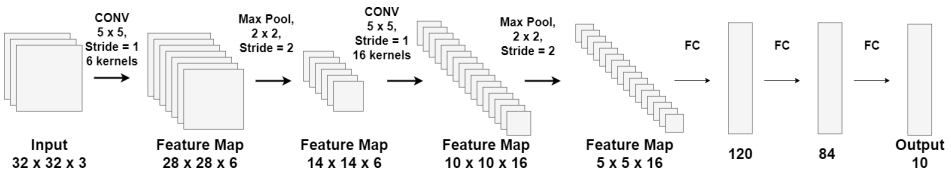


Figure 5.2: LeNet-5 CNN architecture

5.2.1 Performance and Energy Efficiency Evaluation

A. LeNet-5

In LeNet-5, the following parameters are defined to describe the CONV and FC layers. N_i and N_o represent the input and output channel numbers in the CONV layer, or the input and output lengths in the FC layer. The sizes of the ifmap and the ofmap are $R_i \times C_i$ and $R_o \times C_o$, respectively. The size of the kernel is $K \times K$, and the stride is S. In addition, the batch size (Bs) is introduced to improve data parallelism. As a result, the total number of operations in the CONV layer is

$$N_{op} = 2 \times N_i \times N_o \times R_o \times C_o \times K \times K \times Bs \,, \tag{5.1}$$

and the total number of operations in the FC layer is

$$N_{op} = 2 \times N_i \times N_o \times Bs \,. \tag{5.2}$$

Table 5.4: Shape parameters for CONV layers and FC layers in LeNet-5

Layer	N_i	N_o	R_i	C_i	R_o	C_o	K	S
CONV1	3	6	32	32	28	28	5	1
CONV2	6	16	14	14	10	10	5	1
FC1	400	120	-	-	-	-	-	-
FC2	120	84	-	-	-	-	-	-
FC3	84	10	-	-	-	-	-	-

Table 5.5 displays the performance breakdown of the five layers in LeNet-5 at 1V with the core running at 167 MHz. The power was obtained by performing a time-based analysis on the Value Change Dump (VCD) files generated while running post-layout simulations. It is apparent that power consumption decreases

with increasing layer depth, which can be attributed to clock gating reducing switching activities in deeper layers. As for the results of total latency, they were obtained by performing the post-layout simulation in different scenarios. The instruction loading and pipeline latency can be ignored compared to the processing latency.

Table 5.5: Performance breakdown of the five layers in LeNet-5 at 1V. The core runs at 167 MHz.

Layer	Bs	Bit Width (fixed-point)	Power(mW)	Total Latency (us)	Num. of OPs
CONV1	2	8-bit	10.07	353	1411200
CONV2	1	16-bit	8.77	180	480000
FC1	1	16-bit	6.87	36.1	96000
FC2	1	16-bit	6.87	8	20160
FC3	1	16-bit	5.67	1.1	1680
Total	-	-	9.57	578.2	2009040

Table 5.6 displays the measured throughput and energy efficiency. Since the data width differs between layers and PE utilization cannot reach 100% in all layers, the actual throughput is less than the maximum throughput. Take the CONV1 layer as an example; if the batch size is one, there will be only six output channels occupying six PEs, resulting only 37.5% PE utilization. To achieve this, two input frames can be processed simultaneously, doubling PE utilization. The vacant PEs will be deactivated to conserve power.

Table 5.6: Performance and energy efficiency analysis on different LeNet-5 Layers

Layer	Num. of Active PEs	Throughput(GOPS)	Energy Efficiency (TOPS/W)
CONV1	12	3.998	0.397
CONV2	16	2.667	0.304
FC1	16	2.660	0.387
FC2	16	2.520	0.367
FC3	16	1.527	0.269
Total	-	3.475	0.363

B. K-Means

Assuming there are `n` input points that need to be partitioned into `M` clusters, and each point is `d`-dimensional. The number of subtraction, multiplication, and addition operations required to calculate all the distances is $d \times M \times n, d \times M \times n, (d-1) \times M \times n$, respectively, for a grand total of $(3 \times d \times M \times n - M \times n)$ operations. As shown in Figure 5.3, if $d = 2, M = 8, n = 1024$, the total number of operations is:

$$3 \times 2 \times 8 \times 1024 - 8 \times 1024 = 40960 \tag{5.3}$$

Figure 5.3: K-Means Clustering Algorithm example

The performance and energy efficiency analysis on K-Means Clustering Algorithm are shown in Table 5.7.

Table 5.7: Performance and energy efficiency analysis on K-Means Clustering Algorithm

Bit Width(fixed-point)	8-bit
Power(mW)	29
Total Latency(us)	6.2
Throughput(GOPS)	6.606
Energy Efficiency(TOPS/W)	0.228

5.2.2 Comparison with Other Designs

A. CNN Algorithm

The proposed NDP architecture is compared to various MCU and FPGA implementations in Table 5.8. Compared to GAP-8 MCU in [29], the NDP offers 1.6x and 11x higher performance and energy efficiency, respectively. The implementations described in [30] and [31] can achieve 5.8x and 11.5x higher performance compared to the NDP, but at the cost of 12x and 26x lower energy efficiency, respectively.

Table 5.8: Comparison with other MCU and FPGA implementations

Reference	**[29] GAP-8**	**[30]**	**[31]**	**This Work**
Platform	MCU	FPGA	APSoC	Accelerator (CGRA)
Core	8	-	-	1
Dateset	CIFAR-10	MNIST	CIFAR-10	
Network	CNN	LeNet-5		
Data width	INT-8	16-bit fixed-point	-	8-bit or 8/16-bit fixed-point
Clock [MHz]	170	-	650	167
Performance [GOPS]	2.14	20.3	39.88	3.475
Efficiency [GOPS/W]	32.2	30.03	13.99	363

B. K-Means Algorithm

As shown in Table 5.9, the CPU-FPGA implementations in [28] and [32] can achieve 1.2x–8.5x higher than the proposed design, and 730x better energy efficiency. Due to the hardware limitations, only the distance calculation in the K-Means algorithm can be mapped to the NDP. Therefore, comparing the NDP to other K-Means accelerators is not entirely fair. However, the NDP's scalability and low power consumption make it a good choice for accelerating the K-Means algorithm.

Table 5.9: Comparison with existing CPU-FPGA K-Means accelerators

Reference	**[28] Xeon+FPGA**	**[32] Xeon+FPGA**	**This Work**
n	2M	4096-65536	1024
M	8/16	256-1024	8
d	2/4	16	2
Performance	26.2-55.9 (GFLOPS))	7.6-10.6 (GFLOPS)	6.06 (GOPS)
Energy Efficiency	0.312 (GFLOPS/W)	0.312 (GFLOPS/W)	228 (GOPS/W)

Chapter 6
Conclusion

This Chapter draws a conclusion and describes the future work that can be done to improve the design.

6.1 Conclusion

This thesis proposes a near data processing architecture that is optimized for data-intensive applications and aims to solve a variety of problems. Although the proposed NDP was evaluated as an accelerator, it has the potential to couple with modern MCU platforms.

The design was implemented using STM 65nm LPLVT technology. Although dedicated pipeline scheme was added to the architecture, the maximum achievable clock speed is 167 MHz.

For the purpose of evaluating the design, two algorithms, CNN and Kmeans, were mapped onto the hardware with dedicated configurations operating at maximum frequency. Due to the limited bandwidth of SRAM, the proposed design cannot compete with FPGA solutions in terms of performance, but it outperforms them in terms of power efficiency. For CNN algorithm, the power efficiency can be increased by 12x and 26x compared to FPGA implementations, and 11x to MCU implementations. For K-Means algorithm, the proposed hardware surpasses other hardware by over 730x.

6.2 Future Work

There are many opportunities for further work, which can be concluded from three aspects. Flexibility would be enhanced by adding support for additional algorithms, such as the LUT-based activation function in CNN and distance calculation for multidimensional data points in K-Means. The second aspect to explore would be performance, which can be enhanced by adding more `PEs` maximize memory bandwidth utilization. The final one is energy efficiency, in which data-reuse can be investigated further to reduce energy consumption.

Bibliography

[1] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[2] P. Prinz, T. Crawford, J. Hennessy, and D. Patterson, "Computer architecture: A quantitative approach," 2018.

[3] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[4] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings 13*. Springer, 2003, pp. 61–70.

[5] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: an ultra-low-power pulpissimo soc in 22nm fdx," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. IEEE, 2018, pp. 1–3.

[6] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[7] C. Carvalho, "The gap between processor and memory speeds," in *Proc. of IEEE International Conference on Control and Automation*, vol. 5000, no. 10000, 2002, p. 15000.

[8] D. Etiemble, "45-year cpu evolution: one law and two equations," *arXiv preprint arXiv:1803.00254*, 2018.

[9] D. Agrawal, P. Bernstein, E. Bertino, S. Davidson, U. Dayal, M. Franklin, and J. Widom, "Challenges and opportunities with big data: A white paper prepared for the computing community consortium committee of the computing research association," *Computing Research Association*, 2012.

[10] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "A review of near-memory computing architectures: Opportunities and challenges," in *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018, pp. 608–617.

[11] M. Gao and C. Kozyrakis, "Hrl: Efficient and flexible reconfigurable logic for near-data processing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Ieee, 2016, pp. 126–137.

[12] D. G. Elliott, W. M. Snelgrove, and M. Stumm, "Computational ram: A memory-simd hybrid and its application to dsp," in *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*. IEEE, 1992, pp. 30–6.

[13] P. M. Kogge, "Execube-a new architecture for scaleable mpps," in *1994 International Conference on Parallel Processing Vol. 1*, vol. 1. IEEE, 1994, pp. 77–84.

[14] M. F. Deering, S. A. Schlapp, and M. G. Lavelle, "Fbram: a new form of memory optimized for 3d graphics," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 1994, pp. 167–174.

[15] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent ram," *IEEE micro*, vol. 17, no. 2, pp. 34–44, 1997.

[16] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "Flexram: Toward an advanced intelligent memory system," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 5–14.

[17] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors and Microsystems*, vol. 71, p. 102868, 2019.

[18] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 283–295.

[19] H. Lim and G. Park, "Triple engine processor (tep) a heterogeneous near-memory processor for diverse kernel operations," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 1–25, 2017.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[21] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[22] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[23] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2016, pp. 235–244.

[24] J. Davila, A. de Torres, J. M. Sanchez, M. Sanchez-Elez, N. Bagherzadeh, and F. Rivera, "Design and implementation of a rendering algorithm in a simd reconfigurable architecture (morphosys)," in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 2. IEEE, 2006, pp. 6–pp.

[25] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[26] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[28] T. S. Abdelrahman, "Cooperative software-hardware acceleration of k-means on a tightly coupled cpu-fpga system," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–24, 2020.

[29] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.

[30] W. Chen, H. Wu, S. Wei, A. He, and H. Chen, "An asynchronous energy-efficient cnn accelerator with reconfigurable architecture," in *2018 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2018, pp. 51–54.

[31] V. R. Laguduva, S. Mahmud, S. N. Aakur, R. Karam, and S. Katkoori, "Dissecting convolutional neural networks for efficient implementation on constrained platforms," in *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*. IEEE, 2020, pp. 149–154.

[32] M. A. Souza, L. A. Maciel, P. H. Penna, and H. C. Freitas, "Energy efficient parallel k-means clustering for an intel® hybrid multi-chip package," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2018, pp. 372–379.

Appendix A

Some extra material

Algorithm 1 Address Generation in CONV_POOL Mode

```
1: pool_row ← 0
2: pool_col ← 0
3: row ← 0
4: col ← 0
5: kernel ← 0
6: bias ← 0
7: while pool_row ≤ pool_row_size − pool_kernel_size do
8:     while pool_col ≤ pool_column_size − pool_kernel_size do
9:         for i ← 0, pool_kernel_size − 1 do                ▷ Pool kernel row
10:            for j ← 0, pool_kernel_size − 1 do            ▷ Pool kernel column
11:                for ii ← 0, kernel_size − 1 do            ▷ Kernel row
12:                    for jj ← 0, kernel_size − 1 do        ▷ Kernel column
13:                        for kk ← 0, chi − 1 do            ▷ Input channel
14:                            /*Ifmap address calculation*/
15:                            addrA ← [(ii + pool_row + row)×
16:                            column_size + pool_col + col + jj]
17:                            ×chi + kk + addrA_start
18:                            /*Kernel address calculation*/
19:                            addrB ← addrB_start + kernel
20:                            kernel ← kernel + 1
21:                            /*Bias address calculation*/
22:                            addrPsum ← addrPsum_start + bias
23:                        end for
24:                    end for
25:                end for
26:                col ← col + stride
27:                kernel ← 0                    ▷ One ofmap pixel is obtained
28:            end for
29:            row ← row + stride
30:            col ← 0
31:        end for
32:        row ← 0                          ▷ One max-pool result is obtained
33:        pool_col ← pool_col + pool_stride
34:    end while
35:    pool_row ← pool_row + pool_stride
36:    pool_col ← 0
37: end while                               ▷ One ofmap channel is obtained
38: bias ← bias + 1                         ▷ Construct next ofmap channel
```

Algorithm 2 Address Generation in FC Mode

1: $exe\_done \leftarrow 0$
2: $bias \leftarrow 0$
3: $addrB\_pre \leftarrow addrB\_start$
4: **for** $fc\_out \leftarrow 0, out\_len/assoc - 1$ **do**
5: **for** $fc\_in \leftarrow 0, in\_len - 1$ **do**
6: $addrA \leftarrow addrA\_start + fc\_in$
7: $addrB \leftarrow addrB\_pre + fc\_in$
8: $addrPsum \leftarrow addrPsum\_start + bias$
9: **end for**
10: $addrB\_pre \leftarrow addrB$
11: $bias \leftarrow bias + 1$
12: **end for**
13: $exe\_done \leftarrow 1$

Algorithm 3 Address Generation in GP Mode

```
1: exe_done ← 0
2: loop_A_cnt ← 0
3: loop_B_cnt ← 0
4: len_A_cnt ← 0
5: len_B_cnt ← 0
6: while exe_done = 0 do
7:     if loop_A_cnt < loop_A − 1 then
8:         loop_A_cnt < loop_A_cnt + 1
9:     end if
10:    if loop_B_cnt < loop_B − 1 then
11:        loop_B_cnt < loop_B_cnt + 1
12:    end if
13:    if len_A_cnt < len_A − 1 then
14:        if loop_A_cnt = loop_A − 1 then
15:            len_A_cnt ← len_A_cnt + 1
16:        end if
17:    end if
18:    if len_B_cnt < len_B − 1 then
19:        if loop_B_cnt = loop_B − 1 then
20:            len_B_cnt ← len_B_cnt + 1
21:        end if
22:    end if
23:    addrA ← addrA_start + len_A_cnt
24:    addrB ← addrB_start + len_B_cnt
25:    if len_A_cnt = len_A − 1 then
26:        if len_B_cnt = len_B − 1 then
27:            if loop_A_cnt = loop_A − 1 then
28:                if loop_B_cnt = loop_B − 1 then
29:                    exe_done ← 1
30:                end if
31:            end if
32:        end if
33:    end if
34: end while
```

Printed by Tryckeriet i E-huset, Lund 2023

LUND
UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-950
http://www.eit.lth.se