

國立臺灣大學生物資源暨農學院生物機電工程學系



碩士論文

Department of Biomechatronics Engineering

College of Bioresources and Agriculture

National Taiwan University

Master's Thesis

卷積神經網路之深度學習加速器架構設計

Architecture Design of Deep Learning Accelerator
for CNN

邵長威

Chang-Wei Shao

指導教授：方煒 博士

Advisor: Wei Fang, Ph.D.

中華民國 113 年 1 月

January 2024



Acknowledgements



我在台灣大學生物機電工程學研究所，待的時間比一般碩士生多了一年，這三年能過得很充實，學到很多想學的，在學生時期嘗試實作很多想做的，其實最感謝的就是我的指導教授 方煒老師。

生物機電系有個最大的特色，就是同學們的跨領域，系統整合能力很強，其中這需要敢於拋下原本熟識的專業知識，與新的學問做碰撞，身旁的同學們也具備不同背景，一同來到這個像是工程不分系的系所。對於許多未知又高深的學問，想知道適不適合自己，最好的方式就是直接去了解它，接觸他。然而研究所對很多人來說，就是邊工作，邊完成自己畢業，再完成老闆交辦的事項，忙碌的生活節奏又短短的兩年間，嘗試新事物都變得奢侈，代價極高。

方煒老師不論是對自己的導生，還是系統工程課程中，都抱持著非常彈性且動態的教學方式，也曾說過一句話：『台灣的教育都是在教你內容，很少會教你學習的方法』。在老師的指導下，除了學到左右腦交互應用、記憶法則之外，最讓我體悟到我是在一所頂尖研究型大學，台灣第一學府的地方是，方煒教授從來不會強迫並限制學生探索學問的方向，也鼓勵並樂見同學們勇於挑戰。也讓我在研讀碩士時期，能夠接觸我的人生的第二興趣，並作為我的研究主題，因此我很感激，也很珍惜能有這樣的機緣受到方煒老師的照顧，尤其當自己研究目標相當發散的時候，老師總是不厭其煩的提醒，與關心學生。這是我心目中老師的好榜樣。

第二要感謝的老師，是中央大學數學系曾國師老師，也是台大 306 實驗室畢業的學長，口試委員之一。口試後修改論文期間，在結果比較與數據評比

上，給了我很多方向，論文修改的部分也協同方煒老師指導我，頻繁與我進行
週會，檢討我的內容，讓我彌補口試時的缺失，幫助我完成論文。

我也很感謝生物機電系的陳倩瑜老師，葉仲基老師，郭彥甫老師，在我入
學、求知、探索、學習，甚至是心理輔導上，都曾給予過我重要的幫助。實
驗室的鍾興穎學長兼後來的屏科大教授，莊孝穎學長，許展華學弟，也在我與
其他同學們有不同的研究主題時，給予我很大的包容、關心與實質上的幫助，
也很感謝其他位實驗室的學長、同學與學弟們，對於實驗室的維護做的努力。

同時也很謝謝我的女朋友施晴，在研究壓力大，修課忙碌，找不到研究進
步的方向時，也給了我許多生活上，心靈上的關照。也謝謝我的家人，父母與
弟弟，支持我當初考取台大生物機電所，並諒解我研究所比其他人多念了一年的
原因。

邵長威 謹誌

中文摘要



由於深度卷積神經網路 (DCNN) 在人工智慧 (AI) 的應用中對計算力的需求指數成長，對於專用硬體的架構設計來說是一大挑戰。小型系統的功能支援上通常不齊全，而大型晶片變得過於複雜。在不同功能的運算上，同樣仰賴太多模組的切換，導致過多資料搬移，有高延遲與部分模組使用率不高的問題。

這項研究提出了一種基於統計方法的改進結構，包括參數分析、演算法劃分後映射到硬體，以在暫存器傳輸級 (RTL) 建構出一個管線的硬體架構。

研究結果顯示，本論文的架構在單核處理元件 (PE) 上比起其他評比 (e.g, MIT Eyeriss, UCLA DCNN Acc., Google TPU) 的設計，有較佳的硬體重用率、硬體共用性且支援最多功能，也減少了激活函數時的延遲，並確保了所有功能的運算都在核心內完成，且擁有相同的週波時間。

關鍵字：

計算機架構、深度學習加速器、深度卷積神經網路、人工智慧、演算法

Abstract

The demand for computing power in deep convolutional neural networks (DCNNs) for artificial intelligence (AI) applications is exponentially increasing. Designing application-specific hardware architectures poses significant challenges. Smaller systems typically lack essential functionality, while large-scale chips become overly complex. Dynamic switching of modules with different functions results in excessive data movement, leading to high latency and underutilization of certain modules.

This research proposes a statistical-based improved structure comprising parameter analysis, algorithm partitioning, and hardware mapping to construct a pipeline-based architecture in Register Transfer Level (RTL).

The research results indicate that this architecture demonstrates improved hardware reuse, sharing, and functionality on a single processing element (PE) compared to the benchmark (e.g, MIT Eyeriss, UCLA DCNN Acc., Google TPU). It reduces activation function latency and ensures that all functions operate in-core with consistent cycle times.

Keywords:

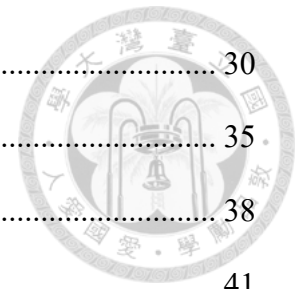
Computer Architecture, Deep Learning Accelerator (DLA), Deep Convolutional Neural Networks (DCNN), Artificial Intelligence (AI), Algorithms.

Table of Content



Acknowledgements	ii
中文摘要	iv
Abstract.....	v
List of Figures.....	viii
List of Tables	x
Chapter 1 Introduction.....	1
1.1 Background.....	1
1.2 Research Scope.....	3
1.3 Research Objectives	4
1.4 Contributions of This Research	6
Chapter 2 Related Work	9
2.1 Exploration of AI and CNN Domains	9
2.2 Comparison of CPU, GPU, ASIC, FPGA in CNN.....	10
2.3 Optimizing ASIC Architectures in AI Computation	12
2.4 Benchmarks for Deep Learning Accelerator	16
Chapter 3 Research Method and Design	17
3.1 Research Gap.....	17
3.2 Deep Learning Accelerator Architecture Design Flow	17
3.3 Popular DCNN Models Analysis	19
3.4 Layers Parameters Statistics	25
3.5 Function and Specification Definition.....	26

3.6 Algorithm Design and Segmentation	30
3.7 Divide-and-conquer	35
3.8 System Flow Design.....	38
3.9 Instruction Set Design	41
Chapter 4 Implementation	45
4.1 Architecture Mapping to RTL	45
Chapter 5 Results and Discussion	49
5.1 Operations Cycle Time Analysis.....	49
5.2 Performance Analysis and Simulation	55
5.3 Data Reuse Rate.....	55
5.4 Design Comparison	57
5.5 Data Path Comparison	60
5.6 Benchmark.....	63
Chapter 6 Conclusions and Future Work	67
6.1 Pros and Cons	67
6.2 Research Contributions.....	69
6.3 Research Limitations	69
6.4 Future Work.....	70
Reference.....	73



List of Figures



Figure 1-1 The hierarchical relationships and positioning of DCNN in AI	1
Figure 1-2 Research scope and position	3
Figure 1-3 Research objectives in the emerging algorithms and increasing data trends..	6
Figure 3-1 Deep learning accelerator architecture design flow.....	18
Figure 3-2 Statistical analysis of convolution and pooling layers in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	20
Figure 3-3 Proportion of convolution and pooling layers in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34]).....	21
Figure 3-4 Distribution of max pooling and average pooling layers across different models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	22
Figure 3-5 Proportion of max pooling and average pooling layers in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	22
Figure 3-6 Layer to layer concept.....	23
Figure 3-7 Layer-to-layer connection quantities in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34]).....	24
Figure 3-8 Proportion of layer-to-layer connections in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34]).....	25
Figure 3-9 Convolution kernel size in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34]).....	27
Figure 3-10 Proportion of different convolution kernel sizes in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	28
Figure 3-11 Definition of input feature map, weight, and output feature map.....	31
Figure 3-12 Convolution with zero-padding followed by a second convolution.	34

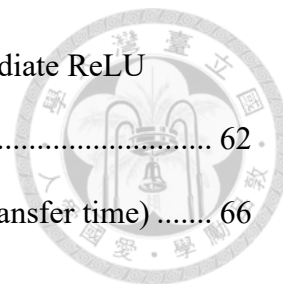
Figure 3-13 Convolution with zero-padding followed by immediate ReLU activation.	34
Figure 3-14 Convolution with zero-padding followed by immediate ReLU activation and pooling	34
Figure 3-15 Flowchart of an eight-stage DLA PE architecture.....	40
Figure 3-16 Instruction set hierarchy: from application-specific to fundamental operations.....	41
Figure 4-1 Caption of DLA PE architecture design details in RTL diagram	47
Figure 4-2 DLA PE architecture design details in RTL diagram.....	48
Figure 5-1 Total cycles for convolution processing per output feature element.....	51
Figure 5-2 Total cycles for average pooling processing per output feature element	52
Figure 5-3 Total cycles for max pooling processing per output feature element.....	53
Figure 5-4 Unified operation time curve for diverse processing modes in this architecture	54
Figure 5-5 Performance of a single PE at different clock rates.....	56
Figure 5-6 Performance of multiple PEs at different clock rates	56
Figure 5-7 Data reuse rates for input feature maps and input weights with varying kernel sizes.....	57
Figure 5-8 Peak throughput comparison of this work and other DLA architectures at identical pe counts and frequencies. (Google TPU v1 achieves 92 TOPS with 65536 PEs at 700MHz, whereas this work achieves 91.75 TOPS.) [17][27][55][57].....	64
Figure 5-9 Performance per single PE at its peak clock rate. The higher the GOPS per PE, the further to the upper left. 1 OPS/PE represents 1 Operation per Second per Processing Element. [17][27][55][57].	65

List of Tables



Table 1-1 Hierarchical relationships and positioning of DCNN in AI with explanations	2
Table 3-1 Statistical analysis of convolution and pooling layers in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	20
Table 3-2 Layer-to-layer connection quantities in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	24
Table 3-3 Convolution kernel size in popular DCNN models (Models' reference sources: [3] [7] [32] [9] [23] [24] [15] [34])	27
Table 3-4 Function and Specification Definition. Fully Connective Layer and Softmax are currently not considered. The kernel size ranges from $k = 1$ to $k = 12$, with consistent treatment of these kernel sizes across different layers as the same hardware operation.....	29
Table 3-5 Operator symbols and their descriptions.....	31
Table 3-6 Algorithm partitioning and hardware mapping specifications.....	37
Table 3-7 Examples of application-specific and fundamental instructions.....	42
Table 3-8 Instruction set architecture in this study's design.....	43
Table 4-1 Environment and Tools	45
Table 4-2 Explanation of DLA PE architecture design details in Figure 4-2.....	46
Table 5-1 Convolution processing cycles - cycle counts for each stage	51
Table 5-2 Average pooling processing cycles - cycle counts for each stage	52
Table 5-3 Max pooling processing cycles - cycle counts for each stage.....	53
Table 5-4 Activation ReLU processing cycles - cycle counts for each stage.....	54
Table 5-5 Convolution of inner product of a tile of input matrix, flow comparison.....	61

Table 5-6 Comparison of Convolution Operation Followed by Immediate ReLU Activation Layer Flows.....	62
Table 5-7 Performance and Functionality Comparison. (Tr. Is data transfer time)	66





Chapter 1 Introduction



1.1 Background

Artificial intelligence (AI) - related applications have reached a preliminary level of maturity in today's world, especially in machine vision and two-dimensional audio signal processing, where deep convolutional neural networks (DCNN) have made significant advancements. Alongside continuous improvements in software algorithms, there is a gap in implementing mature algorithms as hardware solutions.

In Figure 1-1 and Table 1-1, the hierarchical relationships and positioning of DCNN in AI are depicted, indicating that DCNN is just one branch and a specific application within AI.

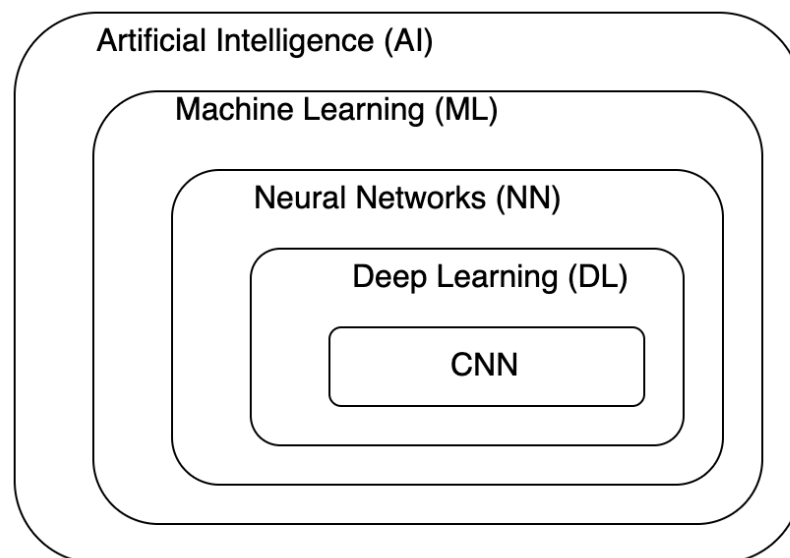


Figure 1-1 The hierarchical relationships and positioning of DCNN in AI

Table 1-1 Hierarchical relationships and positioning of DCNN in AI with explanations

AI	Creating intelligent machines to perform human-like tasks.
ML	Algorithms that enable computers to learn and improve from experience.
NN	Algorithms inspired by the human brain, one kind of ML models.
DL	A subset of ML automating feature extraction, creating multilayered
CNN	Specialized Deep Neural Networks for analyzing 2-D data.

DCNN has achieved remarkable success in diverse fields, including autonomous driving, medical imaging, robotics, security and surveillance, art generation, quality control (QC) in manufacturing, natural language processing, video games and agricultural technology. And notably, in March 2016, Google's AlphaGo defeated world Go champion 4-1 using deep reinforcement learning and custom-designed Tensor Processing Units (TPUs), a custom ASIC. TPUs accelerated the deep learning training process, enabling AlphaGo to learn from a massive dataset of Go games rapidly and accurately [12] .

AI domain acknowledges the significance of three key elements: algorithms (such as various renowned models), data collection (big data), and computing power (e.g., computers or devices capable of executing AI-related operations). AI computing involves other crucial considerations, such as cloud (large-scale servers and data centers) and edge computing (mobile devices like smartphones, digital cameras, surveillance cameras, etc.),

which vary in computing performance and efficiency. Consequently, they exhibit different performance in model training and model inference tasks.



1.2 Research Scope

Given the maturity of DCNN applications and the reliability of algorithms and models based on a few main computational operations, it is appropriate to start researching specialized hardware, specifically Application Specific Integrated Circuits (ASICs). In addition to process technology advancements for improved performance and efficiency, architecture plays a crucial role, which will be further discussed in the literature review. Therefore, the research scope is defined as the architectural design of an edge computing Intellectual Property (IP) core primarily focused on DCNN inference, as depicted in Figure 1-2.

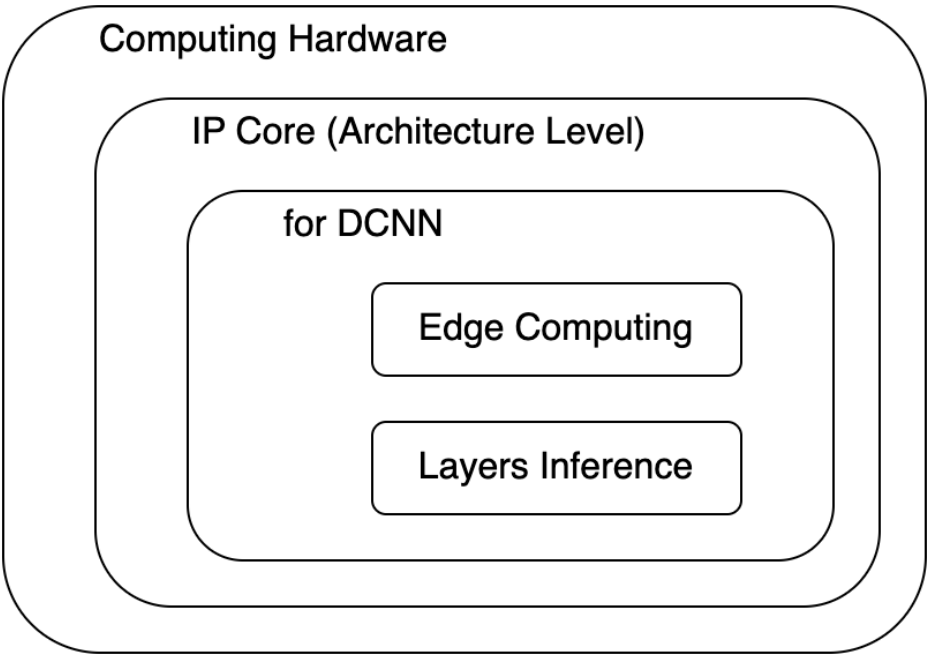
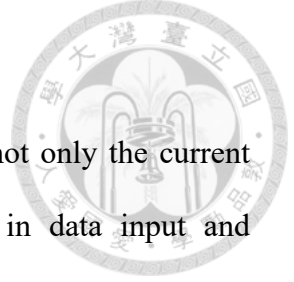


Figure 1-2 Research scope and position



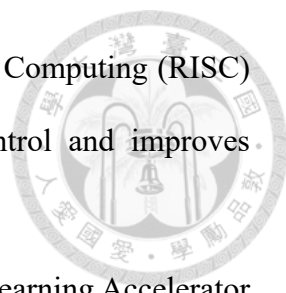
1.3 Research Objectives

In this hardware architecture design, it is crucial to consider not only the current situation but also extension and maintenance, such as changes in data input and computation formats. Therefore, research objectives for this hardware architecture design include the following two key elements of AI as shown in Figure 1-3 on the left and right sides:

- A. **Future Algorithm Support:** To enable emerging algorithms to be interpreted, compiled, or transformed into a universal form supported by this work, this includes models and different types of layers that may be separated or transformed from a certain model.
- B. **Handling Increasing Data Trends:** Designing efficient and high-performance datapath and dataflow systems to mitigate bottlenecks caused by data movement, addressing the potential increase in data throughput resulting from the growing number of model parameters and the ever-expanding global data volume.

Based on these two principles, design is further guided by six fundamental principles, as shown in Figure 1-3 in the middle of the processor:

- A. **Layer-Compatible:** The processor needs to support various common layer computations, with a primary focus on convolutional operations. The reason why the convolutional operation architecture is preferred will be explained in section, 3.2.

- 
- B. **Timing Identical:** Similar to the Reduced Instruction Set Computing (RISC) approach, the fixed-stage pipeline design simplifies control and improves efficiency.
- C. **DLA and PE Scalable:** The design aims to create a Deep Learning Accelerator (DLA) that is scalable to accommodate multiple cores or processing elements (PEs), leveraging parallel computing to achieve improved performance (e.g., Giga floating-point operations per second, GFLOPS). This scalability allows the processor to exhibit non-linearly decreasing performance improvements as the number of cores increases [14] . The single processor IP in this study is intended to be scalable, providing the potential for future enhancements and be expanded to core configurations.
- D. **Efficient Data Movement:** Reducing data movement is critical due to its higher relative energy cost, area cost[30] , and increased cycle times compared to other arithmetic operations.
- E. **Data Reusable:** The concept of data reuse involves utilizing adjacent operands or data stored in PE registers to minimize data movement. This design includes additional reusing channels to reduce data movement.
- F. **Modules and Path Sharing:** Sharing hardware resources between sub-modules and data paths will reduce the IC's area. This goal will be achieved through pipelining and time-division multiplexing to enhance computational speed and resource utilization[8] .

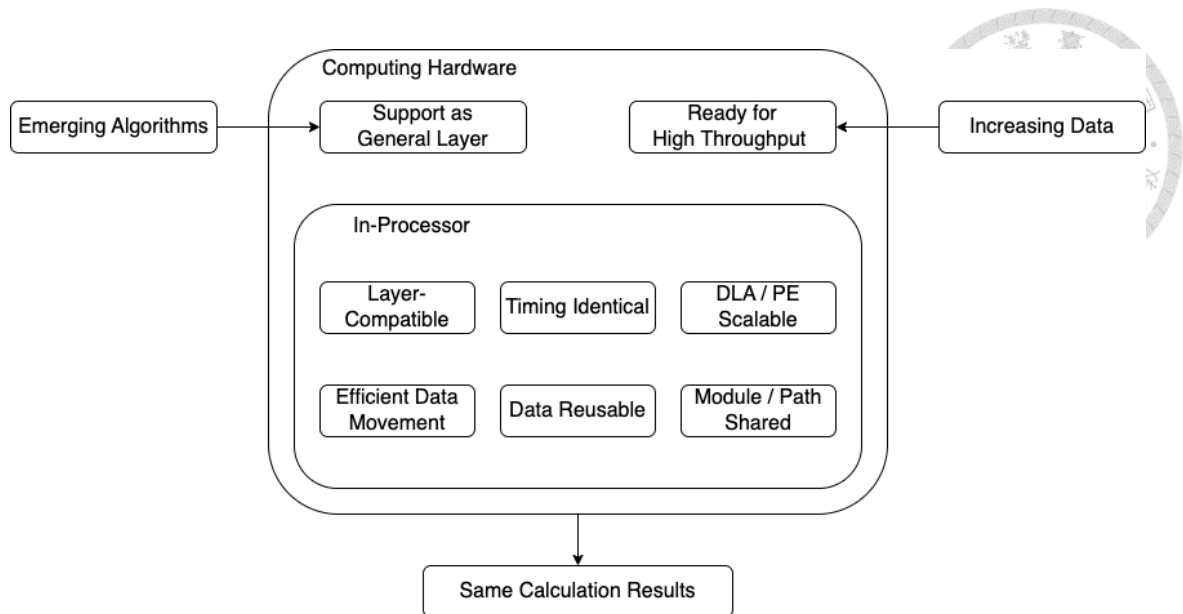


Figure 1-3 Research objectives in the emerging algorithms and increasing data trends

1.4 Contributions of This Research

First, this study adopts a soft-hardware co-design methodology. It begins by analyzing layer parameters from popular DCNN models, then proceeds to define functions and specifications, design algorithms, segment them, and finally, specify hardware mappings. The outcome is a clear system flow chart and function level instruction set for processing elements (PEs).

Second, this research focuses on crafting a single core with comprehensive functionality and minimal latency, diverging from many other papers that delve into issues like Network-on-Chip (NOC) or parallel computing, such as PE grouping. In contrast, this paper presents more detailed insights into the architecture, revealing the internal designs of the core in Register Transfer Level (RTL). This includes critical aspects such as register quantity, individual datapaths, hardware units, and distinctive instruction decoding for each stage.

Third, the benchmark table demonstrates the superior hardware resource sharing and minimal activation latency of the DLA architecture developed in this study. The core processes all functions in-core effectively, avoiding delays associated with off-PE data movement, and maintains uniform cycle time design across all functions, ensuring ease of control and scalability. This approach significantly enhances overall operational efficiency and performance.



Chapter 2 Related Work

This chapter provides a systematically ordered literature review on the topic of "Architecture Design of Deep Learning Accelerator for CNN," the focus of this research.

In the process of designing a novel hardware architecture, it is essential to comprehend the domain and algorithms associated with the hardware application. Therefore, we initiate a discussion on the field of artificial intelligence (AI) and delve into one of its branches, Convolutional Neural Networks (CNN). Subsequently, a comparative analysis is conducted among four major hardware architectures: Central Processing Unit (CPU), Graphics Processing Unit (GPU), Application-Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) are assessed for their merits and drawbacks in the context of AI and CNN operations. Additionally, prominent architectures within the academic community are explored, and an examination is conducted on how commercially available chips from leading IC design companies are optimized for AI computations. Finally, essential benchmarks and metrics within this domain are consolidated.

2.1 Exploration of AI and CNN Domains

Convolutional Neural Networks (CNNs) belong to the field of Artificial Intelligence (AI), specifically under the broader umbrella of machine learning and deep learning. Mathematically, the essence of CNNs lies in linear algebra ($y = w * x + b$), where each component is defined as follows: y signifies the neuron, x denotes the input, w represents the weight, and b is the bias term. Activation functions introduce non-linearity into the network, allowing it to learn complex patterns and relationships in the data. The sigmoid function, for instance, maps the input values to a range between 0 and 1, which

is particularly useful for binary classification tasks. ReLU (Rectified Linear Unit) is renowned for its computational efficiency and its capability to mitigate the vanishing gradient problem. ReLU maps the input values to a range between 0 and positive infinity. As a large-scale simulation system is composed of numerous neurons, the mathematical operations evolve into matrix computations [18] .

Deep Learning and Deep Neural Networks typically refer to structures with one or more hidden layers, where the output of one layer serves as the input for the next. In Convolutional Neural Networks (CNNs), the convolutional layer enhances model bias and reducing flexibility through Receptive Fields and Parameter Sharing [19] .

AI computations are primarily divided into Training and Inference. Training aims to generate a foundational model using techniques such as regression, gradient descent, classification, and backpropagation [18] . On the other hand, DCNN Inference employs the parameters of a pre-trained model for computations. Each layer represents a mathematical operation, comprising various main layers such as convolution, pooling, activation functions, etc. These layers can be sequentially connected or can diverge into unrelated branches. This phenomenon is observed in several DCNN models, including AlexNet [3] , MobileNet [7] [31] , GoogLeNet [9] , ResNet [23] , VGG [24] , DenseNet [15] , and EfficientNet [33] .

2.2 Comparison of CPU, GPU, ASIC, FPGA in CNN

Firstly, based on AMD's 2019 announcement [29] , it was revealed that over the preceding decade (2010-2019), technological advancements constituted the primary driver of performance improvements, contributing to a 40% increase. Microarchitecture emerged as the second most significant factor, accounting for 17% of the improvement [29] . Concerning operations, the relative energy cost associated with data movement

ranked highest, followed by multiplication, with addition being the least among the three. The relative energy cost and relative area cost of floating-point operations exceeded those of integer operations. Additionally, for operations with larger data widths, the relative energy cost and relative area cost were higher than those with smaller data widths (e.g., 32-bit > 16-bit) 錯誤! 找不到參照來源。 . Furthermore, the single-core era was characterized by microarchitecture advancements, voltage scaling, and limitations imposed by power consumption and complexity. In contrast, the multi-core era emerged driven by the pursuit of enhanced throughput but faced constraints related to power consumption and scalability [10] .

There is a tradeoff between flexibility and efficiency. CPUs typically exhibit the highest flexibility, followed by GPUs with better efficiency. FPGAs share a similar characteristic, while ASICs possess optimal efficiency [32] . Dedicated ICs, such as ASICs, demonstrate approximately 1,000 times better relative energy efficiency than CPUs [30] . In terms of relative performance, they surpass CPUs by 10,000-100,000 times, 100-1,000 times more than GPUs [20] . GPUs predominantly handle the computation for AI training, and they are also well-suited for AI inference. In comparison, FPGAs are more suitable for inference tasks than GPUs. CPUs perform poorly in both training and inference, although sometimes they can offer higher flexibility [26] .

Regarding commonly used microarchitecture techniques, pipelining increases delay and area but reduces energy consumption. However, the utilization of pipelining and time-division multiplexing in hardware design can keep the energy cost constant without introducing delays, thereby reducing the required area [8] [5] [13] . These principles contribute to variations in the application of the four computing architectures in AI operations due to differences in their primary application scenarios.

2.3 Optimizing ASIC Architectures in AI Computation

This section will introduce how various architectures optimize AI computation, starting from Central Processing Units (CPUs), Graphics Processing Units (GPUs), and some well-known Application-Specific Integrated Circuits (ASICs).

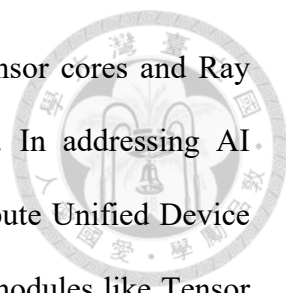
MIPS Technologies introduced the Microprocessor without Interlocked Pipeline Stages (MIPS) architecture in 1985, embodied in the R2000 central processing unit (CPU). The R2000 features a fixed five-stage pipeline, consisting sequentially of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) [52]. In contrast to Single-Cycle CPUs, MIPS CPU aims to minimize the latency of each stage by keeping the length of each stage as short as possible, achieving a high usage rate through pipelining. The MIPS CPU employs the Reduced Instruction Set Computer (RISC) instruction set architecture, simplifying control operations [46] [47]. Additionally, the principles of the memory hierarchy in computer architecture align with the philosophy of "Small is Fast, closer is faster" [48]. The concept of locality of reference encompasses two primary types: Temporal and spatial locality. These types suggest that if a memory location is accessed at one point, it is likely to be accessed again in the near future, and nearby locations will also likely be accessed [42].

Intel's AI on CPU solution in the Ice Lake architecture proposes an Ultra Path Interconnect (UPI) channel uniformly connecting to the last-level cache memory (LLC) of each core. This configuration highlights the significance of more interconnects between cores, emphasizing the importance of data sharing among neighboring cores [11]. Additionally, the transition from the historical Single Instruction, Single Data (SISD) paradigm to Single Instruction, Multiple Data (SIMD) implies extensive parallel processing in AI computations [1]. The CPU can function as an independent computing

unit or be subdivided into tiles for partitioned processing within the CPU. The Intel Math Kernel Library for Deep Neural Networks (MKL DNN) mathematical function library places special emphasis on features such as data quantization, data prefetching, data reuse, data buffering, vectorization, and memory access [6].

The computational core of Google's Tensor Processing Unit (TPU) v1 [35] is a Systolic Array (SIMD) [53], enabling simultaneous operation of all processing elements (PE), leading to high throughput and low latency at the expense of increased power consumption. This array is managed by a Matrix Multiplier Unit (MMU or MXU), containing 256x256 Multiply-Accumulate (MAC) units. TPU adopts a weight stationary architecture. TPU v2, v3 quantizing 32-bit Floating Point inputs to Sign 8-bit and Unsigned 8-bit, enhancing speed, reducing area, and lowering power consumption [17]. Google introduces a non-IEEE standard FP bit format for TPU, with BFP16 Fraction reducing by 3 bits and compensating with a 3-bit Exponent increase. This achieves precision and dynamic range close to FP32 in Deep Neural Network (DNN) applications while reducing memory bandwidth [43]. Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN) constitute 95% of the workload on TPU v1 [2]. Google Cloud demonstrates the computation flow of CPU, GPU, and TPU, where TPU's MXU avoids memory reads and directly passes multiplication results to the MAC. Consequently, Google states that TPUs can achieve high computational throughput in neural network calculations, especially for models dominated by matrix computations, devoid of TensorFlow or PyTorch operations, or when computation volume and time are exceptionally significant [16].

NVIDIA is renowned for its Graph Processing Unit (GPU) designs. Observing its recent system architectures, there is a continuous increase in the number of cores and computational power for Tensor operations. The Turing architecture of the RTX2000



series represents a breakthrough transition, notably introducing Tensor cores and Ray Tracing (RT) cores within each Streaming Multiprocessor (SM). In addressing AI computations, NVIDIA chose to preserve its well-established Compute Unified Device Architecture (CUDA) without altering it but introduced additional modules like Tensor cores and RT cores [38] [39]. The Turing architecture not only supports INT32 but also extends Tensor core support to FP16 with FP32 accumulate, whereas the next generation Ampere architecture extends support for BF16 Tensor to FP32 accumulate [40]. It concurrently enhances memory bandwidth and provides more channels connected within each core. The utilization of Winograd Transform optimizes convolution calculations [37], aiming to reduce multiplications and substitute them with more additions. For Deep Learning, NVIDIA not only augment AI computation support on its GPUs but also designed the Nvidia Deep Learning Accelerator (NVDLA) architecture, which is open source. Examining its system architecture, NVDLA features activation unit, pooling unit, and normalization unit, following convolution buffer. The data flow sequentially undergoes convolution, activation, pooling, and normalization. Additionally, each of these units can Write Back (WB) to memory and supports Direct Memory Access (DMA), providing increased flexibility [36].

The University of California, Los Angeles (UCLA) has introduced its DCNN Accelerator, featuring a system architecture where the Convolution Unit Engine (CU) supports average pooling operations, while max pooling is segregated and placed in the Accumulation Buffer. All operations are conducted in Int16. There are a total of 16 Convolution Units, each containing 9 Processing Elements (PE), allowing support for a maximum Kernel size of 3x3. If the Kernel size exceeds 3, weight filter decomposition is employed. In the case of non-multiples of 3, Zero Padding is applied. For a 1x1 Kernel,

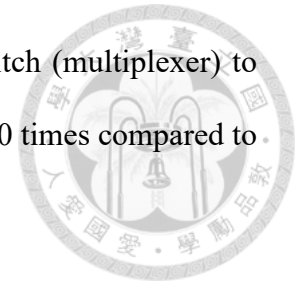
only two PEs within the CU are utilized, with the other seven deactivated to conserve power. These two PEs handle inputs for odd and even channels, respectively [27].

The Energy-Efficient Multimedia Systems (EEMS) group at the Massachusetts Institute of Technology (MIT) introduced the Eyeriss (version 1) DCNN accelerator in 2016 [54] [51]. For data transfer, Eyeriss v1 employs a multi-level memory structure comprising external DRAM, a Global Buffer, and Register Files (RFs) within Processing Elements (PEs). It uniformly transforms two-dimensional matrices into one-dimensional vectors for input, directly converts weights into one-dimensional vectors, and the Input Feature Maps (Fmap) are reordered to two-dimensional Toeplitz matrices [50], feeding the data column-wise. Run Length Encoding (RLE) compression is utilized to minimize off-chip DRAM traffic, and Zero Gate is internally employed within PEs to avoid unnecessary multiplication operations involving zero values, achieving approximately 45% power savings, effectively combining data sparsity and processing power reduction [54].

In a presentation video shared by MIT EEMS Group on YouTube [55], the dataflow of Eyeriss v1 is demonstrated. They optimize data reuse through Row Stationary (RS), where filter rows (weights) are reused horizontally across PEs, Input Feature Map rows (Fmap) are reused diagonally across PEs, and Partial Sums (psum) accumulate vertically across PEs. In convolution layer operations, this approach proves to be 1.4x – 2.5x more energy-efficient than other DNN dataflows.

The Network-on-Chip (NoC) manages the transmission between the Global Buffer (GB) and the PE array. The Global Input Network (GIN) is structured into two layers, Y-bus and X-bus, creating a hierarchical memory structure. The Multicast NoC leads to over 80% energy savings compared to broadcast. Yu-Hsin Chen mentioned that as DNNs become more compact, their Eyeriss v2 (version 2) [56] introduces the concept of

Hierarchical Mech Network (HM-NoC). Only requiring circuit-switch (multiplexer) to switch between four modes, it achieves a throughput speedup over 10 times compared to Eyeriss v1.

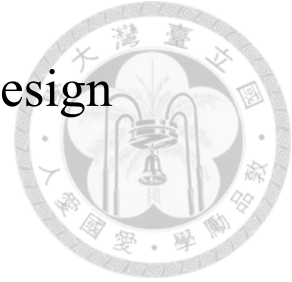


2.4 Benchmarks for Deep Learning Accelerator

For a Deep Learning Accelerator (DLA), the computation of DCNN models constitutes its primary application. Numerous popular models and their algorithms have extensive framework support, such as TensorFlow [44] or Keras [25], enabling direct utilization of these models through APIs. Critical parameters for DCNN models within these APIs include model size, top accuracy, parameters, depth, and inference time on CPU and GPU. Analyzing the model's memory, parameters, and floating-point operations (FLOPs) (multiplication and addition) allows estimation of the required hardware resources for the model [28].

In the realm of DNN hardware, the target metrics include accuracy, power, throughput, cost, external memory bandwidth, required on-chip storage, and core utilization, constituting additional factors. Key benchmark items encompass energy efficiency, external memory bandwidth, and area efficiency (inclusive of process technology) [21]. Other common metrics include core area, the quantity of Multiply-Accumulate (MAC) units, and clock frequency. For Field Programmable Gate Arrays (FPGAs), a comparison may involve the utilization of Look-Up Tables (LUT), Flip-Flops (FF), Digital Signal Processors (DSP), and Random Access Memory (RAM). In the realm of Energy Estimation, the methodology involves summing the processing of each MAC and memory access to achieve normalized energy consumption for relevant comparisons [45].

Chapter 3 Research Method and Design



3.1 Research Gap

In the preceding section on related works, various optimizations for AI computations were discussed, encompassing how traditional CPUs and GPUs optimize AI operations, as well as the development of dedicated ASICs for AI. However, many of these journals, papers, and existing products often require extensive ecosystems and corresponding compilers to serve as the interface between software and hardware. Numerous studies also emphasize the design of data paths across PEs or cores, delving into aspects like Network on Chip (NoC) or Mesh, and even advanced network configurations such as groups or clusters.

This study adopts a different approach, concentrating on the statistical analysis of model parameters and the sequential occurrence of layers. Subsequently it refines the focus to the microarchitecture design within Processing Element (PE) or core while also redefining a simplified instruction set architecture for it.

3.2 Deep Learning Accelerator Architecture Design Flow

In this chapter, a comprehensive explanation of the research methodology and design process will be presented, as depicted in Figure 3-1. The process involves performing a detailed analysis of popular DCNN models to understand their computational characteristics. Subsequently, the algorithms are reorganized and decomposed, with mappings to various hardware units. The design process begins with the creation of individual sub-modules, followed by investigating a pipeline design that aligns with the six design principles outlined in the 1.3 Research Objectives. the complete flowchart of the research design will be presented, along with the development of a control instruction

set for this flow. The final section will showcase the comprehensive processor architecture diagram from a RTL.

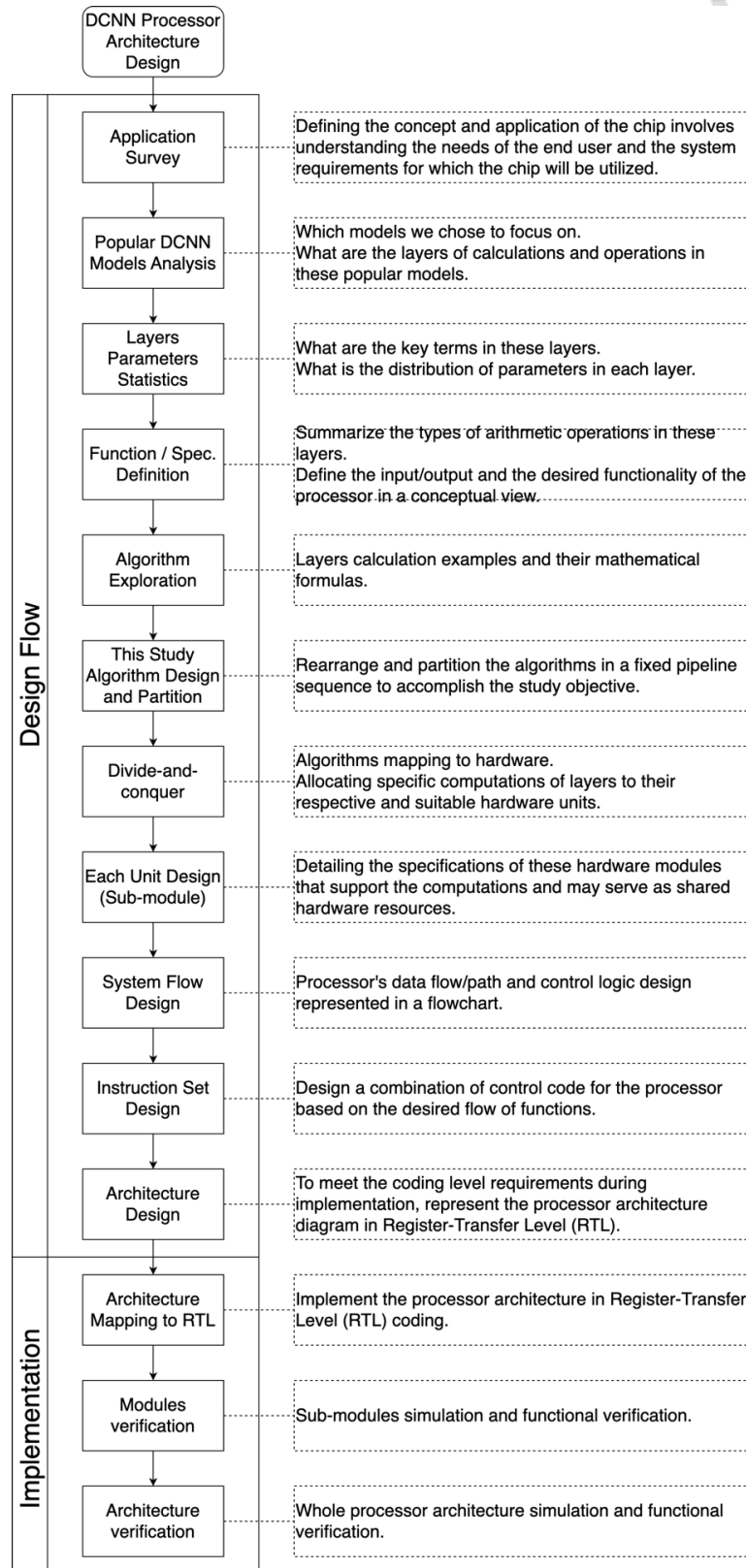
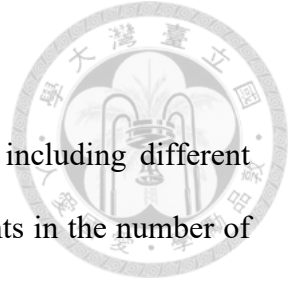


Figure 3-1 Deep learning accelerator architecture design flow



3.3 Popular DCNN Models Analysis

In this chapter, commonly used DCNN models are selected, including different variations with significant changes in algorithms or slight adjustments in the number of layers. Some of these variations are suitable for edge computing, such as MobileNet [7] [31] , while others, like ResNet [23] and VGG [24] , require more computational resources. The focus has been on analyzing the number of convolution and pooling layers, as these two types of network layers demand more significant computational power in DCNNs.

As seen from Table 1-1 and Figure 3-2, the number of convolution layers far exceeds the number of pooling layers. The variations in MobileNet v1, v2, and v3 indicate an increasing trend in the number of convolution layers, which is related to the separable convolution (depthwise convolution, pointwise convolution) algorithm discussed in the previous literature review. Therefore, the layer count statistics do not directly represent the computational workload.

Figure 3-3 also reveals that convolution operations dominate in DCNNs, constituting 96.77% of the layers, while pooling layers only make up a 3.23% portion. Note: The data for the analysis results presented in (Table 3-1 to Table 3-3) and (Figure 3-2 to Figure 3-10) are investigated from this study.

In the previous statistics, Figure 3-2 including pooling layers, we observed two primary types of pooling: Max Pooling and Average Pooling. From Figure 3-4, allows for the analysis of the relative distribution and total count of these two types of pooling

Table 3-1 Statistical analysis of convolution and pooling layers in popular DCNN models

(Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

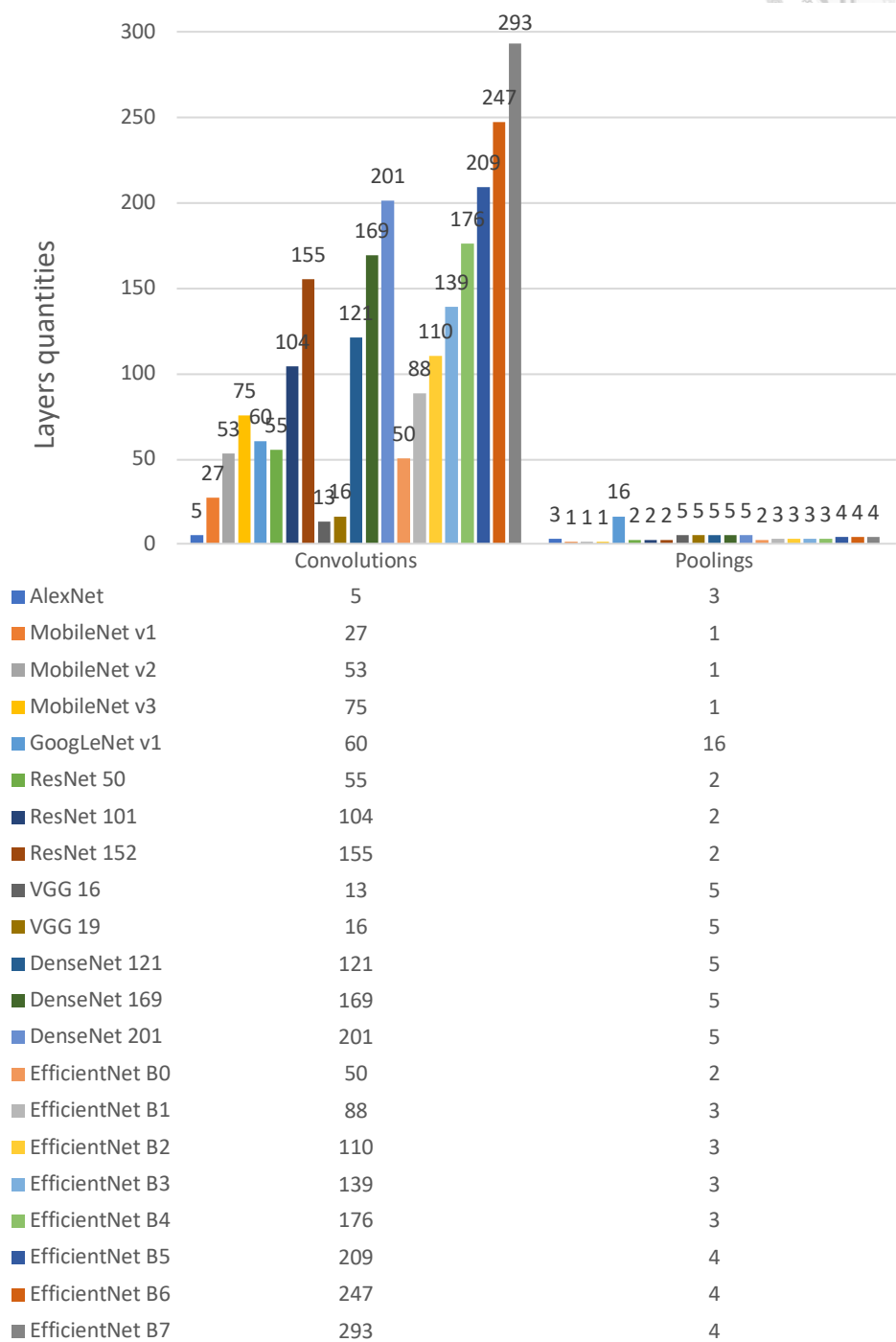


Figure 3-2 Statistical analysis of convolution and pooling layers in popular DCNN models

(Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

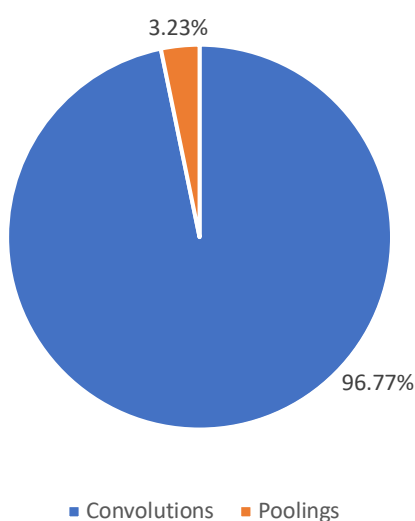


Figure 3-3 Proportion of convolution and pooling layers in popular DCNN models
(Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

layers in each model. It was observed that Max Pooling and Average Pooling can coexist within the same DCNN model. It is possible for a model to utilize only one type of pooling. In Figure 3-5, it is observed that max pooling accounts for 63.29% of the pooling layers, while average pooling constitutes 36.71% of the total. This observation emphasizes the need for careful consideration in the design to accommodate mode switching or different hardware configurations to support the possibility of both types of pooling occurring in combination in the complete computation of a model.

In these models (different from the previous samples), a statistical analysis of the interconnections between network layers was conducted, categorizing them into three

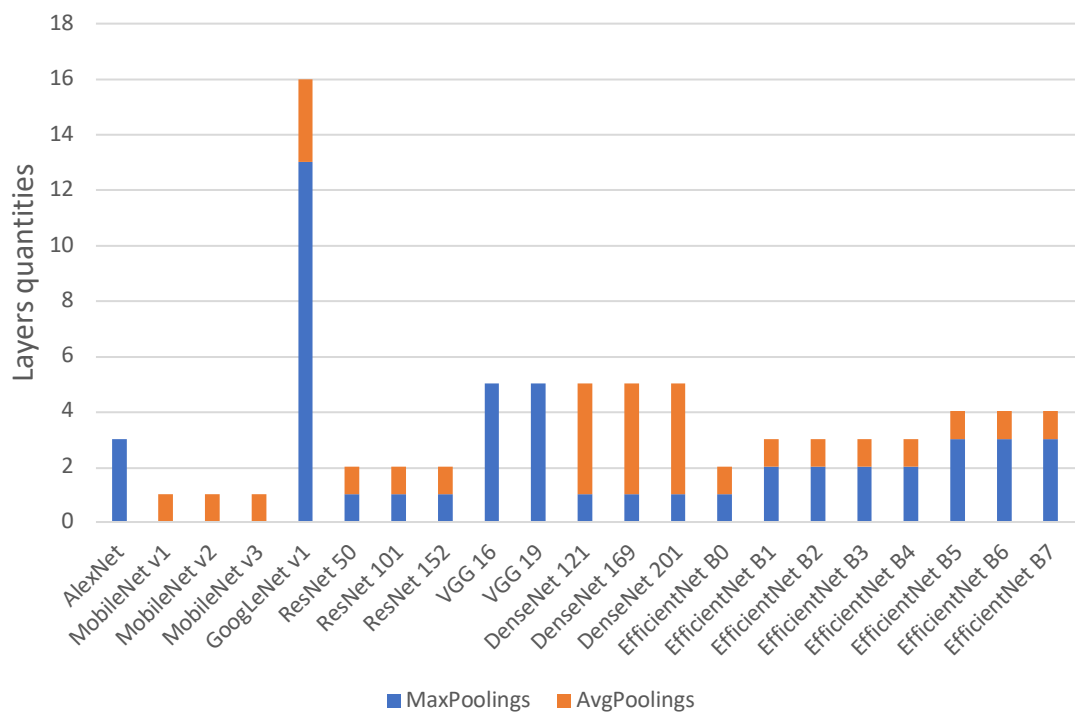


Figure 3-4 Distribution of max pooling and average pooling layers across different models (Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

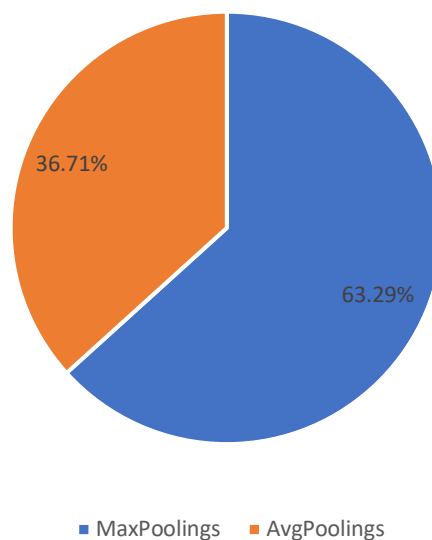


Figure 3-5 Proportion of max pooling and average pooling layers in popular DCNN models (Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

main groups: Convolution to Convolution (C-C), Convolution to Max Pooling (C-MPL), and Convolution to Average Pooling (C-APL). Figure 3-6.

In addition to studying individual layer operations, examining the connections between layers is important, as it relates to the reusability of Partial Sum or Output Feature Map, which is a critical aspect. Figure 3-7 and Table 1-1 shows that C-C connections are the most prevalent, and as models become larger or deeper, the number of C-C connections also increases. However, the occurrences of C-MPL and C-APL are independent of model characteristics (e.g., model size, depth, application domain, parameter quantity, computing resources). The diagram does not include the Pooling-Pooling category, as such connections do not occur in any network layers.

Figure 3-8 reveals that C-C connections constitute 96.13% of the total, while the combined proportion of C-MPL and C-APL connections amounts to 3.88%.

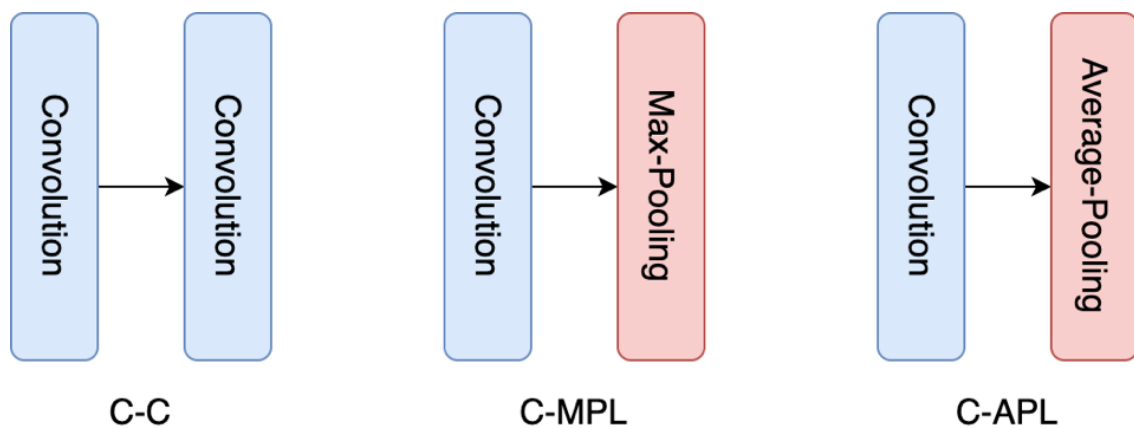


Figure 3-6 Layer to layer concept



Table 3-2 Layer-to-layer connection quantities in popular DCNN models (Models'

reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

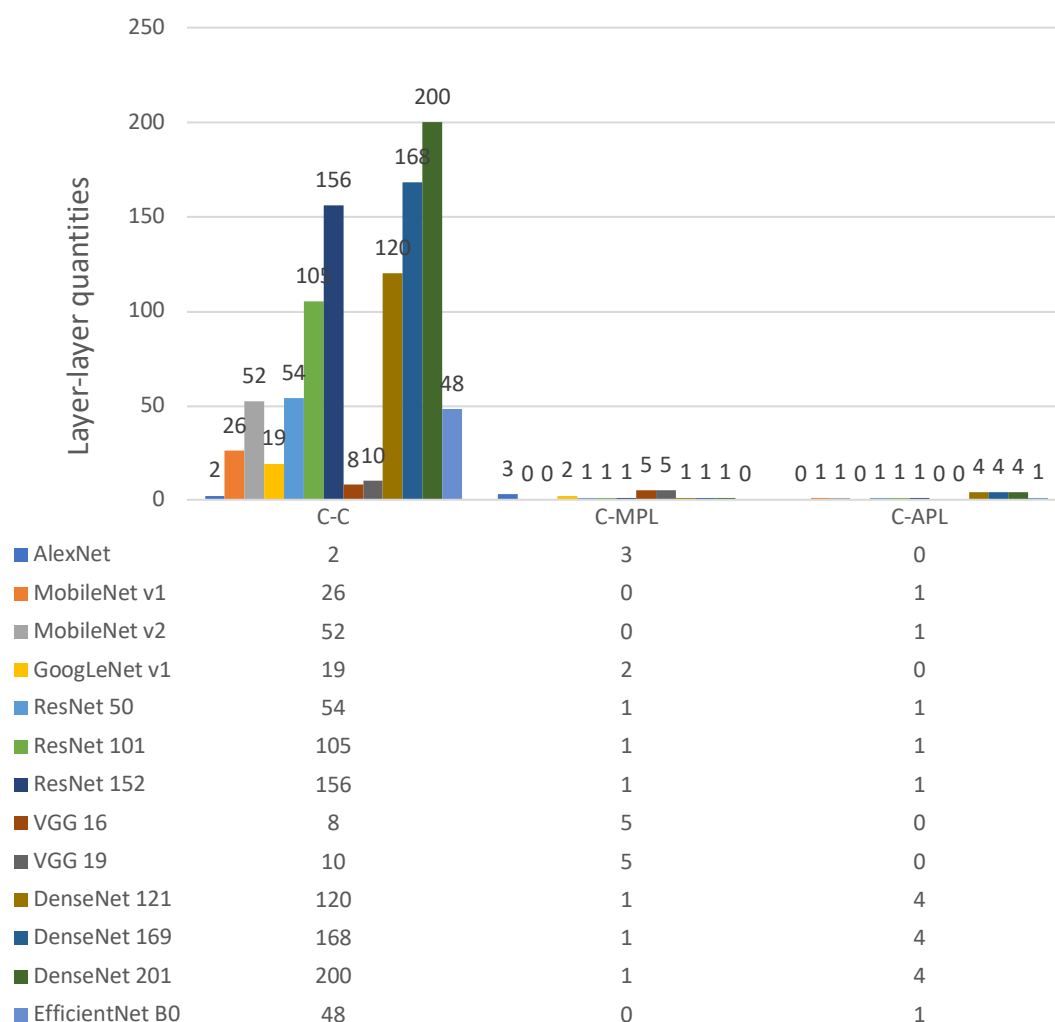


Figure 3-7 Layer-to-layer connection quantities in popular DCNN models (Models'

reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

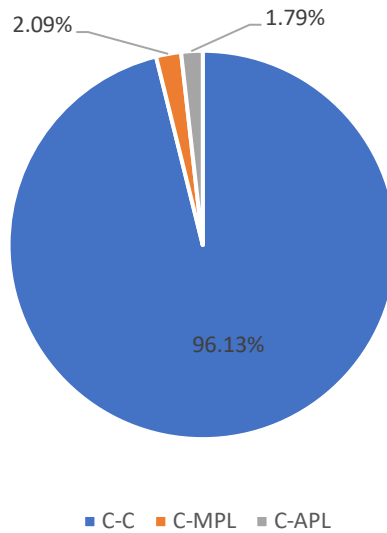


Figure 3-8 Proportion of layer-to-layer connections in popular DCNN models (Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

3.4 Layers Parameters Statistics

Based on the preceding analysis, it was evident that the number of convolution layers is the highest, and the occurrence of layer-to-layer connections is also most frequent. Therefore, a further analysis is conducted on a critical parameter of these convolution layers, which is the "Kernel Size." The kernel size not only affects the computational workload but also influences the two-dimensional dimensions of the output layer.

As observed in Figure 3-9, kernel sizes of 1x1 and 3x3 account for the majority, both being odd square sizes, with no even or asymmetric kernel sizes used in the convolutions. Additionally, it should be noted that although the proportion of 1x1 kernels is high, they are often combined with other kernel sizes in models. In two-dimensional operations, the 1x1 kernel serves the purpose of linear operations by multiplying the matrix with a

constant, frequently employed in the second step of separable convolutions (Pointwise) to match the original channel dimensions (the third dimension's size). For kernel sizes greater than 1x1, they better approximate our understanding of the receptive fields of CNN neurons, such as the 3x3 kernel, while still being referred to as a "filter."

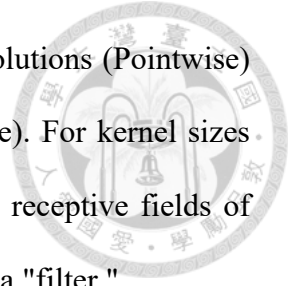


Figure 3-10 reveals that even when considering the inclusion of 1x1 kernels, the proportion of 3x3 kernel sizes remains significant at 54.22%. This observation offers crucial insights for designing the Register file size, data transmission types, and data routing in this research.

Note: In scientific literature, the term "filter" is commonly used to refer to the convolution kernel in DCNNs.

Note: The data for the analysis results presented in (Table 3-1 to Table 3-3) and (Figure 3-2 to Figure 3-10) are investigated from this study.

3.5 Function and Specification Definition

In this section, presenting a structured and comprehensive overview of the functions and specifications require for designing a specialized DLA architecture tailored for Deep Convolutional Neural Networks (DCNNs), based on the earlier findings and analyses in

Table 3-3 Convolution kernel size in popular DCNN models (Models' reference sources:

[3] [7] [31] [9] [23] [24] [15] [33])

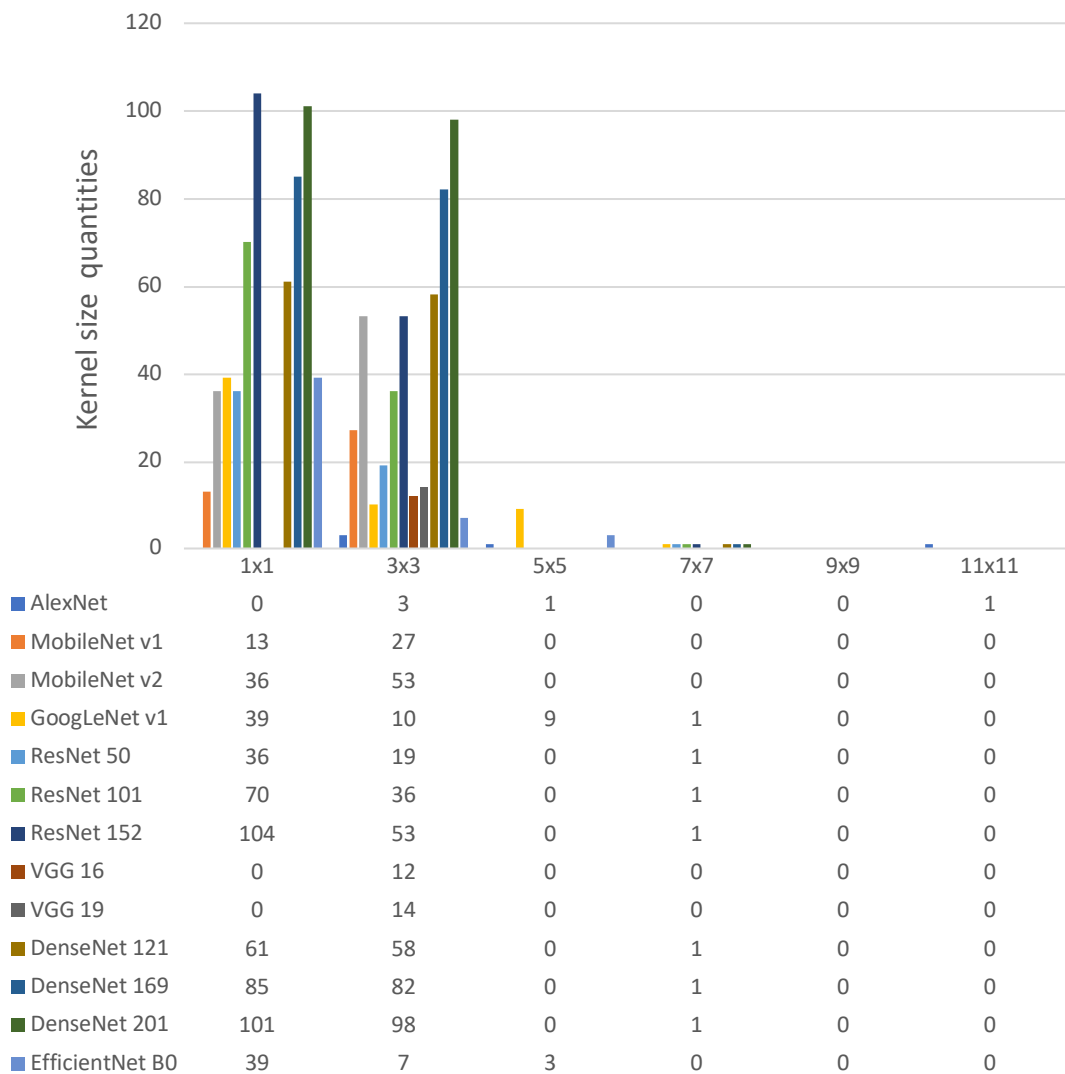


Figure 3-9 Convolution kernel size in popular DCNN models (Models' reference sources:

[3] [7] [31] [9] [23] [24] [15] [33])

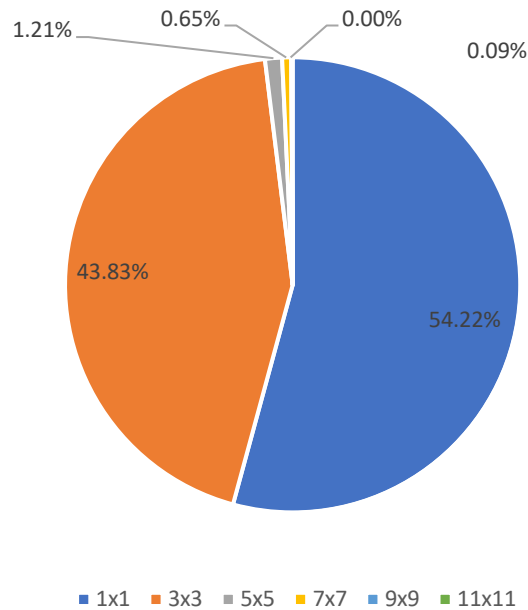


Figure 3-10 Proportion of different convolution kernel sizes in popular DCNN models
(Models' reference sources: [3] [7] [31] [9] [23] [24] [15] [33])

Section 3.3 and 3.4. Primary objective is to optimize the processing of convolutional layers, which have a higher priority than pooling layers and other infrequently occurring layers, with smaller kernel sizes taking precedence over larger ones. Additionally, must consider the Convolution-to-Convolution (C-C) connection sequence.

Table 3-4 Function and Specification Definition. Fully Connective Layer and Softmax are currently not considered. The kernel size ranges from $k = 1$ to $k = 12$, with consistent treatment of these kernel sizes across different layers as the same hardware operation.

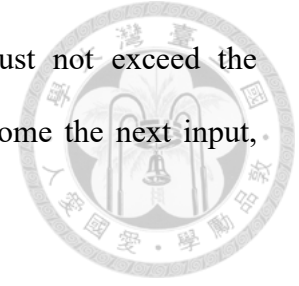
Supported Functions	Function Type	Parameters
Convolution Layer	2D Layer	Kernel size: 0 to 12
		Stride size: 0 to 3
Pooling Layer	Max Pooling	Kernel size: 0 to 12
	Average Pooling	Stride size: 0 to 3
Padding Layer	Zero Padding	Padding size: 0 to 2
	Same Padding	
Activation	Rectified Linear Unit (ReLU)	-
Function Layer		

Other Definitions:

A. Input/Output Considerations:

The input data format should be in a vector, one-dimensional format, which is commonly used to simplify the problem and reduce dimensionality in this algorithm. This enables execution within a single Processing Element (PE) and aligns with the scalability objective of this research.

The output values after processing in this PE must not exceed the dimensions of the input data format since they may become the next input, requiring compatibility in data dimensions.



B. Control Aspects:

The objective is to minimize reliance on large-scale compilers and achieve a more intuitive controller encoding.

C. Hardware Selection:

Priority is given to mode switching or data channel switching, employing compact, fast, and cost-effective units such as Data Selector or multiplexer (MUX) to support pipeline partitioning and timing analysis. The hardware modules should be as small and minimal as possible, removing unnecessary functionalities.

3.6 Algorithm Design and Segmentation

The following Figure 3-11 and Table 3-5 present a set of symbols (operators) that are defined for use in the forthcoming examples and illustrations to facilitate a better understanding of the designed algorithm and its segmentation. These symbols allow for a more concise representation of the algorithm's operations and structure.

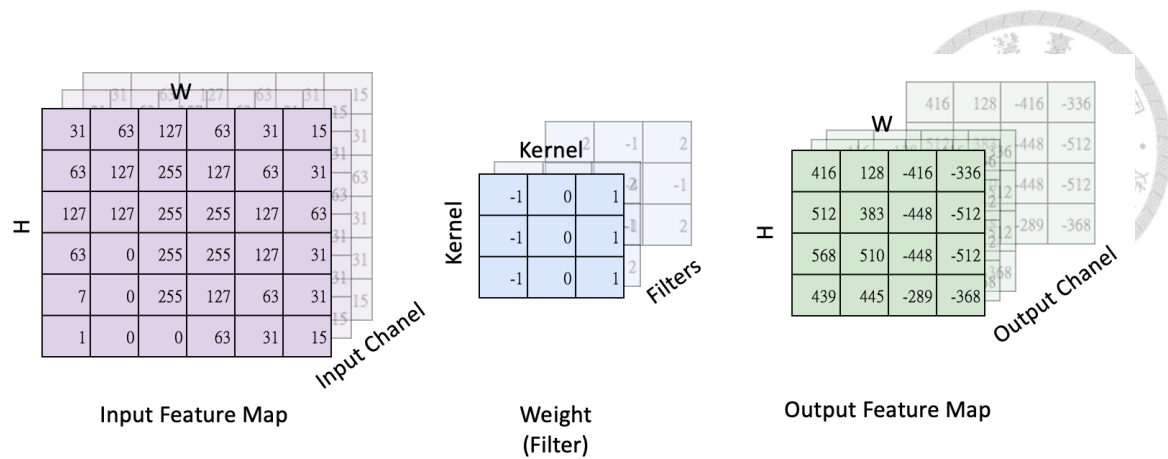
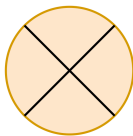
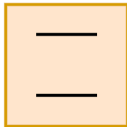


Figure 3-11 Definition of input feature map, weight, and output feature map

Table 3-5 Operator symbols and their descriptions

Operator Symbols	Descriptions
	Convolution: element-wise multiplication of the input feature map with the corresponding kernel (weight) values, followed by summing up the results. (Inner product)
	Output
	Max Pooling: Selects the maximum value within the kernel window for each location in the input feature map during the pooling operation.

Average Pooling:
Average pooling calculates the average value within the kernel window for each location in the input feature map during the pooling operation.

ReLU (Rectified Linear Unit):
ReLU is an activation function exclusively considered in this study, represented by these two symbols, which are equivalent. It introduces non-linearity in the neural network by keeping the output unchanged if the input is positive and setting it to zero if the input is negative.

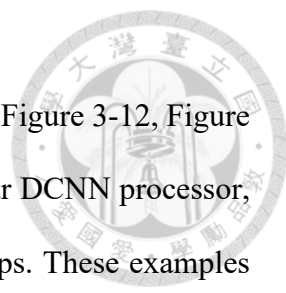
ReLU

Padding:
Zero Padding:
Zero padding adds zero elements around the edges of the input feature map during the convolution or pooling operation.

Replicate Padding:
Replicating border values for padding using the nearest element in each direction.

Input / Output

I. O.



In this section, three primary algorithm examples are presented, Figure 3-12, Figure 3-13 and Figure 3-14 ,each showcasing the desired capabilities of our DCNN processor, with a particular focus on the immediate reuse of output feature maps. These examples demonstrate crucial concepts, such as performing convolution with simultaneous padding and linear filtering, such as ReLU activation. Sequentially, each case will be elaborated on, providing a comprehensive exploration of the proposed functionalities.

Based on our previous analysis, it is evident that convolution is often followed by another convolution operation. In many models, padding with the appropriate size is applied, which allows the output feature map's dimensions to remain consistent with the input feature map's dimensions. Therefore, this requirement is also considered in the algorithm examples presented here.

The first algorithm, Figure 3-12 illustrates the process of performing two consecutive convolutions with zero-padding. The second algorithm, Figure 3-13, example involves directly applying the ReLU activation, a linear adjustment, to the result of a convolution operation with zero-padding. The third algorithm, Figure 3-14, example builds upon the previous one, Figure 3 13, applying the ReLU activation, a linear adjustment, to the result of a convolution operation with zero-padding. Subsequently, either max pooling or average pooling is performed on the output feature map.

However, in addition to the aforementioned three examples, three issues were encountered during the design and derivation of such algorithms as follow:

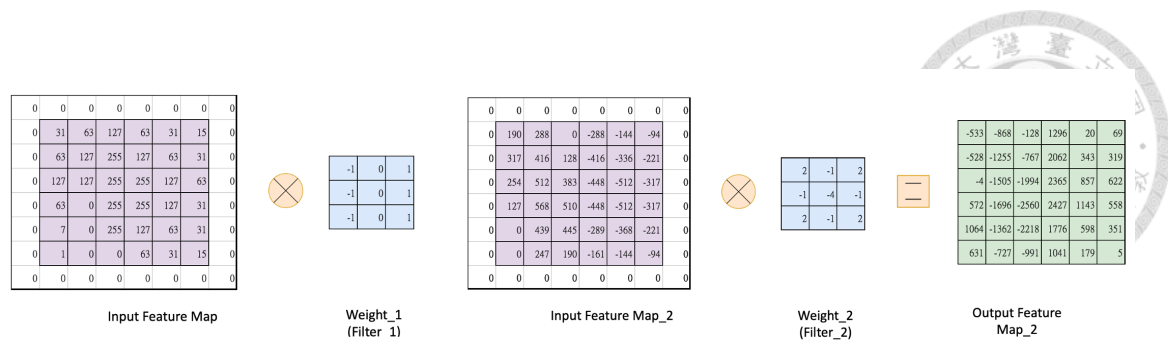


Figure 3-12 Convolution with zero-padding followed by a second convolution.

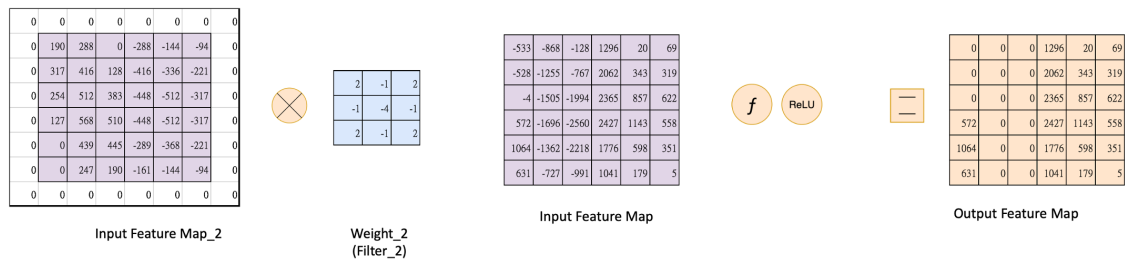


Figure 3-13 Convolution with zero-padding followed by immediate ReLU activation.

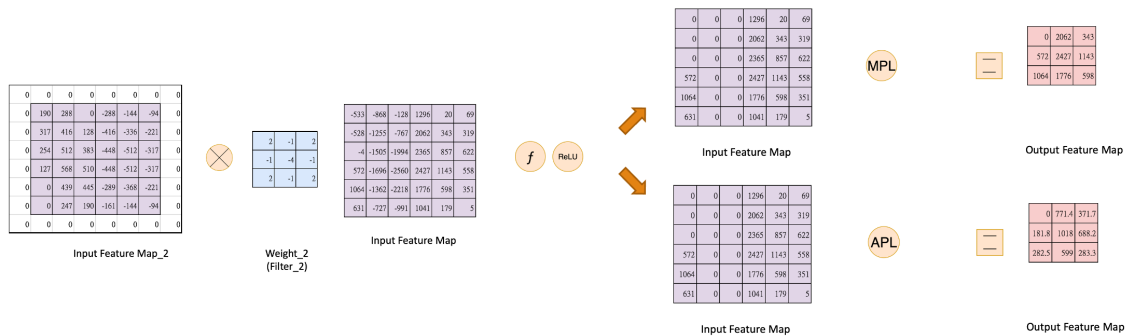


Figure 3-14 Convolution with zero-padding followed by immediate ReLU activation and pooling

First, continuously merging layer-to-layer connections resembles extracting specific cases within special cases, which may decrease the likelihood of alignment with the actual models' required computational processes.

Second, adhering to the Input/Output considerations mentioned in section 3.5, where "the output data dimensions must not exceed the input data dimensions," poses challenges for the pooling operation. Figure 3-14, this operation requires either waiting for multiple outputs from the same PE or simultaneously accessing outputs from different PEs as pooling input. As a result, the pooling operation in algorithm design has been partitioned, enabling output feature map reuse while not mandating the same PE to handle Example Three computations.

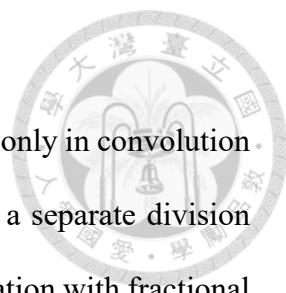
Third, the phenomenon observed in pooling layers also applies to the input data for the second convolution operation, necessitating a similar algorithm partitioning for the second convolution to be executed in individual PEs.

3.7 Divide-and-conquer

Table 3-6 presents a summary of the algorithm partitioning issues discussed in the previous section, indicating whether different input data sources (provided by the system buffer or obtained from individual PE computations) can be supported by a single PE. Additionally, it provides an overview of the arithmetic and logical processing unit specifications required for each function.

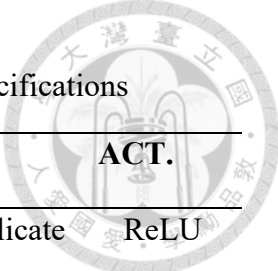
These specifications also encompass the design principles, system assumptions, and requirements formulated based on the research objectives previously outlined.

Several observations can be made:

- 
- A. The arithmetic units for addition and multiplication are used not only in convolution but also in average pooling calculations. To avoid introducing a separate division unit, the division operation will be implemented using multiplication with fractional values, requiring a floating-point multiplier.
- B. ReLU functionality involves frequent comparisons with the value 0 to determine whether to retain or output a value. Therefore, a zero register is necessary to facilitate these comparisons.
- C. Max pooling requires comparators to compare input data values. Due to the two-dimensional data being transformed into a vector as per the design principles, parallel comparators are not suitable. Thus, a feedback loop is essential for this purpose.
- D. The input data, represented as a one-dimensional vector, undergoes sequential processing, where multipliers can only perform constant multiplications on two elements at a time, and accumulators require feedback circuits.

These assumptions also address the intention to achieve hardware component reusability, data channel sharing, and the control complexity of data input/output and data reuse.

Table 3-6 Algorithm partitioning and hardware mapping specifications



Functions	Conv.	Pooling	Padding		ACT.	
Subfunction	-	M-PL.	A-PL.	Zero	Replicate	ReLU
Functions can be processed in a single Processing Element (PE)						
Input data from system	Yes	Yes	Yes	Yes	Yes	Yes
Input data from a PE	No	No	No	Yes	Yes	Yes
Arithmetic and logical processing specifications						
Adder	Yes	No	Yes	No	No	No
Multiplier	Yes	No	Yes	No	No	No
Comparator	No	Yes	No	No	No	Yes
Feedback	Yes	Yes	Yes	No	No	No
Zero Buffer	Yes	No	Yes	Yes	No	Yes
Const. Buf.	No	No	Yes	No	No	No

3.8 System Flow Design

Figure 3-15 Flowchart of an eight-stage DLA PE architecture. The various hardware submodules designed based on the typical layer sequence found in DCNN models are organized into eight main stages. The division of these stages is based on design experience and microarchitecture techniques.

In summary:

Stage 1:

Input data is transferred to the register file, and instruction decoding is performed.

Stage 2:

Operands are transferred from the register file (1 operand/path) and decisions are made regarding padding.

Stage 3:

Each operand on the main data channel is checked for zero values. Weight reuse and input feature map reuse are also processed in this stage. Depending on the operation mode, operands are directed to either the normal data path or the specialized path for Max-Pooling.

Stage 4 & 5:

If no zeros are encountered, two operands are multiplied; otherwise, the multiplier is shut down to save power. As multiplication operations are more

complex and time-consuming than simple logic circuits, two stages are allocated for the multiplication operation. In the case of Max-Pooling, operands are simply transferred.



Stage 6:

The new operand is added to the previous one, and accumulation continues until reset (until $k \times k$ times). For Max-Pooling mode, operands are transferred without accumulation.

Stage 7:

This stage determines whether to activate (ReLU) the data or directly transfer it to the next stage. For Max-Pooling mode, new operands are compared with the previous ones (until $k \times k$ times).

Stage 8:

The output operand (Output Feature Pixel) is transferred to the next location based on the selected output mode/path.

These stages and their operations are defined with strict adherence to the research objectives, design principles, system assumptions, and hardware specifications established in the study.

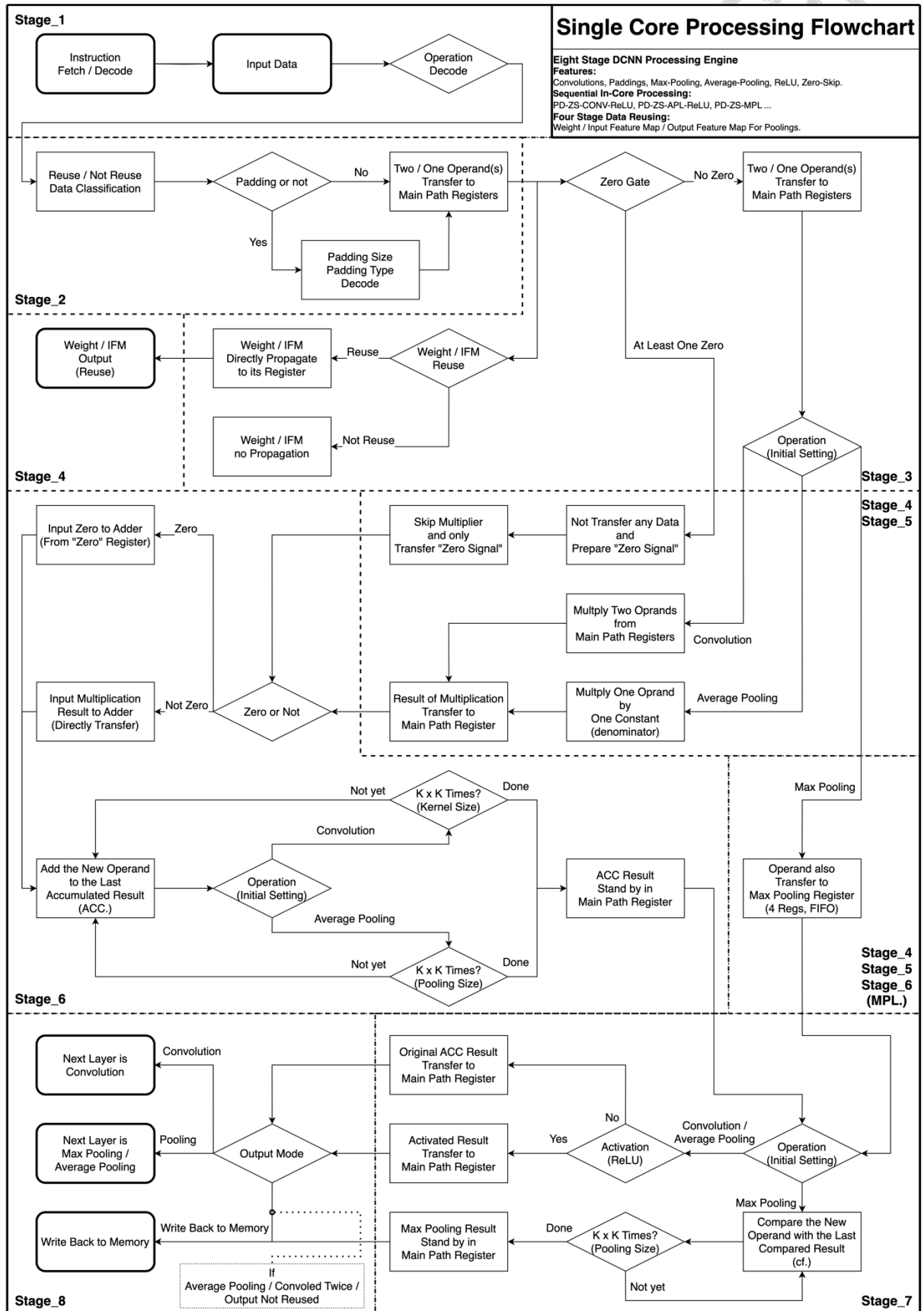
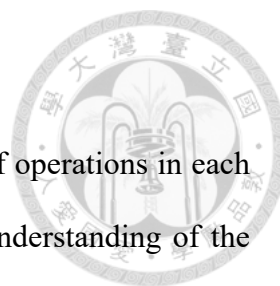


Figure 3-15 Flowchart of an eight-stage DLA PE architecture



3.9 Instruction Set Design

With the established data flow, hardware units, and the order of operations in each stage, the design of instructions can be guided, enabling a better understanding of the decoding sequence.

This approach to the instruction set differs from the typical pursuit of simple instructions as fast as possible, which is commonly found in RISC CPUs. Instead, I prioritize designing instructions based on functionality, following a hierarchy depicted in Figure 3-16. In this hierarchy, higher-level instructions are more application-specific, while lower-level instructions are closer to fundamental operations. Table 3-7, provides examples of such instructions.

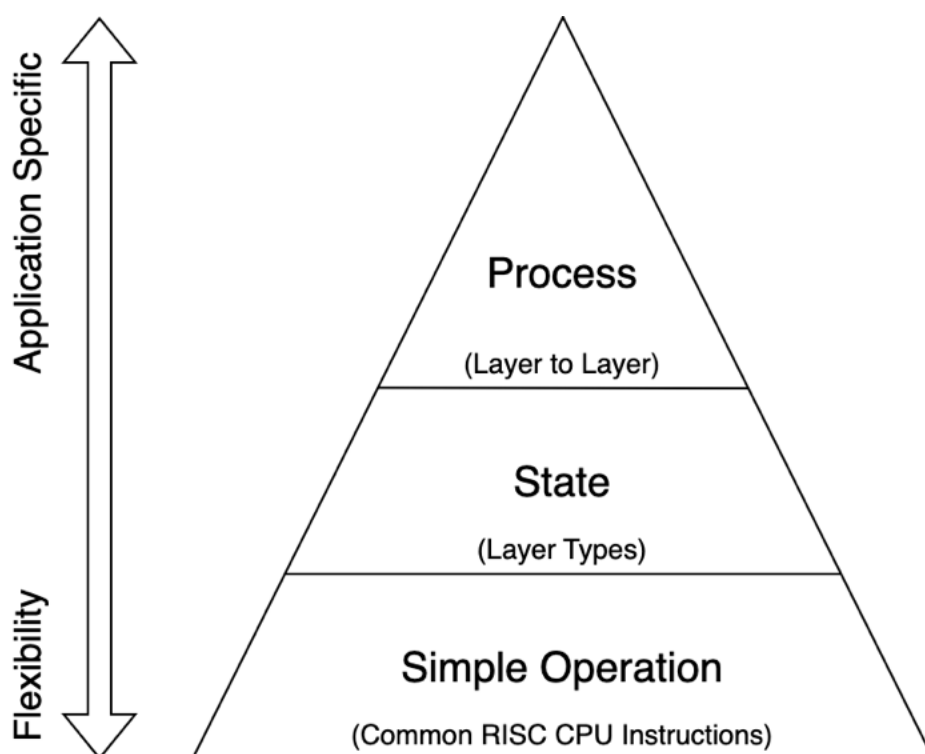


Figure 3-16 Instruction set hierarchy: from application-specific to fundamental operations

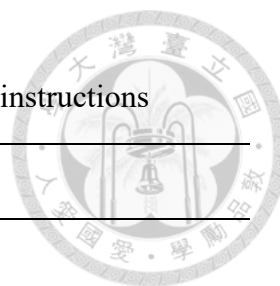


Table 3-7 Examples of application-specific and fundamental instructions

Instruction Hierarchy	Instruction Examples
Layer to Layer	C-C, C-MPL, C-APL ...
Instruction Level	
Layer	Convolution: kernel size, stride, padding type or not, filters parameters.
Instruction Level	Pooling: pooling type, kernel size, stride.
	Activation: ReLU.
Simple	Arithmetic: add, sub, addi, addu, subu, addiu, mul, mult ,div.
Instruction Level	Logical: and, andi, or, sll, srl.
	Data Transfer: load, store, move.
	Others: Conditional Branch, Comparison, Unconditional Jump, System Calls ...

Table 3-8 Instruction set architecture in this study's design

Instruction modes	Opcode	Next Stage Mode	Padding Size	Padding Type	Reuse
	Convolution	Conv	0	non-PD	both
	Max-Pooling	Max-Pooling	1	zero-PD	Weight
	Average-Pooling	Average-Pooling	2	Same-PD	IFM
	Other	Write Back	X	X	Non (X)

Kernel Row Size	Kernel Column Size	Stride Row Size	Stride Column Size	Activation
1	Same as Row	X	Same as Row	Non
2	1	1	1	ReLU
3	2	2	2	
4	3	3	3	
5	4			
6	5			
7	6			
8	7			
9				
10				
11				
X				

Table 3-8 presents the various operation codes, output mode codes, padding sizes, padding types, data reuse channel mode switch codes (two flags to control two channels), kernel row and column sizes, stride row and column sizes, and activation function's flags to determine if ReLU is needed for each calculation.

Each instruction's order is also related to the RTL (Register Transfer Level) combinational circuit's decoding sequence or instruction switching.

In section 5.1 Operations Cycle Time Analysis, further investigation will be conducted to examine how different kernel sizes can impact the processing time required by the processor to produce the output.

Chapter 4 Implementation



4.1 Architecture Mapping to RTL

The experimental tools are established on a personal server with a valid licensed agreement.

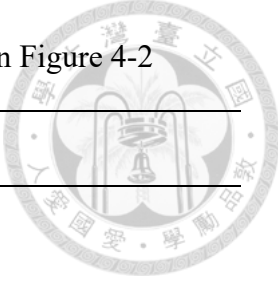
Table 4-1 Environment and Tools

Environment and Tools	
Computer	
Platform	Windows 11 Pro 22H2
CPU	Intel x64 Core I9-9980XE
Memory	DDR4 Quad-Channels 128GB
GPU	Nvidia RTX-2080Ti
HDL Simulation Software	
AMD Xilinx Vivado Edition 2023.1	
Intel Questa Starter FPGA Edition 2022.4	

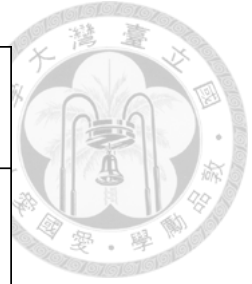
Figure 4-2 DLA PE architecture design details in RTL diagram. This diagram presents the DCNN processor architecture design in RTL view, divided into 8 stages by vertical dashed lines. The spacing between the dashed lines does not indicate any spatial or temporal relationships, as each stage corresponds to one complete clock cycle.

The diagram clearly illustrates the sequential arrangement of different functional hardware units, the presence of feedback circuits, and the number of channels. Both data and control signals propagate from left to right.

Table 4-2 Explanation of DLA PE architecture design details in Figure 4-2



Color-Coded Areas	
Yellow	Control signal lines
Blue	Weight reuse dedicated channel
White	Main processing channel
Purple	Input Feature Map reuse dedicated channel
Pink	Applicable channel for max pooling
Colored Lines	
Black	Data transfer channels
Yellow	Signal control lines
Green dashed	Virtual data transfer channels in pooling mode
Green solid	Additional physical channels introduced
Arrow	
Large hollow arrows	Represent input and output of data in the processor
Register Symbols	
Reg	Register
Zero	Storage for zero values
C	Buffer of constant values



Symbols	Explanation
	Control signal lines
	Weight Reuse dedicated channel
	Main Processing channel
	Input Feature Map Reuse dedicated channel
	Applicable channel for Max Pooling
→	Data transfer channels
→	Signal control lines
→	Virtual data transfer channels in Pooling
→	Additional physical channels introduced
→	Represent I/O data in the processor
◁ 0 1 2 3	Register
▷ 0 1 2 3	Storage for zero values
▷ 0	Buffer of constant values

Figure 4-1 Caption of DLA PE architecture design details in RTL diagram

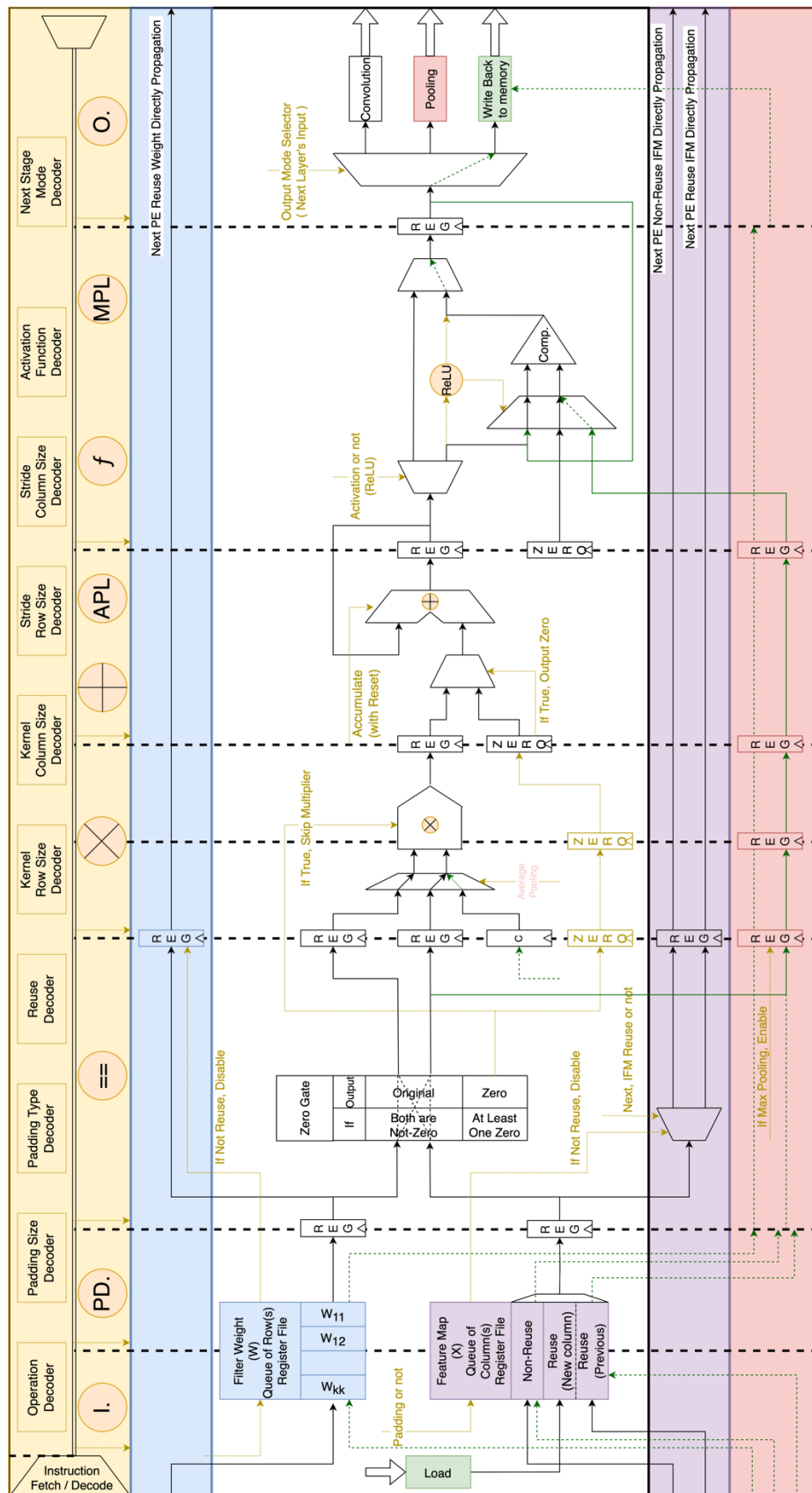


Figure 4-2 DLA PE architecture design details in RTL diagram

Chapter 5 Results and Discussion



5.1 Operations Cycle Time Analysis

This section aims to analyze the total cycle times per processing task (per output feature element) for convolution, average pooling, max pooling and activation in the designed DLA PE. Considering a fixed stage configuration, where each stage represents one cycle time, the investigation will focus on how various parameters impact the overall cycle times. The analysis will be based on four main types of operation codes (Opcodes) within the instruction set architecture.

A. Opcode Type 1: convolution (Figure 5-1, Table 5-1)

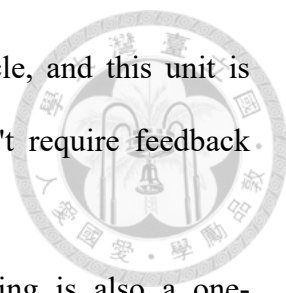
- Accumulating time increases by the square of the kernel size.
- It only obtains one result from the multiplier per cycle time, affecting the overall cycle times.

B. Opcode Type 2: average pooling (Figure 5-2, Table 5-2)

- Accumulating time increases by the square of the kernel size.
- The processing procedure is similar to convolution, as both operations obtain only one result from the multiplier per cycle time, impacting the overall cycle times.

C. Opcode Type 3: max pooling (Figure 5-3, Table 5-3)

- Comparison time increases by the square of the kernel size.

- 
- The comparator only compares two elements in one cycle, and this unit is shared with the activation layer of ReLU, which doesn't require feedback circuits.
 - Similar to convolution and average pooling, max pooling is also a one-dimensional vector operation, and its cycle times are affected by the total number of elements in the kernel.

D. Opcode Type 4: activation (Figure 5-4, Table 5-4)

- The total cycle times are the same as when the kernel size is 1x1, which is the minimum processing time for this processor: 8 cycles.
- Regardless of the number of ReLU data points, in the context of the processor's architecture, processing a batch of n data points is equivalent to individually processing each element n times with a 1x1 data kernel.

Upon observation of the four modes' graphs and tables, it becomes evident that the three curves are identical, shown in Figure 5-4.

Despite the computations originating from different sub-units, the use of RTL and

Table 5-1 Convolution processing cycles - cycle counts for each stage

Convolution	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7	Stage 8	
Kernel Size	IF/ Data In	PD/ Element Out	Zero Gate	Multiplication		ACC.	Activation	Data Output	Total Cycles
1	1	1	1	1	1	1	1	1	8
3	1	1	1	1	1	9	1	1	16
5	1	1	1	1	1	25	1	1	32
7	1	1	1	1	1	49	1	1	56
9	1	1	1	1	1	81	1	1	88
11	1	1	1	1	1	121	1	1	128

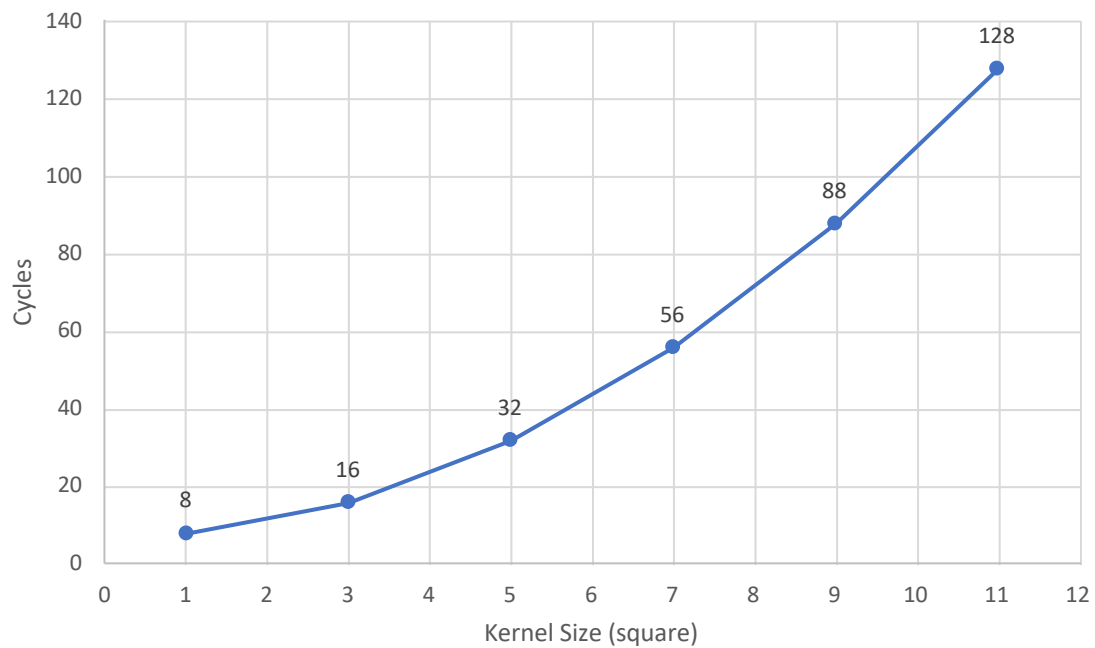


Figure 5-1 Total cycles for convolution processing per output feature element

Table 5-2 Average pooling processing cycles - cycle counts for each stage

Average Pooling	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7	Stage 8	
Kernel Size	IF/ Data In	PD/ Element Out	Zero Gate	Multiplication		ACC.	Activation	Data Output	Total Cycles
1	1	1	1	1	1	1	1	1	8
2	1	1	1	1	1	4	1	1	11
3	1	1	1	1	1	9	1	1	16
4	1	1	1	1	1	16	1	1	23
5	1	1	1	1	1	25	1	1	32
6	1	1	1	1	1	36	1	1	43
7	1	1	1	1	1	49	1	1	56
8	1	1	1	1	1	64	1	1	71
9	1	1	1	1	1	81	1	1	88
10	1	1	1	1	1	100	1	1	107
11	1	1	1	1	1	121	1	1	128

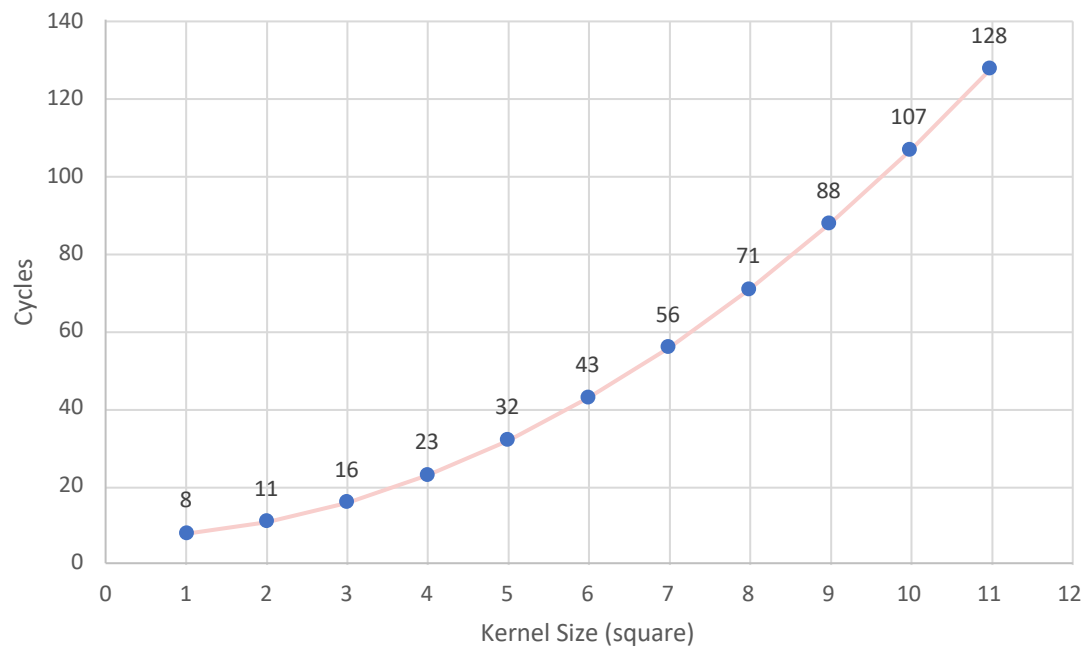


Figure 5-2 Total cycles for average pooling processing per output feature element

Table 5-3 Max pooling processing cycles - cycle counts for each stage

Max Pooling	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7	Stage 8	
Kernel Size	IF/ Data In	PD/ Element Out	Zero Gate	Transfer	Transfer	Transfer	Comp.	Data Output	Total Cycles
1	1	1	1	1	1	1	1	1	8
2	1	1	1	1	1	1	4	1	11
3	1	1	1	1	1	1	9	1	16
4	1	1	1	1	1	1	16	1	23
5	1	1	1	1	1	1	25	1	32
6	1	1	1	1	1	1	36	1	43
7	1	1	1	1	1	1	49	1	56
8	1	1	1	1	1	1	64	1	71
9	1	1	1	1	1	1	81	1	88
10	1	1	1	1	1	1	100	1	107
11	1	1	1	1	1	1	121	1	128

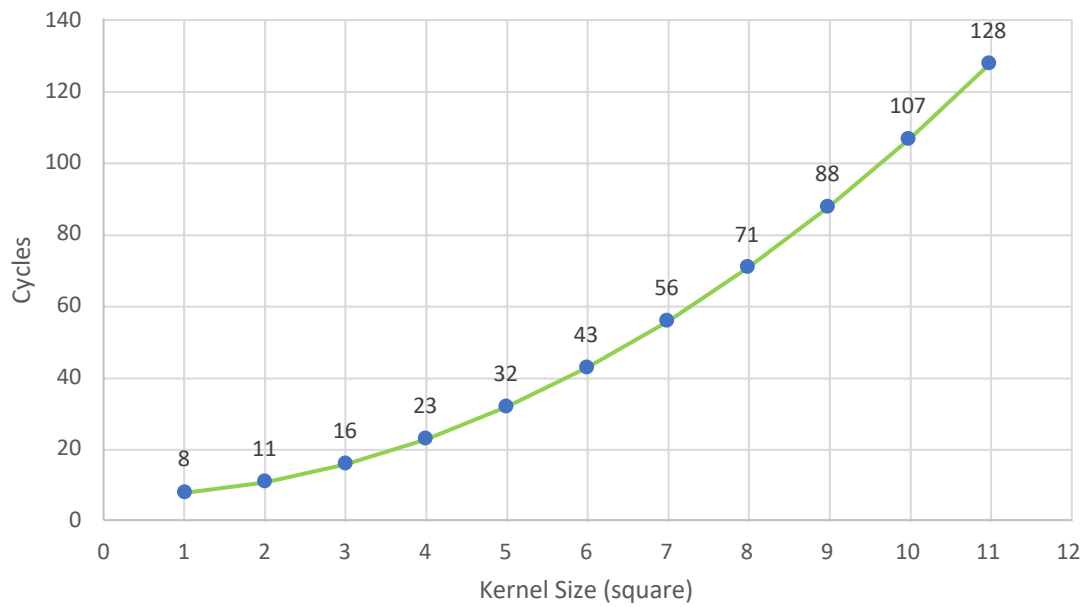


Figure 5-3 Total cycles for max pooling processing per output feature element



Table 5-4 Activation ReLU processing cycles - cycle counts for each stage

Activation	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7	Stage 8	
Kernel Size	IF/ Data In	Element Out	Zero Gate	Transfer	Transfer	Transfer	Activation	Data Output	Total Cycles
1	1	1	1	1	1	1	1	1	8

fixed-stage strategy in the design ensures that the execution time for these instructions is solely dependent on the product of the kernel size: row and column. Additionally, this multiplied value applies to kernels of different operation modes, even though these kernels may have different meanings for different computations of layers in DCNN. Consequently, this results in the same point on the curve when the multiplied value is the same.

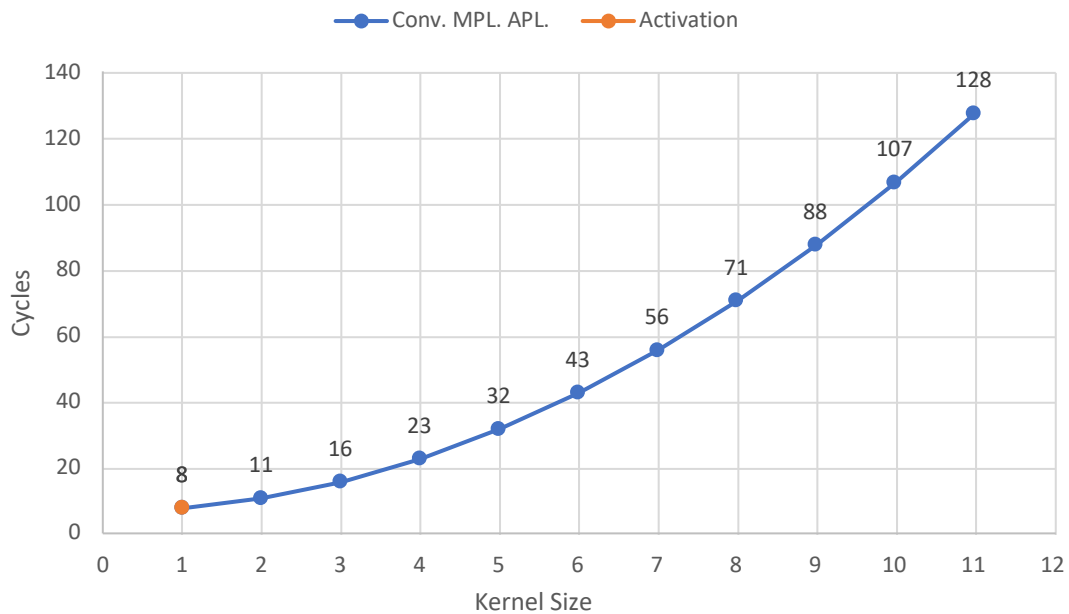
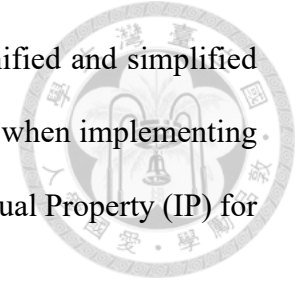


Figure 5-4 Unified operation time curve for diverse processing modes in this architecture

This characteristic is highly advantageous as it allows for a unified and simplified approach in analyzing different models and their computation times when implementing this architecture in a system-on-chip (SOC) or using it as an Intellectual Property (IP) for other computing systems.



5.2 Performance Analysis and Simulation

In Figure 5-5, the performance curve of a single PE at different clock rates is simulated. The actual chip's clock rate may vary due to factors such as chip fabrication process, manufacturing quality, synthesis tools used, Placement and Routing (P&R), and may be directly limited by factors such as Thermal Design Power (TDP), reducing voltage values and clock rates to save power.

Figure 5-6 extends the performance curve from Figure 5-5 to simulate multi-core parallel computing. These results are based on simulations with stable high hardware utilization rates; however, actual performance may decrease due to parameter variations, function switching, or different instruction sequences.

5.3 Data Reuse Rate

With the incorporation of data reuse channels and two separate register files (Input data buffers) with the functionality of data reuse classification, the processor enables full

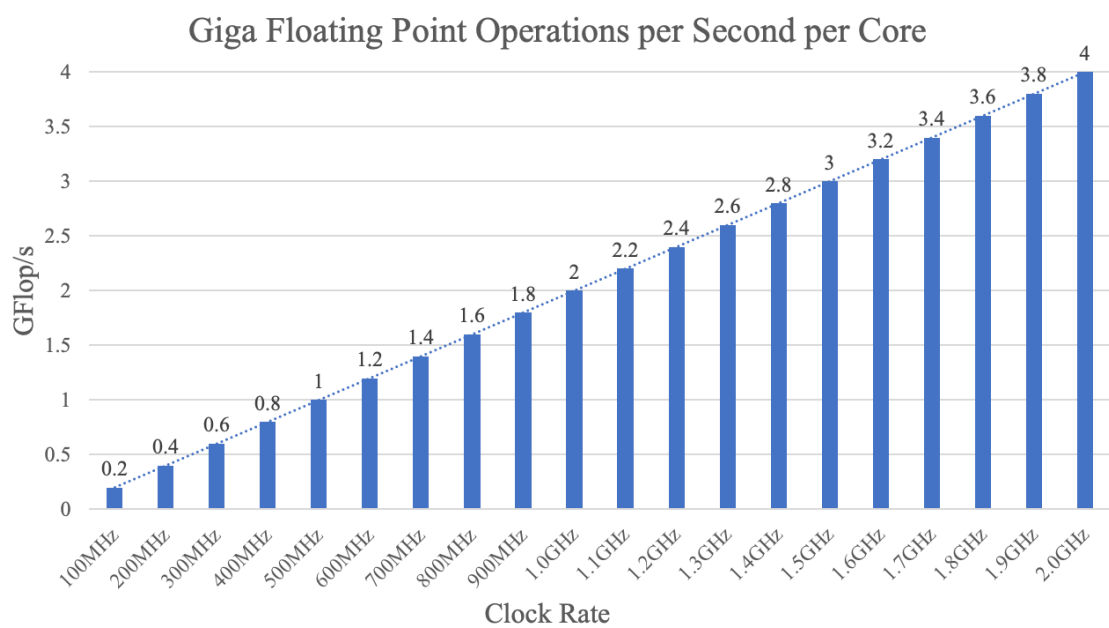


Figure 5-5 Performance of a single PE at different clock rates

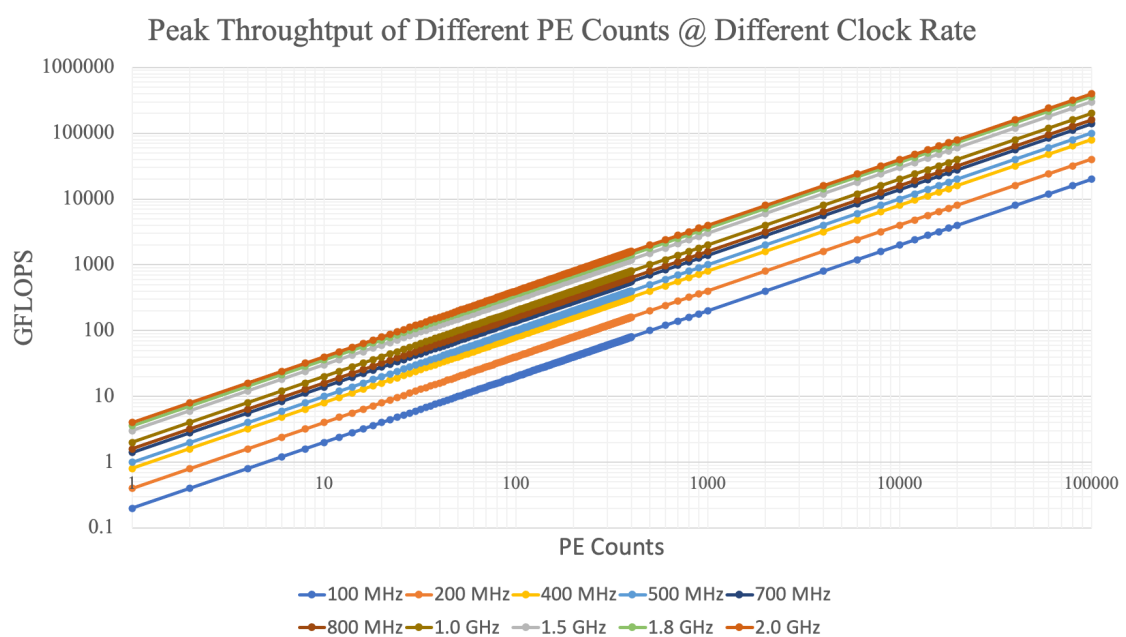


Figure 5-6 Performance of multiple PEs at different clock rates



reusability of weights and their propagation to neighboring processors.

Additionally, the input feature map can achieve a higher reuse rate, particularly with larger kernel sizes.

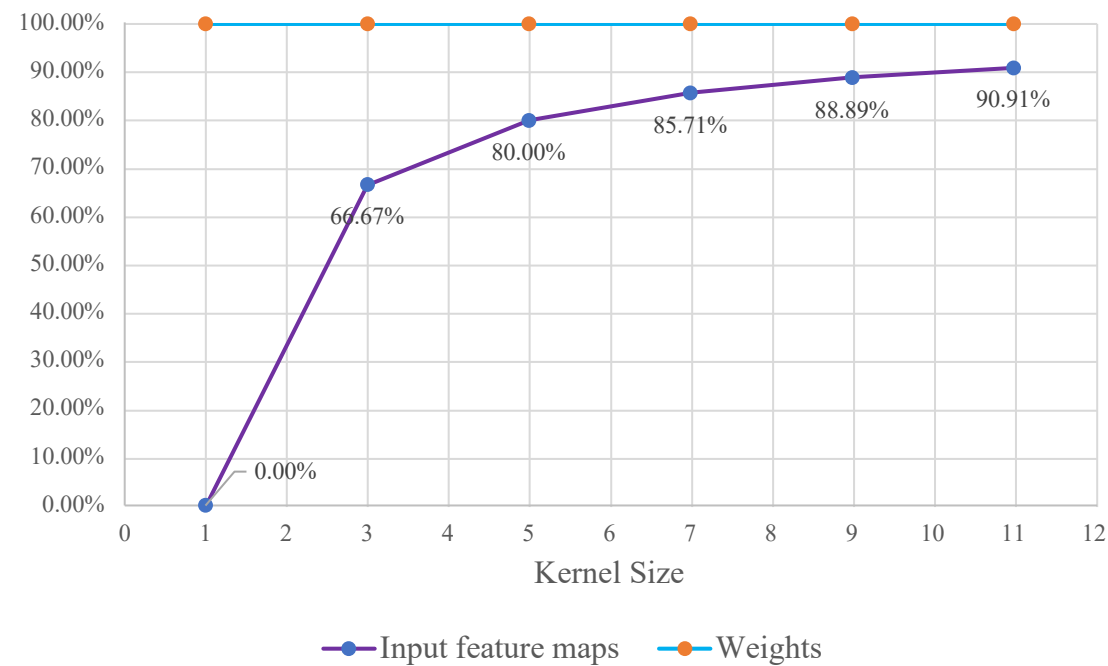


Figure 5-7 Data reuse rates for input feature maps and input weights with varying kernel sizes

5.4 Design Comparison

After conducting cycle time analysis and defining the operations and their respective durations, as well as obtaining information on data reuse rates, our study enables us to make a comprehensive comparison between our designed architecture and well-known

AI architectures mentioned in the earlier literature, as well as typical computing architectures like MIPS CPU [52] :



This study compared to:

A. MIPS CPU [52]

- Advantages: Offers ease of control with lower instruction complexity and a higher instruction level.
- Disadvantages: Limited flexibility is a result of this design's focus on arithmetic logic unit (ALU) and the significant reduction in basic functionalities, as well as the absence of a SISD (Single Instruction, Single Data) architecture.
- Similarities with This Study: Both architectures have fixed stages.

B. Intel Ice Lake [11]

- Advantages: Features a smaller core and a significantly reduced Instruction Set Architecture (ISA), resulting in lower system complexity.
- Disadvantages: Flexibility, generality, compiler and its ecosystem.



C. Google TPU [17]

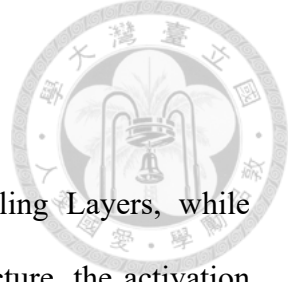
- Advantages: lower system complexity and the need to perform activations in the core.
- Disadvantages: Single core has higher complexity.
- Similarities with This Study: Both architectures implement weight stationary operations and adopt SIMD (Single Instruction, Multiple Data) principles.

D. NVIDIA NVDLA [38] [39] [40]

- Advantages: having lower system complexity.
- Disadvantages: More interconnected path to memory at each unit and NVDLA not need to wait for unnecessary cycles. It also supported reshape operations
- Similarities with This Study: Reconfigurability.

E. UCLA DCNN Accelerator [27]

- Advantages: Shares Max-Pooling with Activation Unit, reducing the hardware complexity. Not Requires large kernel size of weight separation for certain operations.
- Disadvantages: In this design, max pooling and average pooling share the main channels, whereas UCLA's design can separate these two operations into two independent hardware units.
- Similarities with This Study: This design average pooling layer is processed within the convolution unit (CU).



F. MIT EYERISS [54]

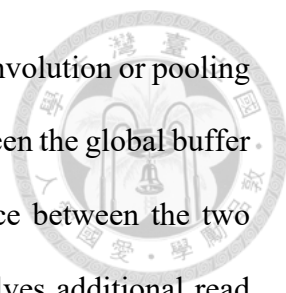
- Advantages: This work's design supports Average-Pooling Layers, while EYERISS lacks this feature. Additionally, in this architecture, the activation layer (ReLU) is executed immediately after convolution in the core, whereas EYERISS requires data to be transferred from the Global buffer to DRAM before performing this operation.
- Disadvantages: This work's single PE is more complex and larger, whereas MIT's design features a well-established mesh network for data routing, which is a capability lacking in this study's architecture.
- Similarities with This Study: Both adopt a row stationary approach, utilize vector input, and incorporate a Zero-Gate mechanism to skip zero-value transfers.

5.5 Data Path Comparison

Comparing these two procedures in Table 5-5, this study's design in the DCNN processor has a more streamlined and efficient approach, with fewer intermediate data transfers and cache writes. This results in reduced latency and improved overall processing efficiency compared to a traditional CPU using conventional instructions.

Table 5-5 Convolution of inner product of a tile of input matrix, flow comparison

This study's architecture	Traditional CPU using basic instructions
1. Instruction decoding	1. Instruction decoding
2. Two operands are input	2. Two operands are input
3. The two operands are multiplied, and the result is transferred to a register	3. The two operands are multiplied, and the result is transferred to a temporary register
4. The temporary result is fed into the accumulator	4. The result in the temporary register is written to the second or third-level cache
5. The two operands' sum is calculated, and the result is transferred to a register	5. Instruction decoding
6. Linear adjustment or waiting for one cycle before transferring the result to the next register	6. The result of the multiplication and the partial sum are written to temporary registers
7. The final result is written back to the buffer or used as the operands for the next computation	7. The two operands' sum is calculated, and the result is transferred to a register
	8. The result in the register is written back to the second or third-level cache

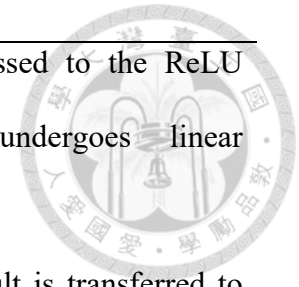


If the ReLU output needs to be immediately used for the next convolution or pooling operation, in MIT EYERISS V1, it requires reading and writing between the global buffer and DRAM before being processed in the PEs. The main difference between the two architectures lies in the MIT EYERISS V1 ReLU step, which involves additional read and write operations between the global buffer and DRAM. On the other hand, in the design of this study's architecture, ReLU only requires one additional cycle, and the data movement occurs internally within the processor, significantly reducing power consumption and latency.

Table 5-6 Comparison of convolution operation followed by immediate ReLU activation layer flows

This study's architecture	MIT EYERISS v1
1. Instruction decoding	1. Instruction decoding
2. Two operands are input	2. Two operands are input
3. The two operands are multiplied, and the result is transferred to a register	3. The two operands are multiplied, and the result is transferred to a register
4. The temporary result is fed into the accumulator	4. The temporary result is fed into the accumulator
5. The two operands' sum is calculated, and the result is transferred to a register	5. The two operands' sum is calculated, and the result is transferred to a register
6. Linear adjustment and the result is transferred to the next register	6. The result is written to the global buffer

-
- | | |
|---|--|
| 7. The final result is written back to the buffer | 7. The result is passed to the ReLU function and undergoes linear adjustment |
| | 8. The adjusted result is transferred to the next register |
| | 9. The final result is written back to DRAM |
-



5.6 Benchmark

In this study, benchmarks are conducted using three different methods. Figure 5-8 , compares the peak throughput (Giga Operations per Second) under similar conditions by aligning the PE counts and clock rates of our study with three target comparisons. However, the magnitude of Google TPU v1's performance [17] far exceeds others, making it unmarkable on the graph; its results are mentioned in the description following the figure title.

In Figure 5-9, chip performance is normalized per single PE for comparison. From the trend lines of operations per second per PE, it is observed that higher values indicate a single PE handling more computational tasks in the same timeframe. However, this does not necessarily translate to better power efficiency or total performance.

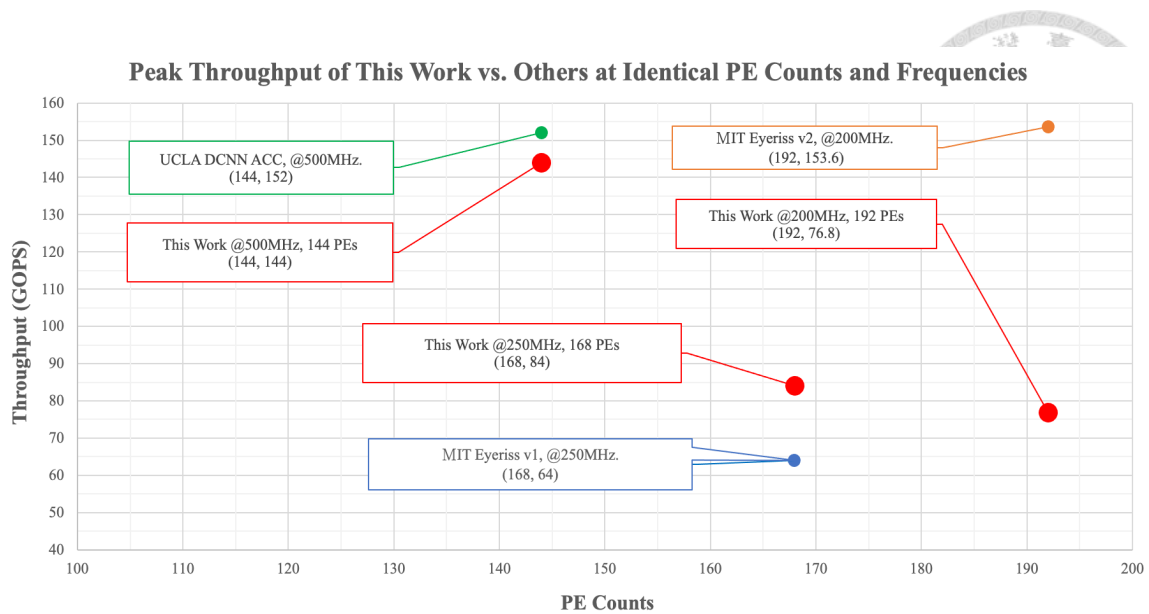


Figure 5-8 Peak throughput comparison of this work and other DLA architectures at identical pe counts and frequencies. (Google TPU v1 achieves 92 TOPS with 65536 PEs at 700MHz, whereas this work achieves 91.75 TOPS.) [17] [27] [54] [56] .

The performance metrics of these five works in Figure 5-9 , while displayed on the same chart, vary in terms of input data dimensions and PE functionality. A simpler microarchitecture with more MACs achieves a higher OPS/PE value. This work features a more intricate PE processing flow compared to others, offering superior functionality and 1-D vector input with straightforward data input control. It demonstrates good performance in the mid-range due to optimal hardware reuse and hardware sharing (refer to Table 5-7).

This metric serves as a fundamental indicator for parallel computation. Within the same performance range, it indicates similar computational capabilities. If additional functionalities are supported, it signifies enhanced processing capacity.

A multi-functional single core reduces hardware module diversity, simplifying system complexity and control difficulty. Minimizing data movement outside the core

can improve power efficiency and reduce latency. This is beneficial for enhancing overall performance when scaling up hardware system.

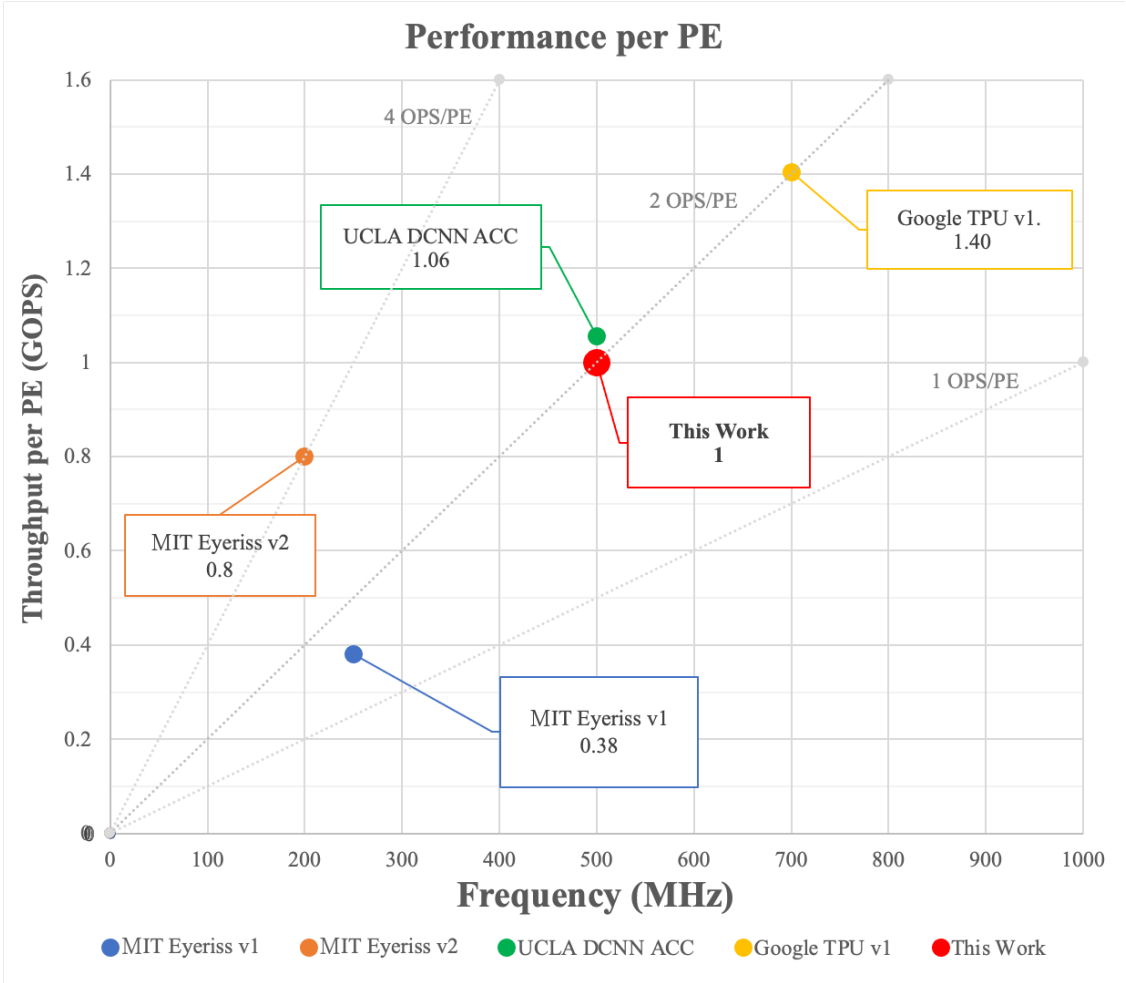


Figure 5-9 Performance per single PE at its peak clock rate. The higher the GOPS per PE, the further to the upper left. 1 OPS/PE represents 1 Operation per Second per Processing Element. [17] [27] [54] [56] .

Table 5-7 primarily compares which functionalities each design supports and whether these functionalities can share computing resources (hardware). Additionally, it compares the forms of PEs and the dimensions of data computation. Finally, it compares where common CNN operations are processed in these designs, and an analysis of function latency is provided, comparing the approximate cycle time required for computation in these DLA architectures for various CNN layers.

Table 5-7 Performance and Functionality Comparison. (Tr. Is data transfer time)

	[55] MIT EYERISS v1	[27] UCLA DCNN ACC.	[17] Google TPU v1	This work
Basic Info.				
Peak Throughput	64 GOPS	152 GOPS	92 TOPS	144 GOPS
Precision	Int 16	Int 16	BFP 16	FP 32
Clock Rate	100-250 MHz	500 MHz	700 MHz	500 MHz
No of PEs	12x14	144	256x256	12x12
No of MACs	168	144	65536	144
Functions				
Max Pooling Support	Yes	Yes	Yes	Yes
Average Pooling Support	No	Yes	Yes	Yes
Activation Function	ReLU	ReLU	ReLU, Sigmoid, tanh	ReLU
Conv. & MPL. HW. Reuse	No	No	No	Yes
Conv. & APL. HW. Reuse	No	Yes	No	Yes
Conv. & ACT. HW. Sharing	No	No	No	Yes
Instruction Type	-	-	CISC	CISC
Function Operation				
PE Calculation Dimension	1-D Vector	2-D Array	2-D Matrix	1-D Vector
PE Type	MAC	Multiplier	MMU	General Function
Convolution	PE	PE(CU)	PE	PE
Max Pooling	Off-PE	Off-PE(CU)	Off-PE	PE
Average Pooling	Not supported	PE(CU)	Off-PE	PE
Activation Function	Off-PE, WB to DRAM	Off-PE(CU)	Off-PE	PE, Immediately
Latency of Functions				
Multiplication Cycle Time	2 cycles	1 cycle	1 cycle	2 cycles
Accumulation Cycle Time	1 cycle	Tr.+1 Cycles (Off CU)	1 cycle	1 cycle
Convolution Cycle Time	K^2+1 cycles	$K+3+Tr.+ACC.$ cycles	$2K-1$ cycles	K^2+7 cycles
Max Pooling Cycle Time	-	$K=2,3.$ Tr.+1,3 cycles (Off)	-	K^2+7 cycles
Average Pooling Cycle Time	Not supported	$K+3+Tr.+ACC.$ cycles	-	K^2+7 cycles
Activation Func. Cycle Time	$\gg 1$ cycles (Read Out)	$\gg 1$ cycles (Read Out)	-	1 cycle

Chapter 6 Conclusions and Future Work



6.1 Pros and Cons

In this chapter, an exploration of the six major research objectives established earlier has been conducted as follows:

1. Layer-Compatible:

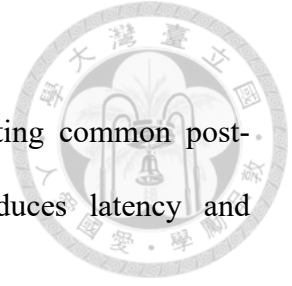
The design focuses on convolutional operations as the core process, allowing various forms of pooling, ReLU, and other network layers to achieve the same results.

2. Timing Identical:

With a minimum of 8 stages, the architecture ensures consistent processing times, even for shorter max pooling operations where an additional 3-cycle delay is introduced. This approach allows turning off the multiplier and accumulator in the certain stages, leading to reduced power consumption. The analysis in section 5.1 shows that processing times remain the same for kernels of identical size.

3. PEs Scalable:

Adhering to the previous point, different processors can perform different layer operations, such as some cores executing convolution while others handle pooling. This flexibility enables the scalability of core numbers based on specific requirements or grouping.



4. Efficient Data Movement:

Beyond energy consumption considerations, incorporating common post-convolution functionalities in the subsequent cycle reduces latency and accelerates processing speed.

5. Data Reusable:

The design implements two reuse channels, reducing the data throughput from system memory or global buffer into the processors array, avoiding bottlenecks, and efficiently reusing neighboring data within the processor (near-memory computation) to reduce data movement power consumption.

6. Module / Path Shared:

The architecture allows the convolutional operation multiplier to be shared by using a denominator value for average pooling. Similarly, the comparator used in max pooling can be shared with a third channel for comparison with zero values, enabling ReLU, the most commonly used activation layer, to share this hardware. Additionally, data transfer channels and other sub-modules are shared, reducing the need for dedicated hardware for other network layers and saving area, especially as non-convolution operations are infrequent based on model statistics presented earlier.

However, the fixed-stage RTL architecture sacrifices some flexibility compared to other edge CPUs. For instance, in order to share the multiplier within convolution and average-pooling, all inputs must be in floating-point format instead of integers, despite

the better precision it offers. Additionally, if there is no requirement for a linear filter before the output, there will be a one-stage (cycle) delay.



6.2 Research Contributions

First, the research methodology of this paper involves a soft-hardware co-design approach. The design flow begins with statistical analysis of layer parameters from popular DCNN models, followed by function and specification definition, algorithm design and segmentation, and finally algorithm partitioning and hardware mapping specifications. The result is a clear system flowchart and instruction set of PEs.

Second, this study concentrates on designing a single core that supports full functionality with low latency, aiming for a balanced design, while most papers focus on issues such as NOC or parallel computing PE grouping. Detailed internal designs of the core in RTL are publicly disclosed, covering register quantity, individual datapaths, hardware units, and distinct instruction decoding for each stage.

Third, the DLA architecture of this research ensures ease of control and scalability due to its consistent cycle time design across all functions. Benchmark tables demonstrate the highest hardware resource sharing and lowest activation latency. With all functions processed in-core, this approach effectively avoids the delays during off-PE data movement, enhancing overall efficiency and performance.

6.3 Research Limitations

Throughout this research, one of the most significant challenges I encountered was the lack of access to free EDA (Electronic Design Automation) tools. These tools are

often highly confidential and require collaboration with semiconductor foundries like Taiwan Semiconductor Manufacturing Company (TSMC) or United Microelectronics Corporation (UMC), making it difficult to obtain free licenses. Additionally, the development process involved verification on FPGA platforms, which come with a hefty price tag and may not provide precise values for computation speed and time.

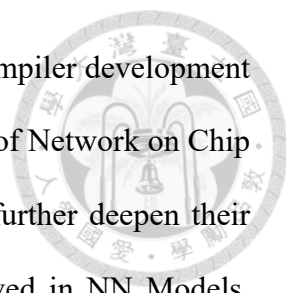
As a result, I ultimately decided to implement and verify the design using licensed software such as Intel Questa and AMD Xilinx Vivado, which allows for functional simulations. Furthermore, in the field of computer architecture, papers related to architectural-level hardware design often have limited detailed design information, and there are few papers that provide results from actual chip fabrication, leading to limited trust in the outcomes presented in the literature.

Compared to software applications or AI model applications in various fields, conducting research in this topic demands a higher entry barrier and is not accessible for enthusiasts.

Indeed, conducting research in this field demands a thorough understanding of algorithms, mastery of microarchitecture design principles, and substantial experience in IC (Integrated Circuit) design, including Verilog HDL and other related skills. The topic's complexity requires significant time and effort, making it a challenging endeavor that demands dedication and expertise.

6.4 Future Work

This is the inaugural investigation of DLA Architecture Design within my lab, with a specific emphasis on microarchitecture and core development. Despite this, numerous branches within this domain remain unexplored. As a result, I suggest that researchers



capitalize on the simplistic instruction set crafted in this study for compiler development to this general-purpose core. This initiative extends to the co-design of Network on Chip (NoC) in both software and hardware. Moreover, researchers can further deepen their exploration on core groups or core clusters with the trends observed in NN Models. Another good avenue is the investigation of optimization between NN model algorithms and compilers. Additionally, the integration of performance-oriented cores and efficiency-oriented cores is proposed as a prospective research direction.

I also recommend that researchers undergo relevant coursework in computer architecture, understand prevalent NN algorithms, comprehend each stage in the PE pipeline within this research, and its instructions before further investigations.



Reference



- [1] Albert Chun Chen Liu, Oscar Ming Kin Law. Jul 2020. Deep Learning-Hardware Design, 3-2 to 3-8. Chuan Hwa Book Co., LTD.
- [2] Albert Chun Chen Liu, Oscar Ming Kin Law. Jul 2020. Deep Learning-Hardware Design, 3-35. Chuan Hwa Book Co., LTD.
- [3] Alex Krizhevsky et al. Jan 2012. ImageNet Classification with Deep Convolutional Neural Networks, NIPS.
- [4] Alfredo Canziani, Eugenio Culurciello, Adam Paszke. Apr 2017. An Analysis of Deep Neural Network Models for Practical Applications. arXiv:1605.07678v4.
- [5] Anantha P. Chandrakasan, S. Sheng, R. W. Brodersen. Apr. 1992. Low-power CMOS digital design. *IEEEJ. Solid-State Circuits*, vol. 27, no. 4, pp. 473-484.
- [6] Andres Rodriguez et al. Jan 2018. Lower Numerical Precision Deep Learning Inference and Training. p.6 and p.18.
<https://www.intel.com/content/dam/develop/external/us/en/documents/lower-numerical-precision-deep-learning-jan2018-754765.pdf>. Accessed: Feb 2022.
- [7] Andrew G. Howard et al. Apr 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861.
- [8] Chia-Hsiang Yang. 2021. VLSI Signal Processing. Lecture. Graduate Institute of Electronics Engineering, National Taiwan University.
- [9] Christian Szegedy et al. Sep 2014. Going deeper with convolutions. arXiv: 1409.4842v1.

- 
- [10] Chuck Moore, April 27, 2011. Data Processing in Exascale-class Computing Systems. The Salishan Conference on High Speed Computing.
- [11] David L. Mulnix. 2017. Intel® Xeon® Processor Scalable Family Technical Overview.
<https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html>. Accessed: Feb 2022.
- [12] David Silver et al. Jan 2016. Mastering the game of Go with deep neural networks and tree search, Nature 529, 484–489 (2016).
<https://doi.org/10.1038/nature16961>.
- [13] Dejan Markovic. 2006. A Power/Area Optimal Approach to VLSI Signal Processing. Ph.D. Thesis, University of California, Berkeley.
- [14] Dejan Marković et al. Aug. 2004, “Methods for True Energy-Performance Optimization” IEEE Journal of Solid-State Circuits, vol. 39, no. 8, pp. 1282-1293.
- [15] Gao Huang et al. Jan 2018. Densely Connected Convolutional Networks. arXiv:1608.06993.
- [16] Google Cloud TPU. 2022. Introduction to Cloud TPU.
<https://cloud.google.com/tpu/docs/intro-to-tpu>. Accessed: Sep 2022.
- [17] Google Cloud TPU. 2022. System Architecture.
<https://cloud.google.com/tpu/docs/system-architecture-tpu-vm>. Accessed: Sep 2022.
- [18] Hung-Yi Lee. Machine Learning. Course (2020, Spring). Electrical Engineering of National Taiwan University.

- 
- [19] Hung-Yi Lee. Machine Learning. Course, ML2021 week3 3/12 Convolution Neural Network (CNN). Electrical Engineering of National Taiwan University.
- [20] Jiin Lai. Sep 2021. Application Acceleration with High-Level Synthesis. Lecture. hls-for-software-define-hardware.pdf. p.8.
- [21] Joel Emer et al., 2017. Hardware Architectures for Deep Neural Networks, ISCA Tutorial p.271-p.282, 2017. <https://eems.mit.edu/wp-content/uploads/2017/06/ISCA-2017-Hardware-Architectures-for-DNN-Tutorial.pdf>. Accessed: Oct 2022.
- [22] Jorge Albericio et al. June 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. ISCA 2016.
- [23] Kaiming He et al. Dec 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385.
- [24] Karen Simonyan, Andrew Zisserman. Apr 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556.
- [25] Keras. 2023. Keras Applications. <https://keras.io/api/applications/>. Accessed: Jan 2023.
- [26] Lauro Rizzatti. November 23, 2018. A Breakthrough in FPGA-Based Deep Learning Inference.
- [27] Li Du, Yuan Du et al. Aug 2017. A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things. IEEE DOI: 10.1109/TCSI.2017.2735490.

- 
- [28] Liang-Gee Chen. Oct 2021. Course: Computing Architecture and System design for AI Machine Learning, lecture Oct 14 – 28.
- [29] Lisa Su. Aug 2019. Delivering the Future of High-Performance Computing. Hot Chips: A Symposium on High Performance Chips. HC31-K1, 7:26-9:50.
- [30] Mark Horowitz, 2014. Computing's Energy Problem (and what we can do about it), IEEE ISSCC 2014.
- [31] Mark Sandler et al. Mar 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381.
- [32] Microsoft Azure Machine Learning, Jun 2023. How to Deploy FPGA Web Service, Figure “Silicon alternatives”. <https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service#fpgas-vs-cpu-gpu-and-asic> .Accessed: Jan 2022.
- [33] Mingxing Tan, Quoc V. Le. Sep 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946.
- [34] Mingyu Gao et al. April, 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. Stanford University. Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2017.
- [35] Norman P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Unit. Google, Inc. CA USA. ISCA 44th. arXiv: 1704.04760.
- [36] NVIDIA Developer. 2018. Hardware Architectural Specification. <http://nvidia.org/hw/v1/hwarch.html#hardware-architectural-specification>. Accessed: Oct 2022.

- [37] NVIDIA Developer. 2018. Winograd Convolution Mode.
<http://nvdla.org/hw/v1/hwarch.html#winoarch>. Accessed: Oct 2022.
- [38] NVIDIA Whitepaper. 2016. NVIDIA Tesla P100. p7, p10 and p11.
<https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>. Accessed: Oct 2022.
- [39] NVIDIA Whitepaper. 2018. NVIDIA TURING GPU ARCHITECTURE. p58-p60. <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>. Accessed: Oct 2022.
- [40] NVIDIA Whitepaper. 2021. NVIDIA AMPERE GA102 GPU ARCHITECTURE Second-Generation RTX. p12-p17. <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf>. Accessed: Oct 2022.
- [41] Olga Russakovsky et al. Jan 2015. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision.
- [42] Peter J. Denning. July 2005. The Locality Principle. Communications of the ACM, Volume 48, Issue 7, (2005), Pages 19–24.
- [43] Shibo Wang, Pankaj Kanwar. Aug 2019. BFloat16: The secret to high performance on Cloud TPUs. Google TPU.
<https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus>. Accessed: Feb 2021.



- [44] TensorFlow. 2023. Module tf.keras.applications.
https://www.tensorflow.org/api_docs/python/tf/keras/applications. Accessed: Jan 2023.
- [45] Tien-Ju Yang et al. 2017. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. IEEE CVPR.
- [46] Ting-Ting Hwang. Computer Architecture Lecture. Feb 2011. (15A-17B) Single-Cycle Processor - 21B Pipelining. Department of Computer Science, National Tsing Hua University.
- [47] Ting-Ting Hwang. Computer Architecture Lecture. Oct 2021. (18AR-21BR) Pipelining. Department of Computer Science, National Tsing Hua University.
- [48] Ting-Ting Hwang. Computer Architecture Lecture. Oct 2021. (22AR-26BR) Memory. Department of Computer Science, National Tsing Hua University.
- [49] Ting-Yang Chen. 2023. An 8.1-to-353 TOPS/W Energy-Aware Deep-Learning Accelerator Supporting Dynamic Neural Networks. Master Thesis. Graduate Institute of Electronics Engineering National Taiwan University.
- [50] Vivienne Sze et al. Dec 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. p2308.
- [51] Vivienne Sze. 2019. Efficient Processing of Deep Neural Networks: from Algorithms to Hardware Architectures.
https://eyeriss.mit.edu/2019_neurips_tutorial.pdf.
- [52] Wikipedia. MIPS architecture processors. Oct 2020.
https://en.wikipedia.org/wiki/MIPS_architecture_processors. Accessed: Feb 2021.



- [53] Wikipedia. Systolic array. Oct 2020. https://en.wikipedia.org/wiki/Systolic_array.

This URL is provided by Google Cloud. Accessed: Feb 2021.

- [54] Yu-Hsin Chen et al., 2016. An Energy-Efficient Reconfigurable Accelerator for Deep Learning Convolution Networks. IEEE ISSCC 2016.

- [55] Yu-Hsin Chen. Aug 2018. Design for Highly Flexible and Energy-Efficient Deep Neural Network Accelerators. <https://www.youtube.com/watch?v=brhOo-7NS4> . Accessed: Oct 2022.

- [56] Yu-Hsin Chen. June 2018. Architecture Design for Highly Flexible and Energy-Efficient Deep Neural Network Accelerators.



