Hardware-Software Co-Exploration And Optimization For Next-Generation Learning Machines

Chen Chunyun

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

2024

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

30 Oct. 2023

Date

Chen Chunyun

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

8 Nov. 2023

.....

Date

.....

Prof. MOHAMED MOSTAFA SABRY ALY

Authorship Attribution Statement

This thesis contains material from 4 papers published or under review in the following peer-reviewed conferences and journals in which I am listed as an author.

Chapter 3 is published as **Chunyun Chen**, Zhe Wang, Jie Lin and Mohamed M. Sabry Aly, "Efficient Tunstall Decoder for Deep Neural Network Compression", in *2021 28th ACM/IEEE Design Automation Conference (DAC)*.

The contributions of the co-authors are as follows:

- Prof. Mohamed M. Sabry Aly and Wang Zhe provided the initial project direction.
- I designed and implemented the hardware accelerator, conducted the experiments, and prepared the manuscript drafts.
- Wang Zhe supported the experiments and gave me the weight data.
- The manuscript was revised by Prof. Mohamed, Dr. Lin Jie, and Wang Zhe.

Chapter 4 is published as 1 conference paper and under review in 1 journal paper: (i) as **Chunyun Chen**, Tianyi Zhang, Zehui Yu, Adithi Raghuraman, Shwetalaxmi Udayan, Jie Lin and Mohamed M. Sabry Aly, "Scalable Hardware Acceleration of Non-Maximum Suppression", in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. (ii) as **Chunyun Chen**, Tianyi Zhang, Lantian Li, Jie Lin, and Mohamed M. Sabry Aly, "ShapoolNMS: Towards Scalable Hardware Acceleration for Non-Maximum Suppression in Object Detection", in Top-tier journal (under review).

The contributions of the co-authors are as follows:

- Dr. Zhang Tianyi, I, and Prof. Mohamed M. Sabry Aly provided the initial project direction.
- I designed and implemented the hardware accelerator, conducted the experiments, and prepared the manuscript drafts.
- Dr. Zhang Tianyi supported the experiments and prepared a Python framework.
- Li Lantian helped with the synthesis and PnR of the accelerator.
- Yu Zehui, Raghuraman Adithi, and Udayan Shwetalaxmi supported the experiments and part of the RTL implementations and verifications.
- The manuscript was revised by Prof. Mohamed, Dr. Lin Jie, and Dr. Zhang Tianyi.

Chapter 6 is published as **Chunyun Chen**, Lantian Li, and Mohamed M. Sabry Aly, "ViTA: A High Efficient Dataflow and Architecture for Vision Transformers", in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

The contributions of the co-authors are as follows:

- I provided the initial project direction, designed and implemented the hardware accelerator, conducted the experiments, and prepared the manuscript drafts.
- Li Lantian helped with the data preparation.
- The manuscript was revised by Prof. Mohamed and Li Lantian.

30 Oct. 2023

Date

Chen Chunyun

"I am a slow walker, but I never walk back."

—Abraham Lincoln

To my dear family

Acknowledgements

Coming to the end of my Ph.D. journey, I would like to take this opportunity to thank all the people who have steadfastly supported me in my research and thesis writing, who were there for the ups and downs of my Ph.D. journey, and who have enriched my Ph.D. journey.

Foremost, I owe my profound appreciation to my supervisor, Prof. Mohamed Mostafa Sabry Aly. His encouragement, wisdom, and unwavering patience have been instrumental in shaping my academic pursuits. I have learned a lot from him, not only in my writing skills but also in my research methodology. He also provided enough freedom for me to explore my research interests and develop my research skills. His attitude towards research and his passion for teaching have inspired me a lot. It was Prof. Mohamed who provided me with the opportunity to work with top researchers in the field of hardware acceleration and Deep Learning from the Agency for Science, Technology and Research (A*STAR).

I appreciate the guidance and feedback from my doctoral committee members: Professors Anupam Chattopadhyay, Liu Weichen, and Kim Tae Hyoung.

Additionally, I extend my gratitude to anonymous reviewers whose valuable comments and suggestions have elevated the quality of my thesis.

I would also like to thank Dr. Darayus Adil Patel, Dr. Dutt Arko, and Dr. Zhu Shien for their help and guidance in my research methodologies and writing. It was Dr. Patel who encouraged me when I was struggling with my research. He is a very nice friend who always helps me when I need help, and I have had many memorable experiences with him. It was Dr. Arko who helped me to improve my writing skills and provided me with many valuable suggestions for my thesis. Li Lantian also helped me review my papers and the thesis before submission, and I am very grateful for her help. We also had many interesting research discussions together. It was Dr. Zhu, and other lab mates in Prof. Liu Weichen's lab, who

started to help and guide me in my research when I was an undergraduate visiting student in Prof. Liu's lab.

I would like to extend my special thanks to Dr. Liu Jiuyang (Sequencer), who led me into the world of Chisel, which boosted my research productivity significantly and made my research more enjoyable even before I started my Ph.D. journey.

My collaborative experiences have been enriched by the team from A*STAR. I'm grateful to Dr. Lin Jie, Dr. Wang Zhe, Dr. Zhang Tianyi, Dr. Geng Xue, Zhao Bin, Jin Bo, and Qu Xuhong. They explain their understanding of the algorithms and hardware design to me selflessly. In particular, Zhao Bin introduced me to many variable knowledge and experiences in the ASIC backends.

Special thanks to my other collaborators from NTU, Chen Xiaowei, Yu Zehui, Liu Nanxi, and Shantanu Raoke for their collaboration in the research projects. My collaborator Wang Yang from Microsoft Research Asia has also provided me with many valuable suggestions for my research. He also shared lots of valuable state-of-the-art research papers with me.

I am also grateful to our assistant manager, Jeremiah Chua for their constant support and assistance in IT matters.

I am very grateful to my friends, Liu Yilun, Ai Junyu, Fan Chaoyan, Dr. Wang Penglin, Dr. Wang Huanyu, Jiang Songting, Dr. Gao Haoyuan, and my friends from NTU, Dr. Zhao Xuejiao, Dr. Wang Tan, Dr. Jiang Yuming, and Dr. Yang Ze. We have had many happy and unforgettable memories together. WeChat group members in "meme group" and "Remain true to our original aspiration" have also provided me with a lot of fun and emotional support.

Lastly, the emotional bedrock of my journey has been my family. I would like to thank my family for psychologically supporting me throughout my study, and overcoming the sleepless nights and the stress of my Ph.D. journey. My parents, Chen Xiaoyong and Tang Xuanping have always been there for me, and I am very grateful for their love and support. My grandfather, Chen Xinqing, who every time I remember always gave me encouragement and support. I miss you so much, Grandpa. My girlfriend, Ma Saijia, has always been there for me, spending time with me, and encouraging me when I am extremely stressed and depressed. I am very grateful for her love and support.

Abstract

In an era dominated by the rapid evolution of Machine Learning (ML), particularly Deep Learning (DL), the efficient deployment of learning algorithms on power- and area-constrained hardware remains a paramount challenge. The scaling up of DL models to trillions of parameters and trillions of computation operations, challenges the modest gains in energy efficiency and memory density derived from silicon scaling, making current DL hardware systems unsustainable.

Therefore, this thesis delivers a comprehensive investigation into hardware-software co-design and optimization for next-generation learning machines, especially the co-design of both the special function hardware and the end-to-end full workload hardware, with detailed system-level impacts of DL hardware accelerators.

The key design metrics for next-generation learning machines are energy efficiency, performance, and area overheads. To enable DL workloads running on resource-constrained hardware platforms, reducing the memory footprint is essential. One way to do this is through efficient entropy coding schemes. Commonly employed Fixed-to-Variable (F2V) entropy coding methods, *e.g.*, Huffman coding and Arithmetic coding, are hardware-unfriendly, and cannot fully benefit from the resulting reduced memory requirement. We introduce adopting Tunstall coding — a Variable-to-Fixed (V2F) coding scheme, for DNN models compression and introduce two Tunstall decoders — the Logic Oriented and the Memory Oriented decoders, with up to a 20× decrease in memory usage and a 100× reduction in energy consumption compared to 32-bit DNNs. Furthermore, these decoders process data 3× to 6× faster than F2V coding schemes.

Apart from the convolutional layer in Convolutional Neural Networks (CNNs) and the Multi-Head Attention (MHA) in Transformers, DL workloads also contain non-linear operations that are not easily parallelizable, posing a challenge in hardware implementations. One of which, Non-Maximum Suppression (NMS), is a critical

step in object detection frameworks, and is a computational bottleneck when mapping the frameworks into the hardware system due to its computational intensity. Existing NMS optimizations don't effectively parallelize on ASIC platforms. The introduced ShapoolNMS overcomes this limitation. Empowered by both low computation complexity and hardware/software co-optimization, ShapoolNMS is up to $42,713\times$ faster than conventional GreedyNMS software implementations.

In this thesis, we also look beyond a single layer in the DL workloads, and introduce two end-to-end workload accelerators for the entire CNN-based and Transformer-based DL workloads, respectively. Current DL accelerators mainly target either the convolutional operations of CNNs or the Multi-Head Attention (MHA) in Transformers. The acceleration of the entire workload is less explored. This thesis introduces CNN-DLA, a Chiplet-based scalable hardware accelerator for CNN-based models with a showcase of ResNet-152, and ViTA for the ViT workload. With the introduced cross-layer optimization dataflow, CNN-DLA achieves significant memory requirement reductions by 84.85%, and the 44-Chiplet achieves a performance of 68 FPS on ResNet-152 with full-HD images as input. Similarly, ViTA reduces memory requirements by 40.5% and adaptable performance of 0.20-16.38 TOPS, with the area and power consumption of 2.00-6.79 mm^2 and 0.22-10.40 W, respectively, making it suited for diverse applications.

Additionally, we also provide detailed guidelines for integrating the introduced accelerators into a real hardware platform, *i.e.*, the PULPissimo System-on-Chip (SoC) platform, including the interfaces, register map, and the finite state machine (FSM) of the integrated accelerators.

Overall, this thesis provides a foundation for a scalable DL accelerator and the hardware-software co-design and co-exploration for learning machines. The introduced methods not only address current hardware limitations but also set a direction for sustainable and efficient DL hardware systems in the future.

Contents

List of Figures

List of Tables

Abbreviations

ADC	Analog-to-Digital Converter
AE	Auto Encoder
AI	Artificial Intelligence
AIM	Associative Index Matching
ASIC	Application Specific Integrated Circuit
AvgPool	Average Pooling
bbox	Bounding Box
BM	Box Memory
BN	Batch Normalization
CIM	Computing In-Memory
CLO	Cross-Layer Optimization
CNN	Convolution Neural Network
COO	Coordinate
CS	Column Stationary
CSC	Compressed Sparse Column
CSF	Compressed Sparse Fiber
CSR	Compressed Sparse Row
CV	Computer Vision
DIMM	Dual-Inline Memory Module
DL	Deep Learning
DLA	Deep Learning Accelerator
DNN	Deep Neural Network
DPM	Deformable Part-based Model

DRAM	Dynamic Random Access Memory
DSA	Domain-Specific Accelerator
EDP	Energy-Delay Product
F2V	Fixed-to-Variable
FC	Fully Connected Layer
FFN	Feed-Forward Network
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FPN	Feature Pyramid Network
FPS	Frames Per Second
FSM	Finite-State Machine
GAN	General Adversarial Network
GB/s	Giga Bytes Per Second
GELU	Gaussian Error Linear Units
GEMM	General Matrix Multiply
GEMV	General Matrix-Vector Multiplication
GLUT	Group of LUTs
GNN	Graph Neural Network
GOPS	Giga Operations Per Second
GPGPU	General-Purpose Graphics Processing Unit
GRU	Gated Recurrent Unit
HMPSoC	Heterogeneous Multi-Processor System on Chip
HOG	Histogram of Oriented Gradients
HWPE	Hardware Processing Engine
IFFT	Inverse FFT
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IMC	In-Memory Computing
IS	Input Stationary
ISC	Integral Stochastic Computing
IoU	Intersection over Union
LN	Layer Normalization

LO	Logic-Oriented
LSTM	Long Short-Term Memory
LUT	Lookup Table
MAC	Multiply-Accumulate
MKICU	MaxPool Kernel Index Compute Unit
ML	Machine Learning
MLP	Multilayer Perceptron
MMIO	Memory-Mapped I/O
MMU	Matrix-Multiplication Unit
MO	Memory-Oriented
MRAM	Magnetoresistive Random Access Memory
MaxPool	Maximum Pooling
NFU	Neural Function Unit
NLP	Natural Language Processing
NMC	Near-Memory Computing
NMS	Non-Maximum Suppression
NVDLA	NVIDIA Deep Learning Accelerator
OS	Output Stationary
OaA	Overlap-and-Add
PCM	Phase Change Memory
PE	Processing Element
PS MNMS	Pyramid Shifted MaxpoolNMS
PSB	Partial Sum Buffer
PSRR-MNMS	PSRR-MaxpoolNMS
PSum	Partial Sum
PnR	Place and Route
RLC	Run-Length Coding
RNN	Recurrent Neural Network
RPN	Region Proposal Network
RR	Relationship Recovery
RRW	Relationship Recovery Way

RS	Row Stationary
ReLU	Rectified Linear Unit
ReRAM	Resistive Random Access Memory
SGD	Stochastic Gradient Descent
SMM	Score Map Memory
SPS	Samples Per Second
SRAM	Static Random Access Memory
STT-MRAM	Spin Transfer Torque MRAM
SumPool	Sum Pooling
TCDM	Tightly Coupled Data Memory
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
TSV	Through-Silicon Via
TTA	Transport-Triggered Architecture
Tanh	Tangent Hyperbolic
V2F	Variable-to-Fixed
VMU	Vector-Multiplication Unit
ViT	Vision Transformer
WS	Weight Stationary

Chapter 1

Introduction

1.1 Background

Deep Neural Networks (DNNs) have witnessed exponential growth and played pivotal roles in numerous applications across various domains of Artificial Intelligence (AI), from image classification [1], object detection [2] to natural language processing [3].

Machine Learning (ML), a prominent subset of AI, involves algorithms that enable systems to learn from data and subsequently make predictions or decisions. Broadly categorized, ML algorithms can be either supervised, where the learning process relies on labeled data (*e.g.*, images with captions or text with sentiment labels), or unsupervised, which involves deriving patterns from unlabeled data (*e.g.*, clustering similar documents or finding patterns in images).

Deep Learning (DL), a more specialized branch of ML, employs artificial neural networks to model complex data structures. One class of networks termed Deep Neural Networks (DNNs) due to their multiple input, output, and hidden layers, are adept at abstracting complex features from the input data, mimicking the structure and function of biological neural networks in animal brains and surpassing the capabilities of shallow neural networks with a single hidden layer. DNNs can be used for various tasks related to images and language modeling, such as image generation [4], text summarization [5], speech recognition [6], *etc.*

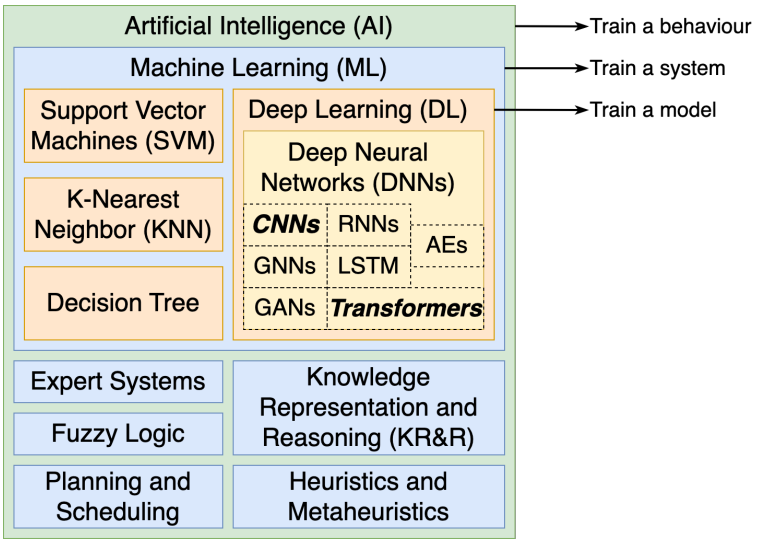


FIGURE 1.1: Relationship between AI, ML, DL, and DNNs.

Fig. 1.1 provides an illustrative representation of the relationship between AI, ML, DL, and DNNs.

DNNs are proliferating in numerous AI applications, thanks to their high accuracy. For instance, Convolution Neural Networks (CNNs), one variety of DNNs, are used in object detection for autonomous driving and have reached or exceeded the performance of humans in some objection detection problems [2]. Commonly adopted CNNs such as ResNet [7] and MobileNet[8] are becoming deeper (more layers) while narrower (smaller feature maps) than early AlexNet [1] and VGG [9].

Nevertheless, as demonstrated in Fig. 1.2, due to the race for better accuracy, the scaling up of DNN models, especially Transformers — another variety of DNNs, to trillions of parameters and trillions of Multiply-Accumulate (MAC) operations, as in the case of GPT-4 [10], during both training and inference, has made DNN models both data-intensive and compute-intensive, carrying heavier workloads on the memory capacity to store weights and computation. Besides, the traditional CNN-based workloads mainly contain compute-intensive convolutions. However, the Transformer contains more operations including both compute-intensive operations such as GEMM, and memory-intensive operations such as element-wise operations, and reduces operations in the non-linear functions, which poses significant challenges for the hardware accelerator design to support all the complex operations in Transformers and trade-off the requirements from both compute-intensive operations and memory-intensive operations and the PPA (performance,

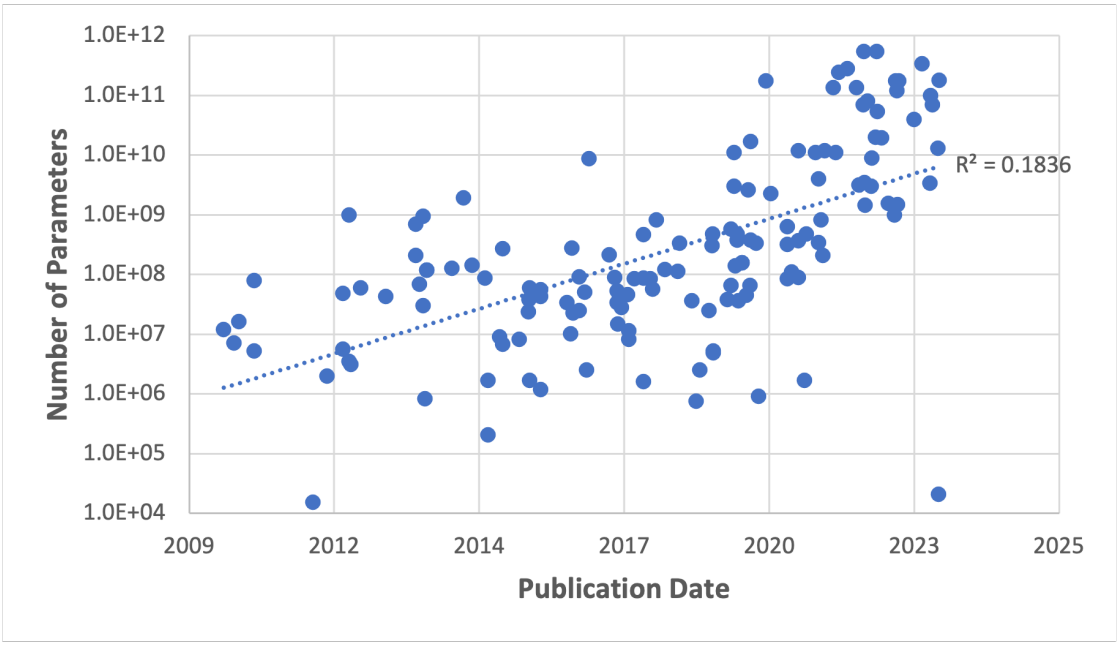


FIGURE 1.2: The number of parameters in DNN models has been increasing exponentially over time. Data from [11].

power, and area) of the chip. These pose a significant challenge for the deployment of these models in an area-efficient and power-efficient manner.

1.2 Motivation

Given these challenges, model compression is a vital research topic to alleviate the crucial difficulties of memory capacity from the algorithmic perspective. Pruning, quantization, and entropy coding are three directions of model compression for DNNs [12].

To handle the increasing computational complexity of DNNs from the hardware perspective, domain-specific accelerators (DSAs) have been developed to accelerate the inference process.

DSAs for DNNs fall into two categories: (1) Special function accelerators that enhance the efficiency of DNN hardware implementations, *e.g.*, accelerators that reduce the memory footprint of the DNN hardware implementations after model compression, accelerators for the special functions in DNN workloads, *etc.* (2) End-to-end workload accelerators, *e.g.*, accelerators for ResNet [7], and ViT [13].

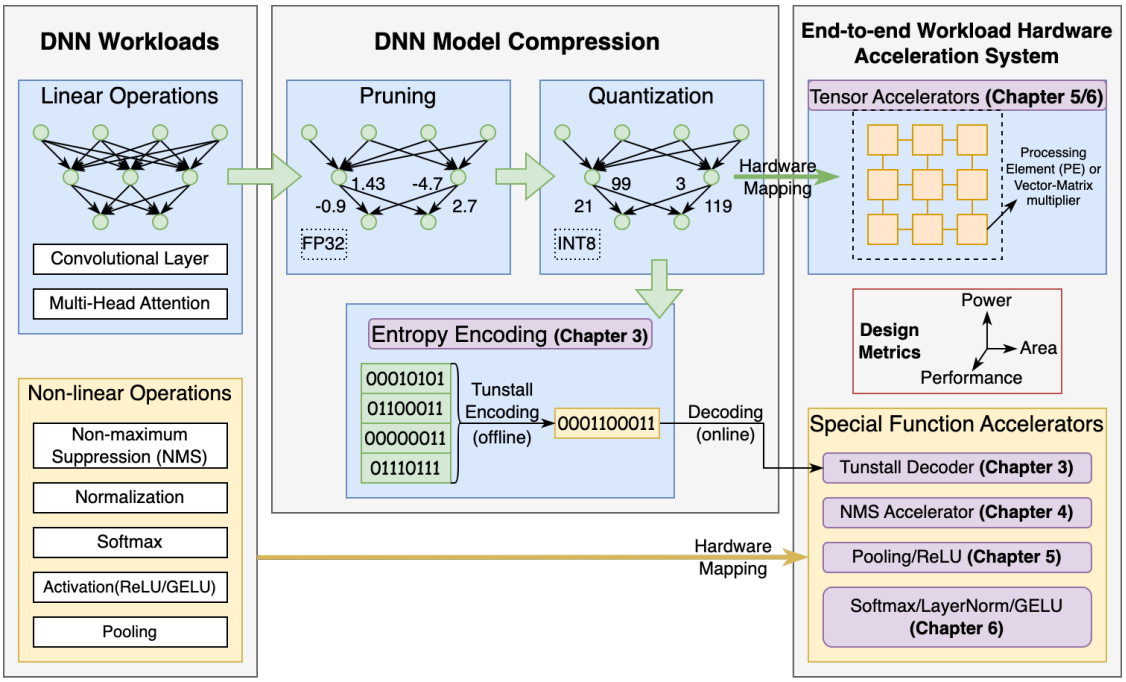


FIGURE 1.3: Overview of hardware-software co-design and optimization for next-generation learning machines.

Fig. 1.3 shows the overview of the hardware-software co-design and optimization for next-generation learning machines.

1.2.1 Need for DNN Models Compression

Pruning removes weights and corresponding computations that have minimal or no impact on accuracy across channels [14], filters [15], connections [12], or entire layers[16]. Quantization compresses DNNs by decreasing the number of bits per variable for features or weights. State-of-the-art quantization methods can compress the weights down to 4 bits from 8 bits without losing accuracy [17–19]. However, further reduction of the number of bits, for example compressing to 2-bits, can bring a noticeable accuracy drop [20].

The effectiveness of pruning and quantization can be enhanced with entropy coding for further model compression. Entropy coding focuses on encoding the quantized values of weights or features in a more compact representation by utilizing the peaky distribution of the quantized values, to achieve a lower number of bits per variable [20], without any accuracy loss. Fixed-to-Variable (F2V) schemes, such

as arithmetic coding [21, 22] and Huffman coding [12, 23] have been adopted in current workloads.

Current F2V methods, however, suffer from high decoding complexity of $O(n \cdot k)$, where n is the number of codewords (quantized values) and k is the reciprocal of compression ratio. Besides, as F2V codewords are of variable length, they can not be indexed (or memory usage would be very inefficient). Therefore, the encoded string needs to be processed on a bit-by-bit basis, which leads to difficulty in developing parallel implementations for the decoding. This is particularly critical since, unlike encoding which can be executed offline, decoding must be processed online. In real-time applications such as autonomous driving, if decoding is not efficient, the inference rate could be reduced. Hence, there is a pressing need for coding algorithms with both high compression ratios and low decoding complexities.

1.2.2 Need for Special Function Acceleration Hardware Solutions

Special function accelerators improve the efficiency of DNNs by supporting new numerical representations. For example, entropy coding reduces the memory footprint of the DNNs by 21.67×. However, this compression method requires a decoder to decode the compressed weights. Special decoders are required to decode the compressed weights while maintaining the throughput of the accelerator.

Special function accelerators are also important in DNN workloads. CNN-based object detectors generate highly overlapped bounding boxes around the ground-truth location of the objects to be detected, hence Non-maximum Suppression (NMS)—a post-processing step—is introduced to filter the overlapping detections to reduce the false positives. While convolution operations have undergone massive performance improvement, thanks to both hardware scalable architectures (*e.g.*, GPUs and DNN accelerators) and algorithmic optimizations (*i.e.*, depth-wise, pruning), the commonly adopted GreedyNMS has not witnessed significant improvement and thus has started to dominate the inference time. This disparity is evident in Fig. 1.4, showing the GreedyNMS's dominance in inference time, where we measure the total inference time—convolution operations are mapped to state-of-the-art GPUs while the GreedyNMS software is mapped on Intel I9-10900X CPU. GreedyNMS is notably time-intensive, consuming 2.76× more time than the entire

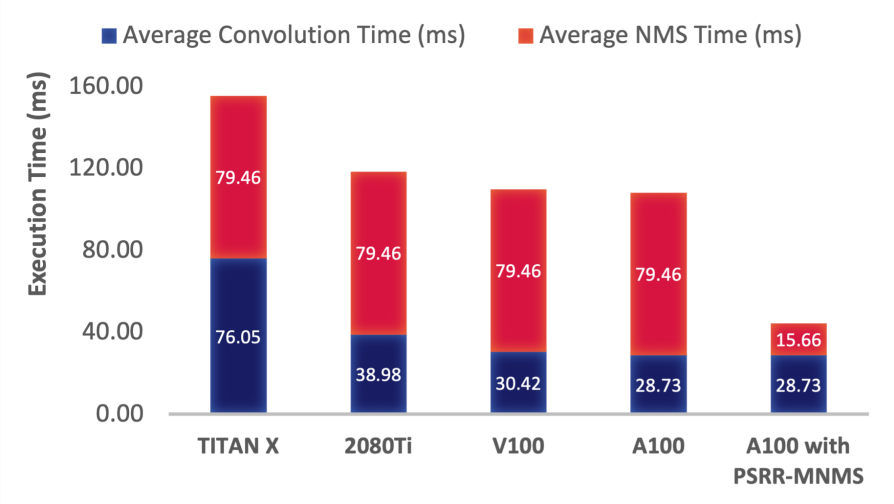


FIGURE 1.4: Execution time of inference on different GPU platforms and GreedyNMS or PSRR-MaxpoolNMS on Intel I9-10900X CPU per image. The detection network is Faster-RCNN with ResNet-50 backbone [7] on the PASCAL VOC benchmarking dataset. Only the top 6,000 bounding boxes are selected in Stage 1. The runtime is evaluated using the Python time library just between the convolution functions and NMS functions, which does not include the data transferring time between GPU and CPU.

inference operations when executing the Faster-RCNN on the A100 GPU with the PASCAL VOC dataset [24]. It is imperative to develop hardware accelerators for efficient NMS algorithms that are scalable by design. In one of the most recent object detection frameworks LOYOv8n [25], NMS occupies a significant portion (36.23%) of the execution time to be executed on NVIDIA GeForce RTX 3090, even if it is accelerated using global NMS instead of per-class NMS, as shown in Fig. 1.5. Meanwhile, energy efficiency and area are critical for IoT devices, where the power of NVIDIA GeForce RTX 3090 is 250W and that of A6000 is 300W, which is not suitable for IoT devices. Therefore, scalable hardware accelerators for efficient NMS algorithms are of the essence.

1.2.3 Need for DNN Acceleration Hardware Solutions

End-to-end workload accelerators empower the entire DNNs with high throughput and low power. Many studies have explored the benefits of implementing DL

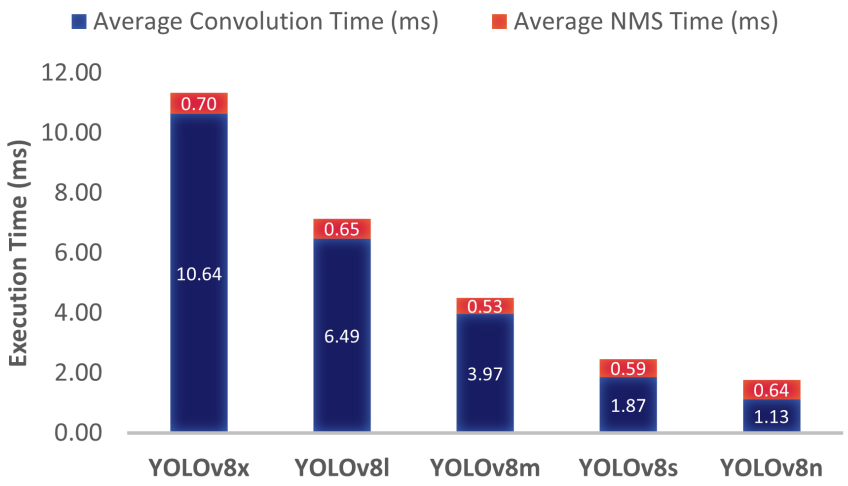


FIGURE 1.5: Average execution time of YOLOv8 models on GeForce RTX 3090. The benchmarking dataset is MS-COCO [26]. The global NMS is computed by the built-in Torchvision NMS executed on the GPU. NMS occupies over 36.2% of the execution time. The runtime is evaluated using the Python time library just between the convolution functions and NMS functions, which does not include the data transferring time between GPU and CPU. With the scalable architectures of GPUs, the convolution is further accelerated and the NMS will dominate the whole pipeline.

applications on hardware platforms such as Application Specific Integrated Circuits (ASICs) [27–31], general-purpose graphics processing units (GPGPUs) [32–34], and Field Programmable Gate Arrays (FPGAs) [35–40] for training and inference. As high-performance convolutions not only require high memory bandwidth but also consume a significant portion of energy on data movements, Near-Memory Computing (NMC) and In-Memory Computing (IMC) are utilized in some studies [41–52] to further optimize the energy consumption by putting the processing logic into memory while own high memory bandwidth. Other than processing convolution directly, some accelerators also yield algorithms such as Fast Fourier Transform (FFT) [33, 34, 53, 54], Strassen algorithm [55] and Winograd algorithm [39, 40, 56–58] after transforming convolution into generalized matrix-vector multiplication (GEMV) or generalized matrix-matrix multiplication (GEMM). Additionally, the exploration of sparsity in DNNs can further enhance performance by reducing energy demands and inference times [27, 31, 57].

Although there are numerous existing hardware solutions for accelerating the deep Convolutional Neural Networks, those inference accelerators either provide an unaffordable latency, occupy a big area, or are not power efficient, thus limiting

their applications in edge devices such as autonomous cars and mobile phones, where Deep Learning inference is required to be processed on the fly with tight latency, low energy consumption, small chip area and extremely high detection accuracy [59]. Moreover, most of them focus on the convolutional layers in CNNs or the matrix multiplications in Transformers while ignoring the special functions and not having system-level solutions. Higher throughput and better accuracy required by modern DNN applications need computer architects to provide comprehensive end-to-end system-level solutions that not only accelerate the typical convolutional layers and matrix multiplications but also the special functions while maintaining a high throughput and good inference accuracy.

Alongside these challenges, even an optimally fine-tuned DNN accelerator is facing a critical memory problem. Simply increasing the memory capacity under the conventional 2D semiconductor process is not a viable solution due to the yield and power constraints. Using Through-Silicon Vias (TSVs) and wafer bonding for 3D integration is one of the most promising solutions to address the vital memory problem [60]. However, as TSVs have 1μm TSV diameter with a pitch of microns, only limited TSVs can be integrated into a single chip hence limiting the improvements in overall system-level energy and execution time and compute-to-memory connectivity [61]. Denser 3D interconnects such as monolithic 3D integration enable fine-grained and dense connectivity across multiple memory layers and logic layers and alleviate the memory wall, IO wall, and logic wall problem [28].

1.3 Research Objective

The aim is to perform a hardware-software co-design and optimization for next-generation learning machines to narrow down the gap between the software and hardware implementations to ensure the real applications of the scaling up DNNs in an energy-efficient and area-efficient manner. The objectives of this thesis are as follows:

- To design and implement special function accelerators to improve the efficiency of DNN hardware implementations.
- To design and implement end-to-end workload accelerators to improve the efficiency of DNN hardware implementations.

- To integrate the introduced accelerators into a real hardware platform.

1.4 Thesis Contributions

The contribution of this thesis is a thorough study of the hardware-software co-design and optimization for next-generation learning machines. The study includes the co-design of the special function hardware, the end-to-end workload hardware, and their full system integration.

- **Special Function Hardware:** We present the special function hardware in two aspects: (1) We introduce Tunstall coding to compress the quantized weights with two hardware decoding schemes to reduce memory usage while maintaining or even improving the inference throughput. (2) We introduce ShapoolNMS — a scalable and parallelizable hardware accelerator for PSRR-MaxpoolNMS [62] — to speed up the NMS process in both one-stage detectors and two-stage detectors with low power.
- **End-to-end Workload Hardware:** We introduce two end-to-end workload hardware accelerators for CNNs and Transformers, respectively: (1) We introduce a scalable CNN accelerator system that supports large input features. In the showcase of ResNet [7], it supports up to full-HD resolution images, ensuring the performance of 68 FPS targeting 28 nm. (2) We introduce ViTA – a scalable architecture with a highly efficient dataflow tailored for the Vision Transformer (ViT) [13] – to accelerate the entire ViT workloads with high area efficiency and power efficiency.
- **Full System Integration:** We introduce the full system integration of proposed innovations in a real hardware platform, *i.e.*, the PULPissimo SoC platform [63], and introduce the register map and the finite state machine (FSM) for the integrated accelerators.

1.4.1 Special Function Accelerators

Chapter 3 adopts the Variable-to-Fixed entropy coding method—Tunstall coding—to address the inefficient decoding problem in deep network compression and present two architectures for decoding.

- It analyzes the efficacy of proposed decoding schemes against different quantization and F2V techniques.
- The decoders are implemented on FPGA as stand-alone components and are integrated into an open-source system-on-chip platform to assess their overheads.

Chapter 4 presents ShapoolNMS, a scalable and parallelizable hardware accelerator, to speed up the NMS process in both one-stage detectors and two-stage detectors with low power.

- It presents a detailed architecture of ShapoolNMS, highlighting the detailed flow of each module.
- It conducts a scalability analysis of ShapoolNMS *w.r.t.* number of bounding boxes.
- It presents a comprehensive design space exploration including power, performance, and area with different ShapoolNMS configurations.
- A performance analysis of ShapoolNMS versus state-of-the-art software and hardware accelerations of NMS is presented.

1.4.2 End-to-end Workload Accelerators

Chapter 5 presents CNN-DLA, an efficient and scalable CNN accelerator system that reduces the processing time of the whole ResNet system with full-HD images.

- It presents a scalable architecture designed to accelerate the entire CNN workloads with a showcase of ResNet, leveraging our memory-centric, hardware-efficient dataflow for heightened area and power efficiency.

- It demonstrates a novel cross-layer optimization technique to reduce the memory requirements and bandwidth requirements of the system.
- It conducts a comprehensive design space exploration.

Chapter 6 presents ViTA (Vision Transformer Accelerator), a scalable and efficient hardware accelerator for the Vision Transformer (ViT) model.

- It demonstrates a scalable architecture designed to accelerate the entire workload of ViT, leveraging our memory-centric, hardware-efficient dataflow for heightened area and power efficiency.
- A novel fused special function module to compute non-linear functions in ViT models, *i.e.*, GELU, Softmax, and LayerNorm, optimizing hardware resource sharing and further improving area and power metrics is also proposed in this chapter.
- A comprehensive design space exploration of the number of ViTA Kernels, and the number of VMUs in each VMU lane, which is used to trade off the area and power consumption is presented.
- It presents a performance analysis of our ViTA architecture with the synthesis results in the 28 nm FD-SOI technology node and a comparison with the state-of-the-art Transformer accelerators.

1.4.3 Full System Integration

Chapter 7 presents the guidelines for integrating proposed innovations into a real hardware platform, *i.e.*, the PULPissimo SoC platform.

- It presents the interfaces, register map, and FSM of the introduced innovations to be integrated into the PULPissimo SoC platform.
- It demonstrates the validation results of the Tunstall decoder on the PULPissimo SoC platform with a four-layer CNN model.
- It also discusses the bandwidth requirements of the introduced ShapoolNMS to be integrated into an SoC platform or with other accelerators.

1.5 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 discusses the background of Deep Learning hardware accelerators. In particular, studies on the foundations of Deep Learning, basic components of Deep Learning accelerators, design metrics of hardware accelerators, and optimizing the memory footprint of Deep Learning workloads are presented.

Chapter 3 introduces Tunstall coding to compress the quantized weights with two hardware decoding schemes. In this Chapter, we also introduce two hardware implementations for online decoding Tunstall codewords to provide suitable throughput for the accelerator.

Chapter 4 introduces a scalable and parallelizable hardware accelerator named ShapoolNMS to speed up the NMS process in both one-stage detectors and two-stage detectors with low power.

Chapter 5 introduces an efficient and scalable Chiplte-based CNN accelerator system that supports the whole ResNet workload with a full-HD resolution image as input with minimal memory footprint and data movement.

Chapter 6 introduces ViTA, a scalable and efficient hardware accelerator with optimized dataflows for Vision Transformer.

Chapter 7 presents the guidelines for integrating the introduced innovations into a real hardware platform, *i.e.*, the PULPissimo SoC platform, including the interfaces, register map, and the FSM of the integrated innovations.

Chapter 8 concludes the dissertation and highlighting the future work.

Chapter 2

Background of Deep Learning Hardware Accelerators

2.1 Deep Learning

2.1.1 Background on Deep Learning

Deep Learning (DL) maps the input features with output features using feature learning in multiple connected convolutional layers where each layer collects several neurons. Each neuron gives a weighted sum of the inputs after a nonlinear activation function. These convolutional layers are called Convolutional Neural Networks (CNNs), which is a key component of Deep Neural Networks (DNNs).

The four ideas are the foundation of DL [64]: (1) Simple Deep Neural Network layers can efficiently construct complex functions; (2) The parameters of these layers can be trained from the examples for approximating the objective functions; (3) Gradient-based methods are utilized during training to minimize the loss functions; (4) The back-propagation algorithms compute the gradient efficiently and automatically.

Fig. 2.1 illustrates popular DNN architectures, layers, and their implementations in the past decade.

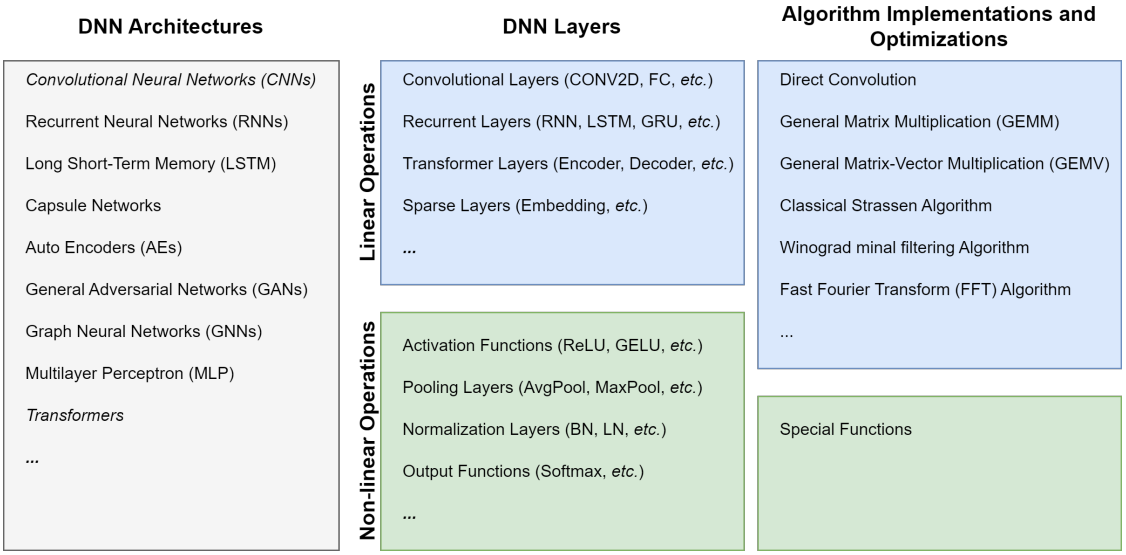


FIGURE 2.1: The illustration of DNN architectures, layers, and their implementations and optimizations.

2.1.2 Training and Inference

DNNs contain two phases, one is called training and another is inference. Training is the process for DNNs to update the weights in the neurons based on the inputs and desired outputs using Stochastic Gradient Descent (SGD). The inference is performed to make predictions, *i.e.*, compute the output features from the input features using the trained DNN model.

Deep reinforcement learning, deep supervised learning, and deep unsupervised learning are three common approaches to training the models. All training data is labeled in supervised learning to classify them or for numeric prediction. No training samples are labeled in unsupervised learning, which is similar to human and animal learning. Reinforcement learning is based on rewarding desired behaviors and punishing desired ones, where perception and interpretation of the environment are done by the reinforcement learning agent. Besides, semi-supervised learning falls between the supervised and unsupervised learning approaches.

2.1.3 Layers in Deep Neural Networks

As shown in Fig. 2.1, the layers in DNNs are the building blocks of the networks. Commonly adopted layers in DNNs are convolutional layers, recurrent layers, transformer layers, and sparse layers. There are also non-linear layers, *e.g.*, activation

TABLE 2.1: Summary of the categories and names of popular DNN layers.

Categories of Popular DNN Layers	**Layers**	**Year**
Convolutional Layers	Fully Connected layer (FC) [65]	*1958*
	Convolution (CONV2D) [66]	*1979*
	Grouped Convolution [1]	*2012*
	Depthwise Convolution (DWCONV2D) [67]	*2016*
Recurrent Layers	Recurrent Neural Network (RNN) [68]	*1986*
	Long Short-Term Memory (LSTM) [69]	*1997*
	Gated Recurrent Unit (GRU) [70]	*2014*
Transformer	Encoder [71]	*2017*
	Decoder [71]	*2017*
Sparse Layers	Embedding Layer	–
Activation Functions	Rectified Linear Unit (ReLU) [72]	1969
	Gaussian Error Linear Units (GELU) [73]	2016
	Sigmoid Activation	–
	Tanh Activation	–
	Leaky ReLU [74]	2013
	Swish [75]	2017
Pooling Layers	Maximum Pooling (MaxPool) [76]	*1990*
	Average Pooling (AvgPool)	–
	Sum Pooling (SumPool)	–
Normalization Layers	Batch Normalization (BN) [77]	2015
	Layer Normalization (LN) [78]	2016
	Instance Normalization [79]	2016
	Weight Normalization [80]	2016
	Group Normalization [81]	2018
	Weight Standardization [82]	2019
Output Functions	Softmax	–
	Heatmap [83]	2014

layers, pooling layers, normalization layers, and output functions. The summary of the categories and names of popular DNN layers is shown in Table 2.1.

The convolutional layers, which are the core building blocks of CNNs, occupy the main computation overheads. Features are extracted by the convolutional layers progressively. Filters, input features, and output features are all 3D, *i.e.*, height, width, and channel. The filters and input features own the same amount of channels, while the number of filters is the same as the number of output channels. Each point in the output features is the sum across all the channels of the filters convoluted with the corresponding input features. Fig. 2.1 and Eq. 2.1 illustrate the convolutional layers, where U is the stride and b is the bias offset.

$$output_{mef} = \sum_{crs} input_{c(U \cdot e+r)(U \cdot f+s)} \times weight_{mcrs} + b_m \tag{2.1}$$

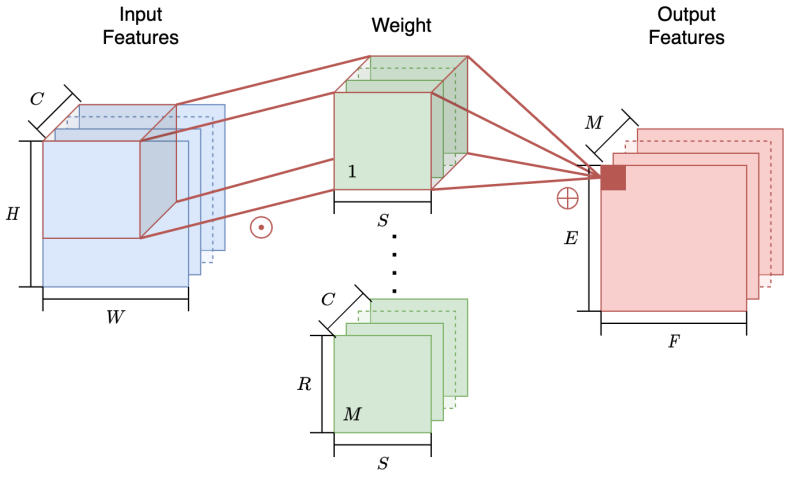


FIGURE 2.2: Illustration of the convolutional layer, where H and W are width and height of the input feature maps, R and S are width and height of the weights, E and F are width and height of the output feature maps. The number of channels of output feature maps is M while that of input feature maps and weights is C.

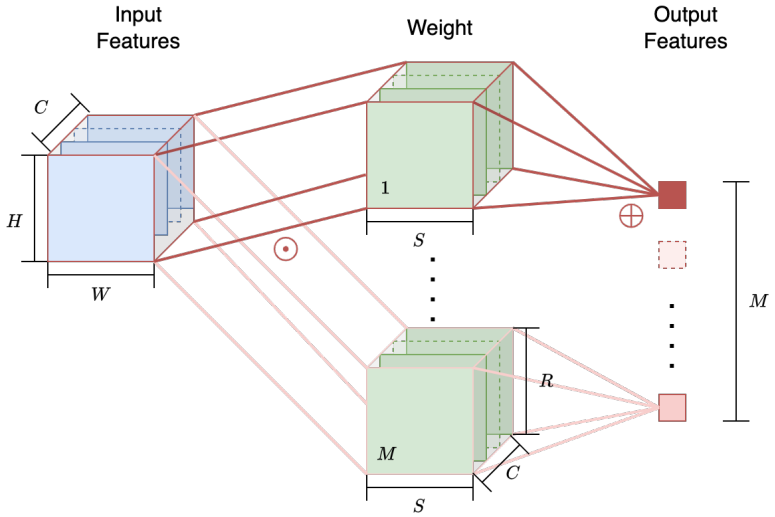


FIGURE 2.3: Illustration of the fully connected layer, where $H = R$, $W = S$, $E = F = 1$, $U = 1$.

Fully connected layers, as Fig. 2.3 and Eq. 2.2 show, perform classification where each point in the output features is the weighted sum of all the input features followed by a bias offset.

$$output_m = \sum_{chw} input_{chw} \times weight_{mchw} + b_m \tag{2.2}$$

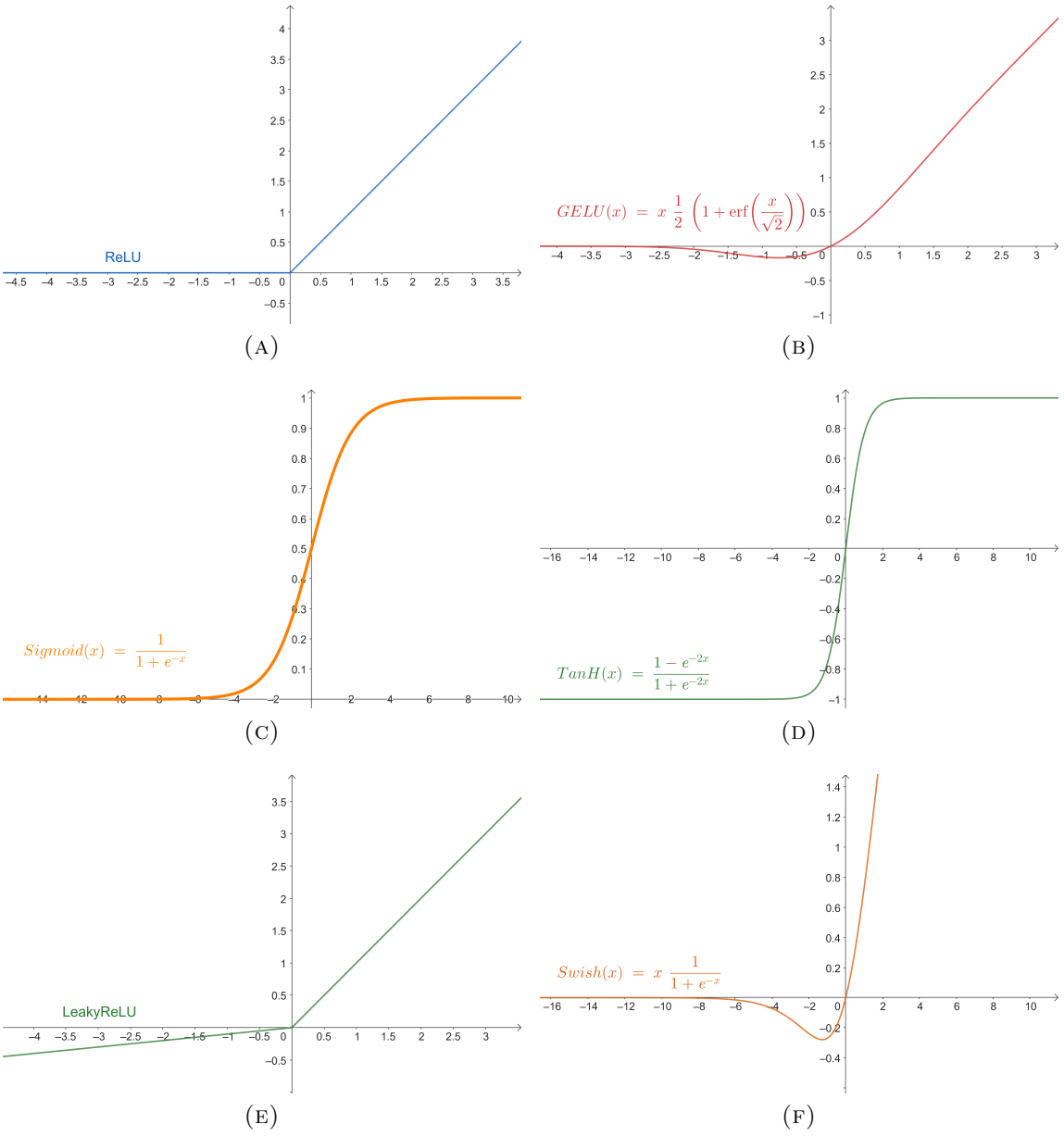


FIGURE 2.4: The illustration of the activation functions. (a) ReLU$(x) = max(0,x)$; (b) GELU$(x) = x \cdot \frac{1}{2}[1+\mathrm{erf}(\frac{x}{\sqrt{2}})]$; (c) Sigmoid$(x) = \frac{1}{1+e^{-x}}$; (d) Tanh$(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$; (e) LeakyReLU$(x) = max(a \cdot x, x)$; (f) Swish$(x) = x \cdot sigmoid(ax)$.

After each convolutional layer or fully connected layer, an element-wise non-linear activation function such as Sigmoid function, Tangent Hyperbolic (Tanh), Rectified Linear Unit (ReLU) [72] and Swish [75], is applied to increase the approximation power of the networks. Otherwise, all the linear fully connected layers and convolutional layers can be represented in one layer. Besides, partial derivatives of the error delta of the corresponding weights are easier to compute by using these nonlinear functions.

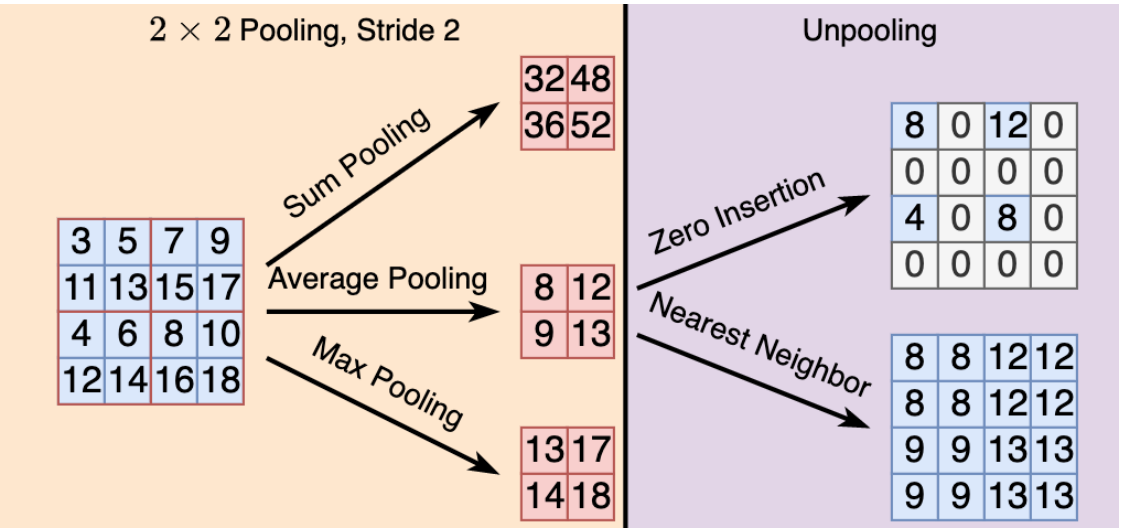


FIGURE 2.5: Various types of pooling and unpooling.

Pooling layers are inserted between convolutional layers periodically to decrease the number of parameters and computation overheads while also mitigating the overfitting problem. Average pooling, max pooling, and sum pooling are three typical pooling operations that are applied channel-wise. Vice versa, unpooling increases the number of parameters.

Normalization layers restrict the input distribution to reduce the Internal Covariate Shift problem during training hence significantly accelerating the training with higher accuracy. Batch normalization [77], weight normalization [80], layer normalization [78], instance normalization [79], group normalization [81] and weight standardization [82] are efficient normalization methods in DNNs.

Batch normalization [77] (Fig. 2.6a) computes the mean and variance statistics across the batch dimension and applies the same normalization to all the features in a layer. It can reduce the internal covariate shift problem and allow for faster and more stable training. However, it may not work well for small batch sizes or recurrent models.

Layer normalization [78] (Fig. 2.6b) computes the mean and variance statistics across the feature dimension and applies the same normalization to each example in a batch. It can work well for recurrent models and does not depend on the batch size. However, it may not be able to capture the heterogeneity among different features in a layer.

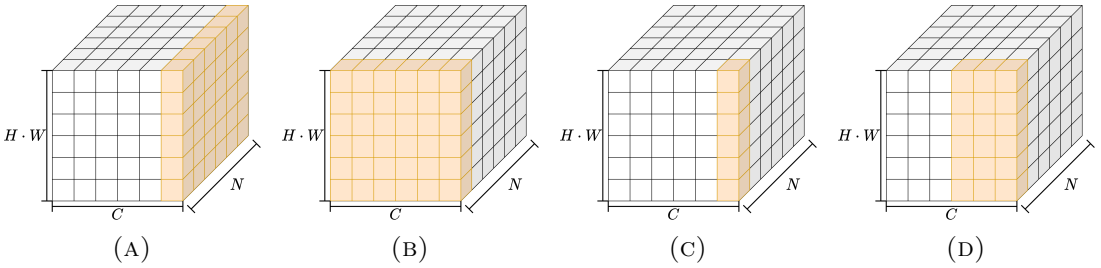


FIGURE 2.6: The illustration of the normalization layers, where N is the batch size. (a) Batch Normalization; (b) Layer Normalization; (c) Instance Normalization; (d) Group Normalization.

Instance normalization [79] (Fig. 2.6c) computes the mean and variance statistics for each feature map separately and applies the same normalization to each spatial location in a feature map. It can help preserve the contrast and style information in image generation tasks. However, it may not be suitable for classification tasks or non-image data.

Group normalization [81] (Fig. 2.6d) divides feature maps by channels into groups and computes the mean and variance statistics for each group separately. It can achieve a balance between batch normalization and instance normalization, and work well for small batch sizes and non-image data. However, it may introduce an additional hyperparameter for choosing the number of groups.

2.1.4 Deep Neural Networks Architectures

Consisting of diverse layers and nodes, several DNN architectures have been developed for different domains or use cases. As illustrated in Fig. 2.1, current DNNs are grouped into these categories: Convolution Neural Networks (CNNs) [66], Capsule Networks [84], Recurrent Neural Networks (RNNs) [68] and Long Short-Term Memory (LSTM) [69], Transformers [71], General Adversarial Networks (GANs) [85], Auto Encoders (AEs) [86], and Graph Neural Networks (GNNs) [87], *etc.*

2.1.4.1 Convolutional Neural Networks

CNNs are one of the most popular DNNs for image classification, object detection, and image segmentation. Its basic building blocks and architectures are proposed in 1979 [66], with the convolutional layer and the downsampling layer.

Benefitting from the powerful GPU computing power, the ReLU [72] activation function and the dropout technique [88], Krizhevsky *et al.* [1] proposed the first pure deep CNN architecture — AlexNet — in 2012, which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It consists of convolutional layers, fully connected layers, and max-pooling layers as the basic building blocks. Grouped convolution is also proposed in AlexNet to reduce the computation overheads and the number of parameters to fit the AlexNet into two GPUs.

The success of AlexNet inspired the development of CNNs in the following years, *e.g.*, VGGNet [9], GoogLeNet [89], ResNet [7], DenseNet [90], *etc.*, making CNNs the most popular DNNs in the ML community in the past decade.

After that, VGGNet [9] won the ILSVRC localization task in 2014 with a deeper network and smaller 3×3 filters. GoogLeNet [89] won the ILSVRC classification task in 2014 with the inception module that consists of multiple filters with different sizes.

He *et al.* introduced the ResNet architecture in 2016 [7] to address the degradation problem [91], which is the difficulty of training very deep neural networks. ResNet, the winner of the ILSVRC in 2015, is the first CNN architecture that outperforms humans with a Top5 error rate of less than 5% in the ImageNet Challenge, which benefits from the residual learning that allows the network to be deeper.

This architecture is notable for its use of residual learning, which enables the training of very deep models without the accompanying increase in errors. The ResNet architecture is composed of four blocks. The first block is a simple convolutional layer with a stride of 2 that reduces the size of the input activations by half. The second to fourth blocks consist of several bottleneck units that have three convolutional layers and one shortcut layer each. The shortcut layer adds the input activations to the output activations of the last convolutional layer in the unit, which helps to preserve the information and alleviate the vanishing gradient problem. The last block is a global average pooling layer and thereafter a fully connected layer to generate the final output.

The ResNet-152 model comprises two main types of bottleneck units, differentiated by their handling of spatial dimensions, as illustrated in Fig. 2.7. **Downsampling Bottleneck Unit**, shown in Fig. 2.7a, is used in the first unit of each block. This unit reduces the spatial dimensions of its input by a factor of 2. Its shortcut

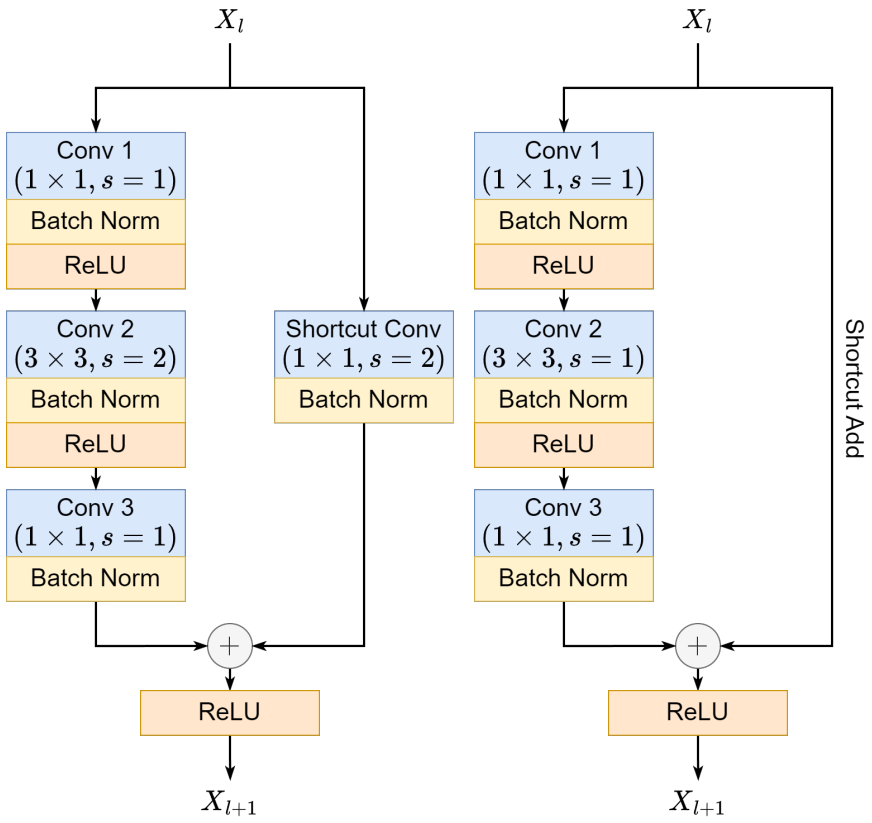


FIGURE 2.7: Two types of bottleneck units in ResNet. Illustrations are ResNet with 50, 101, and 152 layers. (a) Downsampling Bottleneck Unit; (b) Regular Bottleneck Unit.

connection uses a 1×1 convolution with a stride of 2 to align spatial dimensions and channel count before summation. **Regular Bottleneck Unit**, shown in Fig. 2.7b, is used in the other units of each block. This unit preserves the spatial dimensions of its input.

Additionally, all convolutions in the architecture are paired with Batch Normalization and ReLU activations, except for the final convolution, which is followed only by Batch Normalization.

2.1.4.2 Transformer

The Transformer network, introduced by Vaswani *et al.* [71], serves as a pivotal reference for both related works and subsequent models. It encompasses attention mechanisms, a fundamental aspect that was initially presented in the same

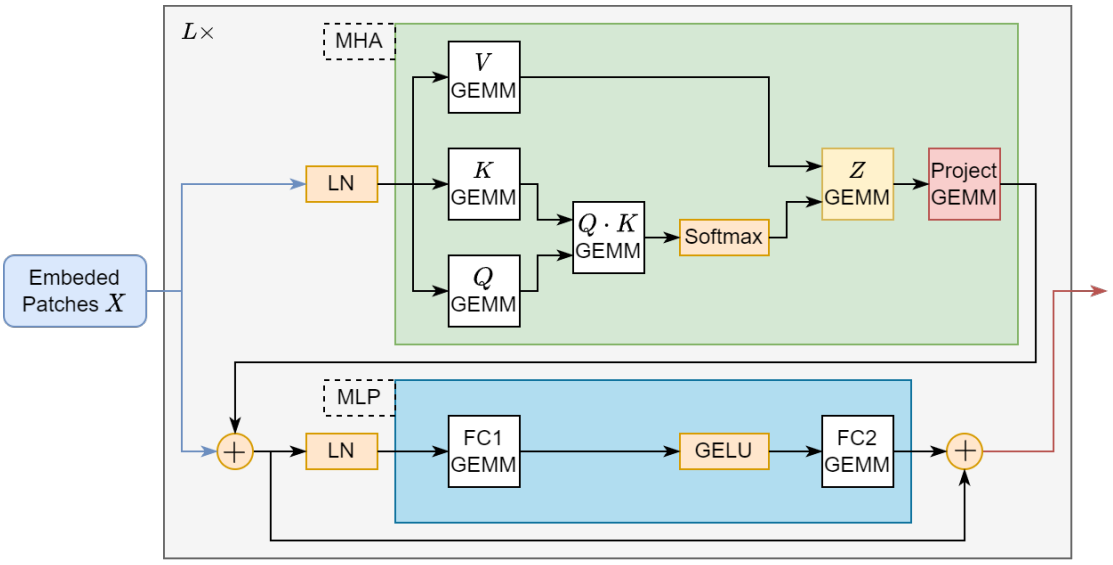


FIGURE 2.8: The Transformer encoder, adopted from [92]. "MHA" is the Multi-Head Attention mechanism, "MLP" is the Multilayer Perceptron, and "LN" is the Layer Normalization.

work. The Transformer architecture comprises two core structures: the encoder and the decoder. These components consist of two sub-layers each, namely Multi-Head Self Attention (MHA, or MSA) mechanisms and position-wise fully connected Feed-Forward Network (FFN) layers, *a.k.a.*, Multilayer Perceptron (MLP) hidden layers. The MHA mechanism allows the network to learn the dependencies and relationships between different elements in the input or output sequence, while the FFN layer applies a non-linear transformation to each element independently. Layer Normalization (LN) and residual connections are universally applied across all sub-layers to facilitate the learning process. Fig. 2.8 illustrates the Transformer encoder.

The MHA mechanism is based on the concept of Scaled Dot-Product Attention, which computes the weighted sum of a set of values (V) according to the similarity between a query (Q) and a set of keys (K). The similarity is measured by the dot product of Q and K, scaled by the square root of the dimension of Q and K (d_k). The weights are obtained by applying a Softmax function to the scaled dot products.

The MHA layer is described by the formula in [71]:

$$\mathrm{MHA}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, \cdots \mathrm{head}_h) \cdot W^O \tag{2.3}$$

where head_i is the output of Scaled Dot-Product Attention, which completes the following operations:

$$[Q, K, V] = X \cdot [W_q, W_k, W_v] \tag{2.4}$$

$$A = \text{Softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) \tag{2.5}$$

$$Z = A \cdot V \tag{2.6}$$

where d_k represents the dimension of both queries and keys.

The Vision Transformer (ViT) [13] is a model proposed for vision tasks that build upon the core concepts of BERT [3], using raw image patches as input. Instead of processing an entire image as a single entity, ViT divides it into a sequence of smaller patches, each consisting of 16×16 pixels. These patches are then treated as tokens, similar to how BERT [3] handles natural language. ViT has achieved remarkable results on various vision tasks, inspiring other transformer-based models, *e.g.*, DeiT [93], PVT [94], TNT [95], and Swin [96] to follow its footsteps. The core operation of these models is the MSA, which is applied to the sequence of image patches, and thereafter fully connected layers that perform non-linear transformations. The main differences among these models lie in how they apply the attention blocks to the input patches for each of their variants.

2.1.5 Applications of Deep Neural Networks

Deep Learning is employed in the expanded scope of signal processing [97], where the term *signal* is extended from classic types namely audio, image, and video to higher-level information representations such as text, language, and documents. The term *processing* is endowed with not only coding, recognition, classification, enhancement, and basic analysis to the tasks, *e.g.*, interpretation, understanding, mining, and retrieval that is more humanistic.

The speech and audio related areas such as speech recognition [6], phone recognition [98], language identification [99, 100], music signal processing [101], music

information retrieval [102, 103] and music generation [104] are highly impacted by DNNs.

DNNs are widely applied in tasks related to images and videos, *e.g.*, image segmentation [96, 105–107], image classification [93, 96, 108–110], image generation [4, 111], image colorization [112], video segmentation [113], video classification [114], object detection [96, 115–117], action recognition [118, 119].

The development of language modeling which is a function that retrievals the probability distribution over sequences of words in a natural language, is accelerated by DNNs. Speech recognition [6], sentiment analysis [3, 120, 121], text summarization [5, 122–124], text information retrieval [125], natural language processing [126] and machine translation [71, 127–129] have been attracting more attention.

Document retrieval [130], generation-based question answering [131], question answering from knowledge graph [132], question answering from relational database [133], retrieval-based question answering [134–137], multi-turn dialogue [138] and image retrieval [139] are applications of DNNs in information retrieval.

DNNs also play an important role in healthcare. For instance, promising results are demonstrated in compound diagnostics spanning dermatology [140], radiology [141], ophthalmology [142], and pathology [143] employed DNNs image classification and object detection.

Some modern transformer models, such as GPT-4 [10], can even perform both NLP and CV tasks in the multitask setting.

2.2 Basic Components of DL Accelerators

As shown in Fig. 2.9, a generalized Deep Learning Accelerator (DLA) consists of an arithmetic unit, a memory hierarchy, and a dataflow controller.

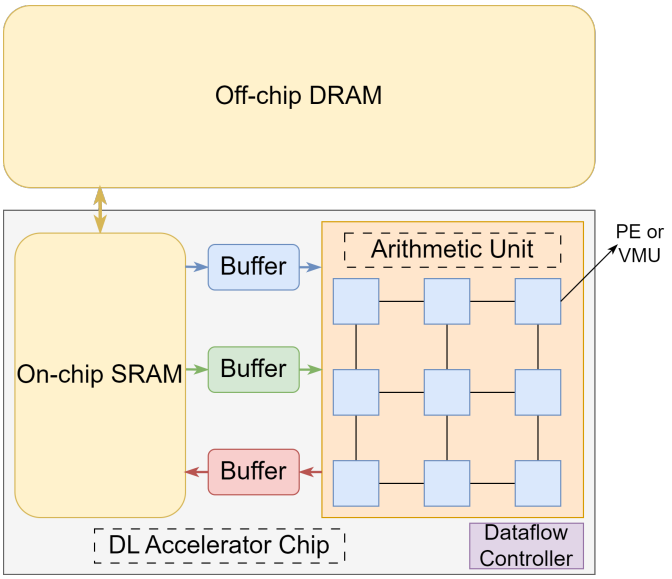


FIGURE 2.9: The illustration of a generalized Deep Learning Accelerator (DLA). The DLA consists of an arithmetic unit, a memory hierarchy, and a dataflow controller. The arithmetic unit performs the arithmetic operations, *e.g.*, convolution, GEMM, and special functions, and the memory hierarchy stores the data. The dataflow controller schedules the dataflow of the operations to reduce the memory access and communication overheads.

2.2.1 Arithmetic Units

Direct convolution operations, one of the most time-consuming operations in CNNs, are mainly performed by the processing elements (PEs) in deep learning accelerators. Fig. 2.10 illustrates a simple PE that performs the multiply-accumulate (MAC) operation. The MAC operation is the core operation in the convolutional layers, which is the weighted sum of the input features and the weights. As shown in Fig. 2.10, the PE consists of a multiplier, an adder, and a register to store the partial sum (PSum). The multiplier multiplies the input feature and the weight, and the adder adds the PSum and the multiplied result. The PSum is stored in the register and is used as the input of the adder in the next cycle or the output of the PE.

GEMM plays an important role in recent DNNs, especially in Transformers. Different from MAC operations, GEMM operations are done by the matrix multiplication unit (MMU) or the more fine-grained vector multiplication unit (VMU). Fig. 2.11 illustrates a simple MMU that consists of four VMUs to perform a 2×2 matrix multiplication operation.

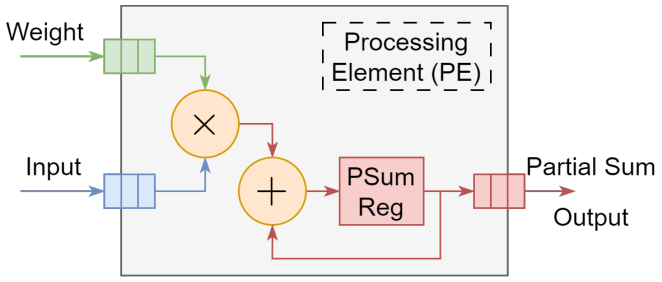


FIGURE 2.10: The illustration of a generalized processing element (PE) in deep learning accelerators. The PE performs the multiply-accumulate (MAC) operation. The inputs of the PE are the input feature and the weight, and the output of the PE is the partial sum (PSum) or the output feature.

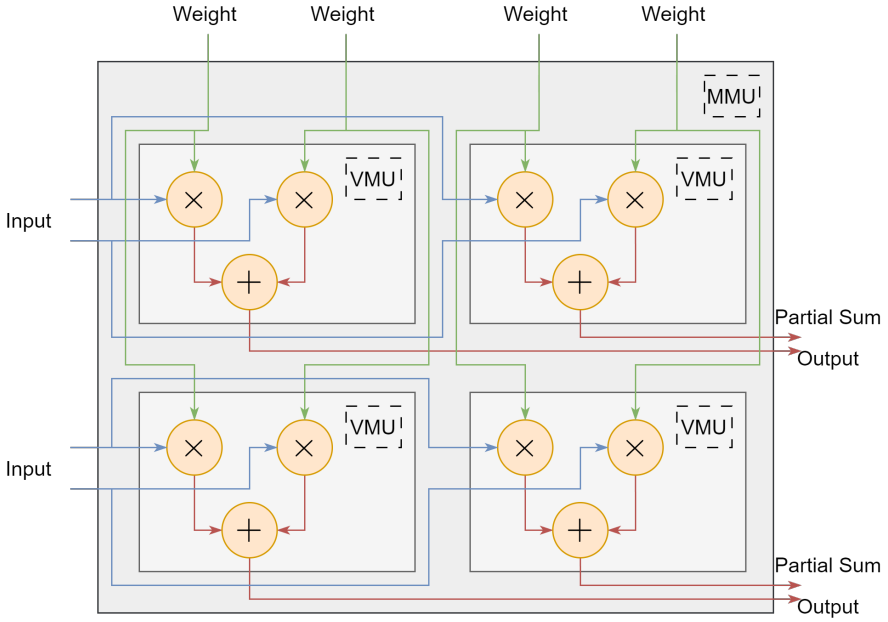


FIGURE 2.11: The illustration of a generalized matrix multiplication unit (MMU) that consists of four vector multiplication units (VMUs). Each VMU performs a 2-element vector multiplication operation. The MMU performs a matrix multiplication operation where the shape of the two input matrices and the output matrix are 2×2. The output matrix is the partial sum matrix or the output feature matrix.

2.2.2 Memory Hierarchy

There are several types of memory in deep learning accelerators, including registers, on-chip Static Random-Access Memory (SRAM), off-chip Dynamic Random-Access Memory (DRAM), *etc*. It takes 10× and 80× to 400× more energy to access on-chip SRAM and off-chip DRAM than registers, respectively [144]. It is critical to design a memory hierarchy that reduces the memory access latency and

increases the memory bandwidth while minimizing the area overheads and power consumption.

Registers or register files are the fastest memory with very low latency, high bandwidth, and low power consumption. However, the capacity of registers is usually very limited, as they require a lot of transistors to implement. Registers are usually used to store a small amount of data with a suitable dataflow to improve data reuse and reduce memory access latency.

On-chip SRAM is the most common memory in deep learning accelerators. One bit of SRAM is usually implemented by six transistors. SRAM has a balance between capacity, density, latency, bandwidth, and power consumption. The capacity of on-chip SRAM is usually larger than that of registers, but smaller than that of off-chip DRAM. On-chip SRAM is usually used to store the weights and the input features of the convolutional layers.

Off-chip DRAM is the most common memory in general-purpose processors. One bit of DRAM is usually implemented by one transistor and one capacitor. It has the largest capacity, density, and the highest energy consumption.

2.2.3 Dataflow Controller

As summarized in Table 2.2, the energy cost of memory access and communication is much higher than that of computation. Dataflow is the way of organizing the data movement and computation on a deep learning accelerator, which is a specialized hardware device that can perform DNN computations faster and more energy-efficiently than general-purpose processors. Dataflow can affect the performance, energy efficiency, and hardware cost of DNN accelerators, depending on the characteristics of the hardware architecture and the DNN model. Therefore, finding the optimal dataflow for a given DNN model and hardware configuration is a challenging problem for DNN accelerator design.

There are several possible dataflows for both direct convolution and GEMM operations. The dataflow controller is responsible for scheduling the dataflow of the operations to reduce the memory access and communication overheads.

TABLE 2.2: Energy cost of arithmetic operation, memory access, and communication for 20 nm from Dally *et al.* [145].

Operation	Detail	Cost	Overheads
Arithmetic	INT8 Add	10 fJ	1.0×
	FP64 Mult	5 pJ	500.0×
Local Memory	8 KB	50 fJ/bit	5.0×
	100 MB	0.7 pJ/bit	70.0×
Global Memory	LPDDR4	4 pJ/bit	400.0×
	SDDR4	20 pJ/bit	2000.0×
Local Communication	on-chip	100 fJ/bit-mm	10.0×
Global Communication	SerDes	10 pJ/bit	1000.0×

In [146], Chen *et al.* proposed four different dataflows for direct convolution operations, *i.e.*, input-stationary (IS), row-stationary (RS), output-stationary (OS), and weight-stationary (WS).

A GEMM operation is formulated as:

$$C_{M\times N} = A_{M\times K} \times B_{K\times N} \tag{2.7}$$

where A is the input matrix, B is the weight matrix, and C is the output matrix, and M, N, and K are the number of rows and columns of the matrix C and the number of columns of the matrix A respectively.

There are three possible dataflows for GEMM operations, *i.e.*, inner product, outer product, and systolic array. The inner product dataflow is the most common dataflow for GEMM operations, which can be classified into six categories, *i.e.*, MNK, NMK, KMN, MKN, KNM, and NKM. Fig 2.12 illustrates the MNK output stationary (OS) GEMM dataflow as an example. In the above six categories, the sequence of M, N, and K represent the order of the for-loop in the GEMM operation, from outer to inner. The outer product dataflow is usually used in the GEMM operations with a large matrix multiplication but is difficult to implement on a large scale. The systolic array dataflow is used in the GEMM operations with a large matrix multiplication, which is easy to implement on a large scale and is adopted in the Google TPU [29].

Similarly, the six GEMM inner product dataflows can be classified into the above four stationaries with the addition of column stationary (CS):

- Weight Stationary (WS): KNM GEMM dataflow.

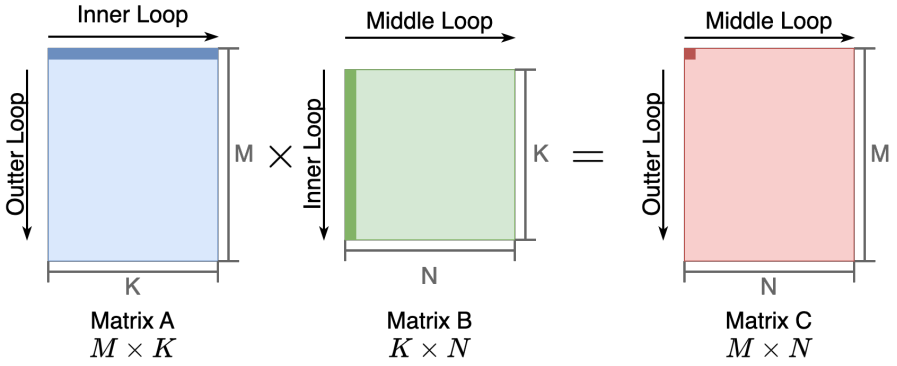


FIGURE 2.12: The illustration of the MNK output stationary (OS) GEMM dataflow. The sequence of M, N, and K represent the order of the for-loop in the GEMM operation, from outer to inner.

TABLE 2.3: Summary of dataflow taxonomy of deep learning accelerators.

Dataflow	Work
WS	NeuFlow [147], Prime [42], Origami [148], ISAAC [43], TPU [29], NVDLA [149]
OS	DianNao [30], Dadiannao [41], ShiDianNao [150], Envision [151], SqueezeFlow [152]
IS	SCNN [153]
RS	Eyeriss [27, 146], Wang *et al.* [154]

- Output Stationary (OS): MNK and NMK GEMM dataflow.
- Input Stationary (IS): KMN GEMM dataflow.
- Row Stationary (RS): NKM GEMM dataflow.
- Column Stationary (CS): NKM GEMM dataflow.

2.3 Design Metrics of Hardware Accelerators

2.3.1 Performance

2.3.1.1 Accuracy

Accuracy is a critical metric for evaluating the task performance of DNNs. Very high accuracy enables DNNs to be applied in a wide range of applications. The accuracy of DNNs is usually evaluated by the Top-1 and Top-5 error rates. The top k error rate is the percentage of the input samples that are not classified correctly in the top k predictions. However, other metrics, such as precision, recall, F1

score, *etc.*, can be used to evaluate the task performance of DNNs depending on the specific task.

Factors such as DNNs architecture, quantization, the evaluation dataset, the training configurations, and tasks *etc.* affect the inference accuracy.

Generally, the accuracy of DNNs is improved by adopting deeper and wider networks, which increases the computation overheads and memory requirements. However, this presents a challenge for hardware accelerators, which have limited computation resources and memory bandwidth.

Techniques such as pruning and quantization can reduce the computation overheads and memory requirements of DNNs, but they also reduce the accuracy of DNNs.

2.3.1.2 TOPS, Latency, and Throughput

TOPS, latency, and throughput are the computation performance metrics of DNN hardware accelerators represented by a measure of operations per second, the time to complete the whole workload, and the number of input samples processed per second, respectively.

The number of operations per second measures the upper bound of the computation performance of a chip or system, *i.e.*, the number of multiply and accumulate operations per second. It is usually reported in the form of Giga operations per second (GOPS) or Tera operations per second (TOPS).

Latency is the time from the input of the workload to the output of the result from the accelerator. Latency is reported in the form of clock cycles, microseconds, or seconds. In real-time applications, the latency of the accelerator is critical, *e.g.*, autonomous driving, where the latency of the accelerator directly affects the safety of the vehicle, and object detection, where the latency of the accelerator directly affects the user experience.

Throughput is the number of input samples processed in a given time slot. Throughput is reported in the form of samples per second (SPS) or frames per second (FPS). High throughput is important for many applications, *e.g.*, video surveillance, where the throughput of the accelerator directly affects the performance of the system.

TOPS defines the peak performance of the accelerator, while latency and throughput define the actual performance of the accelerator, which is affected by the dataflow, memory hierarchy, and flexibility of the accelerator.

2.3.2 Hardware Cost

Power, area, power efficiency, and area efficiency are the hardware cost metrics of DNN hardware accelerators.

Power and area are two basic metrics of hardware. Power is reported in the form of Watts (W) while area is reported in the form of square millimeters (mm^2). They are usually dominated by the memory and interconnects in the deep learning accelerators. Power and area are related to the technology node, *i.e.*, the smaller the technology node, the lower the power and area with the same design.

Power efficiency and area efficiency are metrics that measure the computation capability, *i.e.*, the MAC operations, of a system relative to its power consumption and chip area, respectively. Power efficiency and area efficiency are reported in the form of GOPS/W or TOPS/W and GOPS/mm^2 or TOPS/mm^2, respectively.

The two efficiency metrics are highly related to the architecture and dataflow of the accelerator. The higher the power efficiency and area efficiency, the better the accelerator.

A good accelerator should convert the high-complexity arithmetic operations into low-complexity ones. For example, in Softmax and Layer Normalization operations, the division operations are converted into one reciprocal operation and multiplication operations, respectively.

For the dataflow, data reuse is critical to improve the power efficiency and area efficiency.

2.3.3 Memory Bandwidth

Memory bandwidth is the amount of data that can be transferred from the memory to the processing units per second. It is reported in the form of Giga bytes per second (GB/s).

Peak memory bandwidth is the maximum theoretical rate at which data can be transferred between the memory and the processing units in the accelerator. It's fixed by the memory hierarchy, memory controller, bus width, and clock frequency, *etc*.

The **peak memory bandwidth** of the accelerator should be large enough to accommodate the expected **used bandwidth** of typical workloads to avoid processing unit stalls and low throughput. The **used bandwidth** reflects the actual data transfer rate achieved by the application or workload, which is affected by the application or workload, such as dataflow, and memory access pattern.

However, an inefficient dataflow may lead to a large memory bandwidth requirement, which increases the power and area overheads of the accelerator.

2.3.4 Scalability

Scalability is the ability of the accelerator to scale up the computation resources, *i.e.*, the number of processing units, memory bandwidth, and memory capacity for better performance, *i.e.*, higher TOPS and throughput, and lower latency, or higher efficiency, *i.e.*, higher power efficiency and area efficiency.

The performance of the accelerator is expected to scale linearly with the computation resources in the ideal case. However, the performance of the accelerator may not scale as expected in the real case due to memory access overheads, communication overheads, dataflow inefficiency, and inefficient computation mapping.

Scalability is important for the accelerators that are used in different platforms, *e.g.*, mobile devices, edge devices, and data centers.

2.3.5 Flexibility

Flexibility is the ability of the accelerator to support different DNNs, *i.e.*, different DNN architectures, input sizes, and data types. The flexibility of the accelerator is usually achieved by the programmability of the accelerator with a trade-off between flexibility and hardware cost.

As there are many DNN architectures and workloads, various computation kernels are required to support different DNNs, which increases the complexity of the hardware accelerator, therefore increasing the design effort and chip area. Hardware-software co-design is a promising solution to improve the flexibility of the accelerator with a low hardware cost.

2.3.6 Software Ecosystem

The software ecosystem of the accelerator is critical to the adoption of the accelerator. The accelerator should support the popular deep learning frameworks and end-to-end application workloads.

2.3.7 Special Features

Except for the above metrics, some special features are a plus for the accelerator. For example, numerical representations, mixed precision, and sparsity are three promising techniques to improve the performance of the accelerator.

2.3.7.1 Numerical Representations and Mixed Precision

Mixed precision is a technique that uses different data types for different layers or operations in the DNNs. For instance, Fig. 2.13 illustrates the different floating-point data formats, *i.e.*, FP32, FP16, BF16, and FP8 and fixed-point data formats, *i.e.*, signed and unsigned INT16, and INT8. Different from floating-point data formats, fixed-point data formats use a fixed number of bits only to represent the mantissa while a sequence of fixed-point values shares the same exponent.

Supporting mixed precision is a widely adopted solution to balance the accuracy and performance of the accelerator. The bit width of weights in different layers or channels can be different to balance model precision and size. Table 2.4 summarizes the popular ASIC accelerators that support multiple precision data formats. In [155], 8-bit and 16-bit fixed-point NN layers are supported. Zhang *et al.* [156] proposed a new flexible unit that supports five different precisions in training and inference. LNPU [157] is a highly energy-efficient DNN accelerator that supports

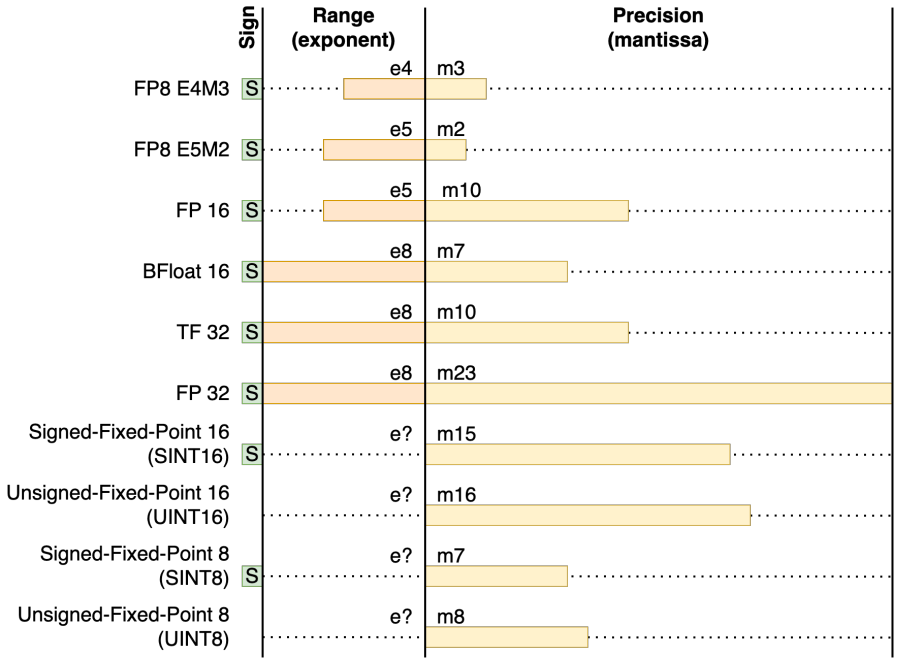


FIGURE 2.13: The illustration of different floating-point and fixed-point data formats.

training and inference, achieving up to 25.3 TFLOPS/W energy efficiency when processing a highly sparse weight layer in 8-bit float-point data format. [158] supports very low-precision data formats such as binary and ternary for aggressive inference performance. In addition, the 16-bit float-point data format is supported for high accuracy in training, hence achieving 24 TOPS inference performance in a 9mm^2 chip. Zhou *et al.* [159] proposed a CNN accelerator that supports mixed precision computations both within-layer and layer-wise. Introducing a fine-grained mixed precision unit enables within-layer mixed-precision operations, which reduces computation area by nearly 50% and dynamic power by around 12.1% in AlexNet and VGG16 compared to the baseline.

2.3.7.2 Sparse Architecture

Sparsity is a technique that reduces the number of MAC operations by skipping the MAC operations with zero input features or weights. Pruning is adopted and well-explored in DNNs [12, 14, 160, 161] to reduce the computation overheads and memory requirements. However, not all accelerators utilize the sparsity of DNNs after pruning.

TABLE 2.4: Mixed-precision hardware

Work	Data Format	Platform	Area	Tech. (nm)	Inference Performance	Freq. (GHz)	Training or Inference	Energy or Area Efficiency
[156][a]	INT4	ASIC	2943 μm^2	28	29.6 GOPS	3.7	Both	4.81 TOPS/W
	INT8				14.8 GOPS			2.09 TOPS/W
	INT16				7.4 GOPS			1.01 TOPS/W
	FP8				14.8 GFLOPS			1.10 TFLOPS/W
	FP16				7.4 GFLOPS			0.55 TFLOPS/W
LNPU[157]	FP8	ASIC	16 mm^2	65	600 GFLOPS	0.2	Both	3.48-25.3 T/W 0-90% sparsity
	FP16				300 GFLOPS			–
[158][b]	Binary	ASIC	9 mm^2	14	24 TOPS	1.5	Both	2.67 TOPS / mm^2
	Ternary				12 TOPS			1.33 TOPS / mm^2
	FP16				1.5 TFLOPS			0.17 TOPS / mm^2
[155]	INT8, INT16	ASIC	19.36 mm^2	65	368.4 GOPS	0.2	Inference	1.28 TOPS/W
[159]	INT8, INT16	FPGA	–	–	–	–	Inference	–

[a] This work proposes one computation unit. Hence, the frequency is computed by 1/delay, where the delay is reported. The inference performance is computed by $2 \times$ #parallel op $\times$ frequency, where the number of parallel operations in one unit is reported. The energy efficiency is computed by 1/energy #op, where the energy per operation is reported.
[b] The energy efficiency is not reported; area efficiency is reported instead.

To reduce the energy efficiency significantly and boost computation speed with fewer computation units, recently works [27, 31, 146, 153, 157] proposed accelerators that support sparse neural networks. As summarized in [162], there are five hardware acceleration methods for sparse neural networks:

- Storing the compressed data in off-chip memory to optimize the memory capacity requirement and improve energy efficiency;
- Storing the compressed data in on-chip memory to optimize the memory capacity requirement and improve energy efficiency;
- Skipping zero elements to improve energy efficiency;
- Reducing ineffectual computation cycles to improve performance and energy efficiency;
- Balancing the workloads of different processing elements to increase performance;

In [57], Liu *et al.* tried to explore the sparsity of the Winograd-based convolution by moving ReLU layers after Winograd transforming, *i.e.*, applying ReLU in the Winograd domain. Lee *et al.* [157] proposed an architecture called LNPU that supports sparse DNNs with mixed precision of FP8 and FP16. With the input load balancer (ILB) to alleviate the imbalanced workload problem caused by irregular sparsity, sparsity is exploited with intra- and inter-channel accumulation, improving PE utilization. Only non-zero weights are kept in the internal

TABLE 2.5: Storage overhead and decoding taxonomy for common encoding methods. Vector d stores n dimensions of a tensor that contains NNZ non-zero elements. (Table adapted from [162])

Format	Storage Overhead (bits)	Decoding Taxonomy
COO	$NNZ \times \sum_1^n \lceil \log_2 d_i \rceil$	Direct
COO-1D	$NNZ \times \lceil \log_2 \prod_1^n d_i \rceil$	Direct
RLC	$NNZ \times B$	Single-step
Bitmap	$\prod_1^n d_i$	Single-step
CSR	$NNZ \times \lceil \log_2 d_1 \rceil + (d_0 + 1) \times \lfloor \log_2 NNZ + 1 \rfloor$	Double-step
CSC	$NNZ \times \lceil \log_2 d_0 \rceil + (d_1 + 1) \times \lfloor \log_2 NNZ + 1 \rfloor$	Double-step

buffers in Cambricon-X [163], while SCNN [153] compressed both non-zero weights and activations in both DRAM and internal buffers with proposed PlanarTiled-InputStationary-CartesianProduct-sparse (PT-IS-CP-sparse) dataflow.

Eyeriss [146] implements data gating to reduce processing energy.

Sparse encoding methods can reduce memory access, increase energy efficiency, and accelerate computation time for accelerators by adapting them to sparse tensors. This is because most sparsity encoding methods only store non-zero elements. These encoding methods are Coordinate (COO), COO-1D, Run-length Coding (RLC), Bitmap, Compressed Sparse Fiber (CSF) [164], Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC) [165]. Table 2.5 summarizes the storage overhead and decoding taxonomy for standard encoding methods suitable for sparse data. [162].

The COO sparsity encoding method is adopted in [166, 167]. COO-1D is utilized in [168–170]. RLC is used in [146, 153, 157, 171]. Bitmap is used in [149, 167, 171, 172]. Mishra *et al.* [173] introduced CSR into the accelerator. Data in [27, 31, 173, 174] is compressed in CSC format. Extensor [166] utilized CSF encoding format.

More specifically, Eyeriss v2 [27] encodes the weights in CSC data format for both on- and off-chip accessing to reduce the bandwidth requirements and energy consumption. Hence 1.2× and 1.3× improvement in speed and energy consumption are obtained by introducing CSC into Eyeriss v2. To best utilize the benefit of CSC data format, the PE in Eyeriss v2 is specially designed, where there are registers to store both address vectors and data vectors in the CSC compressed data, which is shwon in Fig. 2.14.

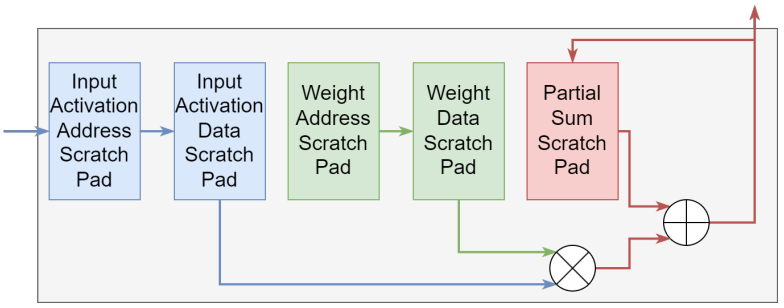


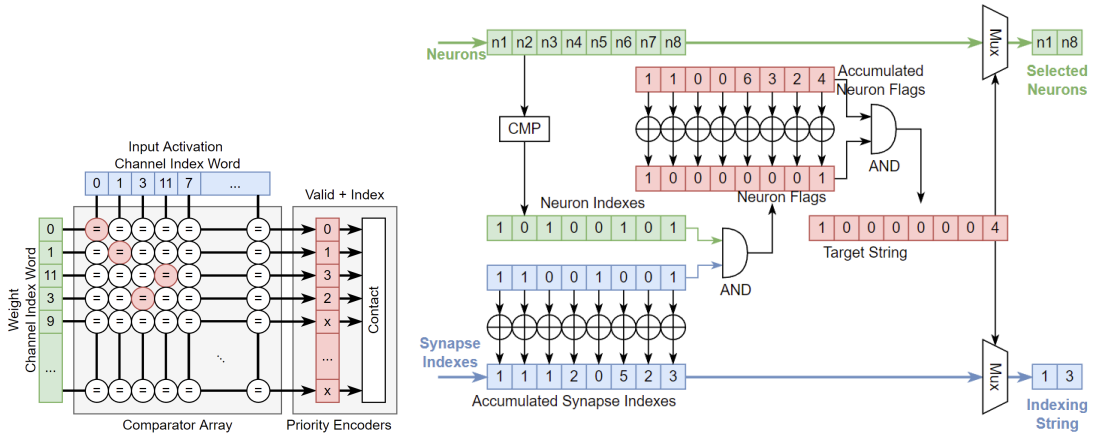
FIGURE 2.14: Simplified Eyeriss v2 PE Architecture. This PE is designed for CSC data format; the data scratchpad stores the data and count vectors while the address scratchpads of input activation and weight store the address vectors of input activation and weight respectively in the CSC compressed data. Figure adapted and simplified from [27].

TABLE 2.6: The hardware accelerators that use sparsity encoding methods.

Work	Tech. (nm)	Area	Freq. (MHz)	Energy Eff. (TOPS/W)	Zero Ratio	Encoding Format	Data Format
EIE [31]	28	40.8 mm^2	800	0.17	70% (Activations) 10% (Weights)	Zero-Skip (Activations) CSC (Weights)	INT4
Envision [151]	28	1.87 mm^2	200	1.30	5-82%	Bitmap	INT16
Sticker [167]	65	7.8 mm^2	200	62.10	both 5%	COO (Activations) CSC (Weights)	INT8
Eyeriss v2 [27]	65	2695k gates	200	0.96	68.98%	CSC	INT8
SNAP [168]	16	2.4 mm^2	260	21.55	both 90%	COO-1D	INT16
SqueezeFlow [152]	65	4.8 mm^2	900	–	–	RLC	INT16

Sticker [167] compresses the sparse weight into CSC format offline and decodes weight on-chip. In addition, there are COO encoders and decoders in Sticker to adopt the COO format for activations online.

Zhang *et al.* [168] designed an associative index matching (AIM) module in SNAP before feeding the data into a multiplier array to find the non-zero partial sum position, i.e., both the input activation and the corresponding weight are non-zero. A similar module is used in Cambricon-S [172] where the bitmap encoding method is used; hence the comparators in SNAP are replaced by AND gates, which is shown in Fig. 2.15b. Envision [151] exploited sparsity by representing the data in bitmap format, storing those flags in a GRD memory, hence reducing both memory fetches and MAC operations. A run-length encoding scheme is utilized in SCNN [153] where the index vector is encoded from the number of zeros between non-zero elements. Hence, only non-zero weights and activations are processed.



(A) AIM in SNAP (B) Neuron selector module in Cambricon-S

FIGURE 2.15: (a) The decoding modules in SNAP, where the comparator array is used to find the matched valid input activation and corresponding weight. Figure adopted and from [168]. (b) The Neuron selector module in Cambrion-S, where AND gates decode the input activation and weight. Figure adopted from [172].

2.4 Optimizing the Memory Footprint of DL Workloads

Even with the benefit of outweighing human accuracy in increasing areas, the utilization of better performance models with deeper and larger neural networks is limited as they not only consume a lot of energy but also require a huge memory capacity to store billions of parameters. Therefore, it is critical to compress the DNNs to mitigate these problems. Moreover, privacy can be preserved on edge devices if all weights are stored on-chip and inference is done on edge devices instead of being processed on the cloud with data transmission.

As shown in Fig. 2.16, pruning, quantization, and entropy coding are three directions of weight compressing for DNNs [12] to reduce the memory footprint.

2.4.1 Pruning

Neural network pruning is one favored method for reducing the number of parameters in the DNN models and computation loads at runtime [175] which consistently removes the components that have minimal or no impact on accuracy across layers, channels, filters, or connections. The compressed model after pruning is power- and

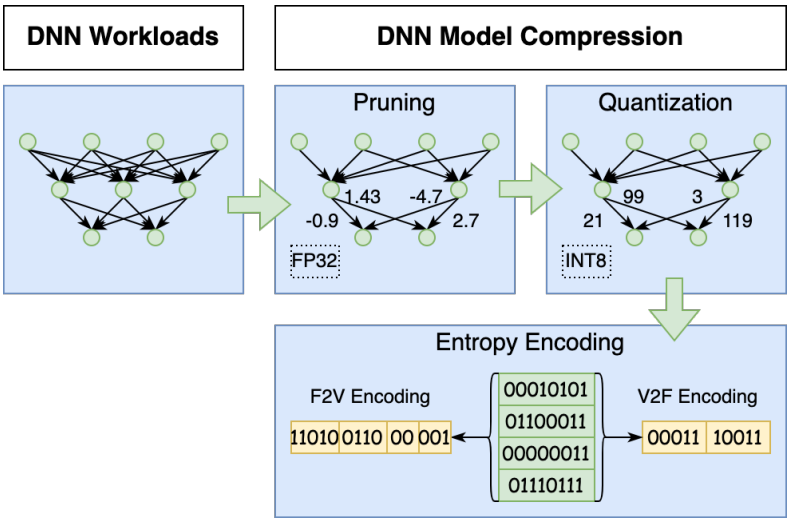


FIGURE 2.16: Pruning, quantization, and entropy coding are three directions of weight compressing for DNNs to reduce the memory footprint.

memory-efficient. Moreover, the high sparsity conducted by pruning not only gives more benefits on the hardware side but also helps the yield of other compression methods such as entropy coding to further reduce the model size.

Pruning approaches have been explored since 1980s by Janowsky [176], Mozer & Smolensky [177] and Karnin [178]. The popularity of DNNs leads to burst attention on deep neural network pruning again.

The state-of-the-art pruning methods are categorized into layer pruning [160, 179], filter pruning [15, 161, 180–182], channel pruning [14, 67, 183, 184], and connection pruning [12, 31, 153, 185–188] due to the removal of components.

Layers are eliminated to compress the DNNs in layer pruning, which is usually used in resource-constrained devices [189]. Authors in [179] proposed discriminative layer pruning based on Partial Least Squares projection. Layers are also pruned in [160].

The depth of the feature maps of the subsequent layers can be reduced by pruning a whole convolutional filter along with their connecting feature maps [182]. Filter pruning firstly evaluates the effect of the filters and ousts some filters based on fine-tuned criteria then retraining the model to minimize the accuracy drop. In [181], low-rank approximations are used to exploit the redundancy of filters in DNNs. Lebedev *et al.* [190] explored group-wise filter pruning using group LASSO. L1

normalization is used in [161] to calculate the importance of filters. The sparsification of full connection layers and separation of convolutional kernels is leveraged in [180]. In ThiNet [15], unimportant filters are recognized via statistical information. In [182], the sensitivity of filters to the auxiliary loss function is utilized for filter-level pruning.

Although multiple output channels that contain different information improve the detection accuracy, this leads to a significant increment of the FLOPS and storage requirements in DNNs training and inference. To reduce the computation loads and memory capacity requirements, unimportant channels should be removed. Several studies have explored the channel pruning problem and how to minimize the performance decrease while pruning more channels. The channel pruning problem is reformulated as a LASSO problem in [14] where the outputs of the pre-trained network are preserved. Network slimming [183] recognizes the insignificant channels via introducing L1 regularization on the batch normalization scaling factors so that they are pushed towards zeros. Then the channels that are less than the pruning threshold are removed. Depth-wise convolution is applied in MobileNets [67] where every input channel is applied to one filter instead of several filters in the normal convolution operations. In [184], a collaborative channel pruning algorithm is proposed to obtain the relationship between channels.

The number of parameters in each convolution layer, which is directly used to analyze the memory and computation requirements, is determined by the number of input and output connections. The connection level pruning is introduced to eliminate unimportant connections, *i.e.*, parameters. There are many heuristics proposed to identify the unimportant parameters. Han *et al.* [12, 31] pruned the DNN model to reduce the total number of parameters and operations. Authors in [188] avoid incorrect pruning by incorporating connection splicing into the network compression and retraining the remaining weights. The ReLU operation is utilized in SCNN [153] to estimate zero-valued weights for connection pruning. In [187], nodes are pruned via the Bayesian point of view.

2.4.2 Quantization

Quantization compresses DNNs by decreasing the number of bits per variable for weights, activations, or gradients. State-of-the-art quantization methods can compress the weights down to 4 bits from 8 bits without losing accuracy [17–19]. However, further reduction of the number of bits, for example compressing to 2-bits, can bring a noticeable accuracy drop [20].

Generally, there are two quantization types, namely, deterministic quantization and stochastic quantization [191]. Quantized values and the real values are one-to-one mapping in deterministic quantization while the quantized values are sampled from the discrete distributions of the real values in stochastic quantization.

Rounding, vector quantization, and quantization as optimization are three techniques in deterministic quantization. Rounding quantization [192, 193] simply maps the real continuous values to discrete quantized weights based on the quantization function such as the $Sign(x)$ function. Vector quantization utilizes a series of information theoretical vector quantization methods, such as structured quantization [194] and scalar quantization. It maps a clustered group of weights using the centroid of each group. Structured quantization contains residual quantization [194] and product quantization. Scalar quantization uses k-means [12, 194] or Hessian-weighted k-means clustering approaches [195]. The quantization problem is also formulated as an optimization problem in Rastegari *et al.* [196], Li *et al.* [197], Hou *et al.* [198].

Random rounding and probabilistic quantization are two stochastic quantization techniques. Quantized values are sampled from the given probabilities of the real values in random rounding [192, 197]. Probabilistic quantization quantized weights from a probabilistic perspective assumed a discrete distribution of weights [199, 200].

2.4.3 Entropy Coding

Entropy coding focuses on encoding the quantized values of weights (or activations) in a more compact representation by utilizing the peaky distribution of the quantized values, to achieve a lower number of bits per variable [20], without any accuracy loss.

Fixed-to-variable (F2V) and variable-to-fixed (V2F) are two types of entropy coding methods. A fixed number of symbols are encoded into a variable length of the codeword in F2V coding schemes. Conversely, V2F coding methods encode multiple symbols to a fixed number of bits. More details about the entropy coding methods are discussed in Section 3.2.

2.5 Chapter Summary

In this chapter, we discuss the foundations of deep learning. More specifically, training and inference, layers in DNNs, DNN architectures, and the applications of DNNs are discussed in detail. We reviewed two typical DNNs, *i.e.*, CNN and Transformer.

Furthermore, we discuss the basic components of deep learning accelerators, including arithmetic units, memory hierarchy, and dataflow controller.

Afterward, we discuss the design metrics of deep learning accelerators, including performance, hardware cost, memory bandwidth, scalability, flexibility, software ecosystem, and special features.

Finally, we discuss three typical DNN compression techniques, namely, pruning, quantization, and entropy coding for reducing the memory footprint of DNNs.

In the following four chapters, we will discuss the hardware acceleration on both specific function hardware and end-to-end workload hardware. More specifically, we will discuss the hardware-friendly DNN coding method — Tunstall coding — with its hardware decoders for accelerating the DNNs after quantization. We will also discuss one scalable and parallel NMS accelerator — ShapoolNMS — to optimize the memory requirements and better the performance of the DL hardware system on the nonlinear operation — NMS. Then we will discuss the hardware accelerators for the end-to-end workloads, *e.g.*, ResNet and ViT. This is represented by the proposed hardware accelerator for the end-to-end ResNet-152 workload and ViT workload, including the non-linear operations, *i.e.*, Softmax and Layer Normalization. Finally, we will discuss the system-level integration of the proposed hardware accelerators.

Chapter 3

Efficient Tunstall Decoder for Deep Neural Network Compression

3.1 Introduction

Power- and area-efficient Deep Neural Network (DNN) designs are key in edge applications. Compact DNNs, via compression or quantization, enable such designs by significantly reducing the memory footprint. Lossless entropy coding can further reduce the size of networks. Currently employed Fixed-to-Variable (F2V) entropy coding schemes such as Huffman coding and Arithmetic coding are inefficient to be decoded in the hardware platforms.

In this Chapter, we introduce a Variable-to-Fixed (V2F) entropy coding method—Tunstall coding—to compress the quantized weights. Tunstall coding can achieve a very high compression ratio, while the decoding mechanism is much more efficient than the F2V decoding—decoding complexity is only $O(n)$, which is much less than that of F2V coding methods. We present two hardware-accelerated decoding schemes, namely Memory-Oriented (MO) and Logic-Oriented (LO), where we develop the hardware implementation for the decoding stage in Chisel [201] and map them on FPGA. Experimental results show that our introduced schemes reduce

The work in this chapter has been published in [Proceedings of the 2021 28th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 1021-1026].

TABLE 3.1: The codebook of the example Huffman tree

Probability	Symbol	Codeword
0.625	A	0
0.250	B	10
0.125	C	11

the memory usage allocated for common DNN weights, *i.e.*, ResNet-50 [7] and MobileNet-v2 [8], by 18× and 2.3×, compared with 32-bit full precision 32-bit 4-bit quantization, respectively, while achieving up to 10× speedup than F2V methods.

3.2 Related Work

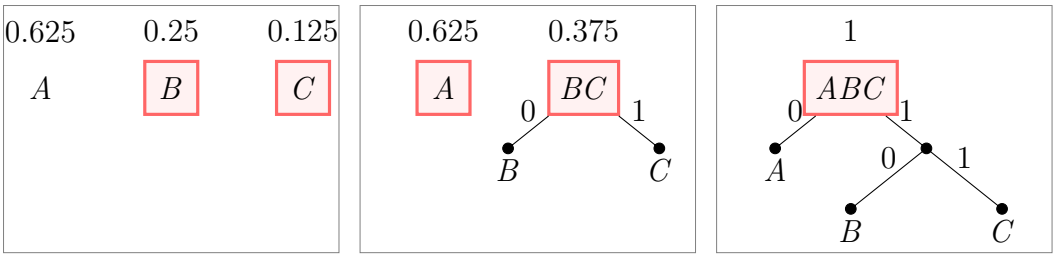
3.2.1 Fixed-to-Variable Entropy Coding Schemes

A fixed number of symbols are encoded into a variable length of the codeword in Fixed-to-Variable (F2V) coding schemes. F2V coding strategies such as Huffman coding [202] and arithmetic coding [203] are explored in DNNs compression [12, 21–23].

In the Huffman coding algorithm, all symbols are first listed in decreasing order of their probabilities. Then a Huffman tree is constructed step by step, where each step adds two symbols with the smallest probabilities to the top of the partial Huffman tree and removes these two symbols from the list. An auxiliary symbol is added to the list to represent the two original symbols that are removed.

Fig. 3.1 illustrates an example of the construction of the Huffman tree. In this example, we have three symbols A, B, and C with probabilities 0.625, 0.25, and 0.125, respectively—these probabilities indicate the likelihood of a weight having this symbol. Firstly, A, B and C are listed according to their probabilities (Fig. 3.1a). Then B and C are replaced by the combined symbol BC with a probability of 0.375 (Fig. 3.1b). Finally, A is combined with BC and is added to the top of the partial tree (Fig. 3.1c). Table 3.1 lists the codebook of this example in Huffman coding. According to this codebook, sequence "AAABABAC" is encoded as a binary string "00010010011".

Arithmetic coding assigns one code to the entire input instead of separating the inputs into individual symbols and replacing each with an integral number of bits



(A) (B) (C)

FIGURE 3.1: The Huffman tree

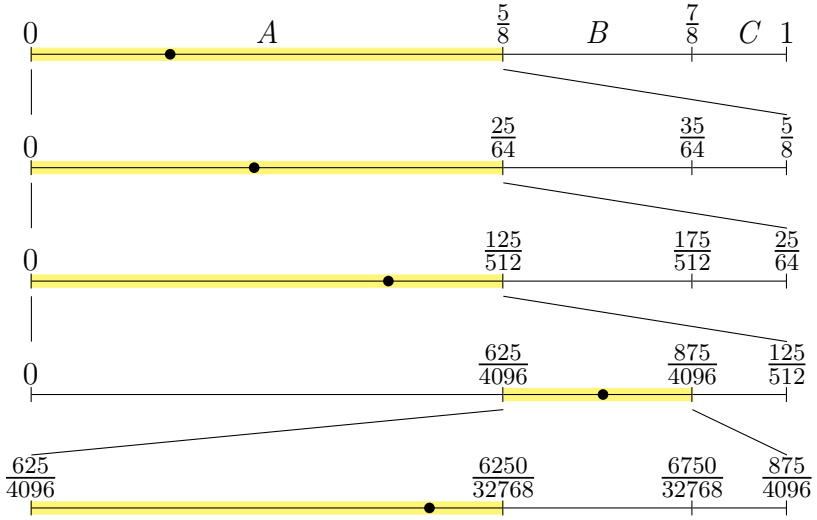


FIGURE 3.2: The first five steps of the Arithmetic Coding example

as Huffman coding does. This coding method starts with a specified interval, and then the inputs are read symbol by symbol, while the interval is narrowed down. For instance, three symbols A, B, and C with probabilities 0.625, 0.25, and 0.125, are assigned the subintervals $\left(0, \frac{5}{8}\right]$, $\left(\frac{5}{8}, \frac{7}{8}\right]$ and $\left(\frac{7}{8}, 1\right]$. As Fig. 3.2 shows, to encode the string "AAABABAC", we start with the interval $(0, 1]$. The first symbol A reduces the subinterval from its 0 point to 62.5% point, whose result is $\left(0, \frac{5}{8}\right]$. The second A reduces $\left(0, \frac{5}{8}\right]$ in the same way to $\left(0, \frac{5^2}{2^6}\right]$, then $\left(0, \frac{5^3}{2^9}\right]$. Step by step, the last C narrows the subinterval from $\left(\frac{5^4\times37}{2^{17}}, \frac{5^5\times31}{2^{19}}\right]$ to $\left(\frac{5^4\times1233}{2^{22}}, \frac{5^5\times31}{2^{19}}\right]$. In this case, any number in the subinterval $\left(\frac{5^4\times1233}{2^{22}}, \frac{5^5\times31}{2^{19}}\right]$, such as $(0.0010111101)_2$ is able to represent the string "AAABABAC".

In Deep Compression [12] and Coreset-base Compression [23], Huffman coding [204] was used to compress the quantized weights. In [21] and [22], another Fixed-to-Variable (F2V) [205] coding method—arithmetic coding [203]—was adopted. However, the decoding stage of these F2V methods can be inefficient, because the

encoded string needs to be processed on a bit-by-bit basis. Therefore, it is difficult to develop parallel implementations for the decoding, F2V codewords are of variable length and can not be indexed (or memory usage would be very inefficient). As a result, we cannot decode multiple symbols per single clock cycle. Besides, F2V coding methods have very high computational complexity for decoding. Given the number of codewords (quantized values) n and the reciprocal of compression ratio k, the decoding complexity is as much as $O(n \cdot k)$. Unlike the encoding stage which can be done offline, decoding must be processed online. In real-time applications, if decoding is not efficient, the inference rate could be reduced.

3.2.2 Variable-to-Fixed Entropy Coding Schemes

Conversely, V2F coding methods encode multiple symbols to a fixed number of bits. In the decoding stage, it is therefore possible to process multiple bits each time and decode multiple symbols per clock cycle. It is also feasible to decode the encoded string in parallel, as the encoded string can be split into fixed-length bit chunks based on the length of codewords.

Fig. 3.3 illustrates an example of the construction of the Tunstall tree. In this example, we have three symbols A, B, and C with probabilities 0.625, 0.25, and 0.125, respectively—these probabilities indicate the likelihood of a weight having this symbol[1]. We construct a 3-bit Tunstall tree to encode the symbols. Specifically, the construction of the Tunstall tree is performed iteratively. We start with a single symbol at the root with probability 1.0. For each iteration, the leaf node with the highest probability will be selected and converted to a subtree with K children, where K is the total number of symbols and $K = 3$ in this example. For instance, at the second iteration (Fig. 3.3b), the leaf node A, which has the largest probability 0.625 (Fig. 3.3a), is selected and converted into a subtree with children AA, AB, and AC. The third iteration converts leaf node AA, whose probability is 0.391, to the father node of a new subtree. The iteration continues until the number of leaf nodes reaches 2^C, where C is the bit size of the Tunstall codewords and $C = 3$ here. The codebook of this example is listed in Table 3.2. According to this codebook, sequence "AAABAAC" is encoded as a binary string "000101010".

[1]The values of these symbols represent DNN weights. In rare cases, two symbols may be combined to represent a single weight. This is done to improve encoding efficiency

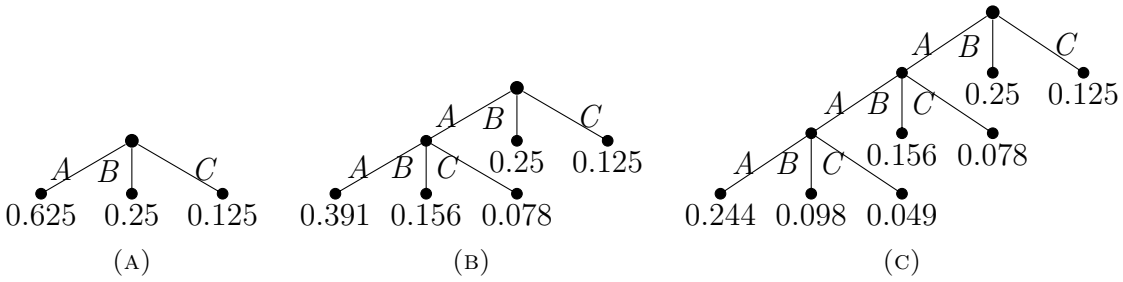


FIGURE 3.3: The Tunstall tree

TABLE 3.2: The codebook of the example Tunstall tree

Codeword	**000**	**001**	**010**	**011**	**100**	**101**	**110**
Symbols	*AAA*	*AAB*	*AAC*	*AB*	*AC*	*B*	*C*

3.3 Memory-Oriented Decoding Scheme

3.3.1 The Decoding Algorithm

As the codewords of Tunstall coding have a fixed length, in the decoding stage, we can process multiple bits each time (versus bit-by-bit processing in F2V techniques) and decode the codeword efficiently by reading from the codebook. For example, if the length of codewords is 8 bits, 8 bits are processed each cycle. Algorithm 1 describes the procedure of Tunstall decoding. In this chapter, the codewords are 10 bits.

Algorithm 1: The procedure of Tunstall decoding

input : A codeword under decoding c,
the codebook b
output: A sequence of decoded *symbols*
Function `SWDecoding`(b, c) **is**
  dictionary value $\leftarrow b[c]$;
  foreach *symbol* **in** *dictionary value* **do**
    Output the decoded *symbol*;
  end
end

The Memory-Oriented (MO) method comprises a dedicated memory segment, located on-chip, that stores the codebook. This method, akin to Algorithm 1, uses codewords as indices to retrieve the corresponding uncompressed weights. Fig. 3.4

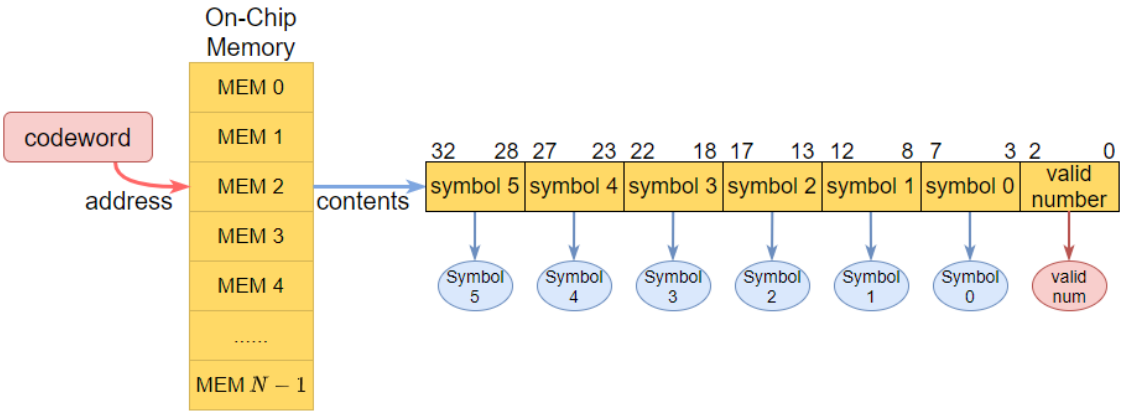


FIGURE 3.4: The memory-oriented hardware decoding method

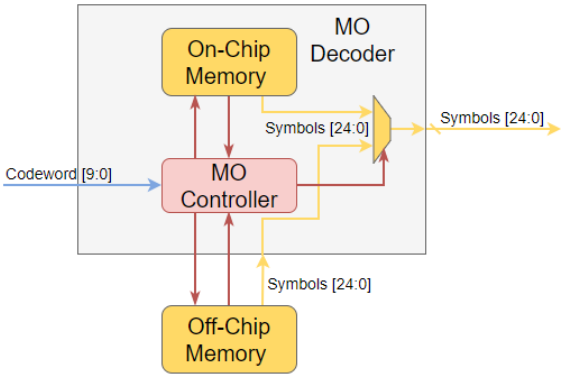


FIGURE 3.5: The architecture of the MO decoder

illustrates this method. N codebook entries are stored in the dedicated small memory segment. When a new codeword is received, the corresponding entry that stores uncompressed weights and the number of valid elements is read.

3.3.2 The Hardware Architecture

As Fig. 3.5 illustrates, the MO decoder contains two main components: a segment of on-chip memory with M entries which stores the dictionary, and the MO controller.

In most situations, the whole dictionary is too large to be stored on-chip, while the latency is too heavy to read off-chip for all tokens. Trading off the size of the on-chip memory and the reading latency, it stores top M frequent codewords on-chip. In this case, the dictionary needs to be updated with the Algorithm 2 introduced in Section 3.3.3.

32 28	27 23	22 18	17 13	12 8	7 3	2 0
symbol 5	symbol 4	symbol 3	symbol 2	symbol 1	symbol 0	valid number
					symbol 0	$3'b001$
				symbol 1	symbol 0	$3'b010$
			symbol 2	symbol 1	symbol 0	$3'b011$
		symbol 3	symbol 2	symbol 1	symbol 0	$3'b100$
symbol 5	symbol 4	symbol 3	symbol 2	symbol 1	symbol 0	$3'b101$

FIGURE 3.6: The data format for the MO decoder. Examples of different valid values are given below, where the empty boxes mean that their values should be ignored by the receiver.

3.3.2.1 On-Chip Memory

The memory stores parts of the complete dictionary. Uncompressed data could be read directly with only small energy consumption if the codeword is contained in the entries—codewords can then be decoded every cycle that generates more than a single weight per cycle. The format of the data stored in the memory is illustrated in Fig. 3.6. The *valid number* field is used to indicate the number of valid symbols in each memory word in the dictionary. For instance, $3'b011$ in the valid field indicates that there are three valid symbols decoded from the current codeword, and the other two symbols in current data which is read out from the memory will be ignored by the receiver. Other examples are listed in Fig. 3.6.

3.3.2.2 MO Decoder Controller

This controller is used to decide whether to read on-chip or off-chip. This is required when the number of entries in a codebook far exceeds the available entries on the on-chip memory. If the current codeword is contained in the dictionary stored in the on-chip memory, the controller will enable the on-chip memory. Otherwise, the controller will disable the on-chip memory, access the uncompressed weights from the off-chip memory, and wait for the response. If the number of on-chip memory entries is larger than the number of codebooks, the decoding can be computed on-chip.

Algorithm 2: Update the codewords

```
Function Updating(oldCodebook) is
    D ← emptydictionary;
    UpdatedCodebook ← emptydictionary;
    // count the frequency
    for codeIdx ← 0 to numberofcodewords do
        if codeIdx ∈ keys(D) then
            D[codeIdx] ← D[codeIdx] +1;
        else
            D[codeIdx] ← 1;
        end
    end
    // sort the counter array
    Sort(values(D));
    // remapping the codewords
    for i ← 0 to size of keys(D) do
        UpdatedCodebook[i] ← oldCodebook[keys(D)[i]];
    end
end
```

3.3.3 Updating the codewords for the MO decoder

To improve the hit ratio of the on-chip dictionary, the dictionary will be sorted and remapped. Algorithm 2 describes the process, where $values(D)$ are the frequencies of the codewords, and $keys(D)$ are the codewords.

Firstly, the frequency of the old codewords is counted. Secondly, the old codewords are sorted by frequency. Then, the codewords are remapped to their rank in the codebook while keeping the corresponding uncompressed symbols.

For instance, in Fig. 3.7, the left one shows the original codebook based on the Tunstall tree. However, in one particular compressed dataset, the frequency of different codewords varied. After analyzing and sorting the codewords based on their frequency as shown in the middle of Fig. 3.7, the codebook and codewords can be updated and remapped using the rank of their frequency.

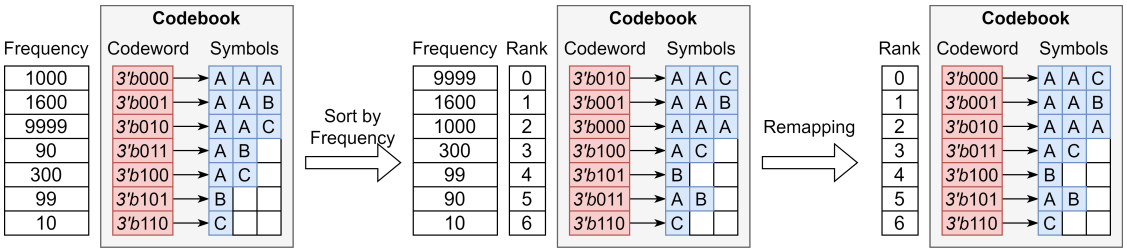


FIGURE 3.7: The example of updating the codewords for the MO decoder based on their frequency. Left: the original codebook; middle: sort the codewords by frequency for one particular compressed dataset; right: remapping the codewords with their rank.

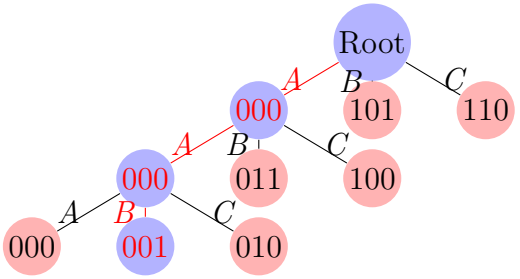


FIGURE 3.8: An example of the Tunstall-Tree Based Hardware Decoding

3.4 Logic-Oriented Decoding Scheme

3.4.1 The Decoding Algorithm

The Logic-Oriented (LO) decoding algorithm follows the reversal of the encoding process in Fig. 3.3. Observed from the Tunstall tree, uncompressed symbols at each level in the Tunstall tree are equal to the rank of nodes' codewords.

Fig. 3.8 illustrates the decoding process, where edges represent the uncompressed symbols. Assume we decode the codeword 001. First, it is compared with three children of the root (codewords 000, 101, and 110). Because 001 is between 000 and 101, node 000 is selected. As the rank of 000 at level 2 is 0, the first decoded symbol is A. Repeating the same process, node 000 at level 3 is selected and the second decoded symbol is A. Finally, the last uncompressed symbol B is obtained.

Algorithm 3 summarizes the flow and is illustrated in Fig. 3.9, where the information of the nodes n contains three kinds of information: the codewords of the nodes (nc), whether the nodes have child nodes (ns) and have further descendants. Fig. 3.12 illustrates the data format of n. The function *Find Rank* is described in Algorithm 4.

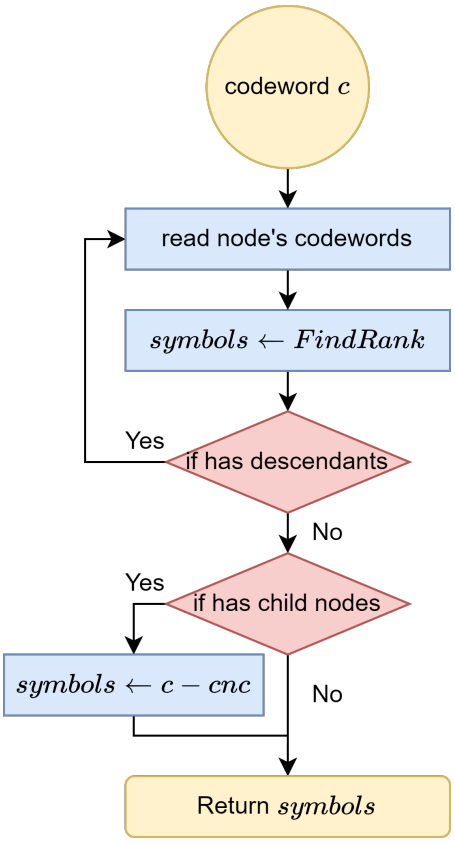


FIGURE 3.9: The algorithm overview of Tunstall-Tree Based decoding method.

3.4.2 The Hardware Architecture

The LO decoder comprises four units: one or multiple sub-decoders, one subtractor, a memory segment to store the symbols, and a controller. Fig. 3.10 illustrates the architecture of a LO decoder with five sub-decoders (six stages). Each decoder will be used to decode one level of the tree. If the tree depth is larger than the number of sub-decoders, the last sub-decoder will be iteratively used to go over each level of the tree.

3.4.2.1 Sub-Decoder

Each sub-decoder generates one symbol (*i.e.*, DNN weight) per clock cycle. As shown in Fig. 3.11, it contains a comparator, a selector, and a node memory. The node memory stores the information of the Tunstall tree nodes, whose data format is illustrated in Fig. 3.12. When a sub-decoder receives a codeword, it compares it with all stored codewords. The comparison will generate '1' or '0' if the stored

Algorithm 3: Decoding the symbols along the Tunstall tree. *nc*: an array of the codewords of the nodes; *ns*: an array of Boolean values of whether the nodes are inner nodes; *na*: an array of node addresses; *ng*: an array of Boolean values of whether the nodes have descendants

input : codeword under decoding c,
depth of current nodes in the Tunstall tree d,
address of the memory a
output: a sequence of uncompressed *symbol*s
Function `Decoding(`c, d, a`)` **is**
    $symbols \leftarrow \emptyset$;
    $nc, ns, na, ng \leftarrow n[d][a]$;
    $curSyb \leftarrow$ `FindRank(`c, nc`)`;
    if *current node has descendants* **then**
        $a \leftarrow na[curSyb]$;
        Decoding the symbol using the function `Decoding(`c, $d+1$, a`)`;
    else
        if *current node has child nodes* **then**
            // *cnc*: current node's codeword
            $cnc \leftarrow nc[curSyb]$;
            $symbols \leftarrow c - cnc$
        else
            current codeword has been decompressed
        end
    end
end

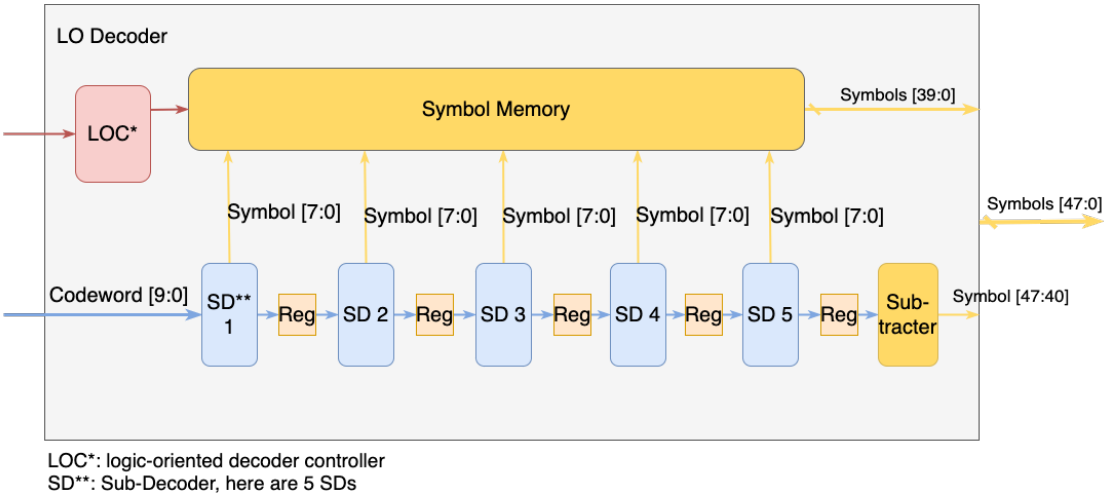


FIGURE 3.10: The architecture of the $N = 6$ stages LO decoder (assuming a 10-bit codeword)

code word is smaller or larger than the input (bits $[9:0]$ of the stored data as shown in Fig. 3.12). The output of all comparisons is then one-hot encoded to

Algorithm 4: Find the rank of the codeword

input : codeword under decoding c,
codewords of the nodes nc
output: the rank of this codeword r
Function `FindRank(`c, nc`)` **is**
    $compResult \leftarrow \emptyset$;
    for $node\ idx \leftarrow 0$ **to** $number\ of\ nodes$ **do**
        if $c \geq nc[node\ idx]$ **then**
            $compResult[node\ idx] \leftarrow 1$;
        else
            $compResult[node\ idx] \leftarrow 0$;
        end
    end
    $onehot \leftarrow compResult \oplus [0, compResult[: -1]]$;
    `// convert the onehot to a integer`
    $r \leftarrow$ `OHToInt(`$onehot$`)`;
end

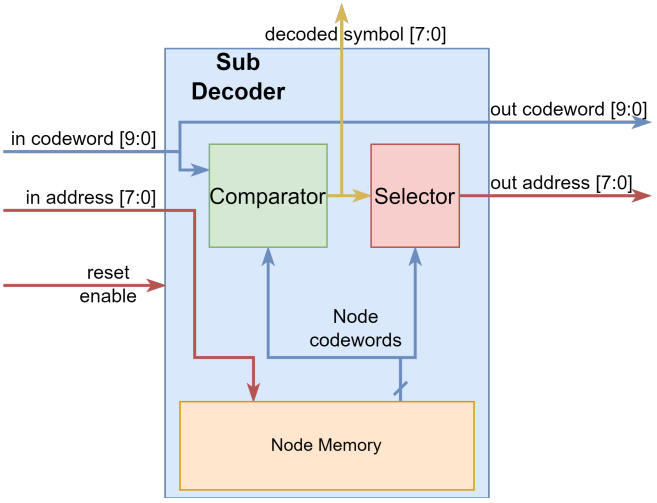


FIGURE 3.11: The architecture of the sub-decoder.

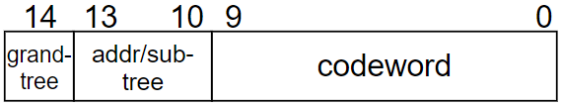


FIGURE 3.12: The data format for the LO decoder

generate the symbol as described in Algorithm 3.

As illustrated in Fig. 3.12, the data stored in the node memory contains three parts: the codeword of the node($[9:0]$), whether the node has descendants(MSB), and

whether the node has child nodes ([10]) or the start address of its child nodes in the memory ([10 : 13]).

3.4.2.2 Subtractor

The subtractor is the last module in the pipeline and is used to decode the leaf nodes by performing the operation described in line 12 of Algorithm 3.

3.4.2.3 Symbol Memory

Symbol Memory stores the decoded weights during the full decoding process. It is split into $N-1$ banks, each bank stores $N-1$ decoded weights of the corresponding codeword. Once the last symbol is obtained from the subtractor or the bank is full, symbols stored in the corresponding bank will be read out.

3.4.2.4 LO Decoder Controller

This module manages the accessibility of the Symbol Memory (described above). It enables writing $N-1$ symbols decoded in the $N-1$ sub-decoders simultaneously, each symbol to its corresponding bank. Decoded symbols are read in order. If the codewords are not fully decoded while the corresponding bank is full, the controller triggers a halt (via clock gating) to all prior stages. Afterward, it evicts the contents of the register bank, while enabling the sub-decoder to refill the bank. This process is repeated till all symbols are decoded.

3.5 Design Metrics and Impact on Performance

3.5.1 Design Metric of MO decoder

3.5.1.1 The codeword bits

Table 3.3 summarizes the impact of codeword bits based on the 28,460,930 quantized 4-bit weights. Increasing the bit of the codeword results in a higher compression ratio, but this yields diminishing returns. Increasing the number of bits from

TABLE 3.3: The impacts of Tunstall codeword bits

Codeword Bit	Comression Ratio *	Memory Cell Bit	Decoder Memory Capacity (KB)	# Codeword **	Codebooks and Codewords Size (MB)	LUT Read (MB)
8	1.6248	44	1.38	5,777,177	5.54	19.22
10	1.6356	55	6.88	4,565,567	5.61	18.82
12	1.6631	95	47.50	3,781,777	6.27	18.66
14	1.6732	114	228.00	3,228,979	9.45	18.82

* weight size / codeword size

** The codewords are compressed from 28,460,930 weights. This value also equals the decoding clock cycles.

8 to 14 bits resulted in a 3% higher compression and 44% speedups of decoding. Furthermore, the decoder memory capacity to store the codebooks would increase significantly as the size of the codebooks and codewords also increase. Considering the trade-offs of resource utilization, read energy consumption, and decoding speed, we set the Tunstall codeword size to be 10 bits in this Chapter.

3.5.1.2 The number of on-chip memory entries

As Section 3.3.1 indicates, increasing the number of memory entries decreases the decoding cycles and consumes less energy as more codebooks are stored on-chip and fewer are read off-chip. However, if the area is limited, only top M frequent codebook pairs are stored on-chip, where M is the entries of the on-chip memory.

3.5.2 Design Metric of LO Decoder

3.5.2.1 The number of registers in one sub-decoder

As Algorithm 4 describes, the number of registers in one sub-decoder reflects the maximum number of nodes it can compare with. If the number of nodes exceeds the maximum number of memory banks, the LO decoder takes additional cycles to obtain the remaining nodes and complete the comparison.

3.5.2.2 The number of sub-decoders

Each sub-decoder executes the Algorithm 3 parallelly. Hence, the number of sub-decoders limits the maximum number of codewords that can be decoded simultaneously. Though more sub-decoders reduce the decoding cycles, it increases the hardware resources required.

TABLE 3.4: The resource utilization and decoding cycles of MO and LO decoders

Resources and Decoding Cycle		Memory-Oriented Decoder		Logic-Oriented Decoder			
		MO 1024 Entries RAM	***MO 64 Entries RAM***	***LO 8 SD****		***LO 1 SD***	
				27 Banks	***10 Banks***	***27 Banks***	***10 Banks***
Resources**	***LUT***	1601	267	6,322	2,140	4,942	1,771
	FF	1184	76	3,201	33	544	33
	LUT RAM	1	13	2,585	1,442	2,797	1,236
	BUFG	1	1	2	1	2	1
Cycle	***ResNet-50***	4,565,527	10,764,540	8,621,733	26,099,734	8,711,905	26,184,537
	MobileNet-v2	743,007	1,848,581	1,081,004	3,621,028	1,117,003	3,655,032

*Sub-Decoder

**The RTLs are generated by the Chisel codes, and synthesized by Vivado 2020.1 without optimization.

3.5.3 Performance Evaluation

We have developed an RTL implementation of both MO and LO, and we ran them on Gensys II FPGA (as a stand-alone component and integrated into the open-source PULPissimo platform [63]).

Table 3.4 lists the resource utilization and the corresponding clock cycles to decode ResNet-50 [7] and MobileNet-v2 [8] of MO decoders with a different number of memory entries and LO decoders with a different number of sub-decoders and memory banks. We found that the 64 entries MO decoder reduces memory capacity by 16× while taking 2.36× and 2.49× more cycles than the 1024 entries one to decode ResNet-50 and MobileNet-v2, respectively. Reducing the number of memory banks from 27 to 10 decreases the resource usage by around 20% while increasing decoding cycles by only 3%. 3× speedup is obtained by adding 7 more sub-decoders.

3.6 System-Level Impact on Deep Learning Accelerators

We quantify the impact of our approach against *32-bit full precision*, *quantized 8- and 4-bit*, and *Huffman encoding*. The size of the uncompressed weight far exceeds the on-chip capacity. Thus, we assume that the weight will be allocated in a slower, but large off-chip memory (*e.g.*, eDRAM, whose energy consumption is 15 pJ/bit). For the compressed situations, codebooks and codewords are stored in the on-chip memory (*e.g.*, SRAM, whose energy consumption is 2 pJ/bit) and decoding is done by reading on-chip LUT (register files, whose energy consumption is 0.2 pJ/bit).

TABLE 3.5: The weights size and estimated memory access energy consumption in different methods

CNN Architectures	Methods	Weights Size (MB)	Estimated Energy Consumption (mJ)
ResNet-50	*full precision (32-bit)*	108.57	13,661.16
	Quantized 8-bit	27.14	3,415.29
	Quantized 4-bit	13.57	1,707.65
	Quantized 4-bit with Tunstall LO decoder	5.85	373.55
	Quantized 4-bit with Tunstall MO decoder	6.13	130.13
	Quantized 4-bit with Huffman coding	5.59	139.30
MobileNet-v2	*full precision (32-bit)*	13.65	1,717.94
	Quantized 8-bit	3.41	429.12
	Quantized 4-bit	1.71	214.74
	Quantized 4-bit with Tunstall LO decoder	0.91	57.17
	Quantized 4-bit with Tunstall MO decoder	1.21	24.02
	Quantized 4-bit with Huffman coding	0.89	20.61

We obtain these values using Destiny [206]. We assume a 32-bit word, whereas the decoder and the accelerator have the same operating frequency.

3.6.1 Main memory entries requirements

Table 3.5 summarizes the memory capacity to store the weights (includes codebooks if compressed) of ResNet-50 and MobileNet-v2 CNNs. Both Tunstall coding and Huffman coding require 18.0× and 2.3× less than full precision and 4-bit weights, respectively, to store the whole ResNet-50. Huffman coding occupies around 4% less capacity than the two Tunstall coding schemes. This is because we can only use 30 out of the 32-bits per word–10-bit codeword and three codewords per memory word, whereas the two extra bits indicate valid codewords as Fig. 3.13a shows.

3.6.2 Estimated memory access energy consumption

Table 3.5 lists the estimated energy consumption to access all weights from the read and write energy consumption of each memory hierarchy from Destiny [206] and the corresponding number of read and write operations. For ResNet-50, the LO decoder consumes 37× less energy to access all the weights compared to full precision and 4.6× less energy compared to start-of-the-art quantized 4-bit weights. MO decoder achieves 110× and 13.2× less energy compared to full precision and quantized 4-bit weights. MO decoder consumes similar energy to the Huffman decoder.

TABLE 3.6: Memory access cycles in different methods

CNN Architectures	Methods	Weights Read Clock Cycle
ResNet-50	***full precision (32-bit) ****	142,303,750
	Quantized **	28,460,750
	Quantized 4-bit with Tunstall LO decoder	8,653,482
	Quantized 4-bit with Tunstall MO decoder	4,620,697
	Quantized 4-bit with Huffman coding	28,460,750
MobileNet-v2	***full precision (32-bit)***	17,895,185
	Quantized	3,579,037
	Quantized 4-bit with Tunstall LO decoder	1,084,028
	Quantized 4-bit with Tunstall MO decoder	796,888
	Quantized 4-bit with Huffman coding	3,579,037

* Stored off-chip, 5 clock cycles per read per weight
** Stored on-chip, 1 clock cycle per read per weight

3.6.3 Weight read clock cycles

As Table 3.6 shows, the LO decoder achieves a 9.9× and 13.9× speedup when reading the weights of ResNet-50 and MobileNet-v2, respectively, compared to full-precision where weights are stored off-chip. MO decoder achieves 18.5× and 13.5× speedup. We provided a detailed discussion on comparing Huffman decoding in Section 3.7.

3.6.3.1 Resource utilization overheads

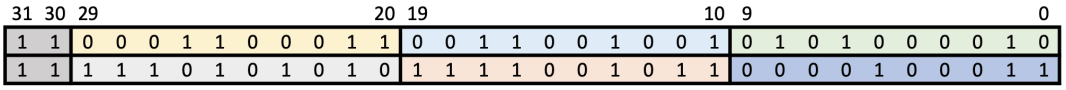
We integrated both decoders into the PULPissimo SoC platform [63] and ran them on FPGA (Gensys II). The MO 1024 RAM decoder and LO 8 SD 27 banks decoder only bring 2.16% and 8.14% resource overheads, which is affordable considering the significant improvements in performance and power. The MO 64 RAM decoder only brings 0.48% overhead for such a system.

3.7 Quantifying Tunstall Vs. Huffman Coding

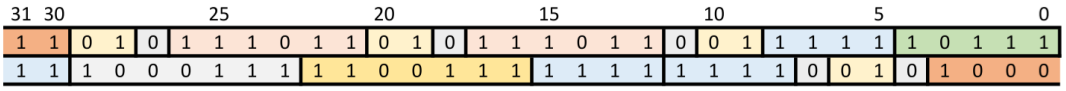
In this section, we quantify the performance of the 7-stage 24 memory banks LO Tunstall decoder and the 1024 entries MO Tunstall decoder with the 256 entries Huffman decoder. The Huffman decoder decodes one weight per clock cycle. The performance for ResNet-50 is summarized in Table 3.7. All implementations are synthesized and mapped to FPGA.

TABLE 3.7: Tunstall decoder and Huffman decoder performance of ResNet-50

Hardware Decoder	Resources Utilizations (# LUT)	Decoding Cycles	Memory Capacity (MB)	Memory Access Consumption (mJ)
7-stage 24 memory banks LO Tunstall decoder	6,322	8,653,482	5.85	373.5
1024 entries MO Tunstall decoder	1,601	4,565,527	6.13	130.1
256 entries Huffman decoder	5,021	28,460,750	5.59	139.3



(A) Tunstall codewords example



(B) Huffman codewords example

FIGURE 3.13: Tunstall and Huffman codewords stored in two 32-bit memory cells.

3.7.1 The Storage of the Codewords

Fig. 3.13 illustrates how Tunstall and Huffman codewords are stored in two 32-bit main memory cells. Different colors represent different codewords. There are six 10-bit Tunstall codewords in Fig. 3.13a which are compressed from 17 symbols while the number bits of Huffman codewords in Fig. 3.13b varies from one to seven, which is extremely irregular. These 64-bit Huffman codewords represent 18 uncompressed symbols. The two most significant bits in Fig. 3.13a represent the number of valid codewords in the 32-bit memory cell.

3.7.2 The Decoding Clock Cycles

3.7.2.1 Software decoding time

Such irregularities render Huffman coding to be slower than Tunstall coding. We implement the Huffman decoding method and Tunstall decoding method in C++ and they are executed and measured on an Intel Core CPU. The software decoding time is summarized in Table 3.8. Tunstall is 1.5× faster than Huffman for the same CNNs, though in some small and simple layers, Huffman decoders perform better than the Tunstall decoder.

TABLE 3.8: Tunstall and Huffman software decoding time (microsecond)

CNN Architectures	**Coding Methods**	**Software Decoding Time (microseconds)**		
		maximum	***average***	***minimum***
ResNet-50	***Tunstall Coding***	121,999	23,073.59	541
	Huffman Coding	178,217	35,786.87	732
MobileNet-v2	***Tunstall Coding***	71,046	3,606.79	302
	Huffman Coding	105,512	5,578.81	75

3.7.2.2 Hardware decoding clock cycles

As Table 3.7 shows, the 1024 entries MO Tunstall decoder and the 7-stages LO Tunstall decoder are 6.23× and 3.29× faster than the Huffman decoder in ResNet-50.

3.8 Chapter Summary

In this chapter, we present Tunstall coding to further compress weights after state-of-the-art quantization. Two hardware-accelerated decoding prototypes are designed and implemented, namely memory-oriented Tunstall decoder and logic-oriented Tunstall decoder. Compared with the full precision 32-bit networks, the MO decoder reduces 19.58× and 21.67× memory usage in the inference stage on ResNet-50 and MobileNet-v2 respectively, while the LO decoder reduces 19.76× and 22.08×. Our designs are around 6× and 3× faster than Huffman coding. Our Tunstall decoders are suitable for integration in the deep learning accelerator to significantly reduce the memory footprint while bringing little latency and resource overheads.

Chapter 4

Scalable Hardware Acceleration of Non-Maximum Suppression

4.1 Introduction

Convolution-based object detectors are grouped into one-stage paradigms such as SSD [207] and YOLO [25] and two-stage paradigms such as Fast-RCNN [208] and Faster-RCNN [209]. As highlighted in Chapter 1, a scalable NMS hardware accelerator that supports high image resolution is needed to alleviate the computational bottleneck caused by NMS.

In this chapter, we present ShapoolNMS—a scalable and parallelizable hardware accelerator for PSRR-MaxpoolNMS [62]—to speed up the NMS process in both one-stage detectors and two-stage detectors with low power. Besides, the scalability of ShapoolNMS is analyzed, and the design space is explored.

ShapoolNMS can achieve 3.00× to 19.71× speedup over the state-of-the-art hardware accelerators of NMS [210] with 2.58× less area when both are targeting 28 nm. Its area is also less than 0.01% than the area of NVIDIA A100 Tensor Core GPU and Google Tensor Processing Unit (TPU). The NVIDIA A100 Tensor Core GPU is fabricated on TSMC's 7 nm N7 manufacturing process with a die size of 826 mm^2 while TPU v1 is fabricated with a die size of 331 mm^2 in a 28 nm process node.

The work in this chapter has been published in [Proceedings of the 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022, pp. 96-99].

4.2 Related Work

4.2.1 Object Detection

Object detection not only classifies the images but also precisely estimates the locations of detected instances of objects in the images. It is the foundation of many other computer vision problems, namely object tracking, image captioning, and instance segmentation. Object detection has attracted a lot of attention in recent years and is used in many applications such as autonomous driving, face recognition, robot vision, human behavior analysis, video surveillance *etc.*

Traditional object detection algorithms were designed based on fine-tuned hand-made features. Viola Jones detector [211, 212], Histogram of Oriented Gradients (HOG) detector [213] and Deformable Part-based Model (DPM) [214] are three typical early detectors.

One-stage and two-stage detectors are the two main paradigms in deep learning-based object detection frameworks. One-stage detectors, *e.g.*, SSD [207] and YOLO [25], predict the location of the bounding boxes and object classification by applying the classifier and location regressor to a set of local windows based on the location, scales, and ratio. A focal loss function is introduced in Retina-Net [215] for detectors to pay more attention to the misclassified training cases.

RCNN [216], a two-stage detector, employs hand-crafted region proposal generation and is further optimized by Fast-RCNN [208]. Fast-RCNN directly crops out the proposal features from the image feature maps instead of separately inputting each proposal into the RCNN to generate proposal features. To generate region proposals, Faster-RCNN [209] trains a Region Proposal Network (RPN). A top-down architecture, Feature Pyramid Network (FPN), is designed in Pyramid Networks [217] to take advantage of the pyramidal feature of convolutional networks in PRN and Fast-RCNN.

4.2.2 Non-Maximum Suppression Algorithms

Non-Maximum Suppression (NMS) is a crucial step in object detection to filter the highly overlapping bounding boxes caused by the similarity of the adjacent

detection windows to reduce false positives. In [218], NMS algorithms are classified into three subcategories, *i.e.*, greedy selection, bounding box aggregation, and learning-based NMS.

GreedyNMS [213], one of the most prominent greedy selection NMS algorithms, is the de-facto widely used stander NMS method, where candidate boxes are sorted in descending order followed by nested loops. In each loop, only the candidate box with the highest confidence score is kept and the others are rejected if they have an intersection over union (IoU) larger than the threshold θ. However, GreedyNMS also accepts wrong bounding boxes [219], which decreases the detection accuracy. Some modified greedy selection NMS algorithms have been proposed recently. SoftNMS [220] decreases the scores of the boxes that are to be suppressed using a linear or Gaussian function instead of directly removing them. An NMS algorithm is proposed in IoU-Net [221] to provide the localization confidence that guides the suppression decision. Adaptive NMS [222] sets the Intersection over Union (IoU) threshold adaptively based on the density of the object. FeatureNMS [223] extends GreedyNMS by leveraging the feature embedding distance to better determine the keeping or removing of the bounding boxes when the IoU is similar to the threshold.

Bounding box aggregation methods suppress several overlapped bounding boxes to only one considering not only the confidential scores but also spatial locations. Viola Jones detector [211, 212] employed a weak learning algorithm that determines the threshold to filter the candidate boxes. In Overfeat [224], the individual candidate boxes are merged using a greedy strategy across the detection classes with similar spatial locations in each scale. MaxpoolNMS [225] and PSRR-MaxpoolNMS [62] utilize MaxPool operations to cluster the overlapped bounding boxes with similar ratios, scales and spatial locations that only the candidate box with the highest confidential score are kept.

NMS is trained to be part of convolutional neural networks in learning-based NMS methods [219, 226–228]. Authors in Gnet [219] redefine NMS as a re-scoring task by allowing detections to amend their scores based on their neighbors. GreedyNMS is integrated into training directly by employing mAP as the loss function in [226].

4.2.3 Non-Maximum Suppression Hardware Acceleration

Hardware-based NMS is less explored. CPU-NMS [229] and GPU-NMS [230, 231] target CPU and GPU platforms. The second generation of GPU-NMS [231] could process 1027 bounding boxes in 0.324 ms on GeForce GTX 1060 at 1.70 GHz. Inspired by GPU-NMS [230], Shi *et al.* [210] implemented a power-efficient NMS accelerator that is able to merge 1000 bounding boxes in 12.79 μs at 400 MHz. However, the scalability of supporting large image resolution is not mentioned in this work. The capacity of the memories, such as box memory (BM), position based bit-table (PBBT) and compressed PBBT limit the support on large image resolution. 1024 bounding boxes are supported in this work. Besides, the previous work by Shi *et al.* works slower on the image with dense objects as their time complexity is $O(mn)$ where n and m are the numbers of bounding boxes before and after the NMS. For a high-resolution image, the number of n and m would significantly increase the resource usage and execution time of related accelerators.

MaxpoolNMS [225] and PSRR-MaxpoolNMS [62] reformulate NMS as MaxPool operation which is inherently parallel and friendly to the hardware. The hardware sorting process is eliminated in C-NMS [232] by clustering the bounding boxes based on the diagonal L1 distance followed by similar MaxPool operations to obtain the final results. However, the accelerated NMS still occupies a significant time compared with the execution time of convolution operations. Additionally, these solutions do not support high image resolution.

The existing methods for NMS have some limitations and drawbacks that motivate our work. Greedy selection methods have high time complexity and accuracy degradation due to sorting and nested loops. Learning-based NMS methods require additional supervision and computation due to integrating NMS into training. Current hardware-based NMS methods are not scalable or efficient enough to handle high image resolution and dense objects due to memory capacity and time complexity constraints.

In this case, a scalable NMS hardware accelerator that supports high image resolution is needed to alleviate the computational bottleneck caused by NMS.

4.3 PSRR-MaxpoolNMS Algorithm Review

PSRR-MaxpoolNMS (PSRR-MNMS) [62] approximates GreedyNMS in both one- and two-stage detectors via **Relationship Recovery** (RR). RR projects the boxes onto a 3D confidence score map and **Pyramid Shifted MNMS** (PS MNMS) eliminates more overlapped boxes.

The confidence score is the probability that an anchor box contains an object. The channels in the 3D confidence score map represent different scales s_0 and ratios r_0 of the anchor boxes, while each cell in the score map indicates the score and spatial information (x, y) of a bounding box (bbox). The scale of each anchor box equals its height times its width, and its ratio equals its height divided by its width. In this Chapter, there are 4 scales and 3 ratios with a downsampling ratio $\beta = 16$. s_0 and r_0 are:

$$s_0 \in [64^2, 128^2, 256^2, 512^2] \tag{4.1}$$

$$r_0 \in [1:2, 1:1, 2:1] \tag{4.2}$$

Hence, the confidence score map consists of 12 channels. Its width and height are $round(\frac{W}{\beta})$ and $round(\frac{H}{\beta})$ respectively, where W and H are the width and height of the images.

As the inputs of stage 1 in two-stage detectors are cells of the confidential score map, PSRR-MNMS is simplified as Single-Channel MNMS without RR and Shifted MNMS, which is called **partial PSRR-MNMS**. The PSRR-MNMS in one-stage detectors or stage 2 of two-stage detectors is called **full PSRR-MNMS**, which includes both RR and PS MNMS.

4.3.1 Relationship Recovery

Relationship Recovery (RR) projects the bounding boxes on the 3D score map after downsampling of spatial locations, as Fig. 4.1 illustrates, to solve the mismatch problem [62].

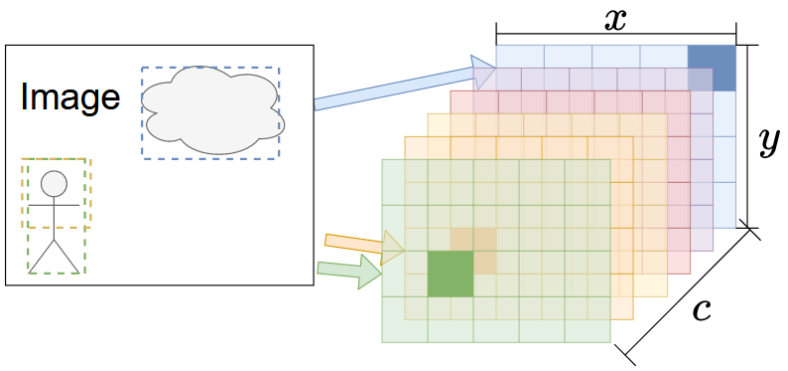


FIGURE 4.1: RR to project the bounding boxes (dashed rectangles) with different scales, ratios, and spatial locations into the 3D score map (highlighted rectangles)

4.3.1.1 Spatial Recovery

Spatial Recovery computes the spatial location (x, y) of the boundary box on the 3D score map based on its central position $[X_c, Y_c]$ and the downsampling ratio β:

$$x = [\frac{X_c}{\beta}] \tag{4.3}$$

$$y = [\frac{Y_c}{\beta}] \tag{4.4}$$

4.3.1.2 Channel Recovery

Channel Recovery projects the bbox to a channel $c(s_0, r_0)$, where s_0 and r_0 are its scale and ratio that are Euclidean-nearest to the box scale s' and ratio r'. Scale and ratio are computed from the candidate box width w' and height h':

$$s' = w' \times h' \tag{4.5}$$

$$r' = \frac{h'}{w'} \tag{4.6}$$

4.3.1.3 Score Assignment

Each cell in the score map only keeps the box with the highest score to remove highly similar boxes that are projected into the same cell. It can be considered as a pre-filtering step based on 1×1 MaxPool within every confidential score map cell to remove highly similar boxes.

4.3.2 Pyramid Shifted MaxpoolNMS

Pyramid Shifted MaxpoolNMS (PS MNMS) consists of Pyramid MNMS and Shifted MNMS. **Pyramid MNMS** consists of four steps to suppress the overlapped boxes. **Shifted MNMS** addresses the edge effect problem. They reduce the maximum number of overlapped boxes via a sequence of MaxPool operations with fine-tuned kernel sizes and strides:

$$h = \sqrt{r \cdot s} \tag{4.7}$$

$$w = \frac{s}{h} \tag{4.8}$$

$$k_x = s_x = max(round(\frac{\alpha w}{\beta}), 1) \tag{4.9}$$

$$k_y = s_y = max(round(\frac{\alpha h}{\beta}), 1) \tag{4.10}$$

where r and s are the ratio and scale of each channel that belongs to Eq. (4.2) and Eq. (4.1). h and w denote the height and width of the anchor boxes in the channel. k_x, k_y and s_x, s_y are kernels and strides in x and y direction. $\alpha \in (0, 1)$ represents the overlap threshold. A larger α would remove more overlapped boxes while missing the objects. The function $round$ rounds the values to the nearest integer. In this Chapter, α equals 0.75 and 0.25 in full and partial PSRR-MNMS respectively as [62, 225] suggest in PASCAL VOC, and α in YOLOv8 detectors equals 0.25.

4.3.2.1 Pyramid MaxpoolNMS

Pyramid MaxpoolNMS (Pyramid MNMS) mainly suppresses overlapped boxes by Single-Channel MNMS, Cross-Ratio MNMS, Cross-Scale MNMS, and Cross-all-Channel MNMS, which are illustrated in Fig. 4.2. Single-Channel MNMS applies MaxPool operations on every single channel separately with different kernel sizes. Cross-Ratio MNMS concatenates score maps at adjacent ratios for each scale with 3D MaxPool on the concatenated maps to remove the overlapped boxes with similar spatial locations and the approximately same scale. Cross-Scale MNMS eliminates the overlapped boxes with similar spatial locations and the same ratio by operating 3D MaxPool on the concatenated score maps at different scales for each ratio. Cross-all-Channels MNMS operates 3D MaxPool on all channels.

4.3.2.2 Shifted MaxpoolNMS

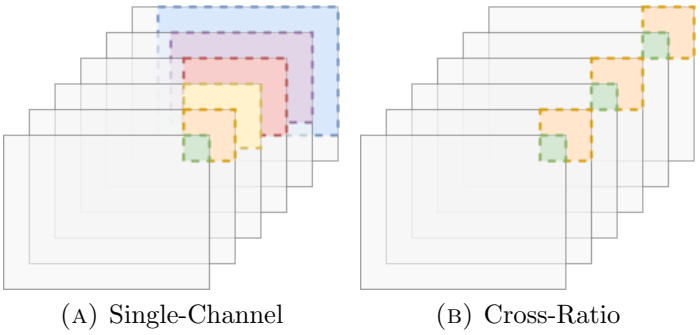
As Fig. 4.3 illustrates, Shifted MaxpoolNMS (Shifted MNMS) alleviates the edge effect problem by padding zeros around the score maps and then doing the Max-Pool operations as Pyramid MNMS, which further reduces the overlapped boxes. For instance, given a kernel size k, $\frac{k}{2}$ zeros are padded around the border of the confidential score maps, followed by the same MNMS steps in Pyramid MNMS.

4.3.3 Hardware/Software Co-optimization

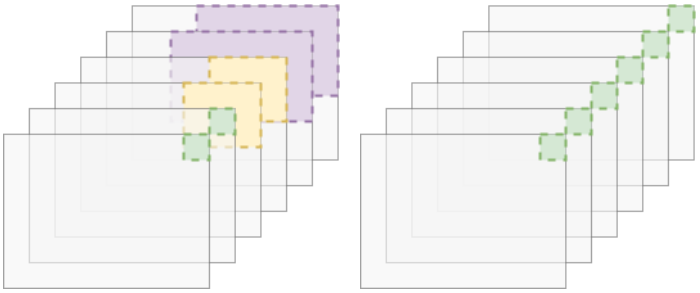
When designing ShapoolNMS, we optimized the original PSRR-MNMS toward a hardware-friendly algorithm.

4.3.3.1 Operation Parallelism

To accelerate the computation, we improve the parallelism of PSRR-MNMS by introducing multiple computation units to process NMS simultaneously.



(A) Single-Channel (B) Cross-Ratio



(C) Cross-Scale (D) Cross-Channel

FIGURE 4.2: The four steps of Pyramid MaxpoolNMS. The 6 channels in the score map represent 2 scales and 3 ratios. Different MaxPool kernels are represented in different colorful dashed rectangles, whose sizes are different.

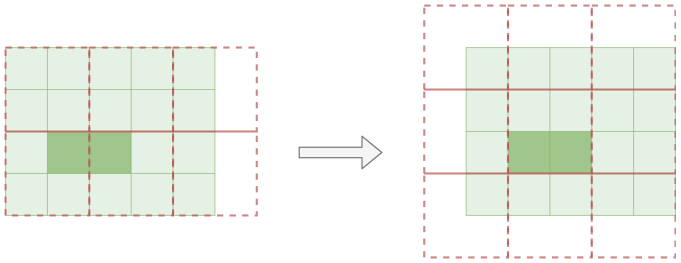


FIGURE 4.3: Shifted MNMS addresses the edge effect problem by padding $\frac{k}{2}$ zeros for a kernel size k, followed by the same MaxPool step in Pyramid MNMS. Two adjacent cells are kept after PS MNMS and only the higher one is kept after Shifted MNMS. MaxPool kernels are represented as red dashed rectangles.

4.3.3.2 Fuse Operations

Unlike PSRR-MNMS, we fused Score Assignment with Single-Channel MNMS as it is treated as 1×1 MaxPool operation to speed up and reduce energy consumption.

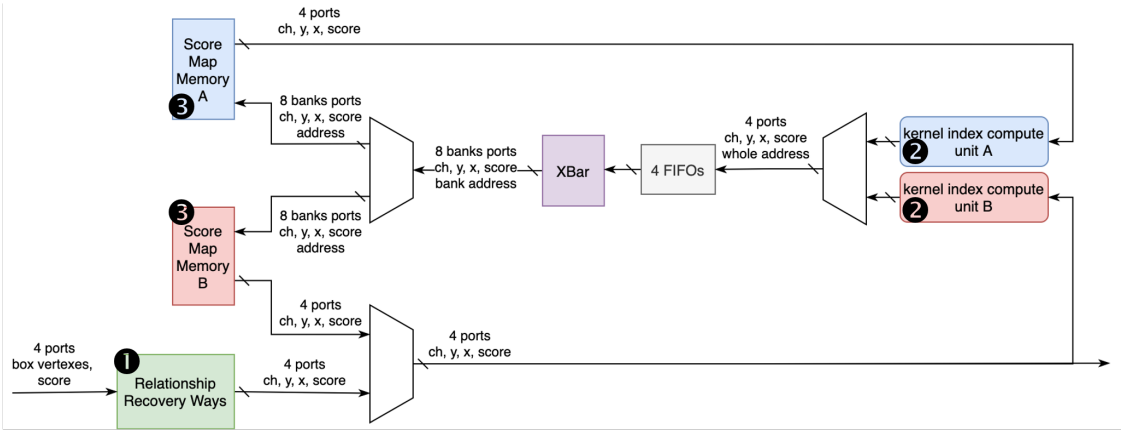


FIGURE 4.4: Overall Architecture of ShapoolNMS. The system architecture is illustrated with 4 RRWs ($N = 4$) and 8 memory banks in SMMs ($M = 8$). The depth of the FIFOs is 4. XBar is introduced in Section 4.4.4. ❶ Relationship Recovery Ways, which are introduced in Section 4.4.2. ❷ MaxPool Kernel Index Compute Unit which is introduced in Section 4.4.3. ❸ Score Map Memory for M memory banks, which is introduced in Section 4.4.5.

4.3.3.3 Converting MaxPool to Temporal Comparing

Unlike PSRR-MNMS where the MaxPool operation is done by comparing all the elements in the MaxPool kernel and finding the maximum number, we converted MaxPool to temporal comparing by comparing the elements in the MaxPool kernel sequentially as they come temporally to reduce the required comparators and temporal storage hence reducing chip area and energy consumption.

4.4 ShapoolNMS Architecture

Fig. 4.4 illustrates the architecture of ShapoolNMS, including Relationship Recovery Ways (RRWs), MaxPool Kernel Index Compute Units (MKICUs), XBar and Score Map Memories (SMMs).

4.4.1 ShapoolNMS Controller

As shown in Fig. 4.5, the four stages of the FSM coincide with the four steps in full PSRR-MNMS, where each stage is further split into two substages—Pyramid MNMS and Shifted MNMS.

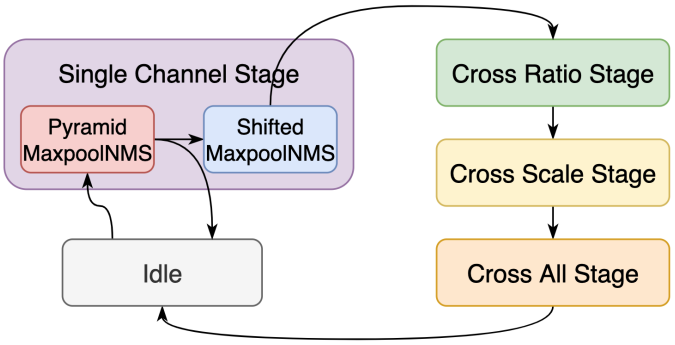


FIGURE 4.5: The Controller FSM simplified state transition diagram

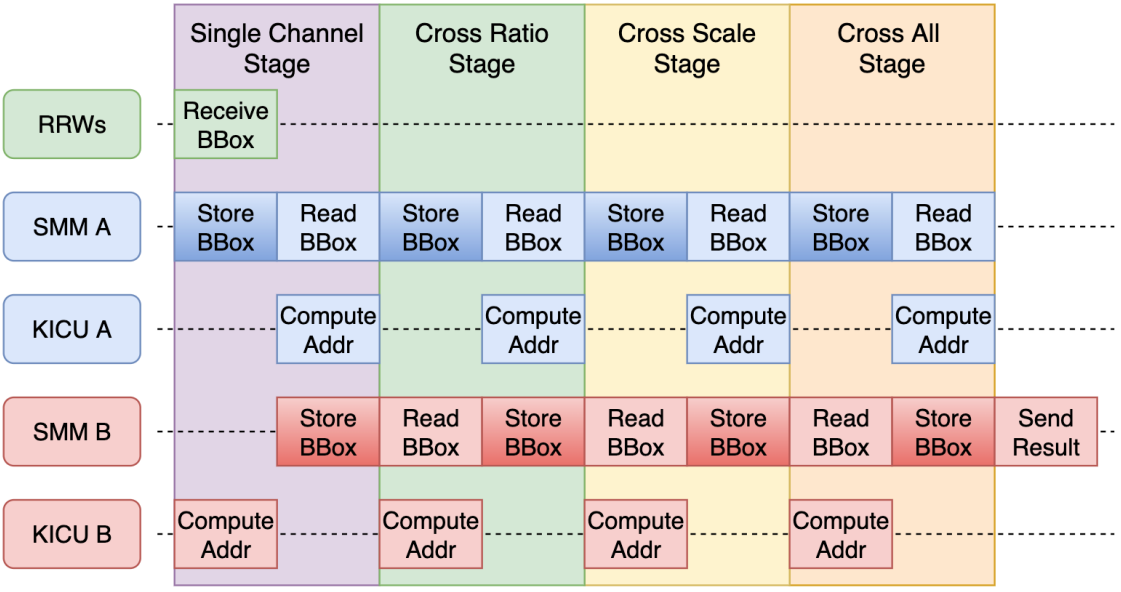


FIGURE 4.6: ShapoolNMS execution steps of full PSRR-MNMS divided into its main phases.

Fig. 4.6 illustrates the detailed operations of main modules in ShapoolNMS to process full PSRR-MNMS, which are described below as well:

- N boxes are input into the RRWs parallelly whose results are written into SMM A via the address computed by MKICU B;
- When all boxes are fed, N cells in SMM A are read simultaneously and MKICU A computes the address for SMM B;
- When SMM A read finishes, SMM B is read and MKICU B computes the address for SMM A;
- Repeat steps 2 and 3 until all stages are finished.

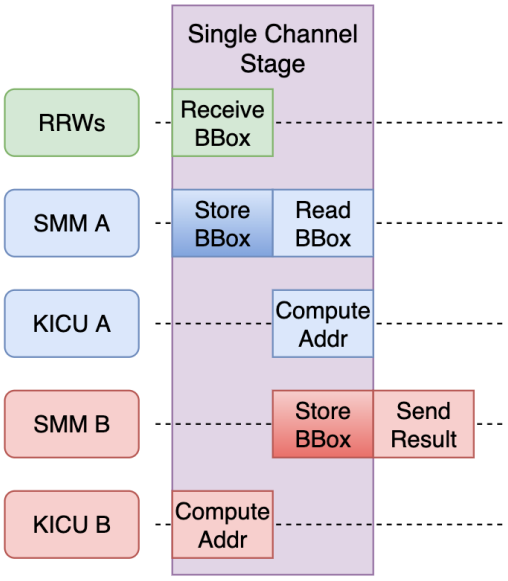


FIGURE 4.7: ShapoolNMS execution steps of partial PSRR-MNMS divided into its main phases.

The process of partial PSRR-MNMS is simpler, which is shown in Fig. 4.7 and the steps are:

- The spatial location of N boxes are written into SMM A via the addresses computed by MKICU B without Relationship Recovery;
- When all boxes are fed, N cells in SMM A are read simultaneously and MKICU A computes the address for SMM B;
- When SMM A read finishes, N cells in SMM B are read out in parallel as outputs.

4.4.2 Relationship Recovery Ways (RRWs)

As Fig. 4.8 illustrates, this module contains N RRWs, where each RRW could process one box and compute its *channel*, x and y in the confidential score map per clock cycle, *i.e.*, Spatial Recovery and Channel Recovery in the algorithm.

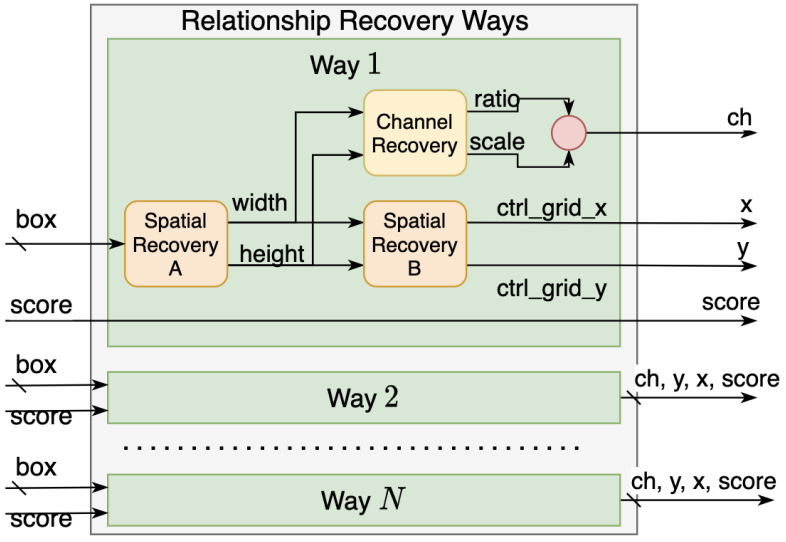


FIGURE 4.8: The structure of Relationship Recovery Ways.

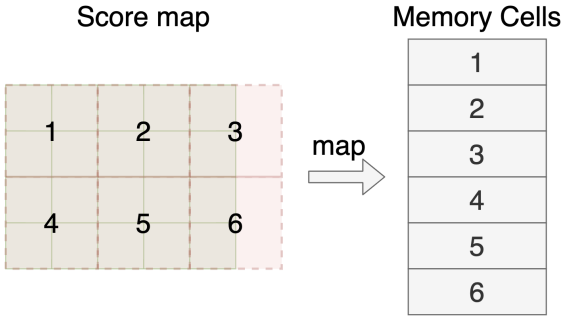


FIGURE 4.9: The MaxPool operation in ShapoolNMS is done by mapping the score map cells with the same ID into the physical memory cells, *e.g.*, the four score map cells in MaxPool kernel 1 are mapped into memory cell 1.

4.4.3 MaxPool Kernel Index Compute Unit (MKICUs)

Fig. 4.9 illustrates the MaxPool operations in ShapoolNMS which are done by mapping the score map cells within the same MaxPool kernel into the same physical memory cell, and only the one with the highest score in one MaxPool kernel is kept in the memory cell.

MKICUs are used to compute the kernel index, *i.e.*, the physical memory address of the bounding boxes. There are two MKICUs in ShapoolNMS for Pyramid MaxpoolNMS and Shifted MaxpoolNMS respectively.

As Fig. 4.10 shows, each MKICU contains four submodules for four PS MNMS steps as introduced in Section 4.3.2, namely Single-Channel MKICU, Cross-Ratio MKICU, Cross-Scale MKICU, and Cross-All MKICU.

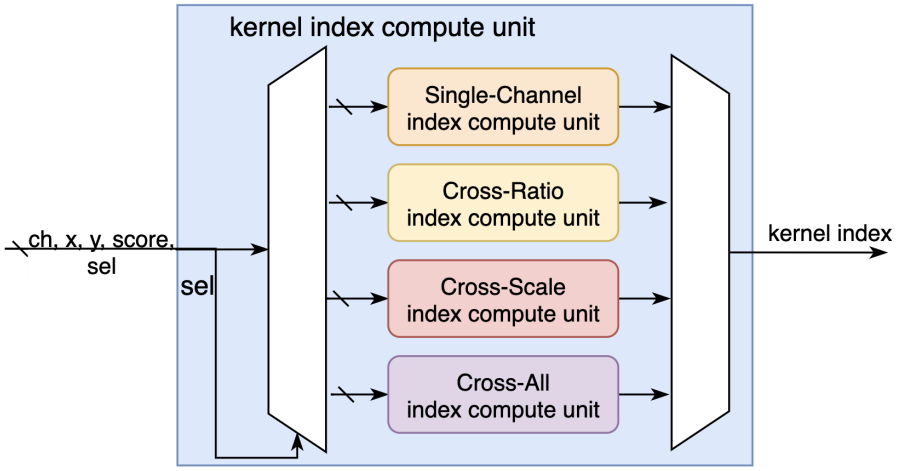


FIGURE 4.10: The structure of MaxPool Kernel Index Compute Unit

FIGURE 4.11: The format of the address, *i.e.*, the kernel index computed by MKICU, *e.g.*, there are 8 memory banks and the image resolution is up to FHD.

4.4.4 XBar

XBar, or crossbar switch, is used to send *score*, *channel*, x, and y to the corresponding memory bank based on the address computed by MKICU. Its ratio is $N \times M$, *i.e.*, its input ports number equals the number of ways (*i.e.*, N in Fig. 4.4) and the output ports number equals the number of memory banks in Score Map Memories (*i.e.*, M in Fig. 4.12).

When multiple bounding boxes target the same memory bank in one cycle, conflict occurs. The address from the FIFOs is specially designed to reduce the conflicts, *e.g.*, the 3 least significant bits of the address are used as bank selection for $M = 8$, which is shown in Fig. 4.11. In this case, the conflicts occur when more than one anchor box has the same at least three significant bits. The XBar also acts as an Arbiter with round-robin scheduling to resolve conflicts.

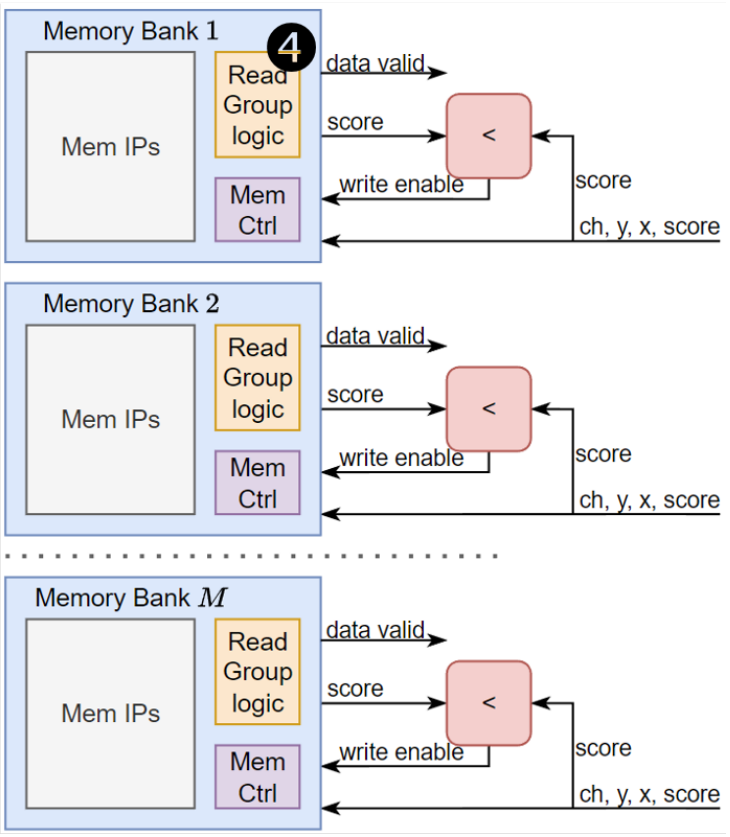


FIGURE 4.12: The structure of Score Map Memory for M memory banks. ❹ One memory bank in the SMM with 2 and 4 memory cells in one read group.

4.4.5 Score Map Memory (SMMs)

4.4.5.1 SMM Write

There are two SMMs: SMM A is used to store the *channel*, x, y, and *score* of the bounding boxes after Pyramid MaxpoolNMS while SMM B stores that of Shifted MNMS. As Fig. 4.12 illustrated, the score of a new box will be compared with the existing one in the memory bank, and it will not be updated into the memory cell unless the new score is greater than the old one. As the memory is read and written at the same time, a dual-port SRAM is required.

4.4.5.2 SMM Read

As described in Section 4.4.1, the data will be read from one SMM to another at each stage. However, less than 1% of the memory cells store valid information as

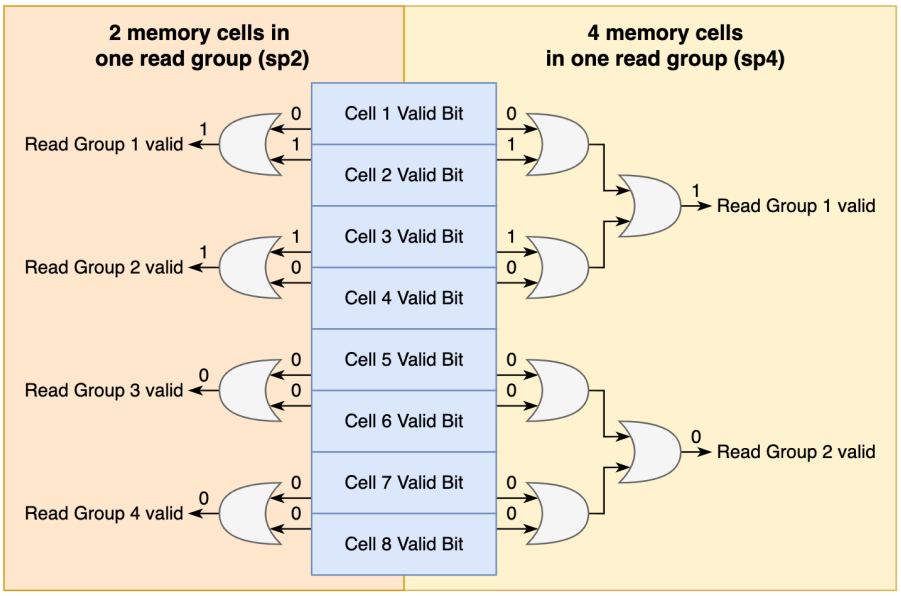


FIGURE 4.13: The structure of one memory bank in the SMM with 2 and 4 memory cells in one read group.

the score map is highly sparse. Extra valid bits and OR-gate trees are introduced to skip those invalid memory cells when reading data hence greatly further boosting the NMS.

In this structure, multiple memory cells are grouped into one Read Group by OR-reducing their valid bits. The outputs of OR-gate trees are checked sequentially. If the Read Group is valid, all memory cells in this Read Group will be read sequentially. Otherwise, the next Read Group will be checked. Fig. 4.13 draws two examples of this structure. The left side illustrates the case where the valid bits of two neighbor memory cells are grouped into one Read Group. When the Read Group bit is valid, then the two memory cells in this group will be read sequentially. In Fig. 4.13, memory cells in the Read Groups 1 and 2, *i.e.*, memory cells 1 to 4, will be read and the Read Groups 3 and 4 will only be checked instead of reading memory cells 5 to 8 one by one. Similarly, four neighbor memory cells are grouped into one Read Group on the right side of Fig. 4.13.

4.5 Design Space Exploration

In this section, we evaluate the performance, power, area, and accuracy of Shapool-NMS under different configurations and parameters. We also provide analysis and

Table 4.1: Experimental setup

Design Tools	
HDL	Chisel
Verilog simulator	Verilator
Logic synthesis	Cadence Genus
Floorplan	Cadence Innovus
Design Parameters	
Parallelism	1/2/4/8
# Mem Banks	1/2/4/8
# Grouped Mem Cells	1/2/4/8/16
Bit width	8/16/20/32 bit
Image Resolution	480p/720p/1080p/2k/4k
Technology Node	Foundry 28 nm FD-SOI
Supply voltage	0.8 V

discussion on the results and their implications.

4.5.1 Experimental Setup

The execution times of ShapoolNMS in different configurations are simulated by Verilator simulator [233] with the Verilog code generated by Chisel [201] and collected in Python scripts. The generated Verilog code is synthesized by Cadence Genus with Foundry 28 nm FD-SOI technology node for both area and power estimation and estimated the memory area and power with CACTI 7.0 [234]. The floorplan and place and route are done by Cadence Innovus. The experimental setup is summarised in Table 4.1.

4.5.2 Design Metrics and Considerations

We examined the following factors that affect the performance, power efficiency, area, and accuracy of ShapoolNMS:

4.5.2.1 Parallelism: Number of Ways and MaxPool Kernel Index Compute Units and Memory Banks

The number of ways and memory banks dictate ShapoolNMS's parallelism. While increased ways reduce execution time, and more memory banks mitigate conflicts

to increase memory access efficiency, both up the area and power costs.

4.5.2.2 Memory Read Efficiency: Grouped Memory Cells

More grouped memory cells elevate memory read efficiency by skipping invalid cells, thereby curtailing read cycles, but also increasing the area and power consumption.

4.5.2.3 Precision: Score and Box Position Bit Width

More bits can represent more values and improve the accuracy, but also increase the area and power consumption.

4.5.2.4 Execution Time: Number of Detection Boxes

This metric impacts both the execution time and the score map's sparsity.

4.5.2.5 Image Resolution

Higher image resolution can improve the detection quality, but also increase the number of boxes and the execution time.

There are trade-offs between these design metrics and the design objectives. We explore various configurations and parameters to find the optimal balance for ShapoolNMS.

4.5.3 Number of Ways and MaxPool Kernel Index Compute Units

Illustrated in Fig. 4.14, doubling the number of parallel computing units (Relationship Recovery Way, MaxPool Kernel Index Compute Units) reduces half of the PS MNMS cycles and also reduces around 30% RR time as conflicts are increasing the processing time when multiple ways target on the same memory bank. The synthesis results show that the 8-way ShapoolNMS is 7.47× faster than the 1-way ShapoolNMS with a 2.69× area overhead.

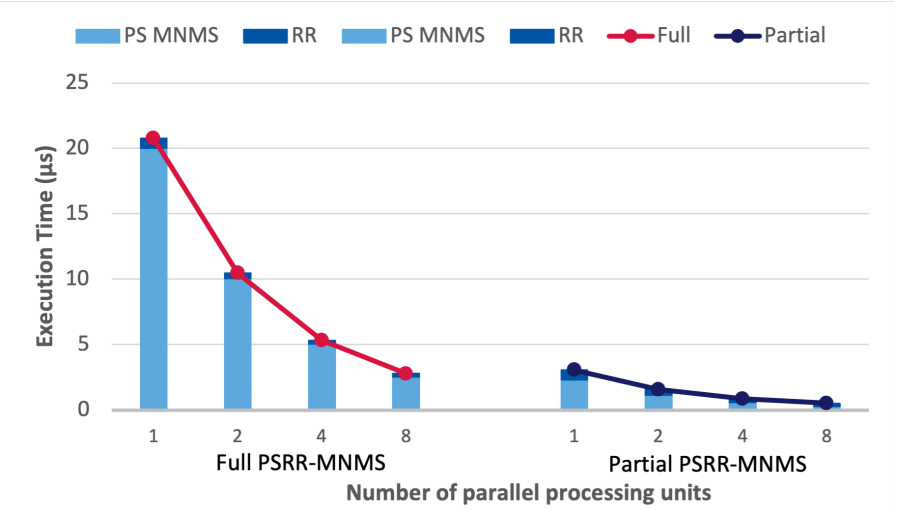


FIGURE 4.14: The execution time of 8-bank ShapoolNMS with various parallel computing units to process 1000 bounding boxes in full PSRR-MNMS (left) and partial PSRR-MNMS (right) at 400 MHz. The data transferring time between GPUs or deep learning accelerators and ShapoolNMS is hidden by the Relationship Recovery process time.

TABLE 4.2: The execution cycles, area, and power of an 8-bit 8-bank ShapoolNMS with a various number of parallel computing units to process full and partial PSRR-MNMS.

way	Full PSRR-MNMS		Partial PSRR-MNMS		Area (mm^2)	Area Overhead	Power (mW)
	Clock Cycle	*Speedup*	*Clock Cycle*	*Speedup*			
1	8,303	1×	1,223	1×	0.0346	1×	3.669
2	4,184	1.98×	631	1.94×	0.0432	1.25×	6.503
4	2,130	3.90×	344	3.55×	0.0597	1.73×	12.631
8	1,112	7.47×	209	5.86×	0.0930	2.69×	23.902

4.5.4 Number of Memory Banks in SMMs

As Fig. 4.15 shows, more memory banks will slightly increase the execution time of PS MNMS when the number of memory banks is larger than that of the ways. The PS MNMS execution time of a 1-way 8-bank ShapoolNMS increased 0.9% than that of a 1-way 1-bank ShapoolNMS due to a slightly larger total number of memory cells to be read and few conflicts happened. However, if the number of memory banks is less than the number of ways, there are more conflicts, hence increasing the execution time. The synthesis results indicate that there is a 12% area overhead to have 8 memory banks compared to only one memory bank, since there are more memory cells in total, which is summarized in Table 4.3.

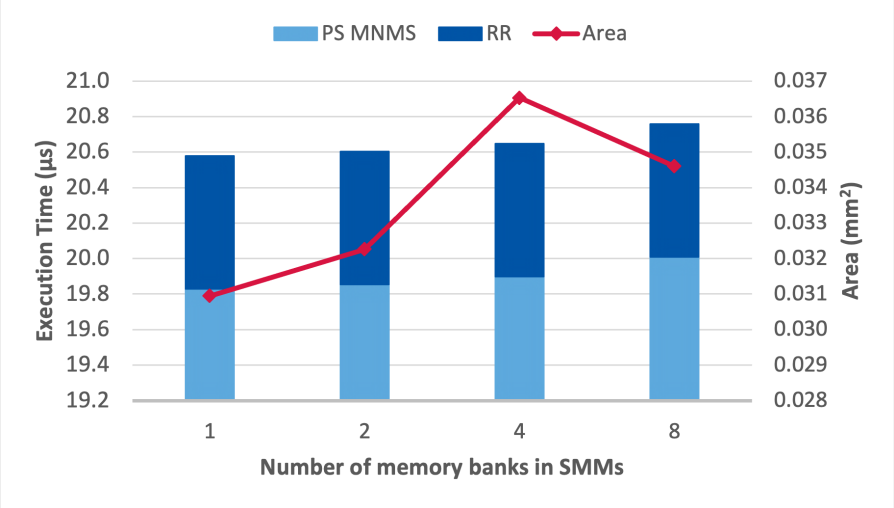


FIGURE 4.15: The execution time and area of an 8-bit 1-way ShapoolNMS with various numbers of memory banks in the score map memories in full PSRR-MNMS at 400 MHz.

TABLE 4.3: The execution cycles, area, and power of an 8-bit 1-way ShapoolNMS with a variable number of memory banks to process full PSRR-MNMS.

Mem Banks	Clock Cycle	Speedup	Area (mm^2)	Area Overhead	Power (mW)
1	8,231	1×	0.0309	1×	3.432
2	8,242	1.00×	0.0323	1.04×	3.505
4	8,259	1.00×	0.0365	1.18×	3.559
8	8,303	0.99×	0.0346	1.12×	3.669

4.5.5 Number of Memory Cells in One Read Group

Fig. 4.16 draws the execution time and area of an 8-bit 1-way 1-bank ShapoolNMS with 1 to 16 memory cell(s) in the read group to skip the invalid memory cells. By grouping 2 memory cells into one read group, the execution time was reduced by 44.3% compared to reading all the data from the memory. The execution time is further reduced to 24.2% and 2% when the number of memory cells in one read group is reduced from 2 to 4 and from 4 to 8 respectively. However, the execution time increased by 14.42% when the number of memory cells doubled from 8 to 16.

The synthesis results show that there is only 5% area overhead to combine 8 memory cells into one read group while obtaining a 2.41× speedup over reading the memory cells one by one, which is summarized in Table 4.4.

In our other experiments, *e.g.*, 8-bit 8-way 8-bank ShapoolNMS, the optimal number of memory cells is 4 for the best computational performance. In summary,

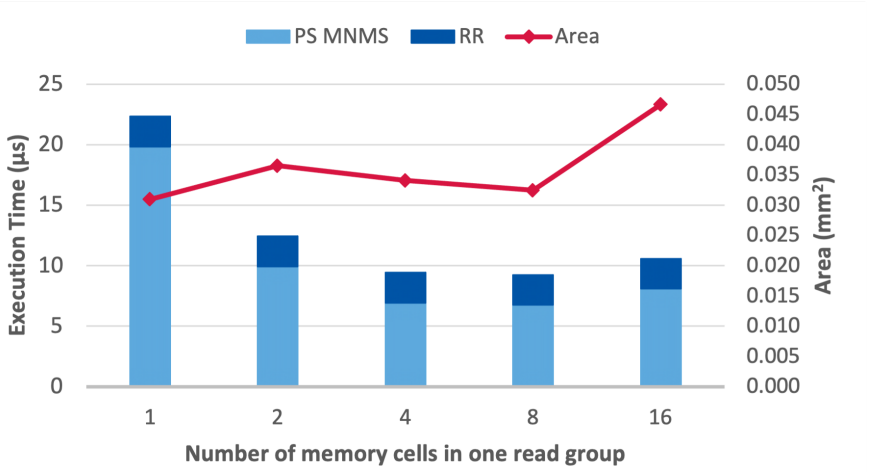


FIGURE 4.16: The execution time and area of an 8-bit 1-way 1-bank ShapoolNMS with various numbers of memory cells in one read group in full PSRR-MNMS at 400 MHz.

TABLE 4.4: The execution cycles, area, and power of an 8-bit 1-way 1-bank ShapoolNMS with a various number of memory cells in one read group to process full PSRR-MNMS.

Read Group	Clock Cycle	Speedup	Area (mm^2)	Area Overhead	Power (mW)
1	8,931	1×	0.0309	1×	3.432
2	4,976	1.79×	0.0365	1.18×	5.675
4	3,772	2.37×	0.0340	1.10×	4.844
8	3,702	2.41×	0.0325	1.05×	4.252
16	4,236	2.11×	0.0466	1.51×	8.030

there is an optimal value of the number of memory cells to achieve the best computational performance due to the sparsity in different parallelisms.

4.5.6 Bit Width of Scores and Box Positions

As both scores and box positions are represented in a decimal number, their precision will affect the mAP, which represents the detection accuracy. The data type of scores and box spatial positions is fixed-point in our hardware.

Table 4.5 summarizes the mAP and area of the 1-way 1-bank ShapoolNMS with varied scores and spatial location bit width, which is also drawn in Fig. 4.17. As summarized, the higher the precision of the value is, the higher the mAP it can achieve. The mAP of the 20-bit ShapoolNMS and 32-bit ShapoolNMS is nearly the same in ResNet-50.

TABLE 4.5: The mAP, area, and power of a 1-way 1-bank ShapoolNMS with varied bit widths on ResNet50 and ResNet152 backbone.

Bit Width	ResNet50		ResNet152		Area (mm^2)	Area Overhead	Power (mW)
	mAP	*Improve*	*mAP*	*Improve*			
8	77.1072	0.0000	78.4019	0.0000	0.0309	1×	3.432
16	77.2967	0.1895	78.5924	0.1905	0.0386	1.25×	3.534
20	77.3316	0.2244	78.7159	0.3140	0.0421	1.36×	3.577
32	77.3331	0.2259	78.7344	0.3325	0.0531	1.72×	3.711

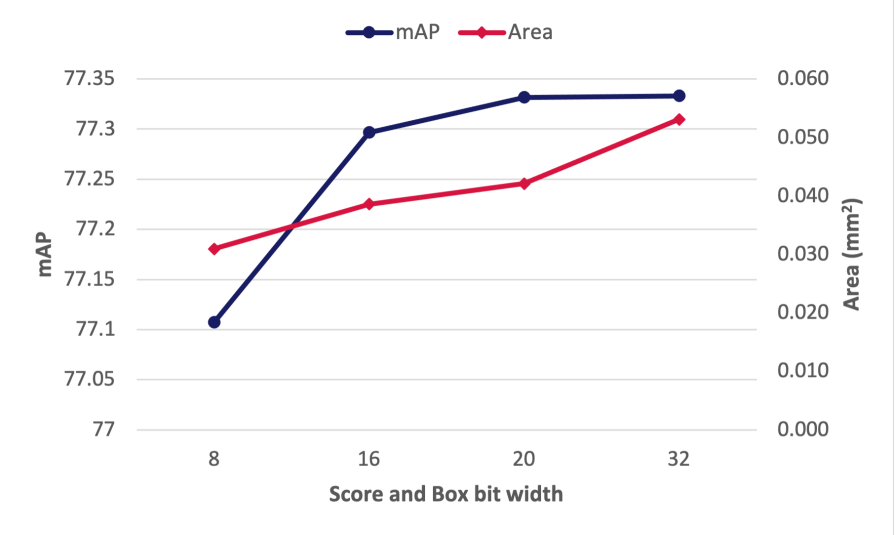


FIGURE 4.17: The mAP and area of a 1-way 1-bank ShapoolNMS with varied bit widths in ResNet-50 backbone.

4.5.7 Number of Detection Boxes

The timing breakdown of a 1-way 1-bank ShapoolNMS to cluster the various numbers of bounding boxes in full PSRR-MNMS is shown in Fig. 4.18. For partial PSRR-MNMS, the execution time of PS MNMS is around 1,223 clock cycles in this case. It is also a linear trend for the execution time of a 1-way 1-bank ShapoolNMS with 2 memory cells in one read group to process a varied number of bounding boxes as shown in Fig. 4.19. We found that the execution time of PS MNMS is nearly constant without read groups in SMMs, while it is a linear trend with read groups in SMMs. The execution time of RR is always the linear trend.

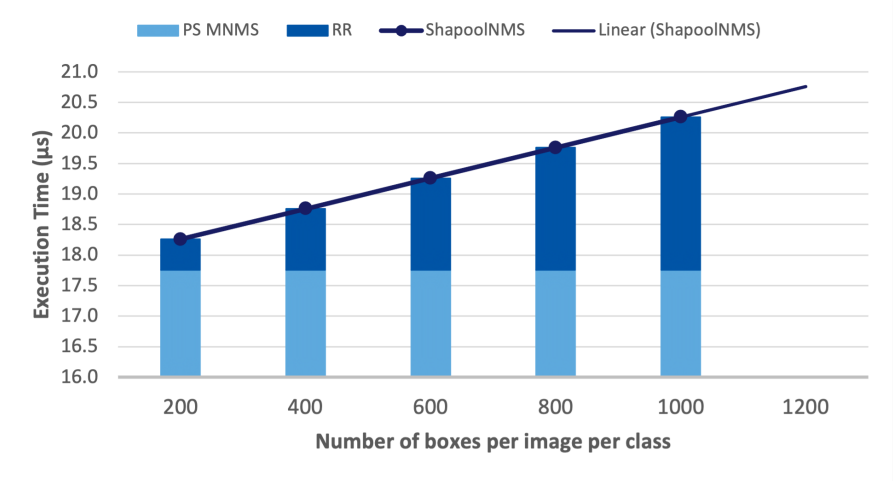


FIGURE 4.18: The detailed execution time for a 1-way 1-bank ShapoolNMS to process various boxes in full PSRR-MNMS at 400 MHz.

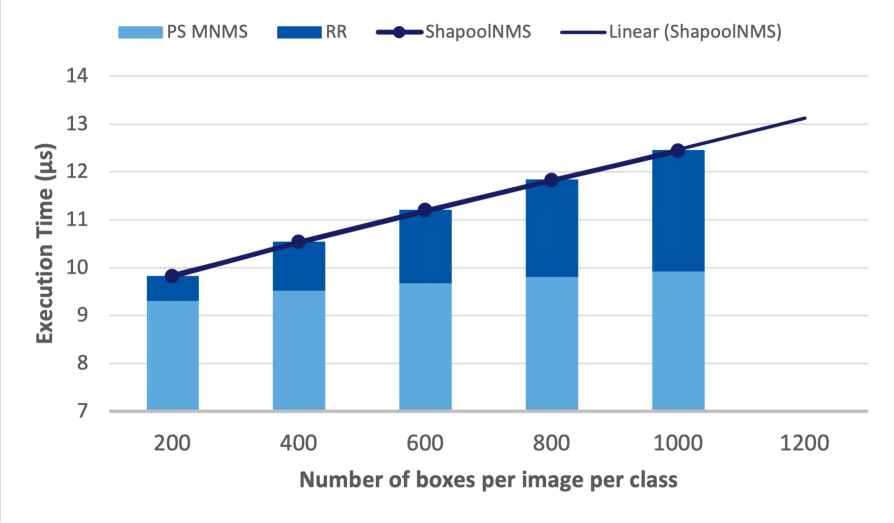


FIGURE 4.19: The detailed execution time for a 1-way 1-bank ShapoolNMS with 2 memory cells in one read group to process various boxes in full PSRR-MNMS at 400 MHz.

4.5.8 Supporting Image Resolution

Higher image resolution is essential for better detection accuracy. In ShapoolNMS, higher input resolution means that the score map is larger, hence requiring a larger Score Map Memory capacity and a longer time to read all the data from the memory when doing PS MNMS.

Table 4.6 lists the execution time without jumping invalid memory cells of PS MNMS, the capacity requirements, and the corresponding chip area in different image resolutions. This is also drawn in Fig. 4.20.

TABLE 4.6: The execution time of PS MNMS, the capacity of SMM, and the area and power in varied image resolutions that is supported by an 8-bit 1-way 1-bank ShapoolNMS.

Image Resolution	PS MNMS Cycles	Memory Size (KB)	Area (mm^2)	Power (mW)
640×480 (480pVGA)	5,080	6.12	0.0309	3.432
720×480 (480p)	5,819	7.03	0.0337	3.433
1280×720 (720p)	14,851	19.80	0.0722	4.072
1920×1080 (1080p)	32,276	44.94	0.1544	4.319
2048×1080 (2K)	34,440	47.96	0.1630	4.320
3840×2160 (4K)	126,144	190.22	0.6623	5.260

[a] This cycle is near constant for any number of bounding boxes if the memory cell is read one by one. However, with the read group structure, a significant speedup can be spotted with a very high image resolution.
[b] The memory capacity of two Score Map Memory when the score is 8-bit.

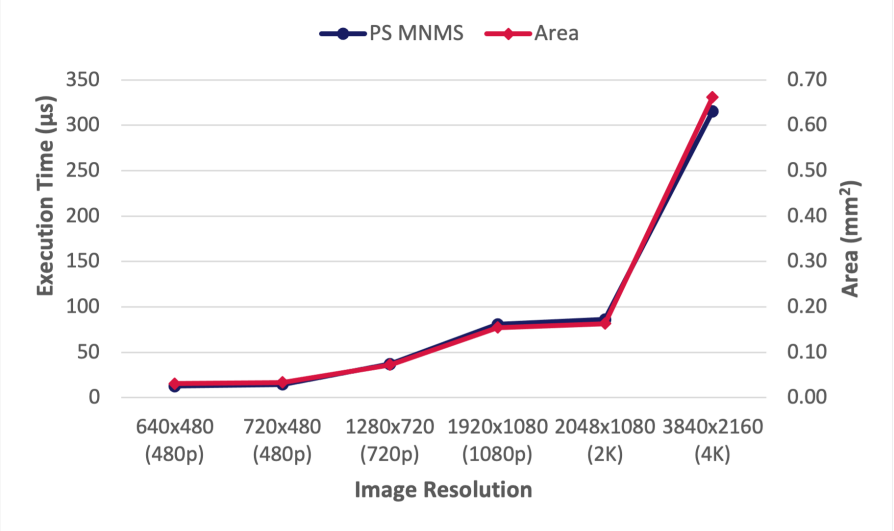


FIGURE 4.20: The execution time of PS MNMS, the capacity of SMM, and the area in varied image resolutions that is supported by an 8-bit 1-way 1-bank ShapoolNMS at 400 MHz. The data transferring time between GPUs or deep learning accelerators and ShapoolNMS is hidden by the Relationship Recovery process time.

4.5.9 DSE Summary

An optimal combination of the parameters is summarised as follows. The number of memory banks in SMMs should be equal to the number of ways and MaxPool Kernel Index Compute Units, otherwise the execution time increases. The number of memory cells in one read group should be 4 or 8 depending on the real-time application to obtain the highest speedup. The bit width of scores and box positions should be as large as possible for a higher accuracy. An optimal value for the bit width can be 20-bit, which can achieve a similar accuracy compared with 32-bit while having a smaller area.

TABLE 4.7: The mAP of ShapoolNMS, PSRR-MNMS, and GreedyNMS with Faster-RCNN frameworks on the PASCAL VOC dataset.

Detection Pipeline	**GreedyNMS**	**PSRR-MaxpoolNMS**	**ShapoolNMS**		
			Score and Box bit width		
			32	***16***	***8***
Faster-RCNN ResNet-50	78.40	77.59	77.33	77.30	77.11
Faster-RCNN ResNet-152	78.70	78.40	78.73	78.59	78.40

4.6 Evaluation

The speedup, accuracy, and scalability of ShapoolNMS with common NMS algorithms, as well as state-of-the-art designs, are quantified in this section. PSRR-MNMS and GreedyNMS software implementations are measured on CPU-based execution. The NMS processing time of ShapoolNMS and its computation results are simulated by the Verilator simulator with the Verilog code generated by Chisel.

4.6.1 Detection Accuracy

Table 4.7 summarizes the mAP of ShapoolNMS with different scores and spatial location bit width. Intuitively, better precision yields higher mAP. Our results also indicate that the mAP of PSRR-MNMS is 0.81% less compared with GreedyNMS in the ResNet-50 backbone, and is 0.3% less in ResNet-152. Our 32-bit ShapoolNMS is 1.07% less than GreedyNMS in ResNet-50 and is 0.03% higher than GreedyNMS in ResNet-152. This is acceptable in ResNet-50 and caused mainly by the difference in data format and precision—IEEE 754 in the software implementations and 32-bit fixed-point in ShapoolNMS.

We also evaluate the mAP of ShapoolNMS on YOLOv8 models [25]. The experiment shows that the mAP of YOLOv8 models computed by ShapoolNMS is 35.9 to 52.6, which is 1.3 to 1.7 less than the one computed by GreedyNMS in float point (37.3 to 53.9). The accuracy drop is introduced by quantization.

4.6.2 Execution Time

Fig. 4.21 illustrates the execution cycles of a typical ShapoolNMS whose execution time is a linear trend with the number of bounding boxes and the 64-parallel compute components GreedyNMS hardware utilizing Bubble and Bitonic sorting

TABLE 4.8: The mAP of ShapoolNMS, PSRR-MNMS, and GreedyNMS on COCO value2017 dataset with YOLOv8 frameworks. mAP^{val} values are for single-model single-scale. The mAP is evaluated with the α of PSRR-MaxpoolNMS and ShapoolNMS equal to 0.25.

YOLOv8 Model	GreedyNMS	PSRR-MaxpoolNMS	ShapoolNMS		
			Score and Box bit width		
			32	16	8
YOLOv8n	37.3	37.0	35.9	35.9	35.9
YOLOv8s	44.9	44.5	43.2	43.2	43.2
YOLOv8m	50.2	49.7	48.8	48.8	48.8
YOLOv8l	52.9	52.3	51.4	51.4	51.4
YOLOv8x	53.9	53.4	52.6	52.6	52.6

TABLE 4.9: The speedup of clustering 1000 and 8000 anchor boxes in different methods compared with GreedyNMS and the chip area of the ASIC works.

Method		NMS Execution Time (μs)		Speedup		Area
		n=1,000	***n=8,000***	***n=1,000***	***n=8,000***	
GreedyNMS[a]		35,000	512,000	1×	1×	–
PSRR-MaxpoolNMS [62][a]		18,000	89,000	1.94×	5.75×	–
GPU-NMS V2 [231][b]		324	–	108.02×	–	–
Shi* et al. *[210][c]		12.79	102.32	2,736.51×	5,003.91×	0.080
ShapoolNMS[d]	minimal area	9.26	22.25	3,781.67×	23,011.24×	0.032
	best performance	1.70	5.70	20,629.19×	89,824.56×	0.096

[a] The data is reported in [62].
[b] Executed on GeForce GTX 1060, whose operating frequency is 1.70 GHz. The execution time of $n = 8,000$ is not reported. The benchmark is the Oscars dataset.
[c] The operation frequency is 400 MHz. The execution time of $n = 8,000$ is projected from the reported execution time of $n = 1,000$ and $n = 10,000$ in a linear trend. The benchmark is the WIDER FACE dataset.
[d] The operation frequency is 400 MHz. The ShapoolNMS with the minimal area is the 8-bit 1-way 1-bank ShapoolNMS with 8 memory cells in one read group, while the ShapoolNMS with the best performance is the 8-bit 8-way 8-bank ShapoolNMS with 4 memory banks in one read group. The execution time of $n = 1,000$ is simulated to process full PSRR-MNMS in the VGG16 network on the PASCAL VOC benchmarking dataset while that of $n = 8,000$ is projected in a linear trend.

algorithms (sorting is a key component in GreedyNMS). ShapoolNMS increasingly outperforms GreedyNMS hardware, GPU-NMS V2 [231], and Shi *et al.* [210] with an increased number of boxes. This is due to the time complexity of GPU-NMS and Shi *et al.* is at least $O(nm)$ [210], while that of ShapoolNMS is $O(n)$, as summarized in Table 4.9.

ShapoolNMS achieves 120.95× to 669.48× speedup compared to executing the full and partial NMS operations in the Faster-RCNN backbone on the PASCAL VOC benchmarking dataset with GreedyNMS hardware. The execution time of the system contains convolution operations and NMS. The convolution is accelerated by A100 GPU, see Fig. 1.4. Obtained results from Fig. 4.23 show that ShapoolNMS can greatly alleviate the performance bottleneck caused by de facto GreedyNMS whose execution time is shown in Fig. 1.4.

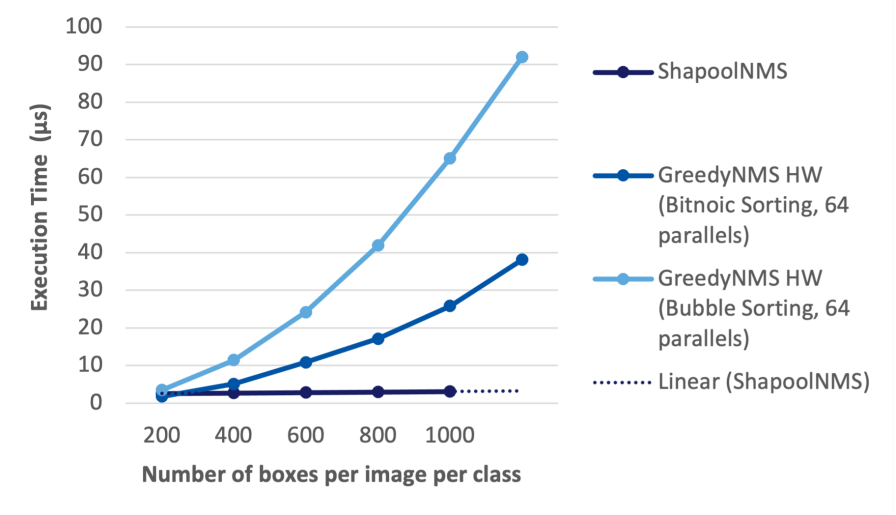


FIGURE 4.21: The execution cycles for a typical ShapoolNMS whose computation time is a linear trend with the number of bounding boxes and a 64-parallel compute components GreedyNMS hardware with Bubble sorting algorithm and Bitonic sorting algorithm to process a varied number of bounding boxes. The data transferring time is not included for a fair comparison with GreedyNMS hardware implementations.

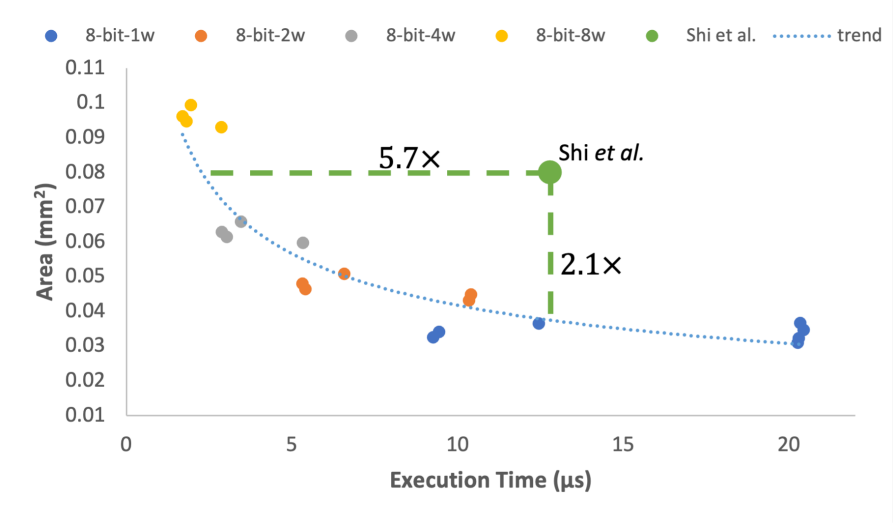


FIGURE 4.22: The execution time vs chip area of the ShapoolNMS with different configurations and the state-of-the-art work proposed by Shi *et al.* [210] (in green) at 400 MHz.

4.6.3 Hardware Utilization

The ShapoolNMS with minimal area and best performance which supports up to 480p VGA image resolution are synthesized by Cadence Genus targeted on Foundry 28 nm FD-SOI manufacturing process, and the Place and Route (PnR) are done by

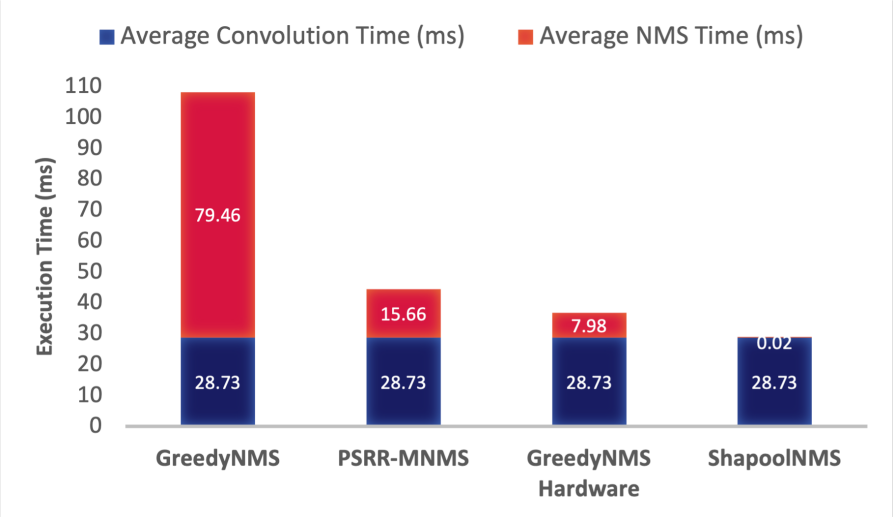


FIGURE 4.23: The timing breakdown of object detection inference time executed on A100 and NMS time of de-facto GreedyNMS, state-of-the-art PSRR-MNMS, 64-parallel compute components GreedyNMS hardware based on Bitnoic sorting algorithm, and 1-way 1-bank ShapoolNMS with 8 memory cells in one read group operated at 400 MHz on PASCAL VOC benchmarking dataset. The data transferring time between GPUs and ShapoolNMS is hidden by the Relationship Recovery process time.

Cadence Innovus. The area of the memory is estimated by CACTI 7.0 [234]. The area of the ShapoolNMS with minimal area after PnR is only 0.031 mm^2, which is only 38.75% of the chip area proposed by Shi *et al.* [210]. The floorplans are shown in Fig. 4.24a and Fig. 4.25a respectively where the black boxes are the SRAMs in the SMMs.

The area breakdowns are illustrated in Fig. 4.26 and Fig. 4.27 respectively. As it shows, the Score Map Memories occupy more than 74% area in the ShapoolNMS with a minimal area while occupying 31.74% area in the ShapoolNMS with the best performance. MaxPool Kernel Index Compute Units in the ShapoolNMS with the best performance occupy the maximum percentage of the chip area, which is 55%.

4.6.4 Power Consumption

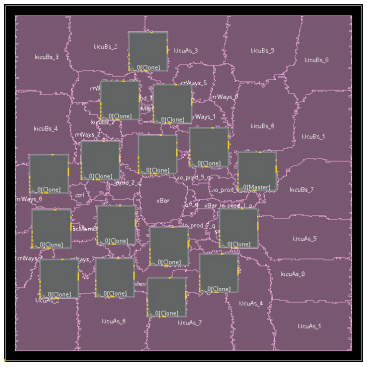
The power of the logic in ShapoolNMS with minimal area and best performance are reported by Cadence Genus while those of the memory are computed by CACTI 7.0 [234]. ShapoolNMS with the minimal area achieves 2.76× and 30.55× better

(A)

Technology Node	Foundry 28 nm FD-SOI
Supply Voltage	0.80 V
Area	0.031 mm^2
Frequency	400 MHz
# way	1
# mem bank	1
Supporting Image Resolution	640×480 (480p VGA)
Power	4.252 mW

(B)

FIGURE 4.24: (a) The layout photograph and (b) chip specification of the ShapoolNMS with minimal area.



(A)

Technology Node	Foundry 28nm FD-SOI
Supply Voltage	0.80 V
Area	0.096 mm^2
Frequency	400 MHz
# Way	8
# Mem Bank	8
Supporting Image Resolution	640×480 (480p VGA)
Power	28.276 mW

(B)

FIGURE 4.25: (a) The layout photograph and (b) chip specification of the ShapoolNMS with the best performance.

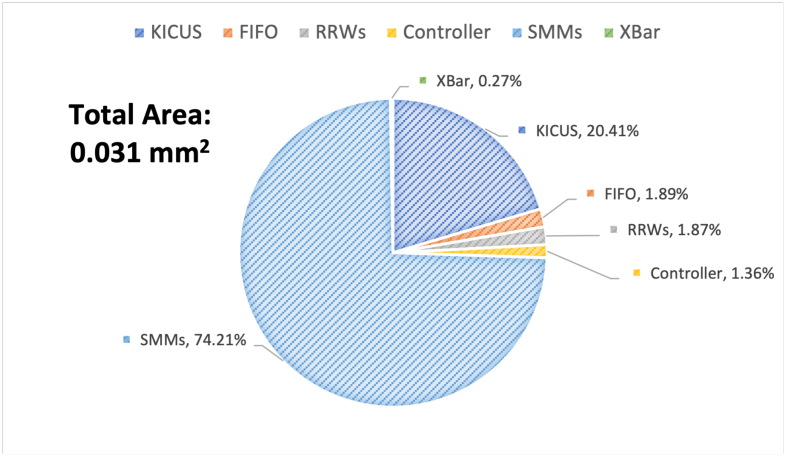


FIGURE 4.26: The area breakdown of an 8-bit 1-way 1-bank ShapoolNMS with 8 memory cells in one read group that supports VGA image resolution.

energy-delay product (EDP) for detecting 1,000 and 8,000 bonding boxes compared with the work proposed by Shi *et al.* [210].

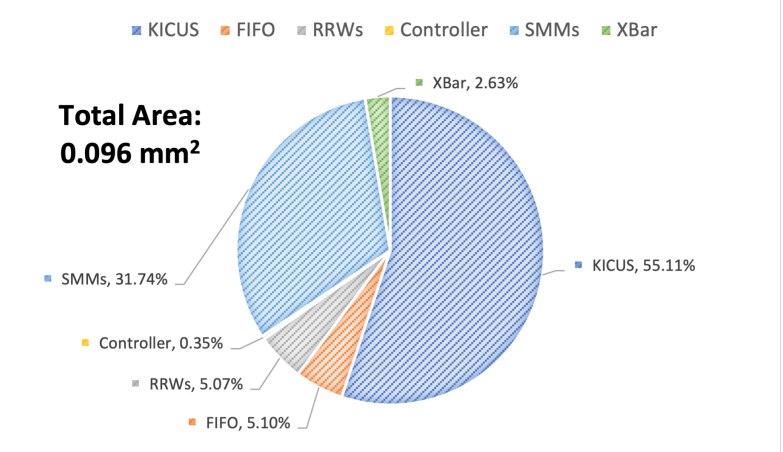


FIGURE 4.27: The area breakdown of an 8-bit 8-way 8-bank ShapoolNMS with 4 memory cells in one read group that supports VGA image resolution.

4.7 Chapter Summary

In this chapter, we have presented a novel highly scalable and parallel NMS hardware architecture — ShapoolNMS — with a time complexity of $O(n)$, which clusters tens of thousands of bounding boxes in very low latency, *e.g.*, the 8-way, 8-bank ShapoolNMS with 4 memory cells in one read group could cluster 8,000 anchors in around 5.70 μs in full PSRR-MaxpoolNMS, *i.e.*, the NMS in one-stage frameworks or the last stage of two-stage frameworks and 4.87 μs in partial PSRR-MaxpoolNMS, *i.e.*, the NMS at the first stage of two-stage frameworks.

ShapoolNMS relies on the parallel and scaleable PSRR-MaxpoolNMS algorithm and is able to support multiple parallel compute units, *i.e.*, Relationship Recovery Ways and MaxPool Kernel Index Compute Units. The 7.47× speedup is obtained when the number of parallel compute units increases from 1 to 8 to cluster 1,000 anchor boxes with a 3.68× area overhead.

The performance of ShapoolNMS can be further boosted by grouping a certain number of memory cells to jump invalid data in the score map memories. The execution time of the ShapoolNMS with 8 memory cells in one read group is reduced by 2.41× to cluster 1,000 anchor boxes with only 1.07× area overhead.

Empowered by both low computation complexity and hardware/software co-optimization, the obtained performance results show that ShapoolNMS achieves a speedup of 1,204.35× to 6,666.14× than GreedyNMS software implementations in the PASCAL VOC benchmarking dataset in the Faster-RCNN pipeline. Compared to

GPU-NMS V2 [231], ShapoolNMS achieves 76.44× to 472.20× speedup to cluster 2,895 anchor boxes. ShapoolNMS is 3.00× to 19.71× faster than the chip proposed by Shi *et al.* [210] to suppress 10,000 anchor boxes. Experiment results indicate that the more bounding boxes are to be suppressed, the more our proposal outperforms GreedyNMS hardware implementations, GPU-NMS V2, and Shi *et al.* ShapoolNMS with the minimal area achieves 2.76× and 30.55× better energy-delay product (EDP) for detecting 1,000 and 8,000 bonding boxes compared with the work proposed by Shi *et al.*

Physical design results show that the chip area of the ShapoolNMS is up to 2.58× less than that of the work proposed by Shi *et al.* [210]. In contrast, the chip area of the ShapoolNMS with the best performance is 0.012% and 0.029% than that of NVIDIA A100 Tensor Core GPU and Google Tensor Processing Unit (TPU) v1 respectively.

ShapoolNMS is suitable for both one-stage and two-stage detection frameworks with any number of anchor boxes and supports very high input resolutions such as Full HD size, which is the essential input size for better detection accuracy. Besides, ShapoolNMS also supports both global NMS and commonly adopted per-class NMS.

Chapter 5

Chiplet-based Scalable CNN Accelerator System

5.1 Introduction

Modern convolutional neural networks have achieved remarkable results in various computer vision tasks. However, as these networks become deeper, they often face the degradation problem, where adding more layers can lead to higher training and test errors.

To address this, He *et al*. introduced the ResNet architecture in 2016 [7]. As introduced in Section 2.1.4.1, there are four blocks in the ResNet, whose output dimensions, weight parameters, and multiply-accumulate operations (MACs) of ResNet are summarized in Table 5.1.

As illustrated in Fig. 5.1, the memory capability requirements of the feature maps in ResNet are up to 262.97 MB even if the memory is reused as much as possible. In ResNet Block 1 Unit 1, there are 135.47 MB INT32 outputs (before ReLU) after the third convolutional layer, which should be stored in the memory and waiting for another 135.47 MB INT32 outputs (before ReLU) of the shortcut convolutional layer to be added to them before being passed to the ReLU layer. The memory requirements of the feature maps decrease to only half of it in Block 2, and so on.

TABLE 5.1: The memory requirement of the INT8 outputs (after ReLU), INT8 weights, and number of MACs in ResNet with a 1080p image as input.

Block	Unit	Weight Layer	# Output	# Weights	# MACs
Block 0	U 0	Conv 1	5.98 MB	9 KB	4.6 G
Block 1	U0	Conv 1	7.97 MB	4 KB	0.5 G
		Conv 2	7.97 MB	36 KB	4.5 G
		Conv 3	31.88 MB	16 KB	2.0 G
		Shortcut	31.88 MB	16 KB	2.0 G
	Other	Conv 1	7.97 MB	16 KB	2.0 G
Block 2	U 0	Conv 1	3.98 MB	32 KB	1.0 G
		Conv 2	3.98 MB	144 KB	4.5 G
		Conv 3	15.93 MB	64 KB	2.0 G
		Shortcut	15.93 MB	128 KB	4.0 G
	Other	Conv 1	3.98 MB	64 KB	2.0 G
Block 3	U 0	Conv 1	1.99 MB	128 KB	1.0 G
		Conv 2	1.99 MB	576 KB	4.5 G
		Conv 3	7.97 MB	256 KB	2.0 G
		Shortcut	7.97 MB	512 KB	4.0 G
	Other	Conv 1	1.99 MB	256 KB	2.0 G
Block 4	U 0	Conv 1	1.00 MB	512 KB	1.0 G
		Conv 2	1.00 MB	2,304 KB	4.5 G
		Conv 3	3.98 MB	1,024 KB	2.0 G
		Shortcut	3.98 MB	2,048 KB	4.0 G
	Other	Conv 1	1.00 MB	1,024 KB	2.0 G

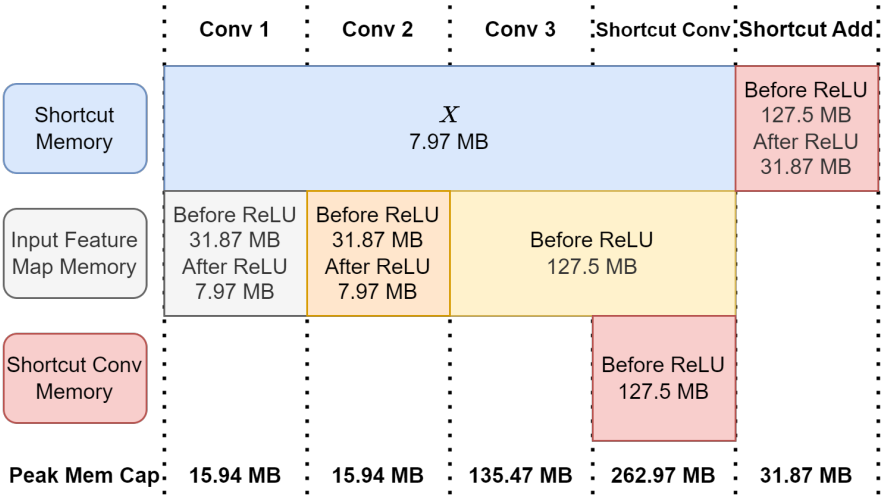


FIGURE 5.1: The memory capability requirements of the feature maps in ResNet Block 1 Unit 1 with a 1080p image as input. The feature map is INT32 and INT8 before and after the ReLU layer, respectively.

As for the weights, there are only up to 36 KB weights in Block 1 while there are up to 2,304 KB weights in Block 4. The decreasing and increasing trends of feature maps and the weights are shown in Fig. 5.2.

As illustrated, the capacity of both shortcut memory and input memory should be

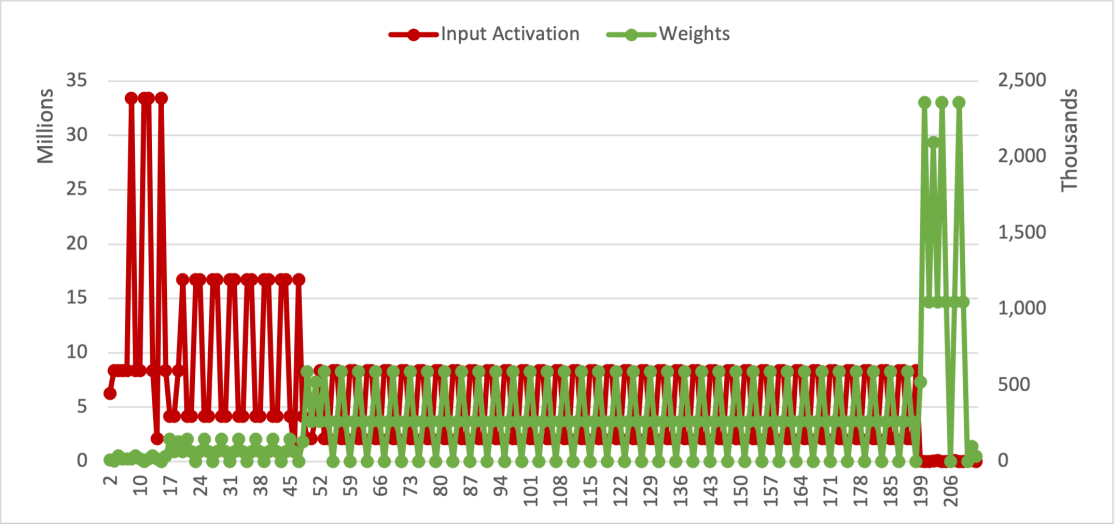


FIGURE 5.2: Number of weight and activation of different layers in ResNet.

at least 262.97 MB, while that of the weight memory should be at least 2,304 KB for one convolutional layer. Those memories would occupy a relatively large chip area.

This imbalance of memory requirements is caused by the dataflow of ResNet, where the feature maps are processed layer-by-layer. The feature maps are stored in the memory and waiting for the next layer to be processed, which leads to large memory requirements.

Besides, the support of deep CNNs with large input features poses a significant challenge to both the on-chip memory capacity and the computational capability of the CNN accelerator system. Within conventional 2D semiconductor processes, the above requirements are difficult to meet due to the limited chip area considering the fabrication yield and cost. Chiplet technology provides a solution to this problem by dividing a relatively large chip into multiple smaller chips, each of which is fabricated on a separate die and then stacked vertically together via TSVs to form a single Chiplet. For stronger computational capability, multiple Chiplets operate parallelly to accelerate the workload. To best utilize the chip-to-chip interconnections and reduce the communication overheads, dataflow should be fine-tuned to mask the communication overheads among Chiplets and increase their computation efficiency.

In this chapter, we propose a Chiplet-based scalable CNN accelerator system that

supports large input features. In the case study, the proposed CNN accelerator system achieves 68 FPS with full-HD resolution images as inputs in ResNet-152. With our novel Cross-Layer Optimization (CLO) dataflow, the memory requirements of the feature maps in ResNet are reduced by 84.85%, improving the area efficiency and power efficiency of the system. In block 3 and block 4, inner product NKM dataflow, *a.k.a.*, column stationary (CS) GEMM dataflow, is applied to minimize the capacity of the register files for weights, reducing 84.03% memory requirements. The support of both NKM-CS GEMM dataflow and MKN-RS GEMM dataflow in the proposed CNN accelerator system enables the system to support various CNN models within the limited on-chip memory and bandwidth.

This CNN accelerator system is implemented on a Chiplet-based architecture — CNN-DLA, where each Chiplet consists of three tiers — one logic layer and two memory layers — that are vertically connected by Through-Silicon Vias (TSVs).

Our contributions are summarized as follows:

- A highly efficient and end-to-end optimized CNN accelerator hardware system that supports up to full-HD resolution images with 68 FPS in ResNet-152.
- A memory-centric Cross-Layer Optimization (CLO) dataflow to reduce 84.85% memory requirements of the features in ResNet, improving the area efficiency and power efficiency of the system.
- A Chiplet-based architecture — CNN-DLA, that supports CLO dataflow, MKN-RS GEMM dataflow, and NKM-CS GEMM dataflow to reduce the memory requirements and both inner- and inter-bandwidths for the features and weights of convolution operations.

5.2 Related Work

5.2.1 Accelerating Networks: Direct Convolution

Most deep learning accelerators adopt the straightforward computation method, direct convolution, which accumulates the product of the inputs and corresponding weights in certain dataflow.

Some members in the DianNao family [30, 41, 235] compute the convolution directly. DianNao [30] exploits the locality properties of large-scale layers for high performance with only 3 mm^2 at 65 nm. The multiplications are done parallelly in the first stage of the Neural Function Unit (NFU) while the partial sums are accumulated via the adder tree in the second stage of NFU. The activation functions are implemented in the last stage of NFU. However, the memory wall becomes the bottleneck when computing classifier layers and convolutional layers with private kernels [41]. DaDianNao [41] mitigates the memory wall problem by processing the convolution with the data from nearby SRAM buffers and eDRAM banks.

NVIDIA Deep Learning Accelerator (NVDLA) [149] is an open-source hardware inference accelerator that provides direct convolution mode, where a wide multiply-accumulate (MAC) pipeline is implemented to parallel process the direct convolutional operations with memory bandwidth optimization features. NVDLA supports not only fully connected layers and convolutional layers but also deconvolution, activation, pooling, and local response normalization *etc*.

In Eyeriss [146], row stationary (RS) dataflow pattern is proposed after analyzing existing dataflow patterns for better throughput and energy efficiency, which is further optimized by exploring the tiling of the MAC operations spatially through channel group dimension across PEs in Eyeriss v2 [27].

Maeri [236] supports arbitrary dataflows by utilizing fine-grained reconfigurable tree-based interconnection network topologies to shape different sizes and numbers of virtual neurons.

5.2.2 Accelerating Networks: GEMM and Transform Optimizations

As Fig. 5.3 shows, a convolution layer can be converted into General Matrix Multiplication (GEMM) or General Matrix-Vector Multiplication (GEMV).

The input matrix is $A \times B$, the weight matrix is $A \times C$ and the output matrix is $A \times C$, where:

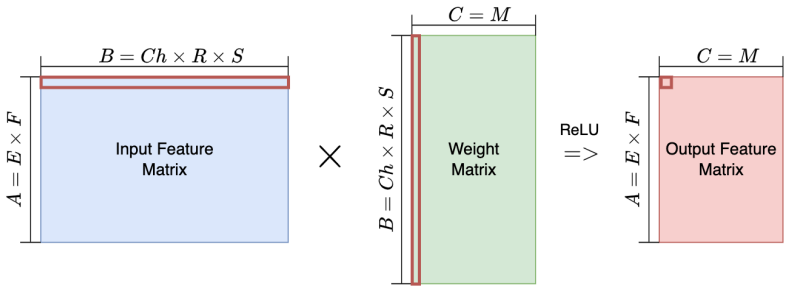


FIGURE 5.3: The transformation from normal convolution to GEMM.

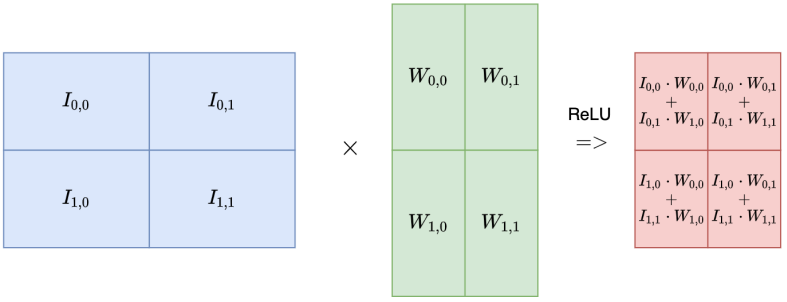


FIGURE 5.4: Large matrix is partitioned into smaller ones via block matrix multiplication to be mapped into limited hardware resources.

$$A = E \times F \tag{5.1}$$

$$B = C \times R \times S \tag{5.2}$$

$$C = M \tag{5.3}$$

where the C is the number of Input channels and Weight channels.

By transforming the convolution layer into GEMM or GEMV, some matrix multiplication optimization algorithms can be applied to further speed up the computation, *e.g.*, Classical Strassen algorithm [237], Winograd minimal filtering algorithm [238] and Fast Fourier Transform (FFT) algorithm [239].

To map a huge matrix multiplication in limited hardware resources, block matrix multiplication is applied to split a huge one into smaller blocks, which is shown in Fig. 5.4.

DaVinci [28, 240] is a scalable DNN architecture where a 3D computation unit is introduced for higher data reuse in GEMM dataflow.

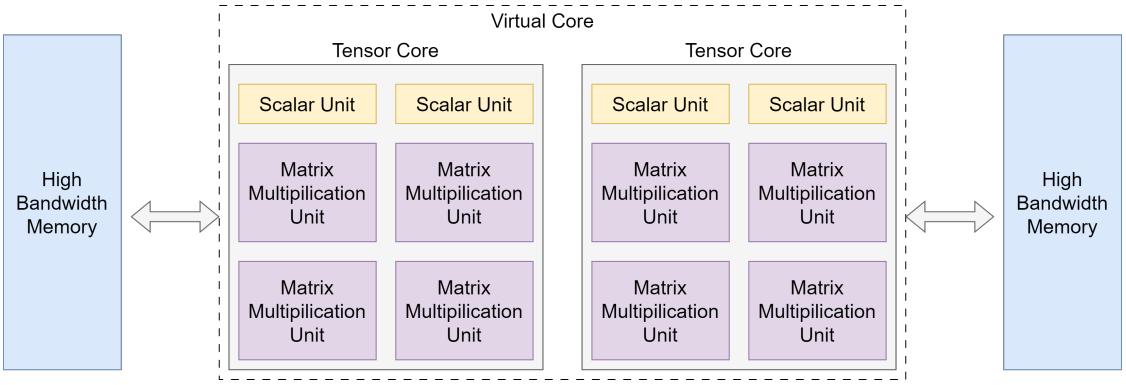


FIGURE 5.5: TPU v4 block diagram [244].

The systolic array algorithm, proposed in 1978 by Kung [241], is very efficient in accelerating GEMM. It offers great data reuses that not only save memory bandwidth but also reduce memory read and write energy consumption while maintaining a simple control flow. One of the most typical systolic array accelerators is Google's Google Tensor Processing Unit (TPU) [29, 242–244]. Fig. 5.5 shows the block diagram of TPU v5.

ShiDianNao [150], one member of the DianNao family, implements its PEs in a systolic array. With a small scale of PEs, it achieves relatively high PE utilization while limiting the computation throughput.

Liu *et al.* [245] proposed a systolic accelerator with three types of data-level parallelization that boost PE utilization.

Classical Strassen algorithm [237] is also re-associated to reduce the computational complexity of GEMMs from $O(N^3)$ to $O(N^{2.807})$ in DNNs. For instance, for 2×2 matrix multiplication $C = A \times B$, where:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \; B = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \tag{5.4}$$

We have:

$$C = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & ef + dh \end{bmatrix} \tag{5.5}$$

$$= \begin{bmatrix} k_5 + k_4 - k_2 + k_6 & k_1 + k_2 \\ k_3 + k_4 & k_1 + k_5 - k_3 - k_7 \end{bmatrix} \tag{5.6}$$

where the k_{1-7} are intermediate values:

$$k_1 = a(f - h) \tag{5.7}$$
$$k_2 = (a + b)h \tag{5.8}$$
$$k_3 = (c + d)e \tag{5.9}$$
$$k_4 = d(g - e) \tag{5.10}$$
$$k_5 = (a + d)(e + h) \tag{5.11}$$
$$k_6 = (b - d)(g + h) \tag{5.12}$$
$$k_7 = (a - c)(e + f) \tag{5.13}$$

In this case, each recursion reduces multiplications by 1/8. Cong *et al.* [55] applied the Strassen algorithm to the convolution transformed in GEMM and reduced the computational complexity. To reduce the number of multiplications, Winograd minimal filtering algorithm [238] is applied while increasing the adders for the transformation, which is efficient in the maths-limited layers with a filter size equal to or smaller than 3. Lavin [56] proposed a minimal 2D Winograd convolution algorithm $F(m \times m, r \times r)$ in Eq. 5.14, where matrix B and G transform the $(m+r-1)\times(m+r-1)$ image tile d and $r \times r$ filter g in the spatial-domain to Winograd-domain activation $B^T dB$ and weights GgG^T, $\odot$ indicates element-wise multiplication.

$$Y = A^T[(GgG^T) \odot (B^T dB)]A \tag{5.14}$$

NVDLA [149] also provides Winograd convolution mode to reduce the number of multiplications for 3×3 filters, where the weight transforming is done offline. However, only limited filter sizes and tiles are supported by Winograd hardware as this algorithm needs specialized transforming matrices, which limits its flexibility.

UniWiG, proposed by Kala *et al.* [246], supports both Winograd and GEMM in the same PE to support both convolution and FC layers therefore increasing its flexibility.

Liang *et al.* [39] proposed an architecture that supports Winograd-based convolution.

Similarly, Fast Fourier Transform (FFT) algorithm [239] converts the data to the more computationally efficient Fourier domain. FFT algorithm reduces the number

of multiplications from $O(RSEF)$ to $O(EF\log_2(EF))$, where E and F are the width and height of the output feature while R and S are the width and height of the filter. Eq. 5.16 shows the computation of the convolution using the FFT algorithm. $\otimes$ indicates convolution operation while $\odot$ indicates element-wise multiplication. Input feature and weight are converted into Fourier domain followed by element-wise multiplications. As the product results are in the Fourier domain, an Inverse FFT (IFFT) operation is required to transform the results back to the spatial domain.

$$output = input \otimes weight \tag{5.15}$$

$$= IFFT(FFT(input) \odot FFT(weight)) \tag{5.16}$$

Previous work in 1997 [247] explored the application of FFT to accelerate inference. Recent works also tried to map FFT on GPU platforms [33, 34]. The architecture proposed by Liang *et al*. [39] also supports the FFT algorithm by simply replacing the transforming matrices.

Zhang *et al*. [53] proposed an accelerator that employed FFT and Overlap-and-Add (OaA) on a CPU-FPGA platform with shared memory to reduce computational requirements and memory access latency. By using OaA, the computational complexity is further reduced to $O(EF\log_2(R))$ [248]. The input feature map is firstly converted by 2D FFT kernels into Fourier domain which is stored in the image buffers later. The output feature map computed from the MAC array is converted back to the spatial domain by 2D IFFT kernels. For the 1×1 convolution, the input feature map and output feature are bypassed as it is inefficient to perform the 1×1 convolution in the Fourier domain.

5.3 CNN-DLA Dataflows

To reduce the memory capability requirements for the ResNet, the memory-centric, hardware-efficient Cross-Layer Optimization dataflow is proposed in CNN-DLA.

Besides, both inner product MNK dataflow and inner product NKM dataflow are supported in CNN-DLA for the flexibility of CNN-DLA to support different CNN models. The MNK and NKM dataflows are introduced in Section 2.2.3 where M,

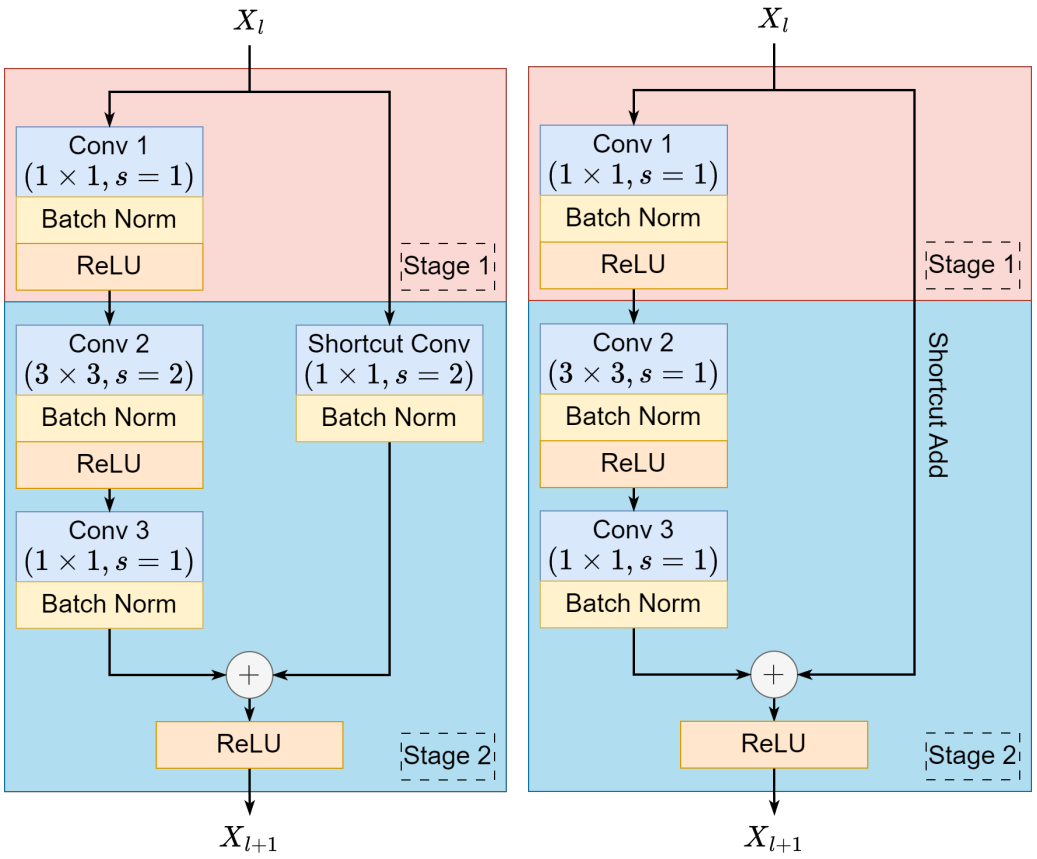


FIGURE 5.6: The two stages in Cross-Layer Optimization dataflow for (a) Downsampling Bottleneck Unit; (b) Regular Bottleneck Unit.

N, and K are the number of rows and columns of the matrix C and the number of columns of the matrix A respectively. The sequence of M, N, and K represent the order of the for-loop in the GEMM operation, from outer to inner.

5.3.1 Cross-Layer Optimization Dataflow Introduction

To reduce the memory capability requirement and improve the area efficiency and power efficiency, Cross-Layer Optimization dataflow is introduced from Block 0 to Block 2 in ResNet, where there is up to 127.5 MB INT32 output, as illustrated in Fig. 5.1.

The Cross-Layer Optimization (CLO) dataflow is proposed to reduce the memory capability requirement and improve the area efficiency and power efficiency. As illustrated in Fig. 5.6, the CLO dataflow consists of two stages for the bottleneck

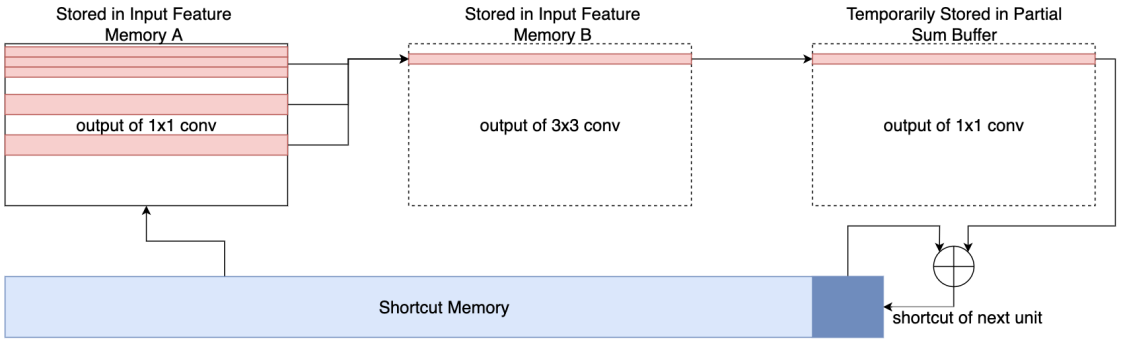


FIGURE 5.7: Cross-Layer Optimization dataflow in GEMM Hardware view

units in ResNet. The first stage includes the first convolutional layer (the first 1×1 convolution) while the second stage includes the other two convolutional layers and adds the shortcut data (or the convolution output of the shortcut convolutional layer if any).

The shortcut convolutional layer is computed in the 2nd stage as the number of the input is only 1/4 of that of the output in this layer to save reading energy consumption, although it can be computed in the 1st stage.

The main idea of this dataflow is to minimize the size of the feature maps that are required to store in the memory for the next computation while maximizing the utilization of memories, *i.e.*, overwriting the memory if possible.

Fig. 5.7 draws the dataflow where the GEMM data is stored in shortcut memory, memory A and B. As it shows, in the first stage, the whole output of the first 1×1 convolution is stored in the input feature memory A, so it can provide all the input data for the next 3×3 convolution, as some rows of the first 1×1 output matrix are required multiple times.

To minimize the memory capacity required to store the outputs, a minimal number of 3×3 convolutions and the last 1×1 convolutions are computed in the second stage. The shape of the output feature matrix is 32×16, which equals the minimal size of the last 1×1 convolutional layer in one iteration, which is also the shape of the 3×3 convolution output matrix in one iteration.

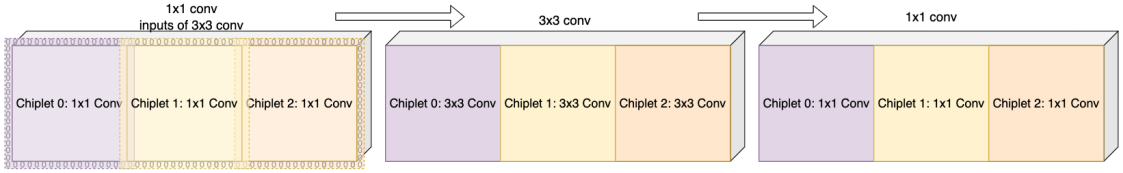


FIGURE 5.8: The CLO dataflow is mapped into Chiplets in Conv2D view

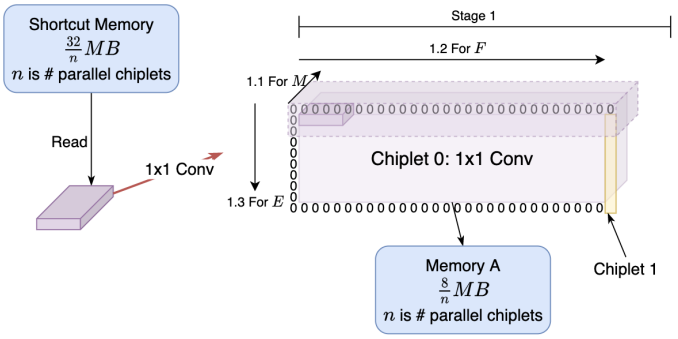


FIGURE 5.9: Detailed sequence of Stage 1 Cross-Layer Optimization dataflow in Conv2D view

5.3.2 CLO Dataflow Partitions and Mapping to Chiplets

The CLO dataflow is partitioned and mapped into several Chiplets, which is illustrated in Fig. 5.8. Communication between the Chiplets within the same stage is only required during the 3×3 convolution.

If there are n Chiplets within the same stage, the memory capacity of Input Feature Memory A and Shortcut Memory can be divided by n. The detailed sequences of these two stages are shown in Fig. 5.9 and Fig. 5.10 respectively.

As Section 5.3.1 describes, an **Inner Product MKN dataflow**, *a.k.a.*, row stationary (RS) or **MNK dataflow**, *a.k.a.*, output stationary (OS) can be applied in the two stages. RS GEMM dataflow is preferred here. Firstly, m output channels are computed. Then the computation area is moved horizontally, and finally, it is moved vertically.

In the 1st stage, the first 1×1 convolution is computed. Its input is loaded from the shortcut memory and read from the input buffer.

Algorithm 5 shows the temporal for-loops in stage 1 of the cross-layer optimization dataflow with RS GEMM dataflow.

Algorithm 5: Temporal for-loops in stage 1 of the cross-layer optimization dataflow with RS GEMM dataflow.

Data: I_0, W_0
Result: O_0
`// RS GEMM Dataflow`
for $a = 0$ **to** $A2_0$ **do**
  for $b = 0$ **to** $B2_0$ **do**
    for $c = 0$ **to** $C2_0$ **do**
      $O_0[a][c] += I_0[a][b] \times W_0[b][c]$;
  ReLU($O_0[a]$);

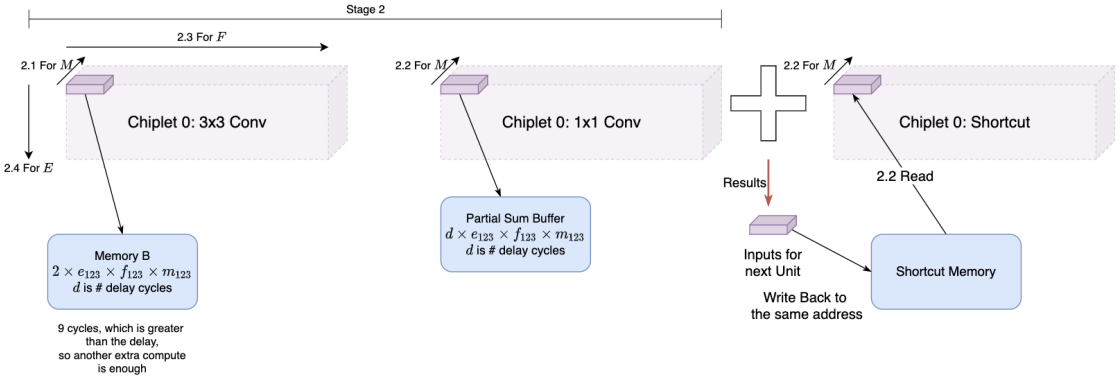


FIGURE 5.10: Detailed sequence of Stage 2 Cross-Layer Optimization dataflow in Conv2D view

The computation sequence is similar in stage 2, which is described in Algorithm 6. Firstly, M output channels of the 3×3 convolution are computed, which are the input channels for the last 1×1 convolution. Secondly, with these outputs, M output channels of the last 1×1 convolution are computed, where the corresponding shortcut data is added (or computed) for the final output of this unit, *which is written into the shortcut memory overwriting the unused shortcut data*. Then the computation area is moved horizontally, and finally, it is moved vertically.

5.3.3 Gains of the CNN-DLA Dataflows

Considering the downsampling bottleneck unit in ResNet, where there is a shortcut convolutional layer, the memory capability requirements of the bottleneck unit in the ResNet-152 with a 1080p image as input, are 262.97 MB, that is, 127.5 MB for the INT32 results of the third convolutional layer before the ReLU function and

Algorithm 6: Temporal for-loops in stage 2 of the cross-layer optimization dataflow with RS GEMM dataflow.

Data: I_0, W_1, W_1, W_{sc}, O_0
Result: O_2
for $a = 0$ **to** $A2_1$ **do**
  // Part 1: 3×3 convolution
  for $b_{3\times3} = 0$ **to** $B2_1$ **do**
    for $c_{3\times3} = 0$ **to** $C2_1$ **do**
      $O_1[a][c_{3\times3}] += O_0[a][b_{3\times3}] \times W_1[b_{3\times3}][c_{3\times3}]$;
  ReLU($O_1[a]$);
  // Part 2: 1×1 convolution
  for $b_{1\times1} = 0$ **to** $B2_2$ **do**
    for $c_{1\times1} = 0$ **to** $C2_2$ **do**
      $O_2[a][c_{1\times1}] += O_1[a][b_{1\times1}] \times W_1[b_{1\times1}][c_{1\times1}]$;
  // Part 3: Shortcut convolution
  for $b_{sc} = 0$ **to** $B2_{sc}$ **do**
    for $c_{sc} = 0$ **to** $C2_{sc}$ **do**
      $O_2[a][c_{sc}] += I_0[a][b_{sc}] \times W_{sc}[b_{sc}][c_{sc}]$;
  ReLU($O_2[a]$);

another 127.5 MB for the INT32 results of the shortcut convolutional layer before the ReLU function.

By adopting the Cross-Layer Optimization dataflow, the memory capability requirements of the bottleneck unit in the ResNet-152 with a 1080p image as input, are 39.845 MB, that is, 7.97 MB for the INT8 results of the second convolutional layer after the ReLU function and 31.875 MB for the INT8 output of this bottleneck unit, which is only 15.15% of the original memory capability requirements.

By supporting both the inner product MKN dataflow and NKM dataflow, the memory capability requirements of the weight buffers are reduced from 2,304 KB (for the second convolutional layers in block 4) to 368 KB (for supporting the CLO dataflow), which is only 15.97% of the original memory capability requirements.

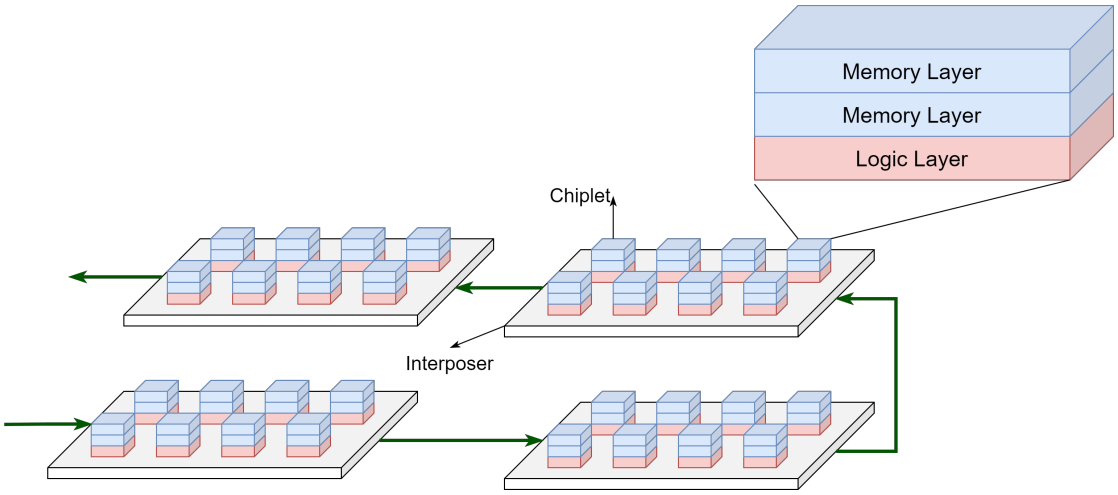


FIGURE 5.11: The illustration of the CNN-DLA Chiplet system. The system consists of several Chiplets in one interposer, where each Chiplet contains one logic layer and two memory layers.

5.4 CNN-DLA Architecture

5.4.1 Architecture Overview

There are two memory layers and one logic layer in one CNN-DLA Chiplet, which is illustrated in Fig. 5.11. The logic layer contains 4 memory blocks, 2 weight blocks, 16 MMUs, 16 PSum Buffers, 16 ReLU units, 4 MaxPool modules, 4 AvgPool modules, and 4 XBars. Each memory layer contains 1 weight memory, and 1 shortcut memory.

Fig. 5.12 illustrates the top architecture of the logic layer in one CNN-DLA Chiplet, whose clock frequency is 200MHz. This chip supports all the convolution operations in Faster-RCNN [209] with ResNet [7] backbone. The main computation modules are the distributed Matrix-Multiplication Units (MMUs), which are scalable.

The input feature map data is stored in the four distributed input memory blocks, while the shortcut data is stored in the shortcut memory, which is in the memory layer.

The two distributed weight blocks store the weight of one convolutional unit in cross-layer dataflow (see section 5.3) while storing the weight of current computation in NKM dataflow.

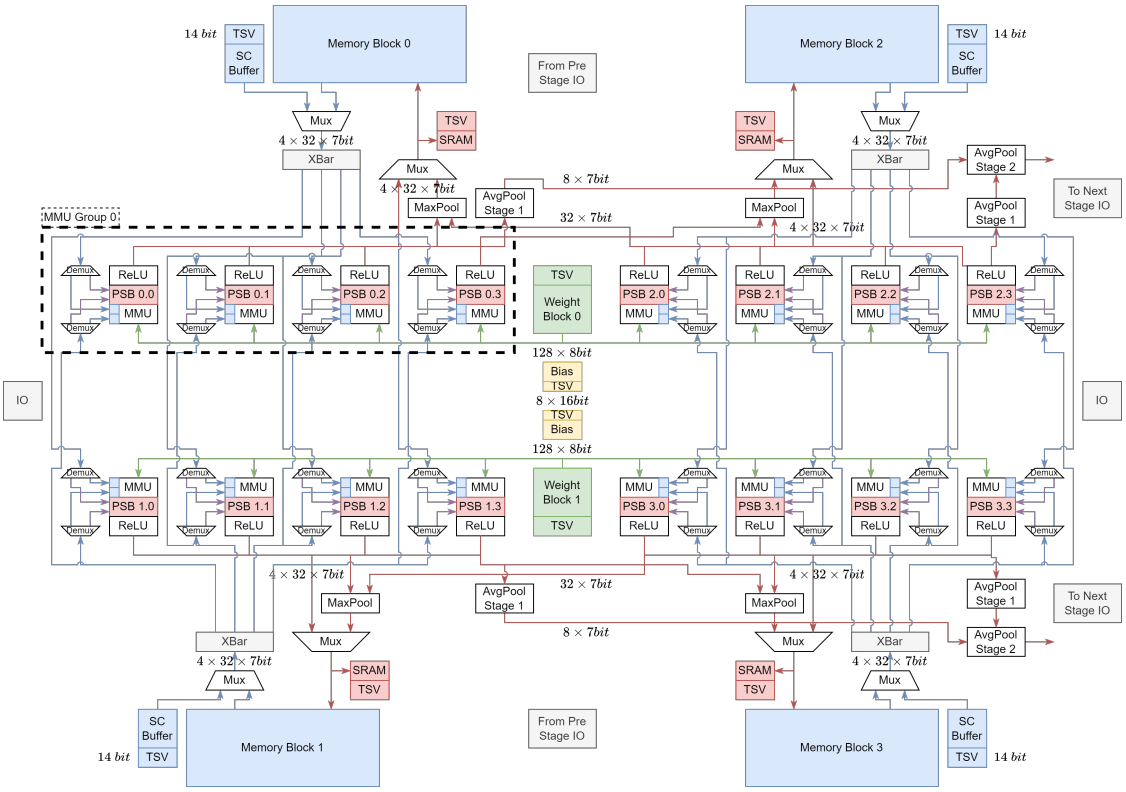


FIGURE 5.12: CNN-DLA top architecture

The input feature map data of current computation is buffered by the FIFOs near the MMUs which is read directly from the corresponding two input memory blocks.

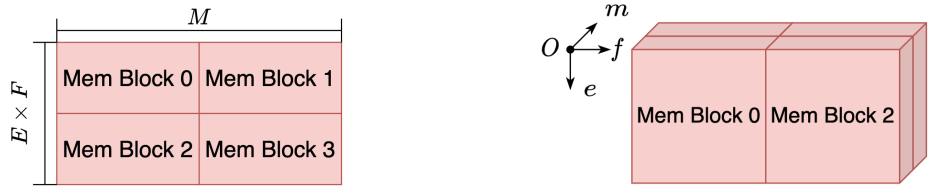
The partial sum is accumulated via the accumulators and stored in the distributed partial sum (PSum) buffers near the MMUs.

ReLU, MaxPool, and Average Pool operations are done in the corresponding distributed function units. Those operations are *directly related* to ResNet.

5.4.2 Convolution Partitions Overview

In CNN-DLA, there are four MMU groups, and each of them computes different partitions of the output feature map to reduce the complexity of data read/write.

More specifically, MMU group 0 (The group on the top left in Fig. 5.12) and group 1 (The group on the bottom left in Fig. 5.12) compute output feature map in different channels. MMU group 0 and MMU group 2 (The one on the top right in Fig. 5.12) compute output feature maps in different columns. The computation results of each MMU group are stored in their own memory blocks.



(A) The storage mapping in GEMM's view (B) The storage mapping in Conv2D's view

FIGURE 5.13: The four memory blocks store different parts of the feature map, which also indicates the computation partitions.

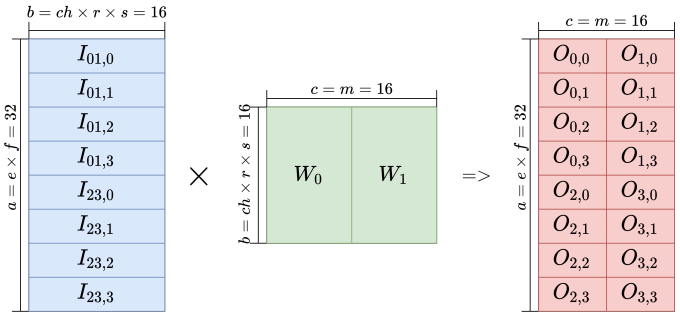


FIGURE 5.14: The block matrix multiplication of the whole chip (four MMU groups) and its shape. The blue rectangle represents input, green rectangle represents weight while the red rectangle represents output.

Fig. 5.13a and Fig. 5.13b illustrate the feature map partitions and the corresponding memory blocks in GEMM's view and Conv2D's view respectively.

Fig. 5.14 draws the block matrix multiplication of the whole chip, *i.e.*, the four MMU groups. The first subscript 01 and 23 in the input matrices indicates the index of its destination MMU group, *e.g.*, 01 means the input matrices are for MMU group 0 and group 1. The first subscript from 0 to 3 in the output matrices indicates the index of the MMU group that computes this output matrix. The second subscript in the input and output matrices shows the index of the MMU inside one group.

As it shows, one CNN-DLA chip computes the following GEMM operations in temporal:

$$O = I \times W, \quad I \in \mathbb{R}^{a \times b}, W \in \mathbb{R}^{b \times c}, O \in \mathbb{R}^{a \times c} \tag{5.17}$$

where I is the input feature matrix, W is the weight matrix, and O is the output feature matrix, and:

$$a = e \times f \tag{5.18}$$

$$b = ch \times r \times s \tag{5.19}$$

$$c = m \tag{5.20}$$

where e is the computation ability on the output channel dimension, f is the output row dimension, ch is the input channel dimension, r and s are the kernel size dimension, and m is the output channel dimension.

As the computation pattern of these four MMU groups is the same, we only show the data sequence, and memory storage of MMU group 0 in the rest of this section.

5.4.3 Matrix-Multiplication Unit (MMU) Group

As the dashed rectangle on the top left side in Fig. 5.12 illustrated, every four MMUs on one corner are grouped to be an MMU group.

5.4.3.1 Structure of MMU Group

One MMU group is split into multiple MMUs to reduce the size of one synchronized computation module to avoid timing issues hence splitting a large matrix multiplication into block matrix multiplication.

As Fig. 5.12 illustrates, there are four MMU groups in one chip and each of them contains four MMUs.

5.4.3.2 Data Mapping in one MMU Group

The block matrix multiplication of one MMU Group is illustrated in Fig. 5.15. As it shows, the input feature matrix is divided into a 4-row-1-column matrix block, which is the same as the output feature matrix with different widths. The four input feature matrix blocks are mapped directly into the four MMUs in one group and the four output feature matrix blocks are computed correspondingly sharing the same weight matrix block.

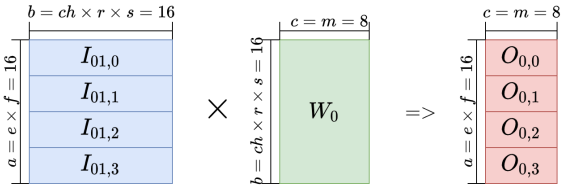


FIGURE 5.15: The block matrix multiplication of one MMU group and its shape.

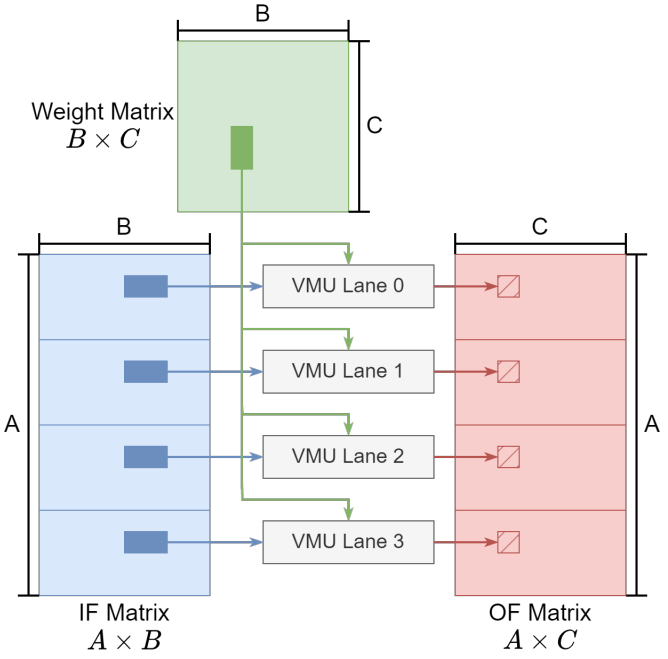


FIGURE 5.16: In temporal, an MMU computes a GEMM operation $o = i \times w$, where i is a 4×16 input feature matrix, w is a 16×8 weight matrix, and o is a 4×8 partial sum matrix.

5.4.3.3 Structure of MMU

As Fig. 5.16 illustrated, one MMU contains four VMU Lanes. In temporal, the MMU computes a GEMM operation $o = i \times w$, where i is a 4×16 input feature matrix, w is a 16×8 weight matrix, and o is a 4×8 partial sum matrix. To reduce the size of one synchronized computation module to avoid timing issues, the input feature matrix is further divided into four 1×16 vectors, which are mapped into the four VMU Lanes respectively. Weights are broadcast to the four VMU Lanes.

As Fig. 5.17 illustrated, the VMU Lane computes a single GEMV operation in temporal. Each VMU Lane consists of 4 Vector-Multiplication Units (VMUs).

Fig. 5.18 illustrates the data path of the VMU, which outputs the inner product result of two vectors, whose length is 16.

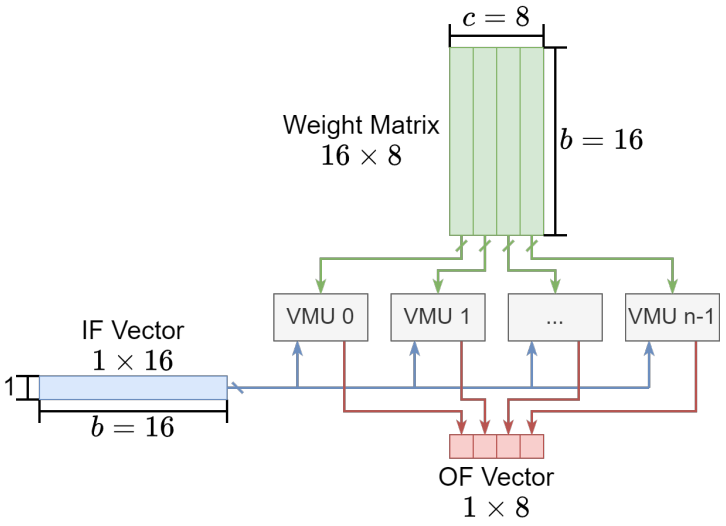


FIGURE 5.17: In temporal, a VMU Lane computes a GEMV operation $o = i \times w$, where i is a $1 \times b$ vector, and w is a $b \times c$ matrix. The matrix w is mapped into $b = 4$ VMUs.

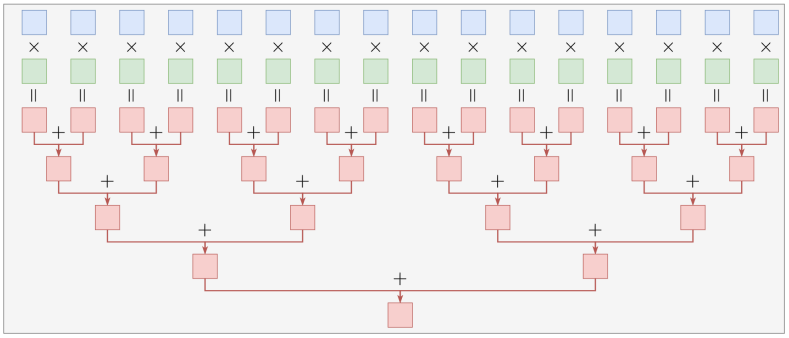


FIGURE 5.18: The computation data path of VMU, that the output is the inner project of two input vectors.

FIGURE 5.19: The block matrix multiplication of one MMU

5.4.3.4 Data Mapping in MMU

The block matrix multiplication of one MMU is shown in Fig. 5.19. Those matrix blocks are mapped into one MMU as Fig. 5.20 draws, where one row represents one VMU.

$I_{0,0-15} \times W_{0-15,0} = O_{0,0}$	$I_{1,0-15} \times W_{0-15,0} = O_{1,0}$	$I_{2,0-15} \times W_{0-15,0} = O_{2,0}$	$I_{3,0-15} \times W_{0-15,0} = O_{3,0}$
$I_{0,0-15} \times W_{0-15,1} = O_{0,1}$	$I_{1,0-15} \times W_{0-15,1} = O_{1,1}$	$I_{2,0-15} \times W_{0-15,1} = O_{2,1}$	$I_{3,0-15} \times W_{0-15,1} = O_{3,1}$
$I_{0,0-15} \times W_{0-15,2} = O_{0,2}$	$I_{1,0-15} \times W_{0-15,2} = O_{1,2}$	$I_{2,0-15} \times W_{0-15,2} = O_{2,2}$	$I_{3,0-15} \times W_{0-15,2} = O_{3,2}$
...	...	...	...
$I_{0,0-15} \times W_{0-15,7} = O_{0,7}$	$I_{1,0-15} \times W_{0-15,7} = O_{1,7}$	$I_{2,0-15} \times W_{0-15,7} = O_{2,7}$	$I_{3,0-15} \times W_{0-15,7} = O_{3,7}$

FIGURE 5.20: The detailed data mapping in one MMU, one row represents one VMU.

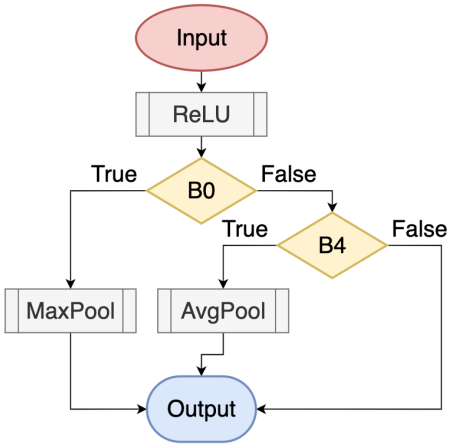


FIGURE 5.21: Flow chart of the special function modules.

5.4.4 Special Function Modules

5.4.4.1 Special Function Modules Overview

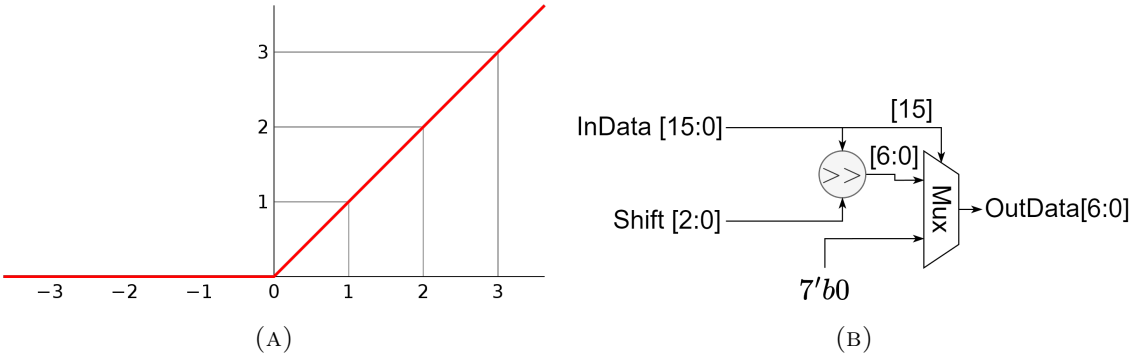
To obtain a better timing result, the special function modules, *i.e.*, ReLU, MaxPool, and Average Pool, are also distributed and located near their input source.

Fig. 5.21 illustrates the flow chart of the special function modules. ReLU is required after each convolutional layer, MaxPool operation is needed at the end of Block 0 in ResNet, and Average Pool is needed at the end of Block 4 in ResNet.

5.4.4.2 ReLU

Fig. 5.22a draws the ReLU function.

As we can obtain from Fig. 5.21, ReLU is required in all situations. Hence, to reduce wire conjunction, ReLU modules are located close to the partial sum buffers (PSB)



(A) (B)

FIGURE 5.22: (a) Plot of the ReLU function. (b) The structure of the ReLU module.

as Fig. 5.12 drew. After ReLU, the bit width of the outputs was reduced from 32 bits to 7 bits.

As illustrated in Fig. 5.22b, the ReLU module is a combinational module whose inputs are 32-bit fixed-point data with a sign bit and a 3-bit shift value to shift this input and outputs a 7-bit fixed-point data without a sign bit. The sign bit is ignored in the output, which means the output is always positive. If the input value is negative, the output value is a 7-bit zero.

5.4.4.3 MaxPool

Fig. 5.23 illustrates the MaxPool operation at the end of Block 0 in ResNet. The convolution is processed horizontally hence the MaxPool operation in this module is also computed horizontally. MaxPool kernels are drawn in colorful transparent 3×3 squares (blue, green, yellow, and red) while the outputs from MMU Groups are shown in colorful small squares with lowercase letters from "a" to "p". The small squares of the outputs in different clock cycles are in different backgrounds (orange, violet, red, and blue).

Fig. 5.24 draws the structure of the MaxPool module while Fig. 5.25 illustrates its operations to compute the MaxPool. As it shows, the MaxPool module reuses the computation datapath of comparing, as the inputs are valid every 12 cycles, while the required throughput of the MaxPool module is low.

MaxPool operation is separated into two stages. As the MaxPool kernel is 3×3, the max values of the three rows are computed by "Max3" modules in the first

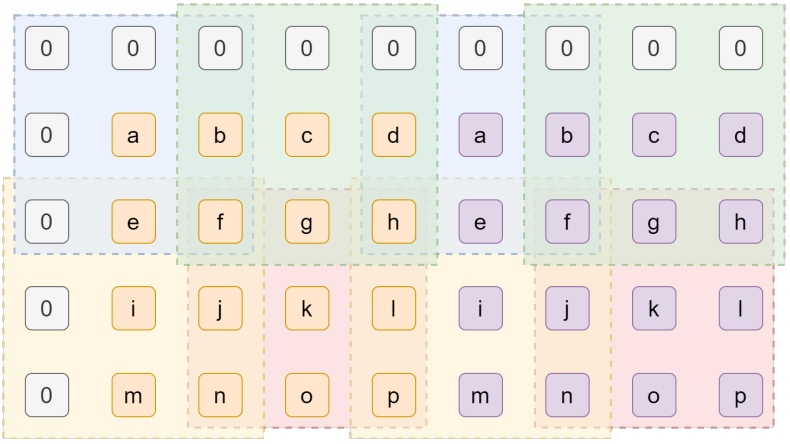


FIGURE 5.23: The MaxPool operation at the end of Block 0 in ResNet. MaxPool kernels are drawn in colorful transparent 3×3 squares while the outputs from MMU Groups are shown in colorful small squares with lowercase letters from "a" to "h". The small squares of the outputs in different clock cycles are in different backgrounds.

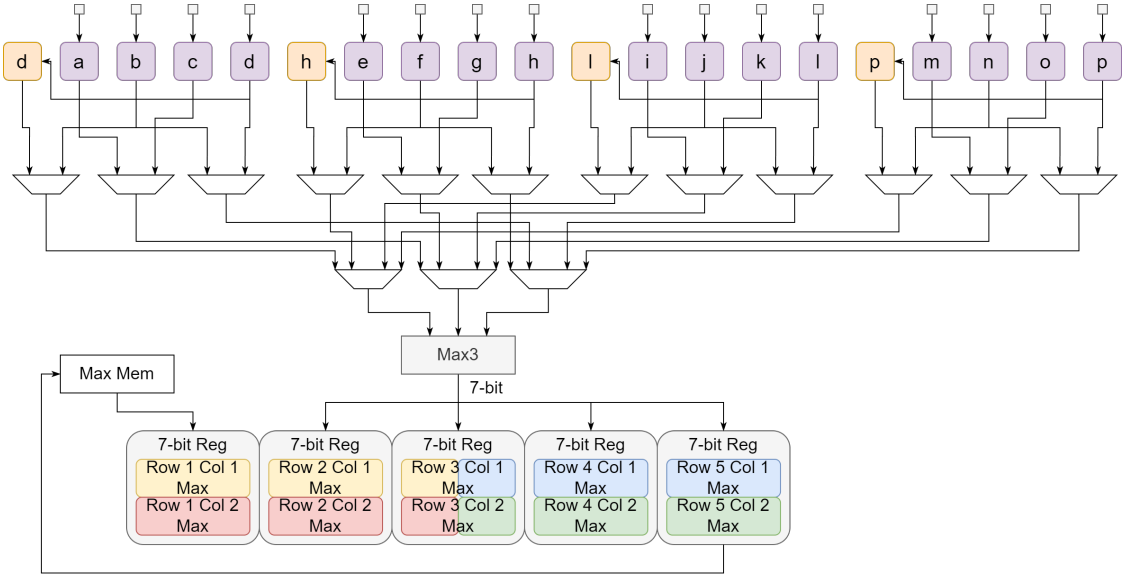


FIGURE 5.24: The structure of the MaxPool module. Computation datapath is reused as the required throughput of the MaxPool module is low.

stage, while the maximum value of these three values is compared by the other "Max3" module in the second stage.

As there are overlaps within the two kernels within the 2×4 outputs, the next outputs, and the next turn of the MaxPool operations, different methods are applied to provide those overlapped data. For the overlap within the 2×4 outputs, the registers in locations "b" and "f" are reused. For the overlap between the two near 2×4 outputs, registers are used to buffer the data in locations d and h. To provide

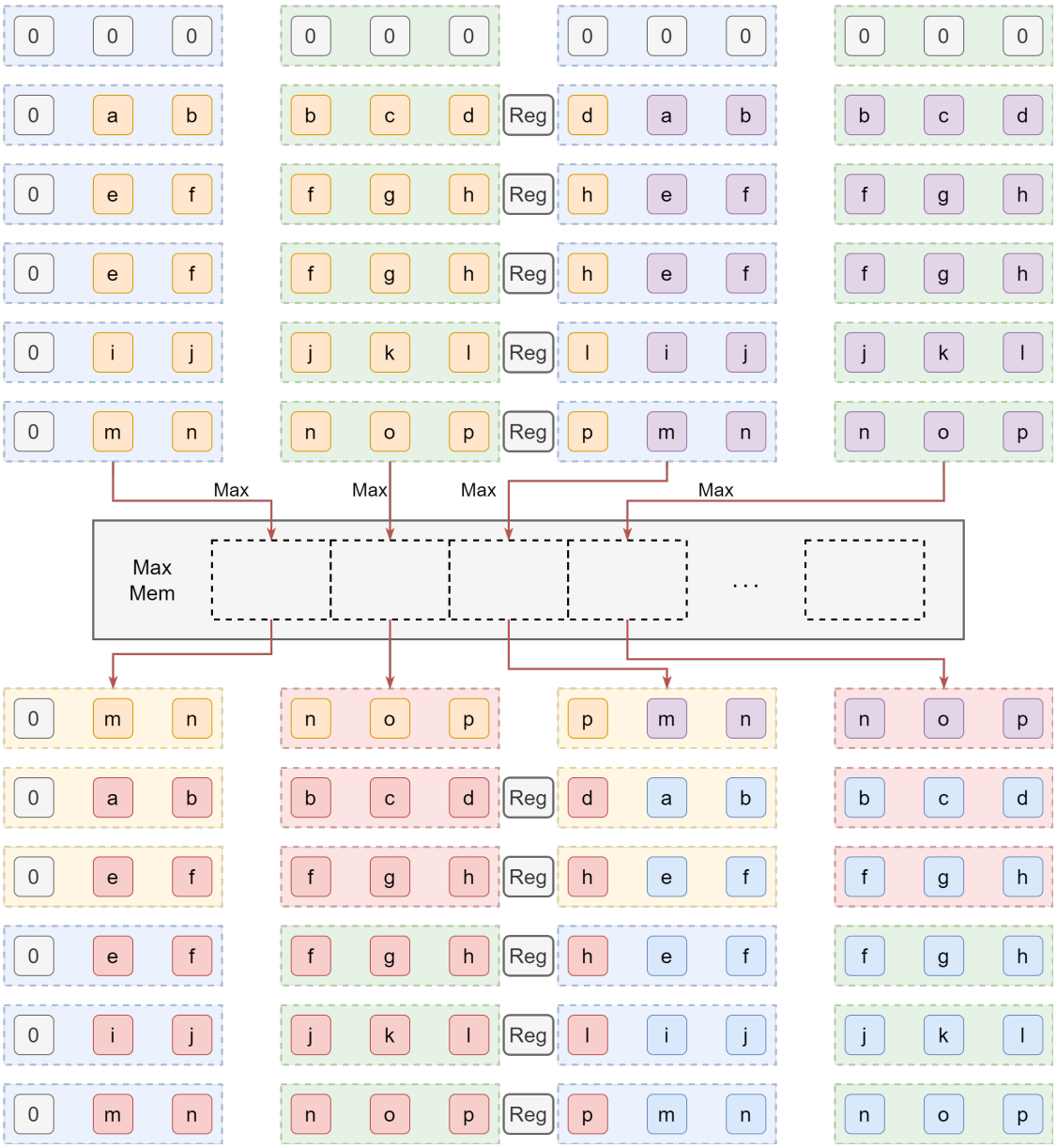


FIGURE 5.25: The data mapping of the MaxPool kernels into the MaxPool module.

the maximum values of the first row in the next turn, multiple SRAM banks are used to store them.

As the outputs from MMU Groups are not valid every cycle to compute the convolutional layer in Block 0, the MaxPool module is controlled by a state machine instead of a pipeline structure. The four states are "Idle", "Read/Max1", "Max2", and "Write", which is shown in Fig. 5.26.

When the ready and valid signals of the input are high, the state moves from

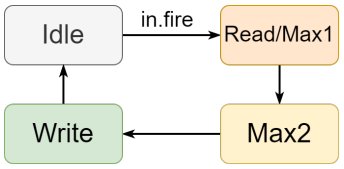


FIGURE 5.26: The state machine of the MaxPool module.

"Idle" to "Read/Max1", where the maximum values of overlapped rows are read correspondingly while the maximum values of the second rows and third rows are computed. The two registers that store the previous d and h are updated in this cycle as well.

The next stage is called "Max2", where the maximum values of three rows are compared, and the outputs are obtained, hence the output is valid in this cycle. The maximum values of the third rows are buffered in registers in this cycle as well.

The maximum values of the third rows are written into the SRAMs correspondingly from the register buffers in the last stage. Then the state goes to "Idle" and waits for the next coming outputs.

5.4.4.4 Average Pool

A 7×7 channel-wise Average Pool operation is applied after Block 4, *i.e.*, take an average of the 7×7 region channel-wise. We optimized the AvgPool operation to reduce the memory bandwidth requirement by accumulating the outputs of the 7×7 convolutional layer within one channel together during the last 1×1 convolutional layer, as illustrated in Fig. 5.27. Meanwhile, the weights of the last convolutional layer are divided by 49 off-chip to get the average value of the 7×7 convolutional layer.

Shown in Fig. 5.28, the AvgPool operation is divided into two stages as the data for the whole AvgPool is distributed. The hardware connections can be found in Fig. 5.12. In Block 4, every MMU computes one row of the output feature map, and the last column of the MMUs is gated as there are only 7 rows in the output feature map. At AvgPool stage 1, the partial sum of 4×7 outputs and 3×7 outputs are channel-wise added separately. At AvgPool stage 2, these two partial sums are added together to get the final Average Pool operation result.

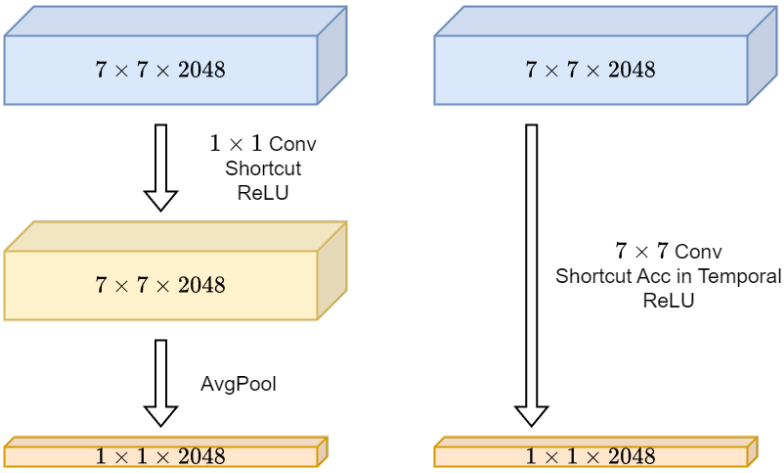


FIGURE 5.27: The Average Pool operation after Block 4. Left: the original Average Pool operation. Right: the optimized Average Pool operation.

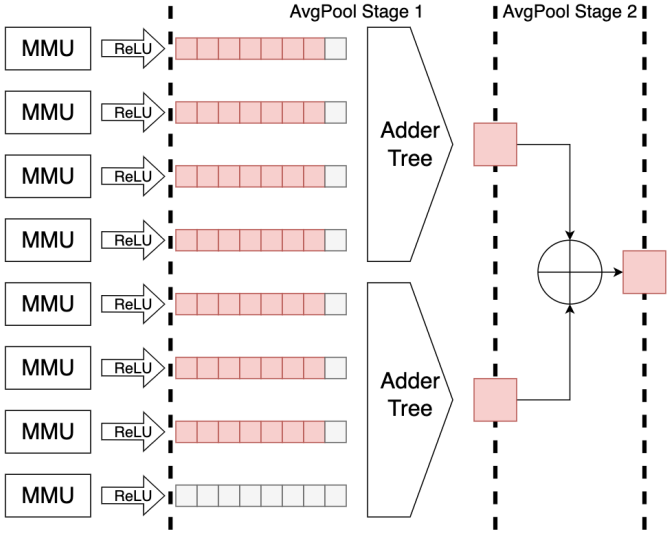


FIGURE 5.28: The two stages of the AvgPool operation. The operation in one channel is shown in this figure. The grad box represents zero values.

5.4.5 Memory Hierarchy

There are several memory blocks in CNN-DLA, including Input Feature Memory, Shortcut Memory, Weight Memory, Weight Buffer, Partial Sum Buffer, MaxPool Memory, and Bias Memory, which is summarized in Table 5.2.

The **shortcut memory** stores the input matrix temporarily for the residential shortcut adding. The **weight memory** and the **bias memory** store the weight matrix and bias for certain units, respectively.

TABLE 5.2: The spec of memories, including location, capacities, word bit, and word count of an 8-Chiplet CNN-DLA accelerator system for ResNet-152 that supports up to Full-HD image resolution.

Mem Name	Weight Mem	Shortcut Mem	Mem A	Mem B	Weight Buffer A	Weight Buffer B	Weight Buffer C	Weight Buffer D	PSum Buffer	MaxPool Mem
Location	ML[a]	ML	LL[b]	LL	LL	LL	LL	LL	LL	LL
Total Capacity (KB)	17807	4080	1,020	255	32	128	144	64	32	420
# Mem Inst	2	2	4	4	2	2	2	2	16	4
Capacity of Each Inst (KB)	8903.5	2040	255	63.75	16	64	72	32	2	105
Word Bit of Each Inst	1,024	896	1024	1024	1024	1024	1024	1024	256	56
Word Count of Each Inst	71228	18652	2040	510	128	512	576	256	64	1920
Port Number	1	1	1	1	1	1	1	1	1	1
IP Word Bit	128	128	64	64	128	128	128	128	128	64
IP Word Count	16000	16000	2048	512	512	512	1024	512	512	2048
# IP in Each Inst	40	14	16	16	8	8	8	8	2	1

[a] ML means Memory Layers.
[b] LL means Logic Layers.

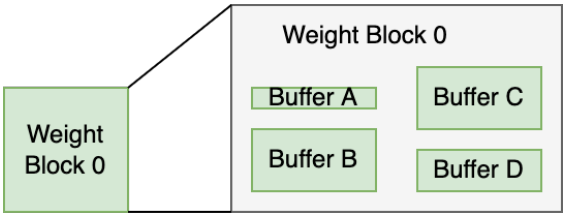


FIGURE 5.29: There are four buffers in one weight block

There are two **weight blocks** in the logic layer, and each of them contains four weight buffers, which are shown in Fig. 5.29. The **bias buffer** stores the bias data for current computation in the GEMM sequence for MMU to read.

There are four Memory Blocks near each MMU Group in the logic layer. In every memory block, there are two input feature memories, A and B. In Block 1 and Block 2 of ResNet, to support Cross-layer Optimization dataflow, **Input Feature Memory A** stores the output feature matrix of the first 1×1 convolutional layer, *i.e.*, the output of the first stage. **Input Feature Memory B** stores part of the output feature matrix of the 3×3 convolutional layer (usually one or two rows are enough). In Block 3 and Block 4 of ResNet, the shortcut is stored in **Input Feature Memory A** to save the number of TSVs and r/w energy consumption as the shortcut is small enough to fit into this memory. The output feature matrix is stored in **Input Feature Memory A** and **Input Feature Memory B** accordingly.

The **shortcut buffer** stores the shortcut data for current computation in the GEMM sequence for MMU Groups to read.

The **Partial Sum Buffer** temporarily stores the partial sums from every MMU for further accumulation to maximize and balance the input and weight reuse and minimize the energy consumption.

5.4.6 Partial Sum Adder

Between each MMU and its PSum Buffer, there is a partial sum adder to accumulate the partial sums stored in the PSum Buffer with the new results from the corresponding MMU.

5.4.7 Input Source XBar

XBar, or Crossbar switch, is used to re-group the input from the sources (*e.g.*, Input Feature Memory *etc.*) to Input Ping-Pong Buffers.

5.5 Evaluation

5.5.1 Experimental Setup

The execution times of CNN-DLA in different configurations are simulated by Verilator simulator [233] with the Verilog code generated by Chisel [201] and collected in Python scripts. Memory IPs (Regfiles and SRAMs) are generated by the ARM memory compilers. The generated Verilog code with corresponding memory IPs is synthesized by Cadence Genus with Foundry 28 nm FD-SOI technology node for both area and power estimation. The floorplan and the place and route (P&R) are done by Cadence Innovus. The experimental setup is summarized in Table 5.3.

5.5.2 Experimental Results

We synthesized CNN-DLA at 200 MHz with the supported input resolution up to 1920×1080 for ResNet-152. The synthesis results are summarized in Table 5.4.

TABLE 5.3: Experimental setup.

Design Tools	
HDL	Chisel
Verilog simulator	Verilator
Logic synthesis	Cadence Genus
Design Parameters	
# Chiplets	44
TOPS/Chiplet @ 200 MHz	3.278
Input	INT 8
Weight	INT 8
Partial Sum	INT 32
Network	Up to ResNet-152
Image Resolution	Up to 1920×1080
Technology node	Foundry 28 nm FD-SOI
Core supply voltage	1.05 V

TABLE 5.4: The experimental results of the CNN-DLA for ResNet-152 with up to 1920×1080 images as input.

CNN-DLA for ResNet-152	**Area (mm^2)**
Memory Layer	
Weight Mem	17.981
Shortcut Mem	6.293
Data TSVs	2.503
Data IOs	0.234
Logic Layer	
Mem Block ×4	4.119
Weight Block ×2	2.481
PSum Buffer ×16	0.996
MMU ×16	7.800
ReLU ×16	0.270
MaxPool ×4	0.248
AvgPool ×4	0.024
Xbar ×4	3.360
Data TSVs	2.503
Data IOs	0.234
Top	
Memory Layer ×2	27.011
Logic Layer	22.034
Summary	
CNN-DLA Area (mm^2)	27.011
Frequency (MHz)	200
GOPS	3276.8

The synthesis results show that the area of the CNN-DLA is 27.011 mm^2 at 200 MHz. The area of the logic layer is smaller than the memory layer, which is 22.034 mm^2.

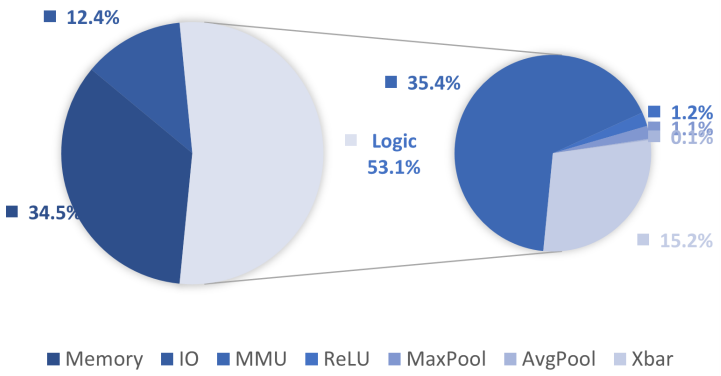


FIGURE 5.30: The area breakdown of the CNN-DLA for ResNet-152 with up to 1920×1080 images as input.

The area breakdown of the logic layer of the CNN-DLA is shown in Fig. 5.30. The area of the memories is 7.595 mm^2, which is 34.5% of the logic layer. The logic occupies 11.702 mm^2, which is 53.1% of the logic layer. The IOs and TSVs occupy 2.737 mm^2, which is 12.4% of the logic layer.

The XBar occupies a chip area of 3.36 mm^2, which is 15.25% of the logic layer. The area of the XBar is relatively large as there are numerous multiplexers in the XBar to re-group the input from the sources to Input Ping-Pong Buffers. To alleviate the wire congestion of these multiplexers, the multiplexers are placed with a relatively large space between them, which leads to a large area of the XBar.

5.5.3 System-level Performance Analysis

We proposed a Chiplet-based CNN-DLA accelerator system and computed the theoretical processing time for computing the inference of ResNet-152.

Table 5.5 shows the processing time and dataflow of each block in ResNet-152 adopting our Chiplet-based CNN-DLA accelerator system.

As it shows, the latency of the convolution blocks in ResNet-152 is 52.75 ms, and the throughput achieves 68 frames per second (FPS).

TABLE 5.5: The processing time and dataflow of each block in ResNet-152 adopting our Chiplet-based CNN-DLA accelerator system.

ResNet-152	**# Chiplets**	**Processing Time (ms)**	**Dataflow[a]**
B0	8	8.08	RS
B1U1-B1U3			CLO RS
B2U1-B2U8			CLO RS
B3U1-B3U16	8	12.1	CS
B3U17-B3U32	8	11.84	CS
B3U33-B3U36	8	6.12	CS
B4U1-B4B3	12	14.61	CS
Summary	**44**	**52.75**	**Hybrid Dataflow**

[a] "RS" is short for Row Stationary dataflow, "CLO RS" is short for Cross-layer Optimization Row Stationary dataflow, and "CS" is short for Column Stationary dataflow.

5.6 Chapter Summary

In this chapter, we introduce a Chiplet-based CNN-DLA hardware accelerator system that supports large input features. In the case study, a 44-Chiplet CNN-DLA accelerator system is designed for ResNet-152 that supports up to full-HD image resolution.

CNN-DLA supports various dataflows, which can be applied to different CNN models. We propose a novel memory-centric Cross-layer Optimization dataflow that reduces the memory requirements of the feature maps by 84.85%, from 262.97 MB to 39.845 MB. By supporting both MKN Row Stationary and NKM Column Stationary dataflow, the required capability of the weight buffer was reduced from 2,304 KB to 368 KB, which is an 84.03% reduction in ResNet.

With the dataflow-aware architecture for ResNet-152, the latency of the 44-Chiplet CNN-DLA is 52.75 ms, and the throughput achieves 68 FPS.

The synthesis results show that the area of such CNN-DLA is 27.011 mm^2 at 200 MHz, where there is one logic layer and two memory layers in the Chiplet.

Chapter 6

ViTA: A Highly Efficient Dataflow and Architecture for Vision Transformers

6.1 Introduction

Transformer-based models have achieved remarkable results in various Natural Language Processing (NLP) and Computer Vision (CV) tasks, such as machine translation [71, 128, 129], sentiment analysis [3, 120, 121], text summarization [5, 122–124], question answering [135–137], image classification [93, 96, 109, 110], object detection [96, 116, 117], semantic segmentation [96, 106, 107], image colorization [112], action recognition [119], and image generation [4, 111]. Some Transformer models, such as GPT-4 [10], can even perform both NLP and CV tasks in the multitask setting. Transformers have become the dominant neural architecture for both NLP and CV domains.

However, the scaling up of Transformer models to trillions of parameters and trillions of Multiply-Accumulate (MAC) operations, as in the case of GPT-4 [10], during both training and inference has made them both compute- and data-intensive. This poses a significant challenge for the deployment of these models in an area- and power-efficient manner.

The work in this chapter has been published in [Proceedings of the 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2024].

TABLE 6.1: Rough area and power for arithmetic modules in different data types in 28 nm 1.05V.

Operator[a]	Area (μm^2)					Power (μW)				
	INT8	***INT32***	***BF16***[b]	***FP16***	***FP32***	***INT8***	***INT32***	***BF16***	***FP16***	***FP32***
Add	23.14	97.16	454.64	471.82	1,184.68	12.34	60.89	443.80	638.88	1,197.96
Subtract	26.04	110.73	455.09	472.70	1,187.20	12.89	63.49	443.89	638.50	1,198.60
Multiply	192.35	2,935.45	630.49	992.27	3,054.07	89.16	1,488.77	392.12	844.71	2,463.09
Square	93.74	1,973.16	287.44	411.97	1,741.87	36.65	794.91	118.07	187.84	839.70
Divide	164.41	9,401.66	991.03	1,604.29	10,105.34	88.73	4,663.43	5,286.32	13,210.73	109,298.13
Square Root	36.79	1,250.85	269.61	522.26	6,053.15	16.03	12,138.32	564.61	3,199.53	89,014.78
Exponent	–	–	1,387.22	1,333.43	4,560.85	–	–	11,901.52	13,663.11	54,780.53

[a] Arithmetic modules are instantiated from Cadence ChipWare IPs and synthesized by Cadence Genus, and all input/output delays are set to zeros. The clock frequency is set to 300 MHz.
[b] BF16 is short for Brain Floating Point 16-bit developed by Google.

Operations in Transformer models are matrix multiplications and non-linear functions. Matrix multiplication consumes over 60% of the execution time of Vision Transformer (ViT) [13] on NVIDIA 2080 Ti GPU [249]. Nevertheless, thanks to both scalable hardware (*e.g.*, GPUs and Transformer accelerators on ASIC [250–254] or FPGAs [255–258]) and algorithmic optimizations (*e.g.*, quantization [92, 109, 259], sparsity [260] and function approximation [252, 253]), this operation has seen significant improvements.

The implementation of the non-linear functions, *e.g.*, GELU, Softmax, and LayerNorm, has been a bottleneck in terms of both area and power efficiency. Non-linear functions occupy nearly 40% and 60% of the execution time and memory usage respectively [249] and occupy 45% of the total area in SwiftTron [254] even if the matrix multipliers are dedicated. However, they have not witnessed significant improvement and remain bottlenecks due to their computational complexity and high throughput requirements.

Moreover, the computation dataflow of Transformer models is less explored, and many Transformer accelerators [254, 257] simply follow the original software implementation dataflow, which can be further optimized to reduce both memory footprint and data movement for achieving higher area- and power-efficiency. As shown in Fig. 6.1, considering the LayerNorm operation before the MHA and the shortcut adding after the Project GEMM, the memory requirements of the Multi-Head Self Attention (MHA) layer in the ViT-Base models to support 224×224 and VGA image resolutions are 1.45 MB and 22.66 MB, respectively, which requires a significant chip area reserved for the memory.

Addressing these concerns, this Chapter introduces ViTA – a scalable architecture with a highly efficient memory-centric dataflow for ViT – to accelerate the entire

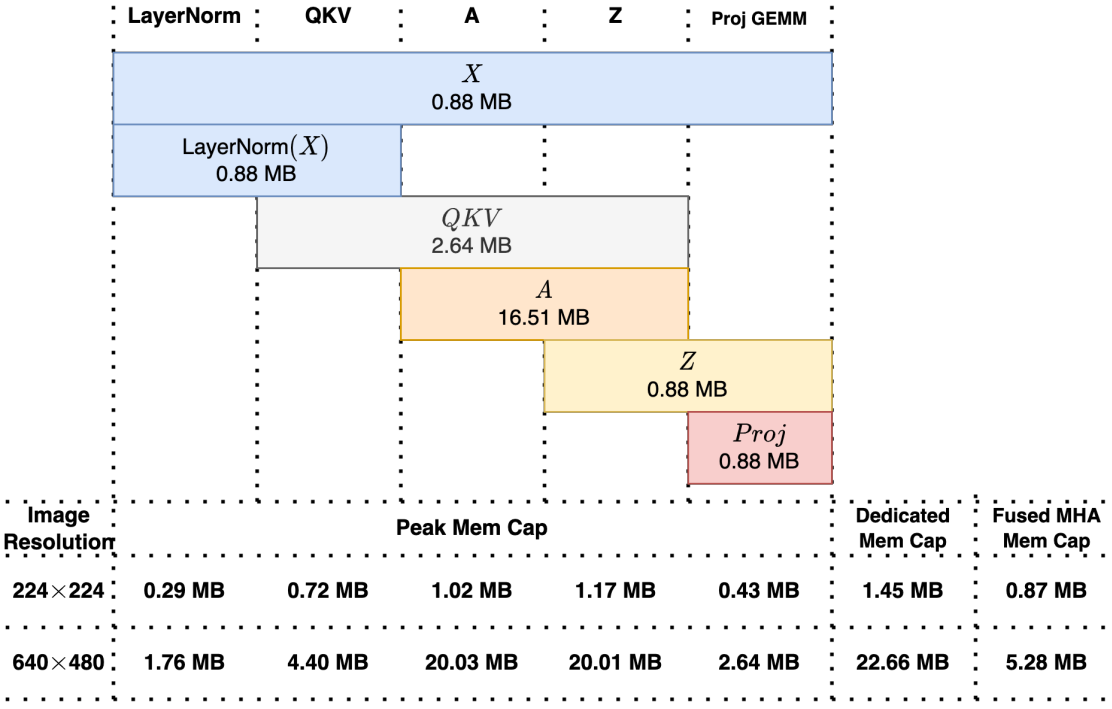


Image Resolution	Peak Mem Cap					Dedicated Mem Cap	Fused MHA Mem Cap
224×224	0.29 MB	0.72 MB	1.02 MB	1.17 MB	0.43 MB	1.45 MB	0.87 MB
640×480	1.76 MB	4.40 MB	20.03 MB	20.01 MB	2.64 MB	22.66 MB	5.28 MB

FIGURE 6.1: The memory capability requirements of the MHA layer in the ViT-Base models, with a 224×224 and VGA resolution RGB image as input in INT8 mode. As illustrated, to support VGA images, 22.66 MB and 20.03 MB memory are required if dedicated memories are used for middle results or memories are reused respectively.

ViT workload with high area efficiency and power efficiency. Our results show that ViTA achieves 16.384 TOPS with an area efficiency of 2.13 TOPS/mm^2 and a power efficiency of 1.57 TOPS/W at 1 GHz, surpassing state-of-the-art transformer accelerators by factors of 27.85× and 1.40×, respectively.

Our contributions are summarized as follows:

- A scalable architecture, ViTA, for the entire ViT workload, exploiting a memory-centric, hardware-efficient dataflow to reduce memory requirement and data movement.
- A novel fused special function module to compute non-linear functions in ViT, *i.e.*, GELU, Softmax, and LayerNorm, optimizing hardware resource sharing and further improving area and power metrics.
- A comprehensive design space exploration of the number of ViTA Kernels, and the number of VMUs in each VMU lane, which is used to trade off the area and power.

- A performance analysis of our ViTA architecture synthesized in the 28 nm FD-SOI technology node, and comparison with the state-of-the-art Transformer accelerators.

6.2 Related Work

6.2.1 Accelerations of Entire Transformers

Most Transformer models [3, 13, 92, 109, 259, 261] still rely on the general-purpose platform GPU for training and inference, which is not energy-efficient. To reduce the memory footprint, inference latency, and power consumption, several works have been proposed to accelerate the Transformer models on GPUs. I-BERT [259] uses lightweight integer-only approximation methods for non-linear operations, such as Layer Normalization, GELU, and Softmax, performing an integer-only end-to-end inference for BERT without any floating-point operations on GPUs. It approximated the GELU (I-GELU) and the experimental (I-EXP) function by second-order polynomials and computed the square-root function (I-SQRT) by the iterative algorithm proposed in [262]. Similarly, I-ViT [109] performs an integer-only end-to-end inference for ViT on GPUs with approximated Shiftmax, Shift-GELU, and I-LayerNorm.

NVIDIA FasterTransformer [92] is a de-facto framework that contains highly optimized Transformer layers for many Transformer models. It offers two quantization modes, where the input and output of GEMM operations are INT8 and INT32/INT8 respectively, while other non-linear functions are FP32. The data types of the basic kernels (operations) in ViT are summarized in Table 6.2.

FlashAttention [263] uses a novel data layout that optimizes memory access patterns on GPUs, reducing the number of cache misses and improving memory efficiency. It also introduces block-sparse attention, which reduces the number of computations required.

Accelerating the entire Transformer workload on ASIC is also explored. SwiftTron [254] proposed an architecture for the Transformer models with very large chip size, *i.e.*, 273 mm^2, and high power, *i.e.*, 33.64 W. Although this architecture adopted the optimized approach for the non-linear functions in [259], the area and

TABLE 6.2: The operations (kernels) in the Transformer and this work, and the corresponding data types.

Platform	**FasterTransformer**		**This Work**	
Datatype	Input	Output	Input	Output
GEMM	INT8	INT32/INT8	INT8	INT32
GELU	FP32	FP32	INT32	INT32
LayerNorm	FP32	FP32	INT32	INT32
Softmax	FP32	FP32	INT32	INT32
Bias Adding	FP32	FP32	INT8	INT32
Residual Adding	FP32	FP32	INT8	INT32

power are still very high, as it directly maps the whole Transformer models to the hardware. Besides, no special synchronizations are adopted in SwiftTron, hence SwiftTron is only executed at low frequency.

Nag *et al*. [257] introduced an architecture for ViT on FPGAs, adopting the non-linear functions units from the former study [255]. The MHA is computed head by head, reducing the memory requirements on-chip. However, five dedicated PE blocks are needed to compute the MHA, decreasing the flexibility of the architecture and the utilization of those PEs, for supporting both MHA and MLP layers.

Approximation schemes are introduced to accelerate the entire Transformer workload. NN-LUT [264] proposed a framework that employs a simple neural network that acts as a universal approximator whose structure is equivalently transformed into a Lookup Table (LUT). It approximated GELU, experimental function, division, and the reciprocal of the square root function by the LUT.

In [252, 253], Wang *et al*. proposed several techniques to utilize similarity and sparsity to achieve high energy efficiency. In [252], approximate computing is adopted for energy-saving where small values are computed with large errors, while large values are computed exactly, matching the error tolerance of attention. The PCSU is introduced in [253] to remove the redundant multiplications whose input vectors are similar.

However, re-training is required in works with dedicated approximated Transformer models to map the general existing models to the corresponding accelerators, which is not flexible and time-consuming for large models.

6.2.2 Accelerations of Individual Operations in Transformers

6.2.2.1 Acceleration on MHA Layer

A^3 [250] first introduced an approximate candidate generation method to reduce the amount of computation in the attention mechanism.

ELSA [251] adopted an approximate self-attention mechanism to reduce the number of computations. Besides filtering out the unimportant tokens, it also introduced hash values and Hamming distances to compute the approximate similarity between a query and each key instead of the dot product.

Although the above works have achieved significant performance improvement on MHA layers, they cannot accelerate the entire workload of Transformer models with optimized dataflow.

6.2.2.2 Acceleration on GELU

GELU is introduced in [73], whose original equation is Eq. 6.1 and can be approximated with Eq. 6.2. From [73], the error function can be approximated as a sigmoid function, as shown in Eq. 6.3. In FasterTransformer [92], GELU is implemented as Eq. 6.2, replacing $\sqrt{\frac{2}{\pi}}$ with a constant value.

$$GELU(x) = \frac{x}{2} \times [1 + erf(\frac{x}{\sqrt{2}})] \tag{6.1}$$

$$\approx \frac{x}{2} \times (1 + \tanh(\sqrt{\frac{2}{\pi}} \times (x + 0.044715 \times x^3))) \tag{6.2}$$

$$\approx x \cdot \sigma(1.702x) \tag{6.3}$$

where σ is the sigmoid function, and erf is the error function, as shown in Eq. 6.4.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x exp(-t^2)dt \tag{6.4}$$

In I-BERT [259], Kim *et al.* proposed I-GELU, which approximates the error function by a second-order polynomial.

Marchisio *et al.* [254] extended this work from GPU to ASIC.

In I-ViT [109], ShiftGELU is proposed based on Eq. 6.3, replacing the sigmoid function with ShiftExp and IntDiv. In [264, 265], GELU is approximated as a whole via Piecewise Linear approximation.

6.2.2.3 Acceleration on Softmax

Softmax is a general operation to predict the class probability in many tasks. Its equation is:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{i=1}^{C} e^{x_i}} \tag{6.5}$$

$$= \frac{e^{x_i - (x)_{\max}}}{\sum_{i=1}^{C} e^{x_i - (x)_{\max}}} \tag{6.6}$$

where x_i denotes the i-th element of the input vector $\mathbf{x}$. C denotes the number of classes. Softmax involves expensive non-linear operations such as exponential operations and divisions, and its computational cost is expensive when dealing with numerous classes and the bit width of the input data is large.

The hardware acceleration of the Softmax algorithm can be classified into three categories:

A. **Direct optimization**: optimize the hardware implementation of exponential and division units directly [264, 266, 267].

B. **Mathematical transformation**: Apply mathematical transforming (*e.g.*, logarithmic transforming) to replace the exponential operations [266].

C. **Mathematical reformulation**: Reformulate the de-facto Softmax equation into other hardware-friendly equations, such as replacing the exponent base e to 2 [268], new Softmax layer based on Integral SC [269].

Yuan [266], for the first time, proposed an efficient hardware implementation of Softmax that uses a logarithmic transformation to eliminate division operations.

In addition to approximating e^{x_i} directly based on LUT (LUT-EXP), Geng *et al.* [267] also proposed another approximation technique using a Piecewise Linear Function (LUT-PLF) as shown in Eq. 6.7. LUTs store α^s, x_l^s, and pre-calculated $y_l^s - \alpha^s * x_l^s$, used to approximate e^{x_i} with one multiply and one addition. Bit shifts approximate multiplication and division. Geng *et al.* proposed six Softmax operation combinations, including exponential functions, power-of-twos, and look-up tables.

$$\begin{gathered} f^s(x) = \alpha^s * (x - x_l^s) + y_l^s = \alpha^s * x + (y_l^s - \alpha^s * x_l^s) \\ x \in [x_l^s, x_r^s], \quad y_l^s = e^{x_l^s}, \quad s \in [1, S] \end{gathered} \tag{6.7}$$

Yu *et al.* [264], Lu *et al.* [265], Ham *et al.* [250, 251], and Wang *et al.* [252] also proposed similar LUT-PLF approximation methods to accelerate the Softmax operation.

In [270], [255], [268], and [271], the exponential function is optimized from base exponent to base 2.

Hu *et al.* [269] reformulated the softmax layer based on integral stochastic computing (ISC) to implement the exponent operations and a logarithmic transformation to avoid the division operations.

Kim *et al.* [259] proposed I-EXP, which approximates the exponential function by a second-order polynomial and a bit shift operation.

6.2.2.4 Acceleration on LayerNorm

LayerNorm is a normalization operation that is widely used in transformer models. During inference, it requires the on-the-fly computation of the mean and variance of the input data across the channel dimension, as shown in Eq. 6.9 and Eq. 6.10 or Eq. 6.11 respectively.

$$\text{LayerNorm}(x_i) = \frac{x_i - \mu}{\sigma} \tag{6.8}$$

where:

$$\mu = \frac{1}{C} \times \sum_{i=1}^{C} x_i \tag{6.9}$$

$$\sigma = \sqrt{\frac{1}{C} \times \sum_{i=1}^{C} (x_i - \mu)^2} \tag{6.10}$$

$$= \sqrt{\frac{1}{C} \times \sum_{i=1}^{C} x_i^2 - \mu^2} \tag{6.11}$$

Eq. 6.10 and Eq. 6.11 are two different ways to compute the variance. In [261], the authors proposed to compute the variances of the input data in a parallel manner on GPUs, as formulated in Eq. 6.11, to increase the instruction execution efficiency and reduce half of the synchronizations.

I-SQRT is proposed in [259] to approximate the square root function via the iterative algorithm proposed in [262]. In I-LayerNorm [109], the square root function is computed via bit-shifting based on the integer iterative algorithm.

6.3 ViTA Architecture

6.3.1 ViTA Architecture and Bus Overview

As illustrated in Fig. 6.2, ViTA contains several kernels for supporting all the operations in the ViT transformer, and the kernels are connected by the buses.

The input bus is used to unicast the input data of the current decoder layer to every ViTA kernel. The weight bus and the bias bus are used to broadcast the weight and bias to every ViTA kernel respectively. The output bus is used to collect the output data of every ViTA kernel and send it to the next decoder layer. The key and value bus is used to collect and broadcast the key and value data to every ViTA kernel.

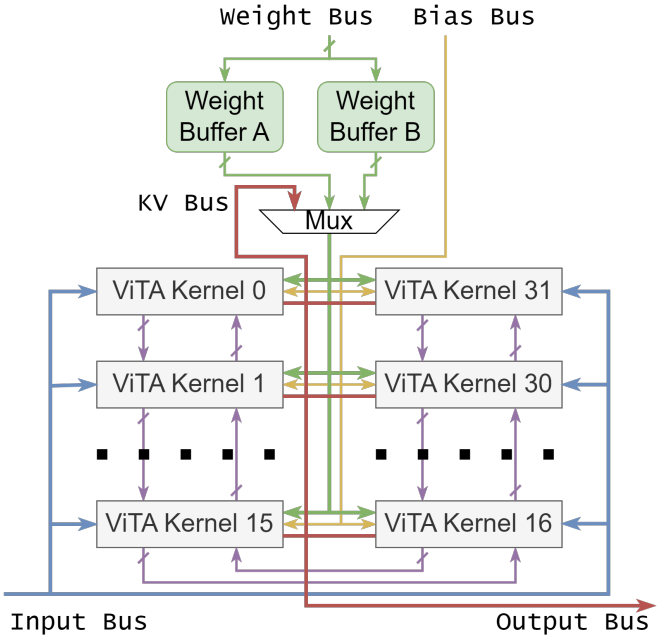


FIGURE 6.2: ViTA architecture overview. There are several buses in ViTA, *i.e.*, input bus (in blue), weight bus (in green), output bus (in red), and bias bus (in yellow). The output bus is also the key and value bus, which is multiplexed with the weight from the weight buffer. ViTA kernels are asynchronous. To compute LayerNorm, $\sum x_i$, and $\sum x_i^2$ are accumulated and updated across the ViTA kernels via the purple bus.

There are two weight buffers in ViTA that act as a ping-pong buffer, whose data are multiplexed to the key and value bus and further broadcast to every ViTA kernel.

6.3.2 ViTA Kernel

As illustrated in Fig. 6.3, each ViTA kernel consists of a VMU Lane, three matrix buffers, an accumulator, two partial sum buffers that act as a ping-pong buffer, and a special function module. The input data, *e.g.*, input matrix, KV matrix, weight matrix, and bias are synchronized to the ViTA kernel through asynchronous FIFOs.

6.3.3 VMU Lane

Temporally, the VMU Lane computes a single GEMV operation, which is illustrated in Fig. 6.4.

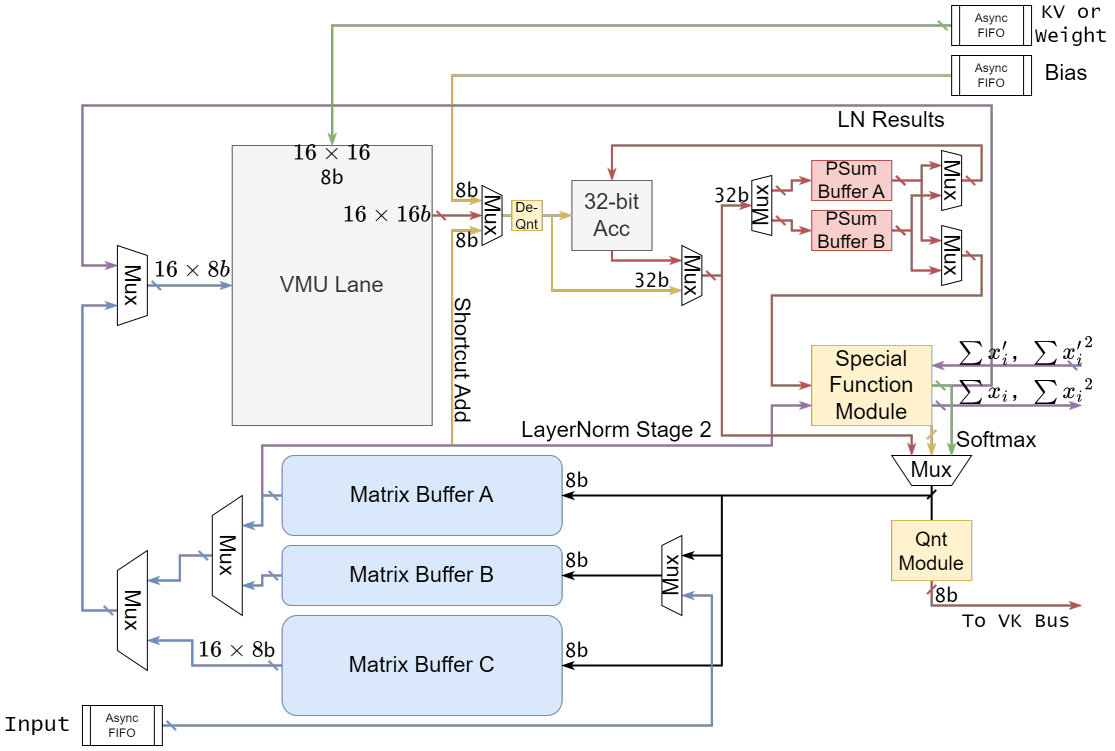


FIGURE 6.3: ViTA kernel overview. Input data, weight data, Key and Value data, and bias data that cross a clock domain crossing are passed into the ViTA kernel via the asynchronous FIFOs.

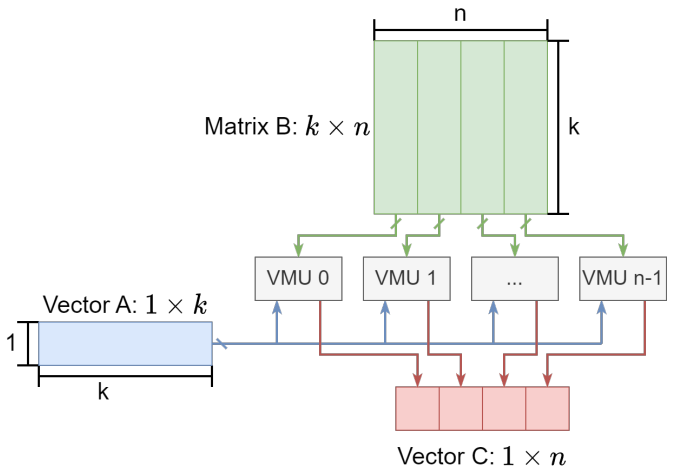


FIGURE 6.4: Temporally, a VMU Lane computes a $c = a \times b$ GEMV operation, where a is a $1 \times k$ vector, and b is a $k \times n$ matrix. The matrix b is mapped into n VMUs.

Each VMU Lane consists of n Vector-Multiplication Units (VMUs).

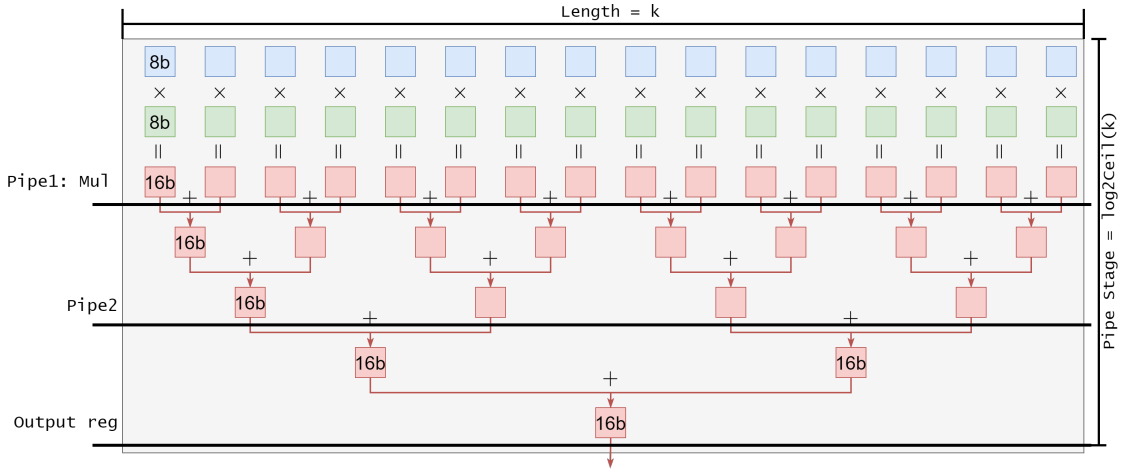


FIGURE 6.5: The computation data path of VMU, that the output is the inner project of two input vectors. The left column indicates the bit width of each value. **All the values are signed**.

TABLE 6.3: The function of the PLF units and adder trees in different non-linear function modes.

Mode	PLF Unit 0-3	PLF Unit 4	Adder Tree 0	Adder Tree 1
Softmax	$f(x) = e^x$	$f(x) = 1/x$	–	$\sum e^{x_i}$
LayerNorm	–	$f(x) = 1/\sqrt{x}$	$\sum {x_i}^2$	$\sum x_i$
GELU	$f(x) = \text{GELU}(x)$	–	–	–

6.3.3.1 Vector-Multiplication Unit (VMU)

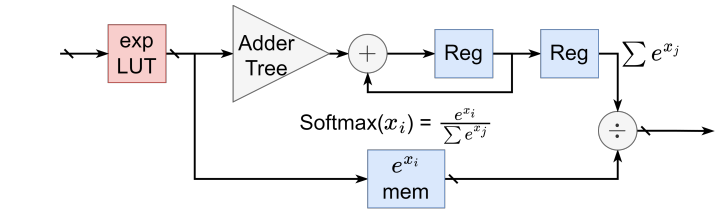
Fig. 6.5 illustrates the data path of the VMU, which outputs the inner product result of two vectors, whose length is k.

6.3.4 Special Function Module

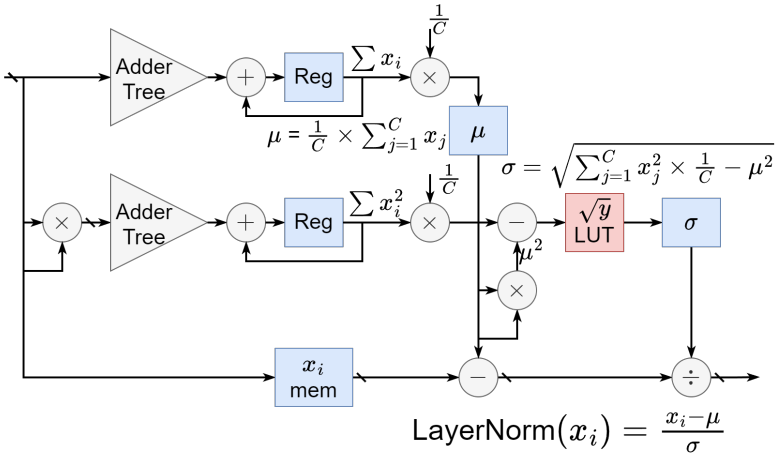
Non-linear functions in the ViT transformer, *i.e.*, GELU, Softmax, and LayerNorm are computed by the Special function module, which is illustrated in Fig. 6.7. The usages of the PLF units and two adder trees in different non-linear function modes are summarized in Table 6.3. The PLF unit is adopted from [264, 267].

6.3.4.1 Softmax

The data paths used only in the Softmax mode are green arrows in Fig. 6.7.



(A)



(B)

FIGURE 6.6: (a) Normal dedicated Softmax and (b) LayerNorm modules.

ViTA computes the Softmax, as formulated in Eq. 6.5, with the approximated exponential function and reciprocal function.

The exponential values e^{x_i} are computed by the PLF Units 0 to 3, and they are stored in the exponential buffer for further use. Meanwhile, the exponential values are also summed up by the adder tree and the accumulator to compute the sum of the exponential values $\sum e^{x_i}$.

Instead of using several expensive dividers, learned from Table 6.1, the vector division is converted to one reciprocal function and one relatively cheaper vector multiplication. After the whole row is processed, the reciprocal of the sum of the exponential values $\frac{1}{\sum e^{x_i}}$ is computed by PLF Units 4, which acts as the reciprocal function. Then the reciprocal of the sum of the exponential values $\frac{1}{\sum e^{x_i}}$ is multiplied by the exponential values e^{x_i}, which are read from the exponential buffer, to compute the Softmax values $\frac{e^{x_i}}{\sum e^{x_i}}$.

As the Softmax is applied row-wise, the corresponding GEMM dataflow should also be row-wise, to reduce the memory requirement of the exponential buffer. As

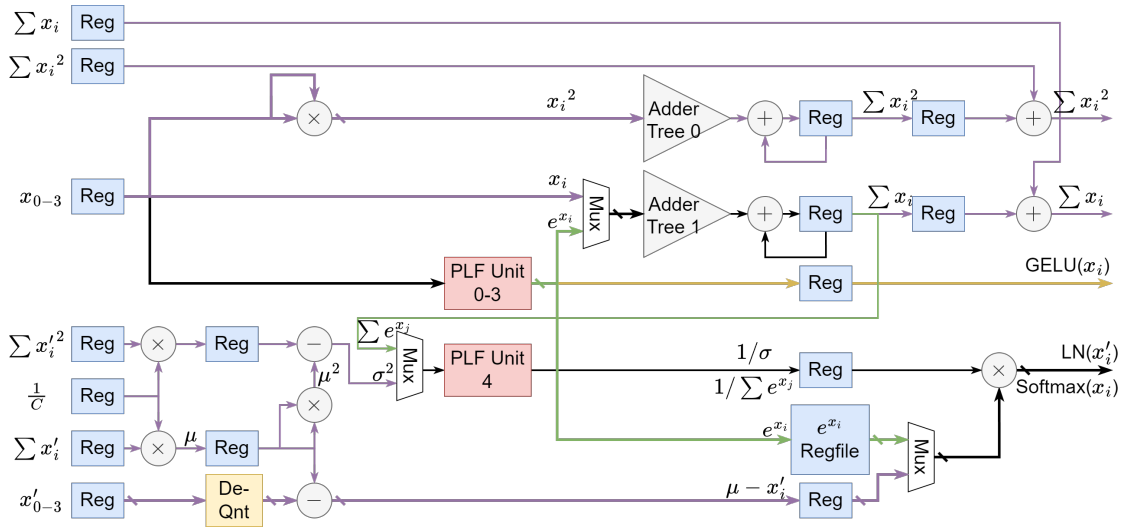


FIGURE 6.7: The computation data path of the Special function module, which is used to compute the non-linear functions, including GELU, Softmax, and LayerNorm. The GELU function, exponential function, reciprocal function, and the reciprocal of the square root function are approximated by the PLF unit. The yellow arrows indicate the data path used only for the GELU, the green arrows indicate the data path used only for the Softmax, and the purple arrows indicate the data path used only for the LayerNorm. The shared data paths are the black arrows.

shown in Table 6.4, the GEMM dataflow is MKN when the Softmax is applied.

6.3.4.2 LayerNorm

The data paths used only in the LayerNorm mode are purple arrows in Fig. 6.7.

ViTA computes the LayerNorm in two stages to reduce the memory requirement of the middle results. In the first stage, the mean μ and the variance σ are computed, while the input x_i is written back to the matrix buffer A after quantization. In the second stage, the input x_i is read from the matrix buffer A after dequantization, then the layer normalization results are computed following Eq. 6.8 with the mean μ and the variance σ computed in the first stage.

There are two formulas to compute the variance σ in Eq. 6.8, which are formulated in Eq. 6.10 and Eq. 6.11. To reduce the latency and improve the throughput, ViTA adopts Eq. 6.11 to compute the variance σ. In this case, the sum of the input x_i and the sum of the squares of the input x_i^2 are computed simultaneously.

However, as the layer normalization is computed across the whole matrix while each ViTA kernel only computes a chunk of the matrix, the sum of the input x_i and the sum of the squares of the input x_i^2 are accumulated across the ViTA kernels for the final result.

As the LayerNorm is computed in two stages, there is no special constraint on the GEMM dataflow.

6.3.4.3 GELU

The data paths used only in the GELU mode are highlighted with yellow arrows in Fig. 6.7.

As summarized in Table 6.4, the GELU function and the second stage of the LayerNorm function are computed simultaneously in the Special function module.

To compute the GELU function, the input data vector is sent to the PLF Units 0 to 3, which act as the GELU function. Then the output from the PLF Units 0 to 3 is output.

As the GELU function is applied element-wise, there is no special constraint on the GEMM dataflow.

6.4 ViTA Dataflows

ViTA supports multiple dataflows to reduce the memory bandwidth and capability requirements. Row Stationary (RS) and Column Stationary (CS) dataflows are adopted for General Matrix Multiply (GEMM) operations. Meanwhile, two Output Stationary (OS) dataflows are introduced to compute special functions while decreasing throughput requirements. A fused multi-head attention dataflow is presented for further optimization. Table 6.4 summarizes the dataflows of each ViT layer, including the name of ViT operations, the corresponding data source and destination of the GEMM computation, the GEMM dataflow, and the special functions before and after the GEMM operation.

TABLE 6.4: Summary of ViTA dataflows, including the name of ViT operations, the corresponding data source and destination of the GEMM computation, the GEMM dataflow, the special functions before and after the GEMM operation, and the dataflow in the view of the special functions.

Layer	Operation		Matrix A	Matrix B	Post Special Function	GEMM Dataflow[a]	(Pre) Special Function	Special Function Dataflow[a]	Matrix C
Input	–		–	–	–	–	–	–	MB B
Pre-Processing	Pre-Processing	Patches embedding	MB[b]B	WB[c]A/B	–	NKM	LN Stage 1	NMK	MB A
Encoder	MHA	$[Q, K, V]$	MB A	WB A/B	LN Stage 2	MKN	–	MNK	MB B/C[d]
		$A = \text{Softmax}(Q \cdot K)$	MB B	MB C	–	MKN	Softmax	MNK	MB A
		$Z = A \cdot V$	MB A	MB C	–	MKN	–	MNK	MB B
		Project GEMM	MB B	WB A/B	–	NKM	LN Stage 1	NMK	MB A
	FFN	FC 1	MB A	WB A/B	LN Stage 2	NKM	GELU	NMK	MB B&C
		FC 2	MB B&C	WB A/B	–	NKM	–	NMK	MB A
Final Classifier	MLP- Head		MB A	WB A/B	–	NKM	–	NMK	–

[a] M, N, and K are the number of rows and columns of the matrix C and the number of columns of the matrix A respectively, and the sequence of M, N, and K represent the order of the for-loop in the GEMM operation, from outer to inner. Fig. 2.12 illustrates the MNK dataflow as an example.
[b] MB is short for Matrix Buffer.
[c] WB is short for Weight Buffer. Weight Buffers A and B are ping-pong buffers for the weight.
[d] The Query matrix is stored in Matrix Buffer B, and the Key matrix and Value matrix are stored in Matrix Buffer C.

6.4.1 GEMM and Special Functions Dataflows in ViTA

6.4.1.1 GEMM Dataflow

GEMM is a foundational operation in Transformer models, responsible for matrix multiplications and additions. A GEMM operation is formulated as:

$$C_{M\times N} = A_{M\times K} \times B_{K\times N} \tag{6.12}$$

where M, N, and K are the number of rows and columns of the matrix C and the number of columns of the matrix A respectively.

There are six ways to implement it in software and hardware using three nested for loops, *i.e.*, MNK, NMK, KMN, MKN, KNM, and NKM. In the above six categories, the sequence of M, N, and K represent the order of the for-loop in the GEMM operation, from outer to inner. Fig. 2.12 illustrates the MNK dataflow as an example.

As summarized in Table 6.5, different permutations impact both performance and energy efficiency with different access patterns for A and B. ViTA supports both MKN, *i.e.*, RS and NKM, *i.e.*, CS dataflows for higher energy efficiency and less memory footprint [146].

Tiling is adopted for large-scale GEMM operations, which is illustrated in Fig. 6.8. Each GEMM operation is distributed across multiple ViTA kernels to enhance

TABLE 6.5: The summary of dataflows, optimum conditions, memory and bandwidth bottlenecks in GEMM, the throughput requirements of special functions, and whether they are supported in ViTA.

Dataflow[a]	**Optimum Conditions**[b]	**In the View of GEMM**		**In the view of Special Functions**	**Supported in ViTA**
		Memory Bottleneck	***Bandwidth Bottleneck***	***Throughput***[c]	
MNK-OS	$M \geq N$	Matrix B	Matrix A, B	n / K/k cycle	✓
NMK-OS	$M < N$	Matrix A	Matrix A, B	n / K/k cycle	✓
KMN-IS	$M \geq N$	Partial Sum Buffer	Matrix B	n / cycle	✗
MKN-RS	$M \geq N$	Matrix B	Matrix B	n / cycle	✓
KNM-WS	$M < N$	Partial Sum Buffer	Matrix A	n / cycle	✗
NKM-CS	$M < N$	Matrix A	Matrix A	n / cycle	✓

[a] OS is short for Output Stationary; IS is short for Input Stationary; RS is short for Row Stationary; WS is short for Weight Stationary; CS is short for Column Stationary.
[b] M and N are the number of rows and columns of the matrix C respectively.
[c] n and k are the computation parallelism of the hardware on N and K dimensions respectively.

parallelism and performance. More specifically, temporally, the ViTA computes a $c = a \times b$ GEMM operation, where a is an $m \times k$ matrix, and b is a $k \times n$ matrix. The GEMM operation is further mapped to m ViTA kernels, where each kernel computes a $c = a \times b$ GEMV operation, where a is a $1 \times k$ vector, and b is a $k \times n$ matrix.

6.4.1.2 Special Function Dataflow

Non-linear operations, critical to Transformer computations, are resource-intensive, occupying 60% of the memory usage in GPUs [249] and 45% area of the accelerator [254]. Therefore, the dataflow in the view of special functions should be carefully designed to improve the power efficiency and area efficiency of the special functions.

Two OS dataflows, *i.e.*, MNK and NMK, have been adopted in the special function module of ViTA to reduce the throughput requirements of the special functions from n per cycle to $n/(K/k)$ per cycle, as summarized in Table 6.5.

To convert the GEMM dataflow to the corresponding special function dataflow, two partial sum buffers are introduced in ViTA, acting as a ping-pong buffer.

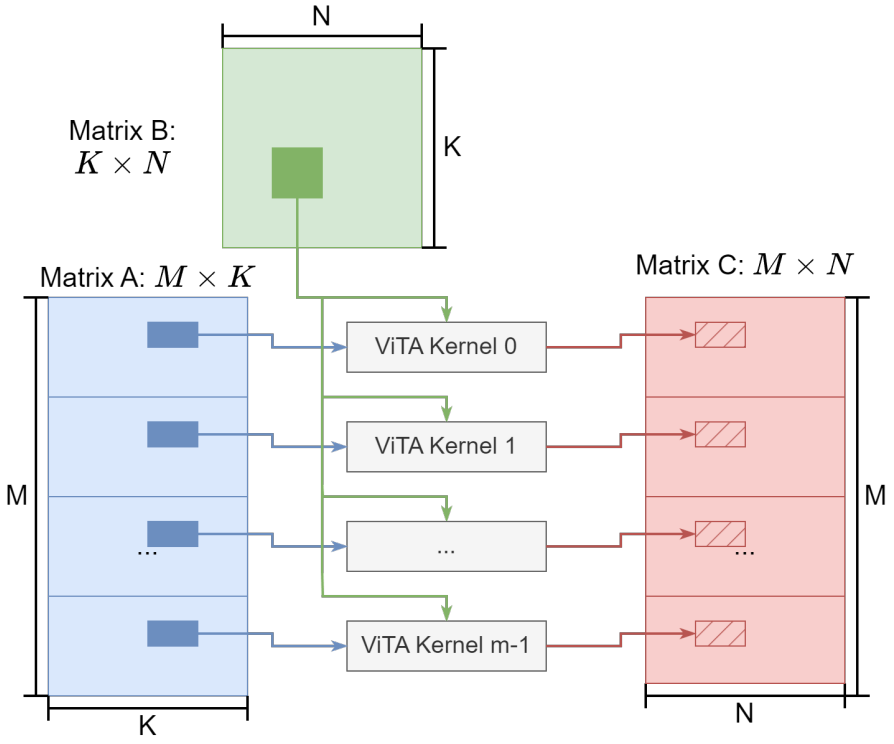


FIGURE 6.8: ViTA uses tiling for large-scale GEMM operations. For every GEMM, the matrix A ($\mathbb{R}^{M \times K}$) is split into m chunks, which are mapped into m ViTA kernels. Temporally, the ViTA computes a $c = a \times b$ GEMM operation, where $a \in \mathbb{R}^{m \times k}$, $b \in \mathbb{R}^{k \times n}$.

6.4.2 Fused Multi-Head Attention Dataflow

A memory-centric, hardware-efficient fused Multi-Head Attention dataflow is proposed in ViTA to reduce the memory capability requirements across the Multi-Head Attention layer.

In this dataflow, MHA is computed per head, and memories are reused during the computation of every self-attention. Algorithm 1 shows the fused multi-head attention dataflow from the input matrix X to the input of the MLP layers, which is illustrated in Fig. 6.9. The function **PerHeadQKV** is shwon in Fig. 6.10, and the functions **PerHeadA** and **PerHeadZ** are illustrated in Fig. 6.11, respectively.

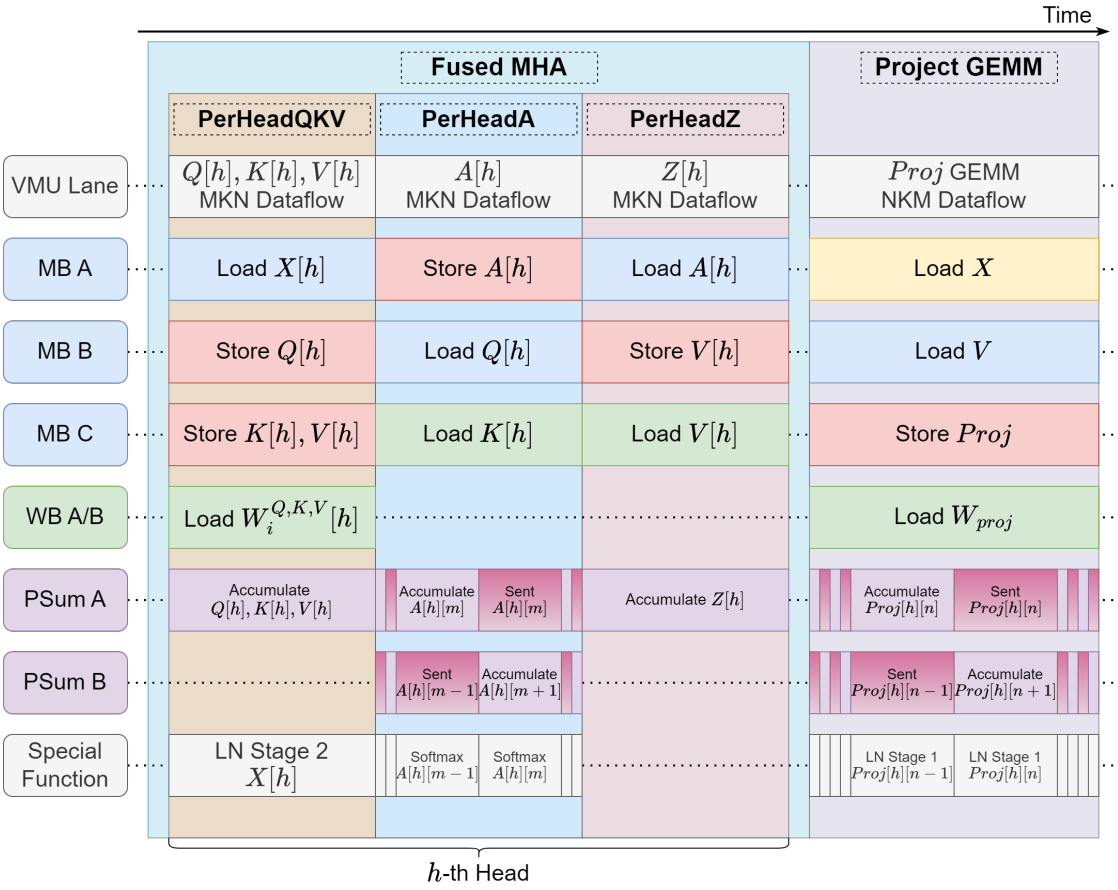


FIGURE 6.9: The fused multi-head attention dataflow in the view of each module, where MB is short for Matrix Buffer, WB is short for Weight Buffer, PSum A and B are Partial Sum Buffer A and B, which are acting as a ping-pong buffer. The GEMMs in **PerHeadQKV**, **PerHeadA**, **PerHeadZ** are MKN dataflow, while the GEMM in **Project GEMM** is NKM dataflow.

6.4.3 Gains of the ViTA Dataflows

Fig. 6.1 demonstrates that the original MHA that supports VGA images requires 22.66 MB when dedicated memories are used for each matrix. With our fused multi-head attention, this requirement drops to 5.28 MB, marking a 76.71% reduction. This reduction translates to the potential for deploying larger or more complex models on constrained hardware. In the MLP layers, while FC 1 results require 3.52 MB, the weights demand 2.25 MB. By adopting dual GEMM dataflows, 48 KB is needed for the weight buffers. Such savings increase both area efficiency and power efficiency.

Algorithm 7: Fused Multi-Head Attention Algorithm

Input: Number of heads H
Input: Matrix $X \in \mathbb{R}^{(N+1)\times D}$ stored in MB A
Input: $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{D\times D_h}$
Input: $W_{proj} \in \mathbb{R}^{H\times D_h\times D}$
Input: $Bias_Q, Bias_K, Bias_V, Bias_{proj}$
Let m, k and n be the computation parallelism of the hardware on the M, K, and N dimensions respectively;
$T_h = H$;
$T_m = ceil((N+1)/m)$;
$T_k = ceil(D/(H \cdot k))$;
$T_n = ceil(D_h/n)$;
Tile X into $T_h \times T_m \times T_k$ blocks of size $m \times k$ each, and tile W_i^Q, W_i^K, W_i^V into $T_h \times T_k \times T_n$ blocks of size $k \times n$ each;
for $1 \le t_h \le T_h$ **do**
    `PerHeadQKV(`H, t_h, $X[t_h]$, $W_i^Q[t_h]$, $W_i^K[t_h]$, $W_i^V[t_h]$, $Bias_Q[t_h]$, $Bias_K[t_h]$, $Bias_V[t_h]$`);`
    `PerHeadA(`H, t_h, $Q[t_h]$, $K[t_h]$`);`
    `PerHeadZ(`H, t_h, $A[t_h]$, $V[t_h]$`);`
`ProjGEMM(`H, X, Z, W_{proj}, $Bias_{proj}$`);`

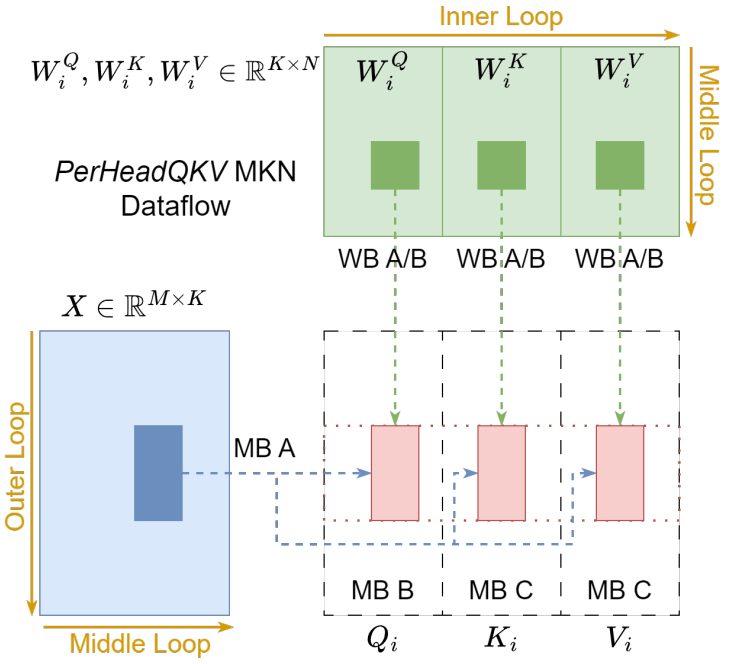


FIGURE 6.10: The dataflow of the **PerHeadQKV** function in Algorithm 1.

Algorithm 8: Project GEMM And Shortcut Add

Function `ProjGEMM` **is**

Input: Number of heads H
Input: Matrix $X \in \mathbb{R}^{(N+1)\times D}$ stored in MB A
Input: $Z \in \mathbb{R}^{H\times(N+1)\times D_h}$ stored in MB B
Input: $W_{proj} \in \mathbb{R}^{H\times D_h\times D}$
Input: $Bias_{proj}$
$T_h = H$;
$T_m = ceil((N+1)/m)$;
$T_k = ceil(D/(H \cdot k))$;
$T_n = ceil(D_h/n)$;
for $1 \le t_h \le T_h$ **do**
  for $1 \le t_n \le T_n$ **do**
    for $1 \le t_k \le T_k$ **do**
      Load $W_{proj}[t_h][t_k][t_n]$ from WB A or B;
      for $1 \le t_m \le T_m$ **do**
        Load $Z[t_h][t_m][t_k]$ from MB B;
        $Proj[t_h][t_m][t_n] += Z[t_h][t_m][t_k] \times W_{proj}[t_h][t_k][t_n]$;
    `/* Shortcut add */`
    for $1 \le t_m \le T_m$ **do**
      Load $X[t_h][t_m][t_n]$ from MB A;
      $Proj[t_h][t_m][t_n] += X[t_h][t_m][t_n]$;
    Send $Proj[t_h][t_m]$ to the special function module for LayerNorm Stage 1;
    Write $Proj[t_h][t_m]$ to MB A;

6.5 Evaluation

6.5.1 Experimental Setup

We implement our ViTA in Chisel [201] and evaluate it for ViT-base with 224×224 RGB images as input. Memory IPs (Regfiles and SRAMs) are generated by ARM memory compilers. The generated Verilog from Chisel and memory IPs are synthesized by Cadence Genus with Foundry 28 nm FD-SOI technology node for both area and power estimation. The experimental setup is summarized in Table 6.6.

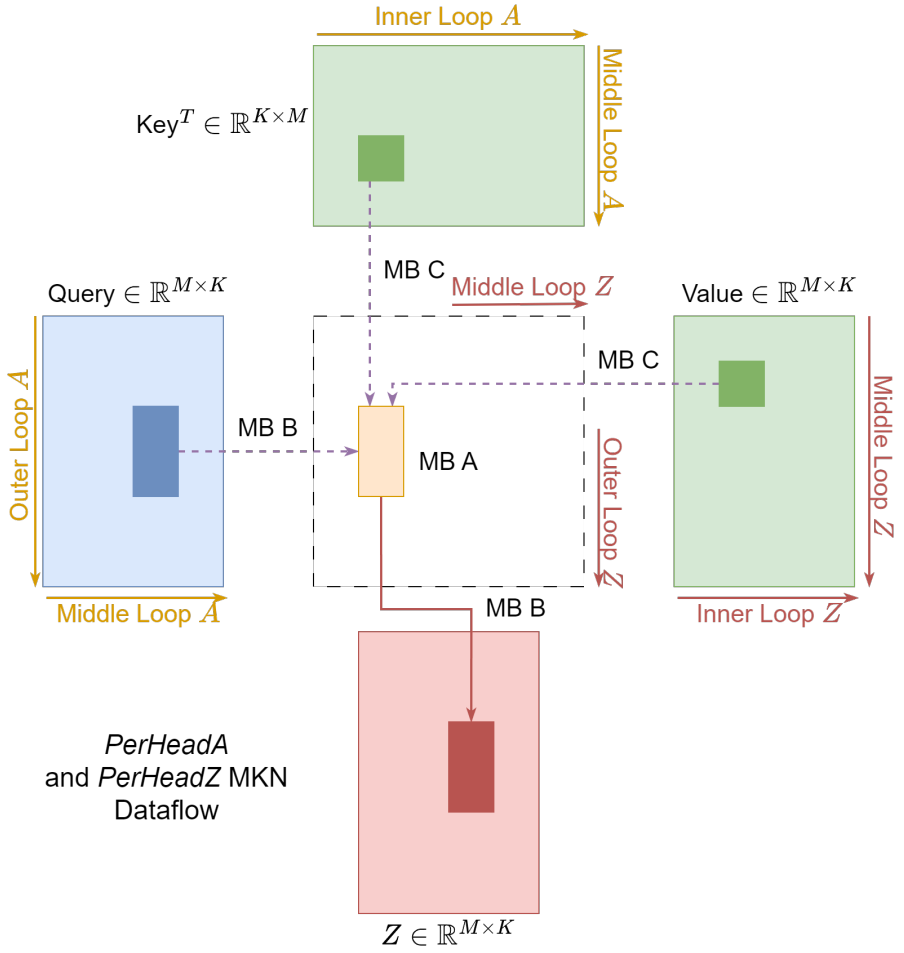


FIGURE 6.11: The dataflow of the **PerHeadA** and **PerHeadZ** functions in Algorithm 1.

TABLE 6.6: Experimental setup

Design Tools	
HDL	Chisel
Verilog simulator	Verilator [233]
Logic synthesis	Cadence Genus
Design Parameters	
VMU length	8/16
# VMU lanes	8/16/32
GEMM input bit width	8 bit
GEMM output bit width	32 bit
Image resolution	224×224
Technology node	Foundry 28 nm FD-SOI
Core supply voltage	1.05 V

TABLE 6.7: The experimental results of a ViTA Huge at different clock frequencies, supporting a 224×224 RGB image as input.

ViTA Huge ($m = 32$, $k = n = 16$)	**200 MHz**		**300 MHz**		**500 MHz**		**1 GHz**	
	Area ***(mm^2)***	***Power*** ***(mW)***	***Area*** ***(mm^2)***	***Power*** ***(mW)***	***Area*** ***(mm^2)***	***Power*** ***(mW)***	***Area*** ***(mm^2)***	***Power*** ***(mW)***
ViTA Kernel								
VMU Lane (256 MAC)	0.0597	22.18	0.0696	37.86	0.0718	65.87	0.0906	158.08
Matrix Buffer A (9.25 KB)	0.0223	3.02	0.0223	4.52	0.0223	7.41	0.0223	14.91
Matrix Buffer B (4.625 KB)	0.0117	1.96	0.0117	2.94	0.0117	4.78	0.0117	9.64
Matrix Buffer C (13.875 KB)	0.0308	3.19	0.0308	4.77	0.0308	7.81	0.0308	15.71
PSumBuffer 2× ***(1 KB*** 2×***)***	0.0293	12.8141	0.0293	19.1992	0.0293	31.9688	0.0293	63.8798
Vector Accumulator	0.0017	0.82	0.0017	1.23	0.0026	2.74	0.0033	7.11
Special Function Module	0.0463	8.96	0.0461	13.32	0.0487	23.66	0.0516	55.16
Top								
ViTA Kernels (32×)	6.46	1,694.37	6.77	2,682.80	6.95	4,616.01	7.67	10,383.70
Weight Buffer 2× ***(48 KB*** 2×***)***	0.02	3.90	0.02	5.84	0.02	9.73	0.02	19.43
Summary								
ViTA	6.48	1,698.27	6.79	2,688.65	6.97	4,625.73	7.69	10,403.13
GOPs	3,276.80		4,915.20		8,192.00		16,384.00	
GOPS/mm^2 or GOPS/W	505.95	1,929.49	724.05	1,828.13	1,175.67	1,770.96	2,131.85	1,574.91
Frame/s[a]	104.483		156.724		261.207		522.414	
Latency (ms)[a]	9.571		6.381		3.828		1.914	

[a] The latencies and frame rates are calculated with the input image resolution of 224×224 on ViT-Base.

6.5.2 Experimental Results

We evaluated the performance (GOPS, throughput, and latency), area, power, area efficiency, and power efficiency of ViTA at different clock frequencies and hardware configurations.

6.5.2.1 ViTA at Different Clock Frequencies

We synthesized ViTA at different clock frequencies, *i.e.*, 200 MHz, 300 MHz, 500 MHz, and 1 GHz, with the same hardware configuration, *i.e.*, $m = 32$, $k = n = 16$, and the supported input image resolution is 224×224, which are summarized in Table 6.7.

ViTA achieves a relatively high area efficiency of 2,131.85 GOPS/mm^2 at 1 GHz and achieves a relatively high power efficiency of 1,929.49 GOPS/W at 200 MHz.

TABLE 6.8: The experimental results of ViTAs with different computation capabilities, supporting a 224×224 RGB image as input.

ViTA @ 300 MHz	ViTA Tiny ($m = k = n = 8$)		ViTA Base ($m = 16$, $k = n = 8$)		ViTA Large ($m = k = n = 16$)		ViTA Huge ($m = 32$, $k = n = 16$)	
	Area ***(mm^2)***	***Power*** ***(mW)***	***Area*** ***(mm^2)***	***Power*** ***(mW)***	***Area*** ***(mm^2)***	***Power*** ***(mW)***	***Area*** ***(mm^2)***	***Power*** ***(mW)***
ViTA Kernel								
VMU Lane ***($k \cdot n$ MAC)***	0.0172	9.4273	0.0172	9.4514	0.0693	37.7591	0.0696	37.8605
Matrix Buffer A	0.0657	4.4080	0.0338	4.3108	0.0448	8.0180	0.0223	4.5165
Matrix Buffer B	0.0338	4.3030	0.0201	3.8947	0.0223	4.5016	0.0117	2.9386
Matrix Buffer C	0.0927	4.4937	0.0472	4.3571	0.0583	8.5747	0.0308	4.7721
PSumBuffer 2×	0.0169	9.8175	0.0169	9.8136	0.0293	19.2023	0.0293	19.1992
Vector ***Accumulator***	0.0009	0.6157	0.0009	0.6152	0.0017	1.2309	0.0017	1.2341
Special Function ***Module***	0.0211	7.1494	0.0210	7.1427	0.0461	13.3097	0.0461	13.3167
Top								
ViTA Kernels ($m\times$)	1.99	321.72	2.51	633.37	4.35	1,481.54	6.77	2,682.80
Weight Buffer 2×	0.02	3.73	0.02	3.73	0.02	5.84	0.02	5.84
Summary								
ViTA	2.00	325.45	2.53	637.10	4.37	1,487.38	6.79	2,688.65
GOPs	307.20		614.40		2,457.60		4,915.20	
GOPS/mm^2 or GOPS/W	153.45	943.94	242.99	964.37	562.84	1,652.30	724.05	1,828.13
Frame/s[a]	11.006		21.158		84.451		156.724	
Latency (ms)[a]	90.856		47.264		11.841		6.381	

[a] The latencies and frame rates are calculated with the input image resolution of 224×224 on ViT-Base.

6.5.2.2 ViTA with Different Hardware Configurations

We synthesized ViTA in different hardware configurations, *i.e.*, $m = k = n = 8$, $m = 16$, $k = n = 8$, $m = k = n = 16$, and $m = 32$, $k = n = 16$, with the same clock frequency, *i.e.*, 300 MHz, and the supported input image resolution is 224×224, which are summarized in Table 6.8.

The area and power of a Tiny ViTA ($m = k = n = 8$) are 2.00 mm^2 and 325.45 mW, respectively, which is 3.40× smaller than the area of a Huge ViTA ($m = 32$, $k = n = 16$) and 8.26× smaller than the power of a Huge ViTA ($m = 32$, $k = n = 16$).

6.5.3 Area and Power Breakdown

The area and power breakdown of the ViTA Kernel in ViTA Huge ($m = 32$, $k = n = 16$) and that in ViTA Tiny ($m = k = n = 8$) at 300 MHz are shown in Fig. 6.12.

As Fig. 6.12 shows, memories occupy 44% area and 38% power in the ViTA kernel in ViTA Huge, due to the efficient ViTA dataflow, achieving a high area efficiency

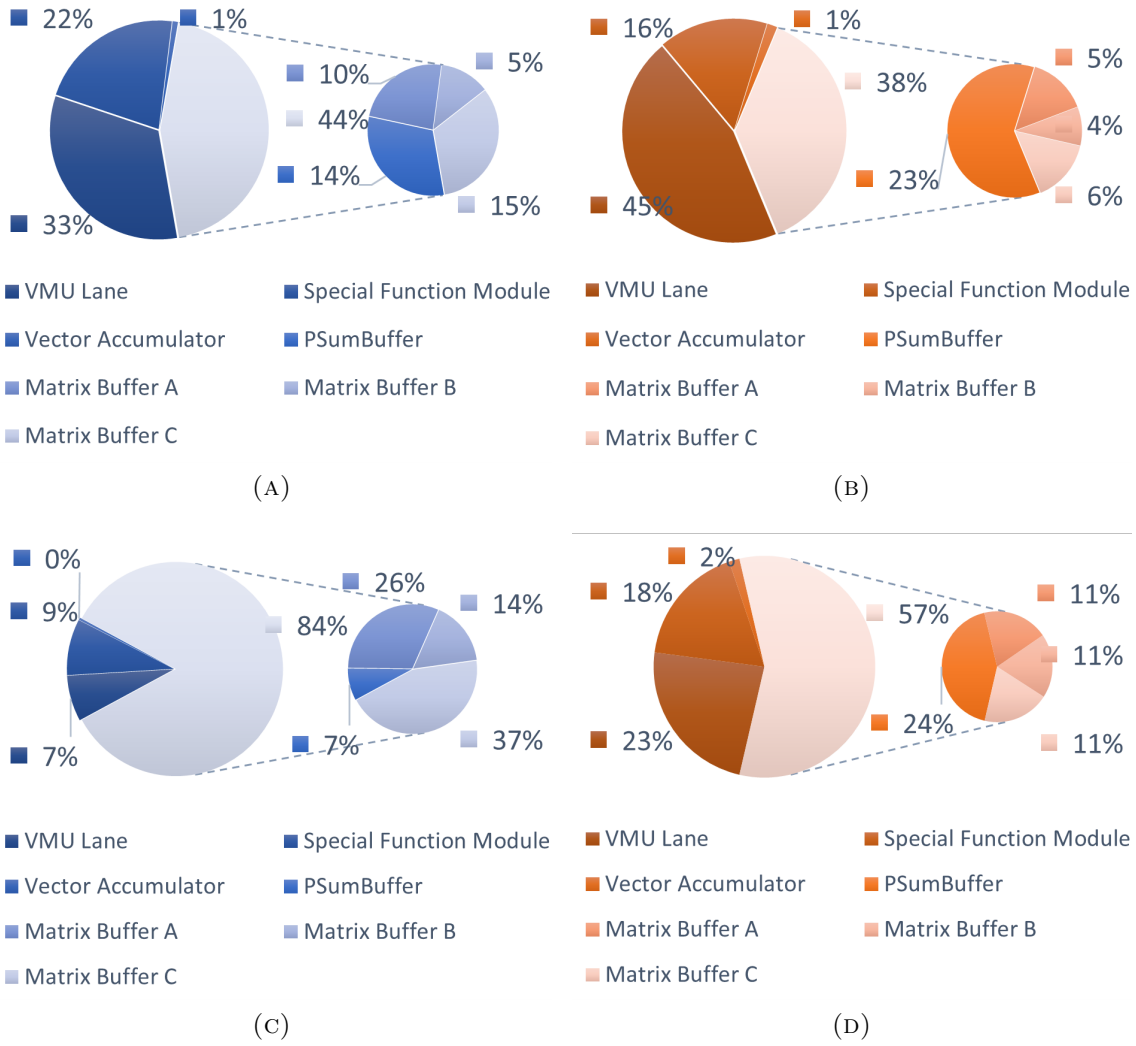


FIGURE 6.12: (a) Area and (b) power breakdown of the ViTA Kernel in ViTA Huge ($m = 32$, $k = n = 16$) at 300 MHz. (c) Area and (d) power breakdown of the ViTA Kernel in ViTA Tiny ($m = k = n = 8$) at 300 MHz.

of 724.05 GOPS/mm^2 and a high power efficiency of 1,828.13 GOPS/W at 300 MHz.

With the number of ViTA kernels decreasing, the capability of the matrix buffers increases, which leads to an increased percentage in the area and power of memories. In ViTA Tiny, memories occupy 84% area and 57% power.

Even with the optimized special function module, it occupies 22% area in ViTA Huge, which is 66% of the VMU lane.

6.5.4 Comparison with Related Works

We summarized the comparisons between the related works and our proposed ViTA architectures, ViTA Tiny at 200 MHz, and ViTA Huge at 1 GHz for area- and power-constraint applications and high-performance applications, respectively, in Table 6.9.

ViTA Huge achieves the best area efficiency of 2.13 TOPS/mm^2 at 1GHz, which is 2.46× higher than the best area efficiency of the related works ELSA [251], even though ELSA does not support the end-to-end workload.

TABLE 6.9: Summary of comparisons between the related works and our proposed ViTA architecture.

Accelerator	**Year**	**Non-Linear Functions**			**End-to-end Workload**	**Tech**	**Area**	**Power**	**Freq.**	**Perf.**	**Power Eff.**	**Area Eff.**	**Type**	**Precision**
		GELU	***Softmax***	***LayerNorm***		***nm***	***mm²***	***mW***	***MHz***	***GOPS***	***GOPS/W***	***GOPS/mm²***		
A3 [250]	2020	✗	INT9	✗	✗	40	2.08	110	1000	221	2,001.41	106.25	ASIC	INT9
ELSA [251]	2021	✗	INT8	✗	✗	40	1.26	969	1000	1,090	1,124.45	865.08	ASIC	INT8 FP16
[252]	2022	✗	INT12	✗	✗	28	6.82	273	510	522	1,913.49	76.54	ASIC	INT12
ViTA [257]	2023	INT8	✗	INT8	✗	28	–	880	150	–	–	–	FPGA	INT8
SwiftTron [254]	2023	INT32	INT32	INT32	✓	65	273	33,640	143	–	–	–	ASIC	INT8
ViTA Tiny (This Work)	2023	INT32	INT32	INT32	✓	28	2.00	217	200	204.80	943.42	102.57	ASIC	INT8
ViTA Huge (This Work)							7.69	10,403	1000	16,384	1,574.91	2,131.85		

It is clear that our architecture achieves a competitive area and power efficiency while supporting the entire ViT workload.

6.6 Chapter Summary

In this chapter, we propose ViTA, a highly efficient and scalable hardware accelerator that accelerates the entire workload of ViT with optimized dataflow and a special function module.

We present a novel memory-centric fused Multi-Head Attention dataflow that reduces the memory requirements of MHA with 224×224 and VGA images as input by 40.5% and 76.71%, from 1.45 MB to 0.87 MB and from 22.66 MB to 5.28 MB, respectively. By supporting multiple GEMM dataflows, ViTA reduces the weight buffer size from 2.25 MB to 48 KB. With a ping-pong buffering scheme on the partial sum buffer, ViTA reduces the throughput of the special functions from n per cycle to $n/K/k$ per cycle.

We also propose a configurable special function module, that supports the approximated non-linear functions, *i.e.*, Softmax, LayerNorm, and GELU, in the ViT model with reused datapath to improve the area efficiency.

Empowered by both optimized dataflows and dedicated hardware modules, ViTA achieves a high area efficiency of 2.13 TOPS/mm^2 and a high power efficiency of 1.57 TOPS/W at 1 GHz while satisfying the end-to-end computation ability of the ViT model.

We also conduct a comprehensive design space exploration of ViTA, and the results show that ViTA can be configured to achieve a wide range of computation capabilities, from 204.8 GOPS to 16.384 TOPS, with a wide range of area and power consumption, from 2.00 mm^2 to 6.79 mm^2 and from 0.217 W to 10.400 W, respectively, suitable for a wide range of applications.

Chapter 7

Full System Integration

7.1 Introduction

In previous chapters, we introduced the design of the proposed hardware acceleration on special functions, *i.e.*, Tunstall decoders for networks after entropy coding, ShapoolNMS for NMS, and end-to-end workload hardware accelerators for CNNs and Transformers. In this chapter, we will introduce the integration of the proposed hardware accelerators into the PULPissimo SoC platform [63], which is an open-source RISC-V SoC platform that contains several submodules such as RISCY core [272] and Hardware Processing Engines (HWPEs) [273]. The PULPissimo SoC platform is illustrated in Fig. 7.1.

7.2 Full System Integration of Tunstall Decoder

We integrated our Tunstall decoders which are introduced in Chapter 3 with the HWPE in the PULPissimo SoC platform, into a new module called **HWPE-Tunstall Wrapper**, which is illustrated in Fig. 7.2. The HWPE is a memory-coupled accelerator that is integrated into the PULPissimo SoC platform to perform the vector-vector dot product operation or the sum of products operation for MAC operations. It reads data from the Tightly Coupled Data Memory (TCDM)

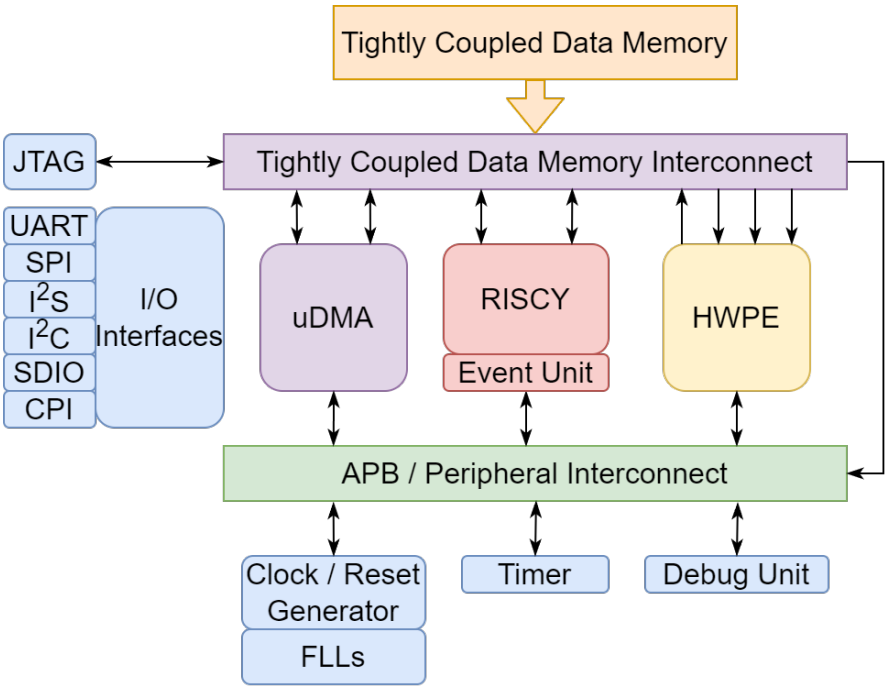


FIGURE 7.1: Overview of the PULPissimo SoC platform. Figure adopted from [63].

via the TCDM Interconnect and its control registers are mapped to the Memory-Mapped I/O (MMIO) space and access the control signals via the APB / Peripheral Interconnect.

As illustrated in Fig. 7.2, there are three input data streams and one output stream for the HWPE: (a) input activation, (b) weight, (c) bias, and (d) output activation. The Tunstall decoder is integrated with the HWPE to decode the encoded weights from the TCDM to the HWPE. The Tunstall decoder reads the codeword or uncompressed weights and the codebook from the TCDM via two input data streams and sends the uncompressed weights to the HWPE via the output data stream.

There are two modes for the HWPE-Tunstall Wrapper, *i.e.*, decoding mode and bypass mode:

- In decoding mode, the Tunstall decoder is initialized with codebooks. During computation, the Tunstall decoder fetches the compressed weights from the TCDM, while other data, *i.e.*, input activations and bias, is sent directly to HWPE from TCDM. Uncompressed weight is then propagated to the HWPE from the Tunstall decoder after decoding.

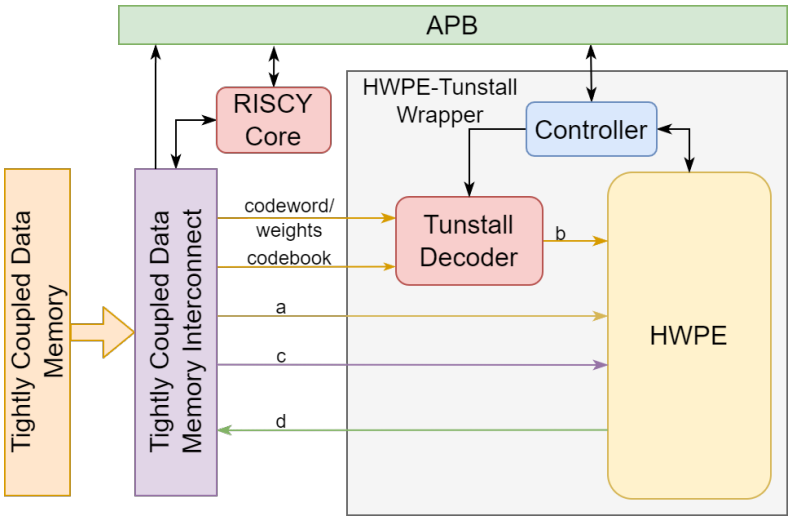


FIGURE 7.2: The introduced Tunstall decoder is integrated with the HWPE in the PULPissimo SoC platform to decode the encoded weights from the TCDM to the HWPE. The integrated module is **HWPE-Tunstall Wrapper**.

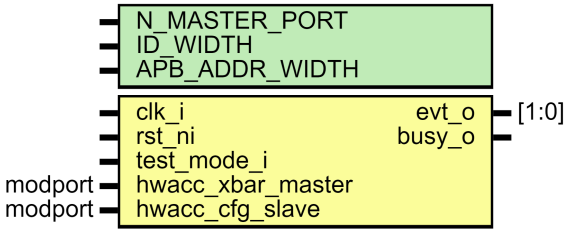


FIGURE 7.3: The interfaces of the HWPE-Tunstall Wrapper.

- In bypass mode, the uncompressed weights are directly bypassed through the Tunstall decoder to HWPE.

7.2.1 Interfaces of the HWPE-Tunstall Wrapper

We inherit the original HWPE interfaces for TCDM accessing and APB bus accessing while only requiring one additional input data stream for the codebook initialization.

As shown in Fig. 7.3, there are five *hwacc_xbar_master* streams, where 5-th is the additional input data stream for the codebook initialization while the 2-nd is used for the compressed weights in the decoding mode or the uncompressed weights in the bypass mode.

The register files are configured via the original *hwacc_cfg_slave* interface.

TABLE 7.1: The additional control registers and job-dependent registers that are used by the integrated Tunstall decoder.

Reg	Offset	Bits	Bit Mask	Content
32	0x0080	31: 0	0xffffffff	Tunstall Codebook Address
33	0x0084	31: 8	0xffffff000	Reserved
		7:4	0x000000f0	Symbol Bits
		3:0	0x0000000f	Bypass mode
34	0x0088	31: 0	0xffffffff	Decoder Trigger

7.2.2 Register File Map in the HWPE-Tunstall Wrapper

Peripheral signals are sent to the HWPE via the APB bus and are stored in Regfiles according to MMIO rules. According to the PULPissimo datasheet, the address from *0x1A10 C000* to *0x1A10 EFFF* is reserved for the HWPE and there are 32 registers used by the HWPE. Three reserved registers are used by the Tunstall decoder, which is summarized in Table 7.1.

Register 32 is used to store the start address of the Tunstall codebook in the TCDM and register 33 is used to configure the number of bits of the symbol (uncompressed weight) and the bypass mode. After register 34 is written, the Tunstall decoder starts to read the compressed weights from the TCDM.

7.2.3 The State Machine in the HWPE-Tunstall Wrapper

The Finite-State Machine (FSM) of the HWPE with a Tunstall decoder is shown in Fig. 7.4. As it illustrated, there are five states:

- Idle: Do nothing;
- Decoder Config: Initializing and configuring the decoder;
- HWPE Config: Initializing and configuring the HWPE;
- Work: the HWPE and the Tunstall decoder are working in parallel and in their own state machines to perform computation;
- Finish: the computation is done;

Except for *codebook_done*, which is the signal from the Tunstall decoder, all the other transition conditions are retrieved from the register files.

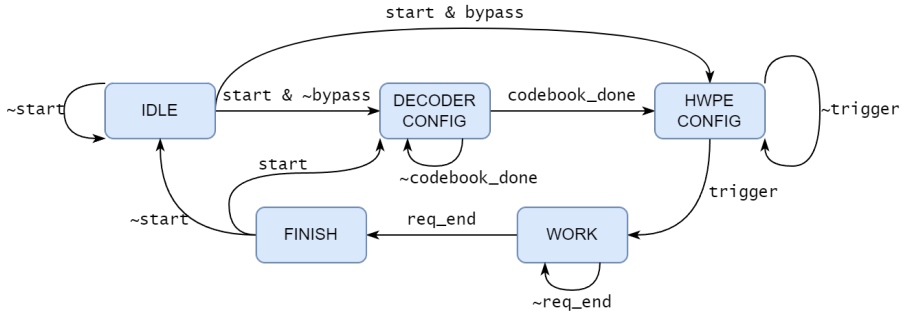


FIGURE 7.4: The FSM of the HWPE-Tunstall Wrapper.

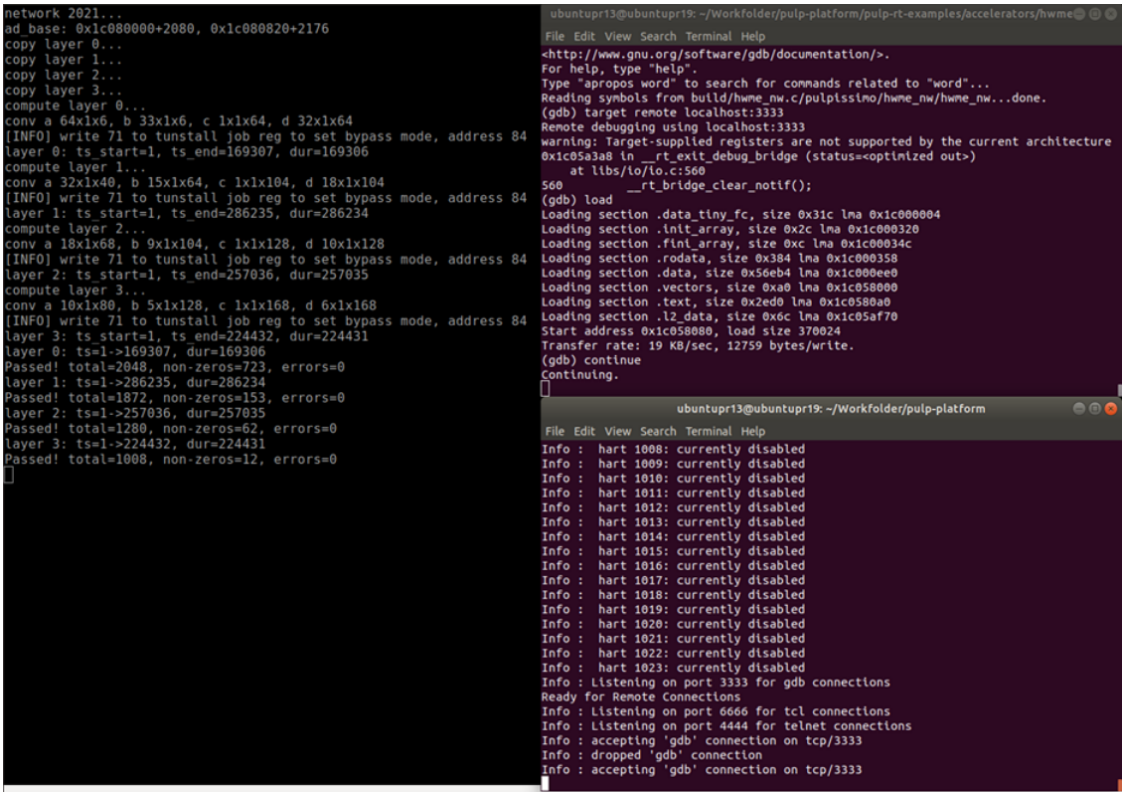


FIGURE 7.5: The FPGA emulation results of the bypass mode. As shown in the left terminal, the computations in the four convolutional layers are correct.

7.2.4 Validation Results

The integration is validated by the simulation and emulation results. We modified the simulation C files and got the correct computation results on our demo four-layer convolution neural network. As illustrated in Fig. 7.5 and Fig. 7.6, the emulation results show that the HWPE with the Tunstall decoder can work correctly on FPGA (Gensys II) in both bypass mode and decoding mode.

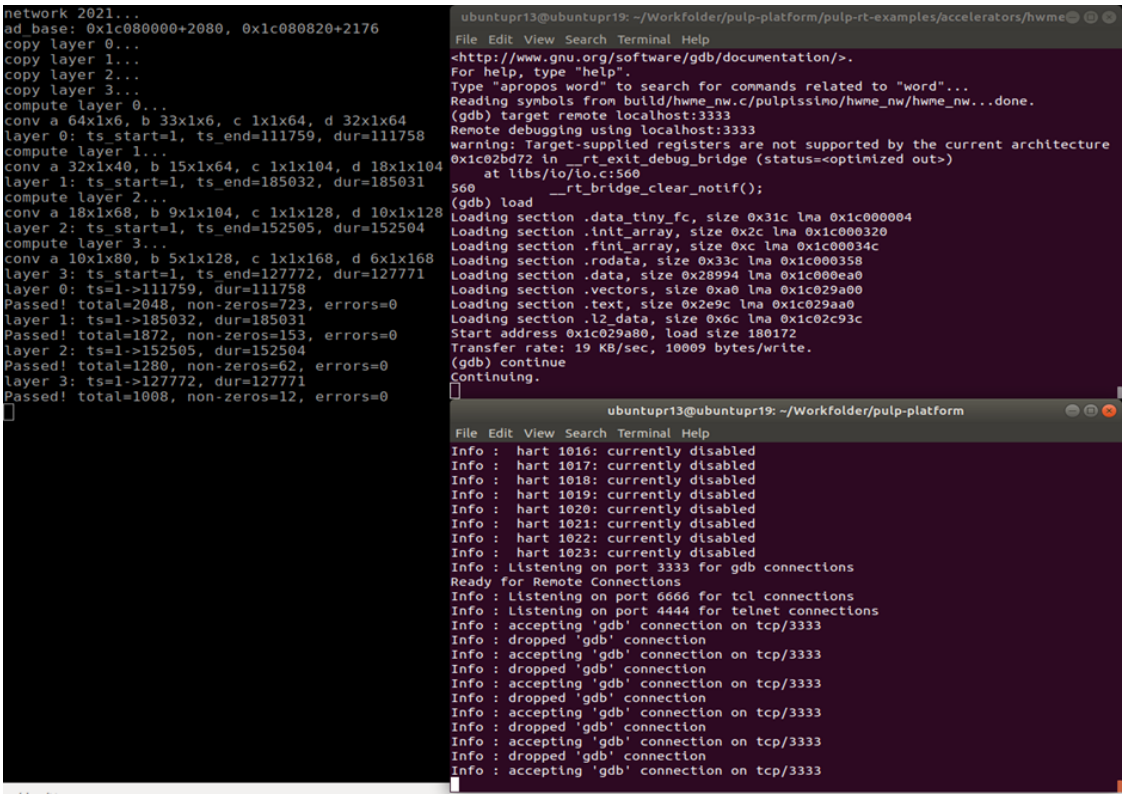


FIGURE 7.6: The FPGA emulation results of the decoding mode. As shown in the left terminal, the computations in the four convolutional layers are correct.

7.3 Full System Integration of ShapoolNMS

In this section, we will introduce the system-level integration of the ShapoolNMS introduced in Chapter 4.

ShapoolNMS is designed to be a coprocessor, which can be a stand-alone chip that connects to other deep learning acceleration chips or GPUs via off-chip communication protocols or can be integrated into a Heterogeneous Multi-Processor System on Chips (HMPSoCs) via on-chip protocols. The HMPSoCs encompass an assortment of processing cores, including CPUs, GPUs, and other deep-learning-based object detection accelerators such as NPUs. In the context of object detection inference, convolutional operations are executed on the aforementioned processing elements, while ShapoolNMS expedites NMS.

7.3.1 I/O and Bandwidth Requirements of ShapoolNMS

The number of I/O requirements of one ShapoolNMS when performing NMS computation is formulated in Eq. 7.1, where I is the maximum support image resolution, B is the bit width of the score, and N is the number of parallel computing units.

$$\#\ \text{data I/O} = (6 \cdot \text{log2ceil}(I) + B) \cdot N \tag{7.1}$$

The required peak bandwidth of one ShapoolNMS when performing NMS computation is formulated in Eq. 7.2, where F is the operating frequency. Please note that the average bandwidth requirement is much less than the peak bandwidth as no data is transferred during PS MNMS.

$$\text{BW} = \#\ \text{data I/O} \cdot F \tag{7.2}$$

7.3.2 Integration with CPU and GPU System as a Standalone Chip

As delineated in Fig. 7.7a, ShapoolNMS can be designed to interface with the GPU through NVLink and with the CPU via the PCIe 5.0 protocol, a standard commonly utilized in NVIDIA GPUs.

The specifications of PCIe 5.0 and NVLink indicate that they offer bandwidths of 128 GB/s and 900 GB/s, which are sufficient for an 8-way 32-bit ShapoolNMS whose peak bandwidth requirement is 50 GB/s at 400 MHz computed from Eq. 7.1 and Eq. 7.2.

7.3.3 Integration with Object Detection Accelerator via On-chip Protocol

As illustrated in Fig. 7.7b, ShapoolNMS interfaces with the object detection accelerator directly. The data transferring time is also hidden by the relationship recovery computation.

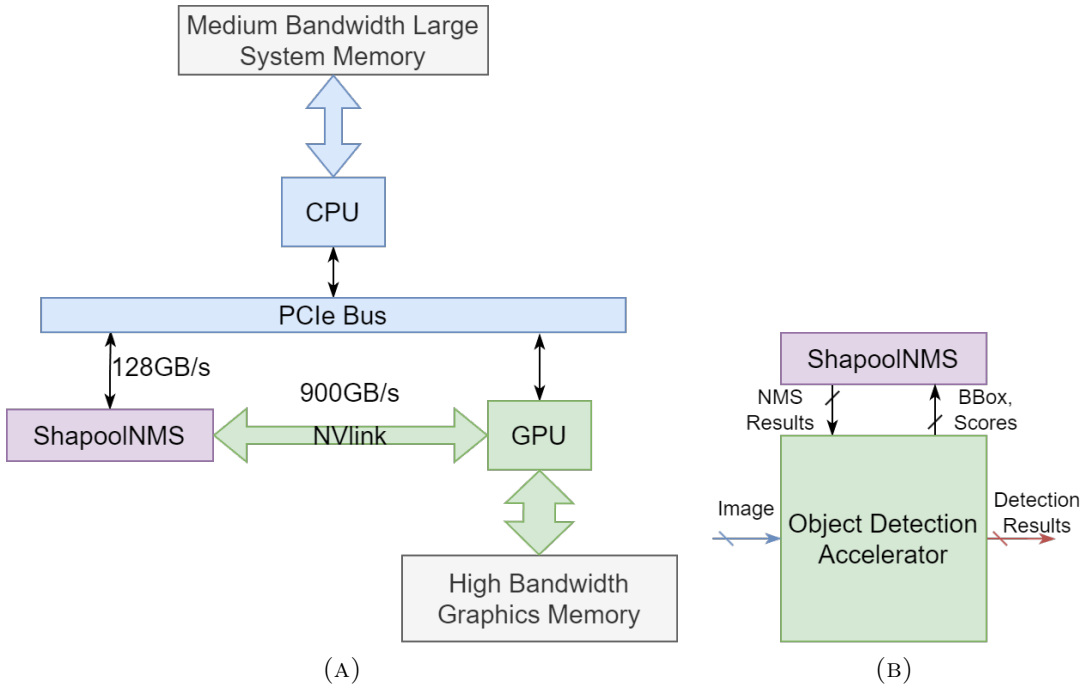


FIGURE 7.7: The HMPSoC system with (a) one CPU, GPU, and ShapoolNMS. (b) one object detection accelerator, and ShapoolNMS.

The peak bandwidth requirement of the interface with an 8-way 32-bit ShapoolNMS is 50 GB/s at 400 MHz computed from Eq. 7.1 and Eq. 7.2.

7.3.4 Integration with PULPissimo

As there is no object detection accelerator and PCIe bus in the PULPissimo SoC platform, we integrated the ShapoolNMS with the PULPissimo via the TCDM Interconnect for data access and the APB bus for control signals, which is shown in Fig. 7.8. The integrated module is **ShapoolNMS Wrapper** where another controller is added for decoding the control signals from the APB bus and sending the control signals to the ShapoolNMS.

7.3.4.1 Interfaces of the ShapoolNMS Wrapper

As illustrated in Fig. 7.9, there are four *hwacc_xbar_master* streams where two of them are used for reading the coordinates and scores of the anchors and one is used for initializing the ShapoolNMS while the other is used for the output data. The register files are configured via the *hwacc_cfg_slave* interface.

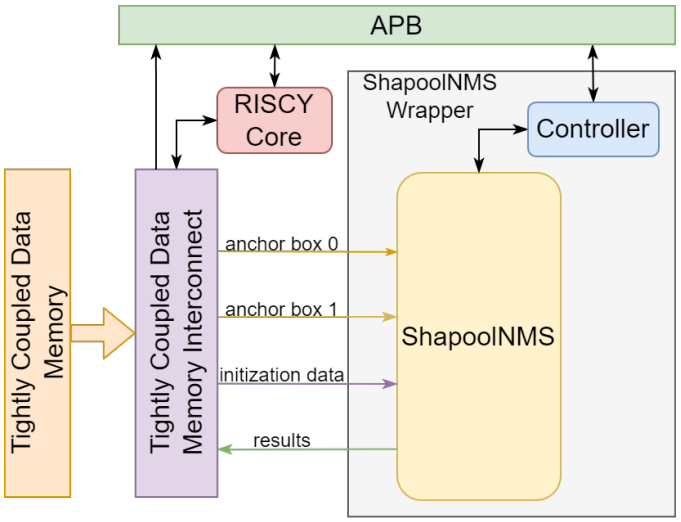


FIGURE 7.8: The introduced ShapoolNMS is integrated with the PULPissimo SoC platform via the TCDM Interconnect for data access and the APB bus for control signals. The integrated module is **ShapoolNMS Wrapper**.

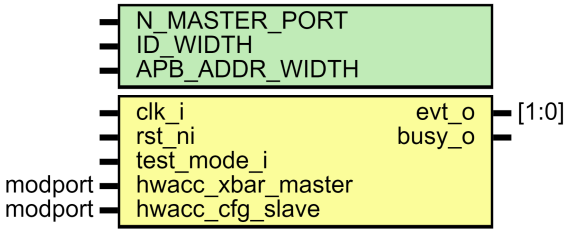


FIGURE 7.9: The interfaces of the ShapoolNMS Wrapper.

7.3.4.2 Register File Map in the ShapoolNMS Wrapper

According to the PULPissimo datasheet, the address from *0x1A11 1000* to *0x1BFF FFFF* is reserved. Therefore, we start the MMIO address of the ShapoolNMS Wrapper from *0x1A11 1000*.

There are 8 control registers used by the ShapoolNMS Wrapper, which is the same as the HWPE and is summarized in Table 7.2. Register 0 and 1 are used to trigger the execution of a job and to acquire the execution status of a job in the ShapoolNMS respectively. Register 2 is used to return the number of jobs that have been finished. The register status is used to return the status of the ShapoolNMS, where status 0 means idle, and status 1 means busy, while register 4 is used to return the ID of the running job. The ShapoolNMS can be soft-cleared by writing 1 to register 5.

TABLE 7.2: The control registers that are used by the ShapoolNMS Wrapper.

Reg ID	Offset	Bits	Bit Mask	Content
0	0x0000	31: 0	0xffffffff	Trigger
1	0x0004	31: 0	0xffffffff	Acquire
2	0x0008	31: 0	0xffffffff	Finished
3	0x000c	31: 0	0xffffffff	Status
4	0x0010	31: 0	0xffffffff	Running job
5	0x0014	31: 0	0xffffffff	Soft clear
6-7	–	–	–	Reserved

Except for the control registers, there are 12 job-dependent registers in the ShapoolNMS Wrapper, which is summarized in Table 7.3. Registers 8 to 11 are used to configure the number of data to be read from the score map memories in the ShapoolNMS. Register 12 is used to configure the start address of the KICU in the ShapoolNMS and once this register is written, the ShapoolNMS starts to initialize the KICUs. If this register is written to 0, the ShapoolNMS will not initialize the KICUs and their values will be kept. The start addresses of the input boxes and the output results are configured by registers 13 and 14 respectively. During the computation, once all the boxes have been sent to the ShapoolNMS, the bit 0 of register 6 will be set to 1 by the CPU. Once the ShapoolNMS finishes the computation, the bit 0 of register 7 will be set to 1 by the ShapoolNMS.

7.3.4.3 The State Machine in the ShapoolNMS Wrapper

As Fig. 7.10 illustrated, there are five states in the ShapoolNMS Wrapper:

- Idle: Do nothing;
- ShapoolNMS Config: Initializing and configuring the ShapoolNMS. The KICUs are initialized from the TCDM;
- Waiting: Waiting for the input anchor boxes from the TCDM;
- Work: ShapoolNMS is working in its own state machine to perform NMS;
- Finish: the computation is done;

Except for *trigger*, which is read from the register files, all the other transition conditions are retrieved from the ShapoolNMS.

TABLE 7.3: The job-dependent registers that are used by the ShapoolNMS Wrapper.

Reg ID	Offset	Bits	Bit Mask	Content
8	0x0020	31: 24	–	Reserved
		23: 12	0x00fff000	Mem read number 1
		11: 0	0x00000fff	Mem read number 0
9	0x0024	31: 24	–	Reserved
		23:12	0x00fff000	Mem read number 3
		11: 0	0x00000fff	Mem read number 2
10	0x0028	31: 24	–	Reserved
		23: 12	0x00fff000	Mem read number 5
		11: 0	0x00000fff	Mem read number 4
11	0x002c	31: 24	–	Reserved
		23: 12	0x00fff000	Mem read number 7
		11: 0	0x00000fff	Mem read number 6
12	0x0030	31: 0	0xffffffff	KICU start address
13	0x0034	31: 0	0xffffffff	Input boxes start address
14	0x0038	31: 0	0xffffffff	Output results start address
15	0x003c	31: 1	–	Reserved
		0	0x00000001	Box received end
16	0x0040	31: 1	–	Reserved
		0	0x00000001	Box send done
17-19	–	–	–	Reserved

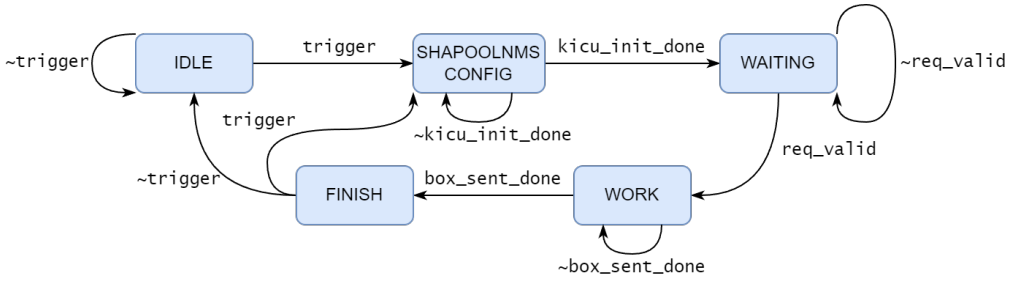


FIGURE 7.10: The FSM of the ShapoolNMS Wrapper.

7.4 Full System Integration of CNN-DLA

We integrated our CNN-DLA which is introduced in Chapter 5 with another controller for decoding the control signals in the PULPissimo SoC platform, into a new module called **CNN-DLA Wrapper**, which is illustrated in Fig. 7.11.

7.4.1 Interfaces of the CNN-DLA Wrapper

There are three *hwacc_xbar_master* streams for data access and one *hwacc_cfg_slave* stream for configuring the register files, which is similar to the HWPE, as shown in

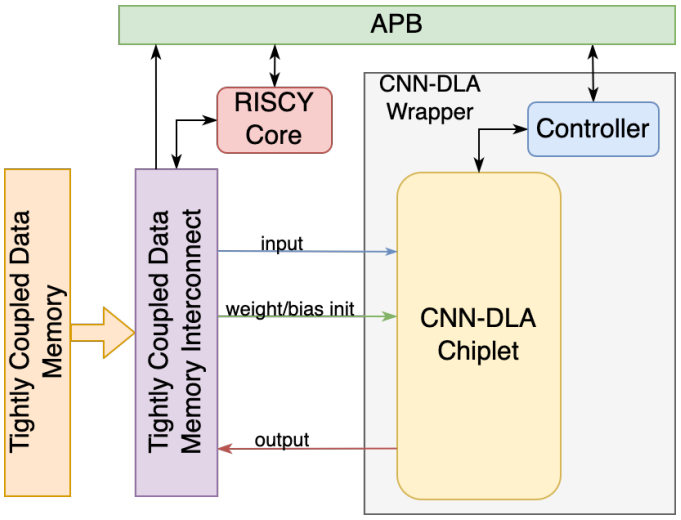


FIGURE 7.11: The introduced CNN-DLA is integrated with the PULPissimo SoC platform via the TCDM Interconnect for weight and bias initialization and input images and the APB bus for control signals. The integrated module is **CNN-DLA Wrapper**.

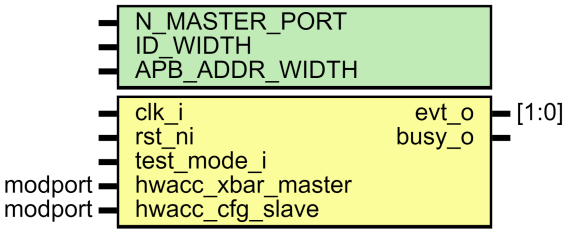


FIGURE 7.12: The interfaces of the CNN-DLA Wrapper.

Fig. 7.12. As the on-chip SRAMs in the memory chips of the CNN-DLA chiplet are initialized with the weights and biases before the computation, there is no bandwidth requirement for the initialization. Therefore, only one *hwacc_xbar_master* stream is required for the initialization. The input images are read from the TCDM via 2-rd *hwacc_xbar_master* stream and the output data is sent to the TCDM via the 3-nd *hwacc_xbar_master* stream during the computation.

7.4.2 Register File Map in the CNN-DLA Wrapper

The MMIO address of the CNN-DLA Wrapper is from *0x1A11 2000* to *0x1A11 2FFF*.

TABLE 7.4: The job-dependent registers that are used by the CNN-DLA Wrapper.

Reg ID	**Offset**	**Bits**	**Bit Mask**	**Content**
8	0x0020	31: 0	0xffffffff	Number of layers
9	0x0024	31: 0	0xffffffff	Total weight size
10	0x0028	31: 0	0xffffffff	Total bias size
11	0x002c	31: 0	0xffffffff	Weight start address
12	0x0030	31: 0	0xffffffff	Bias start address
13	0x0034	31: 0	0xffffffff	Input start address
14	0x0038	31: 0	0xffffffff	Output start address
15	0x003c	31: 28	–	Reserved
		27: 24	0x0f000000	Block ID
		23: 16	0x00ff0000	Unit ID
		15: 12	0x0000f000	Conv layer ID
		11: 8	0x00000f00	Dataflow
		7: 4	0x000000f0	Conv type
		3	0x00000008	Current unit receive
		2	0x00000004	Current unit send
		1: 0	0x00000003	Special Function
16	0x0040	31: 24	–	Reserved
		23: 13	0x00fff000	Output height
		11: 0	0x00000fff	Output width
17	0x0044	31: 24	–	Reserved
		23: 13	0x00fff000	Input channel
		11: 0	0x00000fff	Output channel
18	0x0048	31: 0	0xffffffff	Current weight start address
19	0x004c	31: 0	0xffffffff	Current bias start address

The CNN-DLA Wrapper has the same control registers as the HWPE and the ShapoolNMS and 12 additional job-dependent registers are used, which is summarized in Table 7.4.

Registers 8 to 14 are configured during the initialization and are configured with the number of layers, the total number of weights, the total number of biases, the start addresses of the weights, biases, input images, and output results respectively.

Register 15 to 19 are used during the computation. To compute one convolutional layer, the CNN-DLA is configured with the information of the current layer, *i.e.*, the index of the current block, unit, and convolutional layer in the ResNet, the type of the convolutional layer, *i.e.*, 7×7 convolution with a stride of 2, 3×3 convolution with a stride of 1 or 2, and 1×1 convolution, and the optimal dataflow for better energy efficiency and latency. Those are configured by register 15. The shape of the output feature map is configured by register 16 and the number of input channels and output channels are configured by register 17. The start addresses of

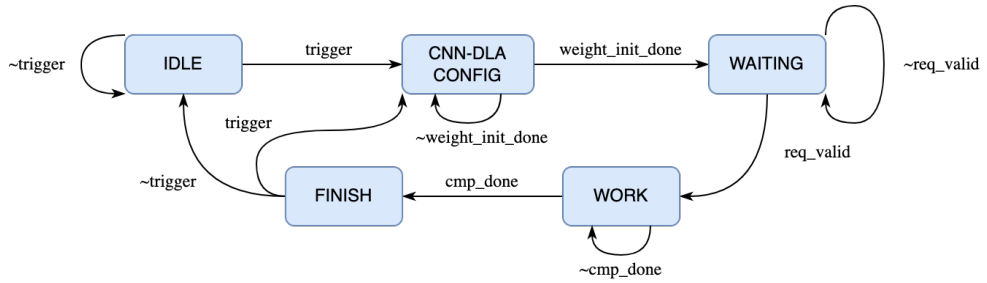


FIGURE 7.13: The FSM of the CNN-DLA Wrapper.

the weights and biases of the current convolutional layer are configured by registers 18 and 19 respectively.

7.4.3 The State Machine in the CNN-DLA Wrapper

The FSM of the CNN-DLA Wrapper is shown in Fig. 7.13. As it illustrated, there are five states:

- Idle: Do nothing;
- CNN-DLA Config: Initializing and configuring the CNN-DLA. The weights and biases are initialized from the TCDM;
- Waiting: Waiting for the input images from the TCDM;
- Work: the CNN-DLA is working in its own state machine to perform inference. The input images are read from the TCDM;
- Finish: the inference is done;

Except for *trigger*, which is read from the register files, all the other transition conditions are retrieved from the CNN-DLA.

7.5 Full System Integration of ViTA

ViTA, introduced in Chapter 6, is integrated into the PULPissimo SoC platform with another controller for decoding the control signals, into a new module called **ViTA Wrapper**, which is illustrated in Fig. 7.14.

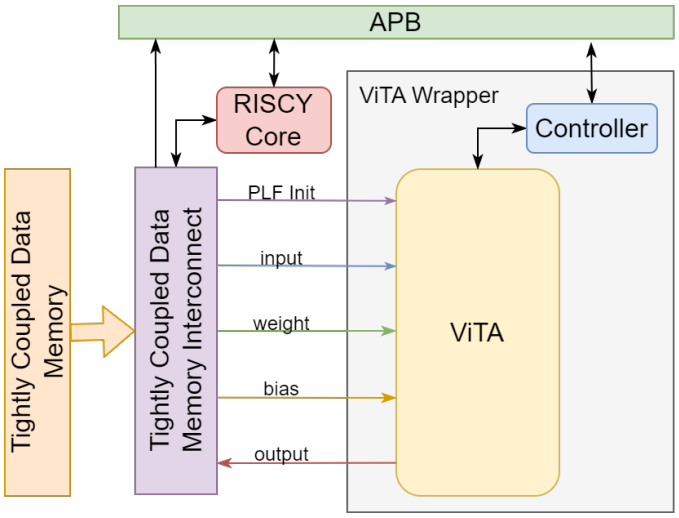


FIGURE 7.14: The introduced ViTA is integrated with the PULPissimo SoC platform. The data is read from the TCDM and the control signals are sent via the APB bus. The integrated module is **ViTA Wrapper**.

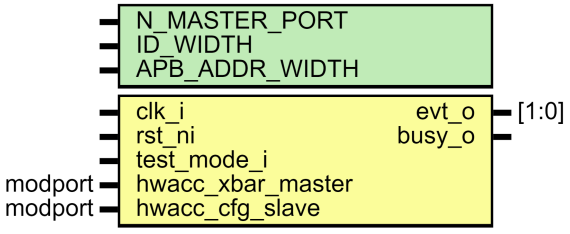


FIGURE 7.15: The interfaces of the ViTA Wrapper.

7.5.1 Interfaces of the ViTA Wrapper

As shown in Fig. 7.15, there are five *hwacc_xbar_master* streams where three of them are used for reading the input, weight, and bias from the TCDM while the fourth is used for the output data, which is similar to the HWPE, while the last one is used to initialize the PLF memories in the ViTA for the special functions.

7.5.2 Register File Map in the ViTA Wrapper

The MMIO address of the ViTA Wrapper is from *0x1A11 3000* to *0x1A11 3FFF*.

The ViTA Wrapper has the same control registers as the HWPE, the ShapoolNMS, and the CNN-DLA. As summarized in Table 7.5, there are 12 additional job-dependent registers used by the ViTA Wrapper.

TABLE 7.5: The job-dependent registers that are used by the ViTA Wrapper.

Reg ID	Offset	Bits	Bit Mask	Content
8	0x0020	31: 0	0xffffffff	Weight start address
9	0x0024	31: 0	0xffffffff	Bias start address
10	0x0028	31: 0	0xffffffff	Input start address
11	0x002c	31: 0	0xffffffff	Output start address
12	0x0030	31: 16	–	Reserved
		15: 14	0x0000c000	Layer type
		13: 12	0x00003000	Operation ID
		11: 10	0x00000c00	GEMM dataflow
		9: 8	0x00000300	Matrix A source
		7: 6	0x000000c0	Matrix B source
		5: 4	0x00000030	matrix C destination
		3: 2	0x0000000c	Pre-spcial function
		1: 0	0x00000003	Post-special function
13	0x0034	31: 24	–	Reserved
		23: 12	0x00fff000	M
		11: 0	0x00000fff	K
14	0x0038	31:20	–	Reserved
		19: 8	0x000fff00	N
		7: 0	0x000000ff	Heads/channels
15	0x003c	31: 0	0xffffffff	PLF 0 address
16	0x0040	31: 0	0xffffffff	PLF 1 address
17	0x0044	31: 0	0xffffffff	PLF 2 address
18	0x0048	31: 0	0xffffffff	PLF 3 address
19	0x004c	31: 0	0xffffffff	PLF 4 address

Registers 8 to 11 are configured with the start addresses of the weights, biases, input images, and output results respectively of the current operation. However, only the start operation and the last operation should be configured with the start addresses of the input images and the output results respectively while the other operations should be configured with 0.

The information on the current operation is configured by registers 12 to 14, including the type of current layer, *i.e.*, pre-processing, encoder, or final classifier, the operation ID, the dataflow of the GEMM operation, the source of the matrix A, *i.e.*, matrix buffer A/B/C or matrix buffers B and C, the source of the matrix B, *i.e.*, weight buffers or matrix buffer C, the destination of the matrix C, the pre-special functions and post-special functions, the shape of the GEMM operation, *i.e.*, M, K, and N, and the number of heads or channels.

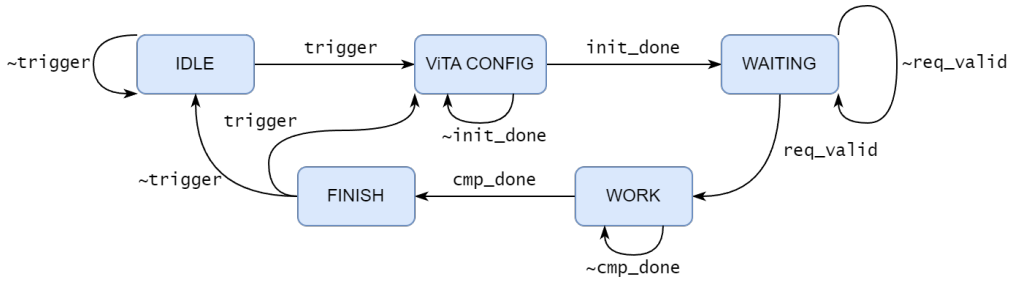


FIGURE 7.16: The FSM of the ViTA Wrapper.

In the ViTA, the PLF memories are configured to perform different special functions in different operations. Therefore, the start addresses of the PLF memories are configured by registers 15 to 19.

7.5.3 The State Machine in the ViTA Wrapper

Fig. 7.16 shows the FSM of the ViTA Wrapper. The ViTA Wrapper has five states:

- Idle: Do nothing;
- ViTA Config: Initializing and configuring the ViTA;
- Waiting: Waiting for the data from the TCDM;
- Work: the ViTA is working in its own state machine to perform inference. Input, weight, and bias are read from the TCDM;
- Finish: the inference is done;

7.6 Chapter Summary

In this chapter, we introduced the full system integration of the proposed hardware accelerators.

We first introduced the full system integration of the Tunstall decoder proposed in Chapter 3 with the HWPE in the PULPissimo SoC platform into a module called **HWPE-Tunstall Wrapper** via the TCDM Interconnect for data access and the APB bus for control signals. The interfaces, register file map, and the FSM of

the HWPE-Tunstall Wrapper are introduced. The integration is validated by both the simulation and the FPGA implementation in both bypass mode and decoding mode.

Then, we introduced the full system integration of the ShapoolNMS proposed in Chapter 4. ShapoolNMS requires 50 GB/s peak bandwidth at 400 MHz, which is sufficient for the PCIe 5.0 and NVLink protocols. We also integrated the ShapoolNMS with the PULPissimo SoC platform into a module called **ShapoolNMS Wrapper** via the TCDM Interconnect and the APB bus. The interfaces, register file map, and the FSM of the ShapoolNMS Wrapper are introduced.

We also introduced the full system integration of the CNN-DLA proposed in Chapter 5 and the ViTA proposed in Chapter 6 with another controller for decoding the control signals in the PULPissimo SoC platform, into a new module called **CNN-DLA Wrapper** and **ViTA Wrapper** respectively. The interfaces, register file map, and the FSM of the CNN-DLA Wrapper and the ViTA Wrapper are introduced respectively.

Chapter 8

Conclusions and Future Works

8.1 Conclusions

As the size and computation complexity of DNNs increase exponentially while memory density keeps steady and silicon scaling also hits the bottleneck, narrowing down the gap between the software and hardware implementations to ensure the real applications of DNNs is essential. The state-of-the-art optimization techniques on DNN hardware implementations are presented as the first step towards achieving this goal. Chapter 2 presents a literature survey of the foundations of deep learning, the basic components of deep learning accelerators, the design metrics of deep learning accelerators, and optimizing the memory footprint of deep learning workloads. The hardware-software co-design and optimization for next-generation learning machines is the main focus of this thesis.

The first contribution is the design and implementation of two special function accelerators to improve the efficiency of DNN hardware implementations. Chapter 3 presents two efficient decoders for Tunstall entropy coding for DNN compression, which is designed to narrow the huge gap between the DNN scaling and memory scaling. By employing the Tunstall entropy coding scheme on the DNNs, the MO decoder reduces 19.58× and 21.67× memory usage in the inference stage on ResNet-50 and MobileNet-v2 respectively, while the LO decoder reduces 19.76× and 22.08× compared with the full precision 32-bit networks while providing few overheads on the whole system. The two Tunstall decoders designed by us are around 6× and 3× faster than Huffman coding.

Chapter 4 proposed another special function accelerator — ShapoolNMS for the NMS algorithm to alleviate the computational bottleneck in the object detection framework. Results show that ShapoolNMS achieves a speedup of $1,204.35\times$ to $6,666.14\times$ than GreedyNMS software implementations in the PASCAL VOC benchmarking dataset in the Faster-RCNN pipeline. Increasing the parallelism can further improve the performance of ShapoolNMS.

The second field is the design and implementation of two hardware accelerator systems for end-to-end DNN workloads. The first one is introduced in Chapter 5, a 3D-stacked Chiplet-based deep learning accelerator — CNN-DLA, which is designed to accelerate the CNN-based workloads such as the object detection framework with a ResNet-152 backbone that supports up to full HD images with 68 FPS. We propose a cross-layer optimization algorithm to reduce the memory requirements of the accelerator by 84.85%. By supporting hybrid dataflows, *i.e.*, both row stationary and column stationary, the memory capacity requirements of the weight buffer are reduced by 84.03%.

Chapter 6 presents the second one, an end-to-end hardware accelerator for the recent state-of-the-art vision transformer — ViT. We also present a cross-layer optimization algorithm to reduce the memory requirements for Multi-Head Attention by 40.5% and for weight buffer by 97.9%. ViTA can be configured to achieve a wide range of computation capabilities, from 204.8 GOPS to 16.384 TOPS, with a wide range of area and power consumption, from 2.00 mm^2 to 6.79 mm^2 and from 217 mW to 10.40 W, respectively, suitable for a great variety of applications.

Finally, we showcase the full system integration of the introduced accelerators into the PULPissimo SoC in Chapter 7.

8.2 Future Works

Although much work has been devoted to the hardware acceleration of DNNs, this area is still in its infancy, following the rapid development of DNN models. Contributions proposed in this thesis are only a small step towards the goal of achieving efficient hardware acceleration of DNNs and can be further extended in various directions.

- **Training Stage of DNNs:** In this thesis, we focus on the inference stage of DNNs. Moreover, the training stage of DNNs is also a critical part of the whole DNN framework. The training stage of DNNs is more computationally intensive and contains more types of operators than the inference stage. In the future, we will focus on the training stage of DNNs and propose hardware solutions to accelerate the training stage of DNNs. This will make both CNN-DLA and ViTA more complete.

- **Accelerator for both the encoder and decoder of the transformer:** In the end-to-end hardware accelerator we developed for ViT, the accelerator can be made more flexible to support both the encoder and decoder of the transformer. This will make ViTA stronger.

- **Accelerator for general DNNs**: An interesting direction is to explore the possibility of using one proposed accelerator to accelerate CNNs, transformers, and future DNNs. As for the ViTA, extra flexibility for various activation functions and variations of Softmax and LayerNorm operations are offered by the configurable special function module. This allows for its adoption in most Transformer encoder-based LLMs and non-linear functions in Transformer decoders and future DNN models. However, due to the increasing size of the KV cache during decoding inference, ViTA faces limitations with Transformer decoder-based LLMs. The current on-chip memory constraints restrict the number of tokens to 197, which falls short of the thousands of millions required for real-world applications. As long as the operations can be transformed to GEMM or element-wise operations that are supported by ViTA, it can support the new DNN models well. Future research efforts could be directed towards improving the flexibility of the proposed work to support not only the traditional CNN models, currently popular Transformer-based LLMs, but also the future DNNs.

In the next few years, the hardware-software co-design and optimization for next-generation learning machines towards both area-efficient and energy-efficient will continue to be a critical research topic as the DNN models will continue to grow in size and computation complexity, which will pose more challenges to hardware designers.

By employing such end-to-end hardware solutions in real AI applications, the emission of carbon dioxide can be reduced compared with current solutions. Besides, this solution also enables edge devices to execute real-time AI applications with higher energy efficiency and enriches the user experience by accelerating workloads and running more accurate DNN models.

List of Author's Awards, Patents, and Publications[1]

Patents

- **Chunyun Chen**, and Mohamed M. Sabry Aly, "ViTA: A Highly Efficient Dataflow And Architecture For Vision Transformers", *NTU Ref: 2023-380*, Sep. 2023.
- **Chunyun Chen**, and Mohamed M. Sabry Aly, "Register-Transfer-Level Implementation Of Decoders For Decoding A Codeword of A Tunstall Code", *NTU Ref: 2023-077*, Mar. 2023.
- **Chunyun Chen**, and Mohamed M. Sabry Aly, "ShapoolNMS: Towards Scalable Hardware Acceleration Of Non-Maximum Suppression For Object Detection", *NTU Ref: 2022-441-01-SG PRV*, *Singapore Provisional Patent Application No.: 10202300546V*, Mar. 2023.
- **Chunyun Chen**, Zhe Wang, Jie Lin, and Mohamed M. Sabry Aly, "Decoders For Decoding A Codeword Of A Tunstall Code", *NTU Ref: 2021-031-02-PCT*, *Singapore Patent Application No. 10202113561S*, *PCT Patent Application No. PCT/SG2022/050887*, Dec. 2022.

[1]The superscript * indicates joint first authors

Journal Articles

- **Chunyun Chen**, Tianyi Zhang, Lantian Li, Jie Lin, and Mohamed M. Sabry Aly, "ShapoolNMS: Towards Scalable Hardware Acceleration for Non-Maximum Suppression in Object Detection", submitted to a top-tier journal (under review).
- Xue Geng, Zhe Wang, **Chunyun Chen**, Qing Xu, Chao Jin, Manas Gupta, Zhenghua Chen, Mohamed M. Sabry Aly, Jie Lin, Min Wu, Xiaoli Li, "From Algorithm to Hardware: A Survey on Efficient and Safe Deployment of Deep Neural Networks", in *IEEE Transactions on Neural Networks and Learning Systems*.
- Tianyi Zhang, **Chunyun Chen**, Yun Liu, Xue Geng, Mohamed M. Sabry Aly, Jie Lin, "PSRR-MaxpoolNMS++: Hardware-Accelerated Non-Maximum Suppression", submitted to a top-tier journal (under review).

Conference Proceedings

- **Chunyun Chen**, Zhe Wang, Jie Lin and Mohamed M. Sabry Aly, "Efficient Tunstall Decoder for Deep Neural Network Compression", in *Proceedings of the 2021 28th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1021-1026, doi: 10.1109/DAC18074.2021.9586173.
- **Chunyun Chen**, Tianyi Zhang, Zehui Yu, Adithi Raghuraman, Shwetalaxmi Udayan, Jie Lin and Mohamed M. Sabry Aly, "Scalable Hardware Acceleration of Non-Maximum Suppression", in *Proceedings of the 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 96-99, doi: 10.23919/DATE54114.2022.9774717.
- **Chunyun Chen**, Lantian Li, and Mohamed M. Sabry Aly, "ViTA: A High Efficient Dataflow and Architecture for Vision Transformers", in *Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2024.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012. 1, 2, 15, 20

[2] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017. 1, 2

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 1, 23, 24, 127, 130

[4] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021. 1, 24, 127

[5] Adam Roberts, Hyung Won Chung, Anselm Levskaya, Gaurav Mishra, James Bradbury, Daniel Andor, Sharan Narang, Brian Lester, Colin Gaffney, Afroz Mohiuddin, et al. Scaling up models and data with `t5x` and `seqio`. *arXiv preprint arXiv:2203.17189*, 2022. 1, 24, 127

[6] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012. 1, 23, 24

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. xxiii, 2, 3, 6, 9, 20, 44, 57, 95, 109

[8] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018. 2, 44, 57

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 2, 20

[10] OpenAI. Gpt-4 technical report, 2023. 2, 24, 127

[11] Epoch. Parameter, compute and data trends in machine learning. https://epochai.org/data/pcd, 2022. xxiii, 3

[12] Song Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 3, 4, 5, 34, 38, 39, 40, 41, 44, 45

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 3, 9, 23, 128, 130

[14] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017. 4, 34, 39, 40

[15] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017. 4, 39, 40

[16] Rahul Mishra et al. A survey on deep neural network compression: Challenges, overview, and solutions. *arXiv preprint arXiv:2010.03954*, 2020. 4

[17] Jungwook Choi et al. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018. 4, 41

[18] Dongqing Zhang et al. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[19] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017. 4, 41

[20] Stefano Recanatesi et al. Dimensionality compression and expansion in deep neural networks. *arXiv preprint arXiv:1906.00443*, 2019. 4, 41

[21] Deniz Oktay et al. Scalable model compression by entropy penalized reparameterization. *arXiv preprint arXiv:1906.06624*, 2019. 5, 44, 45

[22] Simon Wiedemann et al. Deepcabac: Context-adaptive binary arithmetic coding for deep neural network compression. *arXiv preprint arXiv:1905.08318*, 2019. 5, 45

[23] Abhimanyu Dubey et al. Coreset-based neural network compression. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 454–470, 2018. 5, 44, 45

[24] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010. 6

[25] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 6, 63, 64, 87

[26] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014. xxiii, 7

[27] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9 (2):292–308, 2019. xxiv, 7, 29, 35, 36, 37, 99

[28] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE Computer Society, 2019. 8, 100

[29] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. 28, 29, 101

[30] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014. 29, 99

[31] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3): 243–254, 2016. 7, 35, 36, 37, 39, 40

[32] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008. 7

[33] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013. 7, 103

[34] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014. 7, 103

[35] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54, 2017. 7

[36] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[37] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[38] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE transactions on computer-aided design of integrated circuits and systems*, 37(1):35–47, 2017.

[39] Yun Liang, Liqiang Lu, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(4):857–870, 2019. 7, 102, 103

[40] Afzal Ahmad and Muhammad Adeel Pasha. Ffconv: an fpga-based accelerator for fast convolution layers in convolutional neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(2):1–24, 2020. 7

[41] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014. 7, 29, 99

[42] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016. 29

[43] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar.

Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3): 14–26, 2016. 29

[44] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 541–552. IEEE, 2017.

[45] Farnood Merrikh-Bayat, Xinjie Guo, Michael Klachko, Mirko Prezioso, Konstantin K Likharev, and Dmitri B Strukov. High-performance mixed-signal neurocomputing with nanoscale floating-gate memory cell arrays. *IEEE transactions on neural networks and learning systems*, 29(10):4782–4790, 2017.

[46] Avishek Biswas and Anantha P Chandrakasan. Conv-sram: An energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 54(1):217–230, 2018.

[47] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 740–753, 2019.

[48] Cheng-Xin Xue, Wei-Hao Chen, Je-Syu Liu, Jia-Fang Li, Wei-Yu Lin, Wei-En Lin, Jing-Hong Wang, Wei-Chen Wei, Ting-Wei Chang, Tung-Cheng Chang, et al. 24.1 a 1mb multibit reram computing-in-memory macro with 14.6 ns parallel mac computing time for cnn based ai edge processors. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 388–390. IEEE, 2019.

[49] Hossein Valavi, Peter J Ramadge, Eric Nestler, and Naveen Verma. A 64-tile 2.4-mb in-memory-computing cnn accelerator employing charge-domain compute. *IEEE Journal of Solid-State Circuits*, 54(6):1789–1799, 2019.

[50] Shihui Yin, Zhewei Jiang, Jae-Sun Seo, and Mingoo Seok. Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks. *IEEE Journal of Solid-State Circuits*, 55(6):1733–1743, 2020.

[51] Peng Yao, Huaqiang Wu, Bin Gao, Jianshi Tang, Qingtian Zhang, Wenqiang Zhang, J Joshua Yang, and He Qian. Fully hardware-implemented memristor convolutional neural network. *Nature*, 577(7792):641–646, 2020.

[52] Kavitha Madhu, Saptarsi Das, Abhishek Tyagi, Ankur Deshwal, Joonho Song, and Seungwon Lee. Transport triggered near memory accelerator for deep learning. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021. 7

[53] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 35–44, 2017. 7, 103

[54] Jinhua Lin and Yu Yao. A fast algorithm for convolutional neural networks using tile-based fast fourier transforms. *Neural Processing Letters*, 50(2): 1951–1967, 2019. 7

[55] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International conference on artificial neural networks*, pages 281–290. Springer, 2014. 7, 102

[56] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016. 7, 102

[57] Xingyu Liu, Jeff Pool, Song Han, and William J Dally. Efficient sparse-winograd convolutional neural networks. *arXiv preprint arXiv:1802.06367*, 2018. 7, 35

[58] Yulin Zhao, Donghui Wang, and Leiou Wang. Convolution accelerator designs using fast algorithms. *Algorithms*, 12(5):112, 2019. 7

[59] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019. 8

[60] Qiuling Zhu, Berkin Akin, H Ekin Sumbul, Fazle Sadi, James C Hoe, Larry Pileggi, and Franz Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *2013 IEEE international 3D systems integration conference (3DIC)*, pages 1–7. IEEE, 2013. 8

[61] Mohamed M Sabry Aly, Tony F Wu, Andrew Bartolo, Yash H Malviya, William Hwang, Gage Hills, Igor Markov, Mary Wootters, Max M Shulaker, H-S Philip Wong, et al. The n3xt approach to energy-efficient abundant-data computing. *Proceedings of the IEEE*, 107(1):19–48, 2018. 8

[62] Tianyi Zhang, Jie Lin, Peng Hu, Bin Zhao, and Mohamed M Sabry Aly. Psrr-maxpoolnms: Pyramid shifted maxpoolnms with relationship recovery. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15840–15848, 2021. 9, 63, 65, 66, 67, 69, 88

[63] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini. Quentin: an ultra-low-power pulpissimo soc in 22nm fdx. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, 2018. doi: 10.1109/S3S.2018.8640145. xxix, 9, 57, 59, 155, 156

[64] Yann LeCun. 1.1 deep learning hardware: Past, present, and future. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 12–19. IEEE, 2019. 13

[65] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958. 15

[66] Kunihiko Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. *IEICE Technical Report, A*, 62(10):658–665, 1979. 15, 19

[67] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 15, 39, 40

[68] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986. 15, 19

[69] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 15, 19

[70] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. 15

[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017. 15, 19, 21, 22, 24, 127

[72] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4):322–333, 1969. 15, 17, 20

[73] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016. 15, 132

[74] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013. 15

[75] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017. 15, 17

[76] Kouichi Yamaguchi, Kenji Sakamoto, Toshio Akabane, and Yoshiji Fujimoto. A neural network for speaker-independent isolated word recognition. In *ICSLP*, 1990. 15

[77] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015. 15, 18

[78] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. 15, 18

[79] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016. 15, 18, 19

[80] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in neural information processing systems*, 29:901–909, 2016. 15, 18

[81] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018. 15, 18, 19

[82] Siyuan Qiao, Huiyu Wang, Chenxi Liu, Wei Shen, and Alan Yuille. Micro-batch training with batch-channel normalization and weight standardization. *arXiv preprint arXiv:1903.10520*, 2019. 15, 18

[83] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. *Advances in neural information processing systems*, 27, 2014. 15

[84] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. *arXiv preprint arXiv:1710.09829*, 2017. 19

[85] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014. 19

[86] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006. 19

[87] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008. 19

[88] Stephen José Hanson. A stochastic version of the delta rule. *Physica D: Nonlinear Phenomena*, 42(1-3):265–272, 1990. 20

[89] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015. 20

[90] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 20

[91] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994. 20

[92] Fastertransformer, 12 2022. URL `https://github.com/NVIDIA/FasterTransformer/tree/d2923c035b1ee7f8d05b883f052361a673b0e9ad`. xxiv, 22, 128, 130, 132

[93] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021. 23, 24, 127

[94] Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 568–578, 2021. 23

[95] Kai Han, An Xiao, Enhua Wu, Jianyuan Guo, Chunjing Xu, and Yunhe Wang. Transformer in transformer. *Advances in Neural Information Processing Systems*, 34:15908–15919, 2021. 23

[96] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021. 23, 24, 127

[97] Li Deng. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, 3, 2014. 23

[98] Li Deng and John Platt. Ensemble deep learning for speech recognition. In *Proc. Interspeech*, 2014. 23

[99] Gregoire Montavon. Deep learning for spoken language identification. In *NIPS Workshop on deep learning for speech recognition and related applications*, pages 1–4. Citeseer, 2009. 23

[100] Ignacio Lopez-Moreno, Javier Gonzalez-Dominguez, Oldrich Plchot, David Martinez, Joaquin Gonzalez-Rodriguez, and Pedro Moreno. Automatic language identification using deep neural networks. In *2014 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5337–5341. IEEE, 2014. 23

[101] Hendrik Purwins, Bo Li, Tuomas Virtanen, Jan Schlüter, Shuo-Yiin Chang, and Tara Sainath. Deep learning for audio signal processing. *IEEE Journal of Selected Topics in Signal Processing*, 13(2):206–219, 2019. 23

[102] Stefan Uhlich, Franck Giron, and Yuki Mitsufuji. Deep neural network based instrument extraction from music. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2135–2139. IEEE, 2015. 24

[103] Filip Korzeniowski and Gerhard Widmer. Feature learning for chord recognition: The deep chroma extractor. *arXiv preprint arXiv:1612.05065*, 2016. 24

[104] Aishwarya Bhave, Mayank Sharma, and Rekh Ram Janghel. Music generation using deep learning. In *Soft Computing and Signal Processing*, pages 203–211. Springer, 2019. 24

[105] Swarnendu Ghosh, Nibaran Das, Ishita Das, and Ujjwal Maulik. Understanding deep learning techniques for image segmentation. *ACM Computing Surveys (CSUR)*, 52(4):1–35, 2019. 24

[106] Huiyu Wang, Yukun Zhu, Hartwig Adam, Alan Yuille, and Liang-Chieh Chen. Max-deeplab: End-to-end panoptic segmentation with mask transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5463–5474, 2021. 127

[107] Yuqing Wang, Zhaoliang Xu, Xinlong Wang, Chunhua Shen, Baoshan Cheng, Hao Shen, and Huaxia Xia. End-to-end video instance segmentation with transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8741–8750, 2021. 24, 127

[108] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017. 24

[109] Zhikai Li and Qingyi Gu. I-vit: integer-only quantization for efficient vision transformer inference. *arXiv preprint arXiv:2207.01405*, 2022. 127, 128, 130, 133, 135

[110] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021. 24, 127

[111] Ming Ding, Zhuoyi Yang, Wenyi Hong, Wendi Zheng, Chang Zhou, Da Yin, Junyang Lin, Xu Zou, Zhou Shao, Hongxia Yang, et al. Cogview: Mastering text-to-image generation via transformers. *Advances in Neural Information Processing Systems*, 34:19822–19835, 2021. 24, 127

[112] Manoj Kumar, Dirk Weissenborn, and Nal Kalchbrenner. Colorization transformer. *arXiv preprint arXiv:2102.04432*, 2021. 24, 127

[113] Mennatullah Siam, Chen Jiang, Steven Lu, Laura Petrich, Mahmoud Gamal, Mohamed Elhoseiny, and Martin Jagersand. Video object segmentation using teacher-student adaptation in a human robot interaction (hri) setting. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 50–56. IEEE, 2019. 24

[114] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014. 24

[115] Zhaowei Cai, Quanfu Fan, Rogerio S Feris, and Nuno Vasconcelos. A unified multi-scale deep convolutional neural network for fast object detection. In *European conference on computer vision*, pages 354–370. Springer, 2016. 24

[116] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer, 2020. 127

[117] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable detr: Deformable transformers for end-to-end object detection. *arXiv preprint arXiv:2010.04159*, 2020. 24, 127

[118] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, and Luc Van Gool. Temporal segment networks: Towards good practices for deep action recognition. In *European conference on computer vision*, pages 20–36. Springer, 2016. 24

[119] Chiara Plizzari, Marco Cannici, and Matteo Matteucci. Spatial temporal transformer network for skeleton-based action recognition. In *Pattern Recognition. ICPR International Workshops and Challenges: Virtual Event, January 10–15, 2021, Proceedings, Part III*, pages 694–701. Springer, 2021. 24, 127

[120] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019. 24, 127

[121] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. 24, 127

[122] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020. 24, 127

[123] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International Conference on Machine Learning*, pages 11328–11339. PMLR, 2020.

[124] Weizhen Qi, Yu Yan, Yeyun Gong, Dayiheng Liu, Nan Duan, Jiusheng Chen, Ruofei Zhang, and Ming Zhou. Prophetnet: Predicting future n-gram for sequence-to-sequence pre-training. *arXiv preprint arXiv:2001.04063*, 2020. 24, 127

[125] Bhaskar Mitra and Nick Craswell. Neural text embeddings for information retrieval. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 813–814, 2017. 24

[126] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016. 24

[127] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, et al. Tensor2tensor for neural machine translation. *arXiv preprint arXiv:1803.07416*, 2018. 24

[128] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019. 127

[129] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742, 2020. 24, 127

[130] Keet Sugathadasa, Buddhi Ayesha, Nisansa de Silva, Amal Shehan Perera, Vindula Jayawardana, Dimuthu Lakmal, and Madhavi Perera. Legal document retrieval using document vector embeddings and deep learning. In *Science and information conference*, pages 160–175. Springer, 2018. 24

[131] Jun Yin, Xin Jiang, Zhengdong Lu, Lifeng Shang, Hang Li, and Xiaoming Li. Neural generative question answering. *arXiv preprint arXiv:1512.01337*, 2015. 24

[132] Xiao Huang, Jingyuan Zhang, Dingcheng Li, and Ping Li. Knowledge graph embedding based question answering. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 105–113, 2019. 24

[133] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017. 24

[134] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*, pages 2042–2050, 2014. 24

[135] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019. 127

[136] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[137] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020. 24, 127

[138] Tsung-Hsien Wen, David Vandyke, Nikola Mrksic, Milica Gasic, Lina M Rojas-Barahona, Pei-Hao Su, Stefan Ultes, and Steve Young. A network-based end-to-end trainable task-oriented dialogue system. *arXiv preprint arXiv:1604.04562*, 2016. 24

[139] Lin Ma, Zhengdong Lu, Lifeng Shang, and Hang Li. Multimodal convolutional neural networks for matching image and sentence. In *Proceedings of the IEEE international conference on computer vision*, pages 2623–2631, 2015. 24

[140] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017. 24

[141] Mark Cicero, Alexander Bilbily, Errol Colak, Tim Dowdell, Bruce Gray, Kuhan Perampaladas, and Joseph Barfett. Training and validating a deep convolutional neural network for computer-aided detection and classification of abnormalities on frontal chest radiographs. *Investigative radiology*, 52(5): 281–287, 2017. 24

[142] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O'Donoghue, Daniel Visentin, et al. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine*, 24 (9):1342–1350, 2018. 24

[143] Pornpimol Charoentong, Francesca Finotello, Mihaela Angelova, Clemens Mayer, Mirjana Efremova, Dietmar Rieder, Hubert Hackl, and Zlatko

Trajanoski. Pan-cancer immunogenomic analyses reveal genotype-immunophenotype relationships and predictors of response to checkpoint blockade. *Cell reports*, 18(1):248–262, 2017. 24

[144] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014. 26

[145] William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020. xxxi, 28

[146] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016. 28, 29, 35, 36, 99, 142

[147] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 682–687, 2014. 29

[148] Lukas Cavigelli and Luca Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, 2016. 29

[149] NVIDIA. Nvidia deep learning accelerator, 2018. URL `http://nvdla.org/`. 29, 36, 99, 102

[150] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015. 29, 101

[151] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi. In *ISSCC*, 2017. 29, 37

[152] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. Squeezeflow: A sparse cnn accelerator exploiting concise convolution rules. *IEEE Transactions on Computers*, 2019. 29, 37

[153] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45 (2):27–40, 2017. 29, 35, 36, 37, 39, 40

[154] Hong-Yi Wang and Tian-Sheuan Chang. Row-wise accelerator for vision transformer. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 399–402. IEEE, 2022. 29

[155] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Leibo Liu, and Shaojun Wei. A 1.06-to-5.09 tops/w reconfigurable hybrid-neural-network processor for deep learning applications. In *Symposium on VLSI Circuits*, 2017. 33, 35

[156] Hao Zhang, Dongdong Chen, and Seok-Bum Ko. New flexible multiple-precision multiply-accumulate unit for deep neural network training and inference. *IEEE Transactions on Computers*, 2019. 33, 35

[157] Jinsu Lee, Juhyoung Lee, Donghyeon Han, Jinmook Lee, Gwangtae Park, and Hoi-Jun Yoo. 7.7 lnpu: A 25.3 tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16. In *ISSCC*. IEEE, 2019. 33, 35, 36

[158] Bruce Fleischer, Sunil Shukla, Matthew Ziegler, Joel Silberman, Jinwook Oh, Vijavalakshmi Srinivasan, Jungwook Choi, Silvia Mueller, Ankur Agrawal, Tina Babinsky, et al. A scalable multi-teraops deep learning processor core for ai trainina and inference. In *IEEE Symposium on VLSI Circuits*, 2018. 34, 35

[159] Xian Zhou, Li Zhang, Chuliang Guo, Xunzhao Yin, and Cheng Zhuo. A convolutional neural network accelerator architecture with fine-granular mixed precision configurability. In *ISCAS*, 2020. 34, 35

[160] Xiaoxi He, Zimu Zhou, and Lothar Thiele. Multi-task zipping via layer-wise neuron sharing. In *Advances in Neural Information Processing Systems*, pages 6016–6026, 2018. 34, 39

[161] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. 34, 39, 40

[162] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proceedings of the IEEE*, 2021. xxxi, 35, 36

[163] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016. 36

[164] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *IPDPS*, 2015. 36

[165] Fred G Gustavson. Some basic techniques for solving sparse systems of linear equations. pages 41–52, 1972. 36

[166] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *MICRO*, 2019. 36

[167] Zhe Yuan, Jinshan Yue, Huanrui Yang, Zhibo Wang, Jinyang Li, Yixiong Yang, Qingwei Guo, Xueqing Li, Meng-Fan Chang, Huazhong Yang, et al. Sticker: A 0.41-62.1 tops/w 8bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers. In *IEEE symposium on VLSI circuits*, 2018. 36, 37

[168] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W Keckler, and Zhengya Zhang. Snap: A 1.67—21.55 tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos. In *VLSIC*. IEEE, 2019. xxiv, 36, 37, 38

[169] Hyeong-Ju Kang. Accelerator-aware pruning for convolutional neural networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 2019.

[170] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *FCCM*, 2019. 36

[171] Jeff Jun Zhang, Parul Raj, Shuayb Zarar, Amol Ambardekar, and Siddharth Garg. Compact: On-chip compression of activations for low power systolic array based cnn acceleration. *TECS*, 2019. 36

[172] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *MICRO*, 2018. xxiv, 36, 37, 38

[173] Asit K Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. Fine-grained accelerators for sparse machine learning workloads. In *ASP-DAC*, 2017. 36

[174] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017. 36

[175] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando De Freitas. Predicting parameters in deep learning. *arXiv preprint arXiv:1306.0543*, 2013. 38

[176] Steven A Janowsky. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600, 1989. 39

[177] Michael C Mozer and Paul Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In *Advances in neural information processing systems*, pages 107–115, 1989. 39

[178] Ehud D Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE transactions on neural networks*, 1(2):239–242, 1990. 39

[179] Artur Jordao, Maiko Lie, and William Robson Schwartz. Discriminative layer pruning for convolutional neural networks. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):828–837, 2020. 39

[180] Sourav Bhattacharya and Nicholas D Lane. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 176–189, 2016. 39, 40

[181] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014. 39

[182] Pravendra Singh, Vinay Sameer Raja Kadi, Nikhil Verma, and Vinay P Namboodiri. Stability based filter pruning for accelerating deep cnns. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1166–1174. IEEE, 2019. 39, 40

[183] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pages 2736–2744, 2017. 39, 40

[184] Hanyu Peng, Jiaxiang Wu, Shifeng Chen, and Junzhou Huang. Collaborative channel pruning for deep networks. In *International Conference on Machine Learning*, pages 5113–5122. PMLR, 2019. 39, 40

[185] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2017. 39

[186] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. PMLR, 2017.

[187] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. *arXiv preprint arXiv:1705.08665*, 2017. 40

[188] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. *arXiv preprint arXiv:1608.04493*, 2016. 39, 40

[189] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4876–4883, 2020. 39

[190] Vadim Lebedev and Victor Lempitsky. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2554–2564, 2016. 39

[191] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018. 41

[192] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015. 41

[193] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015. 41

[194] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 41

[195] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. *arXiv preprint arXiv:1612.01543*, 2016. 41

[196] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016. 41

[197] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. 41

[198] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*, 2016. 41

[199] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *NIPS*, volume 1, page 2, 2014. 41

[200] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. *Advances in neural information processing systems*, 28:2575–2583, 2015. 41

[201] Jonathan Bachrach et al. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012. 43, 79, 122, 147

[202] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 44

[203] Ian H Witten et al. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. 44, 45

[204] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009. 45

[205] Majid Rabbani. Jpeg2000: Image compression fundamentals, standards and practice. *Journal of Electronic Imaging*, 11(2):286, 2002. 45

[206] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1543–1546, 2015. doi: 10.7873/DATE.2015.0733. 58

[207] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 63, 64

[208] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015. 63, 64

[209] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015. 63, 64, 109

[210] Man Shi, Peng Ouyang, Shouyi Yin, Leibo Liu, and Shaojun Wei. A fast and power-efficient hardware architecture for non-maximum suppression. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 66(11):1870–1874, 2019. xxvii, 63, 66, 88, 89, 90, 91, 93

[211] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001. 64, 65

[212] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004. 64, 65

[213] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005. 64, 65

[214] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. A discriminatively trained, multiscale, deformable part model. In *2008 IEEE conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2008. 64

[215] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017. 64

[216] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014. 64

[217] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017. 64

[218] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*, 2019. 65

[219] Jan Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4507–4515, 2017. 65

[220] Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S Davis. Soft-nms–improving object detection with one line of code. In *Proceedings of the IEEE international conference on computer vision*, pages 5561–5569, 2017. 65

[221] Borui Jiang, Ruixuan Luo, Jiayuan Mao, Tete Xiao, and Yuning Jiang. Acquisition of localization confidence for accurate object detection. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–799, 2018. 65

[222] Songtao Liu, Di Huang, and Yunhong Wang. Adaptive nms: Refining pedestrian detection in a crowd. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6459–6468, 2019. 65

[223] Niels Ole Salscheider. Featurenms: Non-maximum suppression by learning feature embeddings. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 7848–7854. IEEE, 2021. 65

[224] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013. 65

[225] Lile Cai, Bin Zhao, Zhe Wang, Jie Lin, Chuan Sheng Foo, Mohamed Sabry Aly, and Vijay Chandrasekhar. Maxpoolnms: getting rid of nms bottlenecks in two-stage object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9356–9364, 2019. 65, 66, 69

[226] Paul Henderson and Vittorio Ferrari. End-to-end training of object class detectors for mean average precision. In *Asian Conference on Computer Vision*, pages 198–213. Springer, 2016. 65

[227] Chaitanya Desai, Deva Ramanan, and Charless C Fowlkes. Discriminative models for multi-class object layout. *International journal of computer vision*, 95(1):1–12, 2011.

[228] Li Wan, David Eigen, and Rob Fergus. End-to-end integration of a convolution network, deformable parts model and non-maximum suppression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 851–859, 2015. 65

[229] Rasmus Rothe, Matthieu Guillaumin, and Luc Van Gool. Non-maximum suppression for object detection by passing messages between windows. In *Asian conference on computer vision*, pages 290–306. Springer, 2014. 66

[230] David Oro, Carles Fernández, Xavier Martorell, and Javier Hernando. Work-efficient parallel non-maximum suppression for embedded gpu architectures. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1026–1030. IEEE, 2016. 66

[231] David Oro García, Carles Fernandez Tena, Xavier Martorell Bofill, and Francisco Javier Hernando Pericás. Work-efficient parallel non-maximum suppression kernels. *Computer journal*, (bxaa108):1–15, 2020. 66, 88, 93

[232] Huiyu Mo, Leibo Liu, Wenping Zhu, Qiang Li, Hong Liu, Shouyi Yin, and Shaojun Wei. A multi-task hardwired accelerator for face detection and alignment. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(11):4284–4298, 2019. 66

[233] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, 2004. 79, 122, 148

[234] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to understand large caches. *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, 147, 2009. 79, 90

[235] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. *ACM SIGARCH Computer Architecture News*, 43(1):369–381, 2015. 99

[236] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018. 99

[237] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969. 100, 101

[238] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980. 100, 102

[239] E Oran Brigham and RE Morrow. The fast fourier transform. *IEEE spectrum*, 4(12):63–70, 1967. 100, 102

[240] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801. IEEE, 2021. 100

[241] Hsiang T Kung and Charles E Leiserson. Systolic arrays for (vlsi). Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1978. 101

[242] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, 2020. 101

[243] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The design process for google's training chips: Tpuv2 and tpuv3. *IEEE Micro*, 41(2):56–63, 2021.

[244] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–14, 2023. xxvii, 101

[245] Bosheng Liu, Xiaoming Chen, Ying Wang, Yinhe Han, Jiajun Li, Haobo Xu, and Xiaowei Li. Addressing the issue of processing element under-utilization in general-purpose systolic deep learning accelerators. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 733–738, 2019. 101

[246] S Kala, Babita R Jose, Jimson Mathew, and S Nalesh. High-performance cnn accelerator on fpga using unified winograd-gemm architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2816–2828, 2019. 102

[247] Souheil Ben-Yacoub. Fast object detection using mlp and fft. Technical report, IDIAP, 1997. 103

[248] Tyler Highlander and Andres Rodriguez. Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add. *arXiv preprint arXiv:1601.06815*, 2016. 103

[249] Hamid Tabani et al. Improving the efficiency of transformers for resource-constrained devices. In *DSD*, pages 449–456. IEEE, 2021. 128, 143

[250] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. Aˆ 3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020. 128, 132, 134, 153

[251] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 692–705. IEEE, 2021. 132, 134, 152, 153

[252] Yang Wang, Yubin Qin, Dazheng Deng, Jingchuan Wei, Yang Zhou, Yuanqi Fan, Tianbao Chen, Hao Sun, Leibo Liu, Shaojun Wei, et al. A 28nm 27.5 tops/w approximate-computing-based transformer processor with asymptotic sparsity speculating and out-of-order computing. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3. IEEE, 2022. 128, 131, 134, 153

[253] Yang Wang, Yubin Qin, Dazheng Deng, Xiaolong Yang, Zhiren Zhao, Ruiqi Guo, Zhiheng Yue, Leibo Liu, Shaojun Wei, Yang Hu, et al. A 28nm 77.35 tops/w similar vectors traceable transformer processor with principal-component-prior speculating and dynamic bit-wise stationary computing. In *2023 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, pages 1–2. IEEE, 2023. 128, 131

[254] Alberto Marchisio, Davide Dura, Maurizio Capra, Maurizio Martina, Guido Masera, and Muhammad Shafique. Swifttron: An efficient hardware accelerator for quantized transformers. *arXiv preprint arXiv:2304.03986*, 2023. 128, 130, 133, 143, 153

[255] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. In *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pages 84–89. IEEE, 2020. 128, 131, 134

[256] Georgios Tzanos, Christoforos Kachris, and Dimitrios Soudris. Hardware acceleration of transformer networks using fpgas. In *2022 Panhellenic Conference on Electronics & Telecommunications (PACET)*, pages 1–5. IEEE, 2022.

[257] Shashank Nag, Gourav Datta, Souvik Kundu, Nitin Chandrachoodan, and Peter A Beerel. Vita: A vision transformer inference accelerator for edge applications. *arXiv preprint arXiv:2302.09108*, 2023. 128, 131, 153

[258] Yonghao Chen, Tianrui Li, Xiaojie Chen, Zhigang Cai, and Tao Su. High-frequency systolic array-based transformer accelerator on field programmable gate arrays. *Electronics*, 12(4):822, 2023. 128

[259] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pages 5506–5518. PMLR, 2021. 128, 130, 133, 134, 135

[260] Liu Liu, Zheng Qu, Zhaodong Chen, Fengbin Tu, Yufei Ding, and Yuan Xie. Dynamic sparse attention for scalable transformer acceleration. *IEEE Transactions on Computers*, 2022. 128

[261] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021. 130, 135

[262] Richard Crandall and Carl B Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006. 130, 135

[263] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022. 130

[264] Joonsang Yu, Junki Park, Seongmin Park, Minsoo Kim, Sihwa Lee, Dong Hyun Lee, and Jungwook Choi. Nn-lut: neural approximation of non-linear operations for efficient transformer inference. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 577–582, 2022. 131, 133, 134, 138

[265] Haodong Lu, Qichang Mei, and Kun Wang. An efficient piecewise linear approximation of non-linear operations for transformer inference. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 206–206. IEEE, 2023. 133, 134

[266] Bo Yuan. Efficient hardware architecture of softmax layer in deep neural network. In *SOCC*, 2016. 133, 134

[267] Xue Geng, Jie Lin, Bin Zhao, Anmin Kong, Mohamed M Sabry Aly, and Vijay Chandrasekhar. Hardware-aware softmax approximation for deep neural networks. In *ACCV*, 2018. 133, 134, 138

[268] Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Alberto Nannarelli, Marco Re, and Sergio Spanò. A pseudo-softmax function for hardware-based high speed image classification. *Scientific reports*, 2021. 133, 134

[269] Ruofei Hu, Binren Tian, Shouyi Yin, and Shaojun Wei. Efficient hardware architecture of softmax layer in deep neural network. In *ICDSP*, 2018. 133, 134

[270] Meiqi Wang, Siyuan Lu, Danyang Zhu, Jun Lin, and Zhongfeng Wang. A high-speed and low-complexity architecture for softmax function in deep learning. In *APCCAS*. IEEE, 2018. 134

[271] Jacob R Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. Softermax: Hardware/software co-design of an efficient softmax for transformers. In *DAC*, 2021. 134

[272] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017. 155

[273] F. Conti, P. D. Schiavone, and L. Benini. Xnor neural engine: A hardware accelerator ip for 21.6-fj/op binary neural network inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018. ISSN 0278-0070. doi: 10.1109/TCAD.2018.2857019. 155