Design of high-performance and energy-efficient accelerators for Convolutional Neural Networks

by

Mahmood Azhar Qureshi

B.Eng., National University of Sciences and Technology, Pakistan, 2013

M.S., University of Engineering and Technology Taxila, Pakistan, 2018

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2021

Abstract

Deep neural networks (DNNs) have gained significant traction in artificial intelligence (AI) applications over the past decade owing to a drastic increase in their accuracy. This huge leap in accuracy, however, translates into a sizable model and high computational requirements, something which resource-limited mobile platforms struggle against. Embedding AI inference into various real-world applications requires the design of high-performance, area, and energy-efficient accelerator architectures. In this work, we address the problem of the inference accelerator design for dense and sparse convolutional neural networks (CNNs), a type of DNN which forms the backbone of modern vision-based AI systems. We first introduce a fully dense accelerator architecture referred to as the NeuroMAX accelerator. Most traditional dense CNN accelerators rely on single-core, linear processing elements (PEs), in conjunction with 1D dataflows, for accelerating the convolution operations in a CNN. This limits the maximum achievable ratio of peak throughput per PE count to unity. Most of the past works optimize their dataflows to attain close to 100% hardware utilization to reach this ratio. In the NeuroMAX accelerator, we design a high-throughput, multi-threaded, log-based PE core. The designed core provides a 200% increase in peak throughput per PE count while only incurring a 6% increase in the hardware area overhead compared to a single, linear multiplier PE core with the same output bit precision. NeuroMAX accelerator also uses a 2D weight broadcast dataflow which exploits the multi-threaded nature of the PE cores to achieve a high hardware utilization per layer for various dense CNN models.

Sparse convolutional neural network models reduce the massive compute and memory bandwidth requirements inherently present in dense CNNs without a significant loss in accuracy. Designing sparse accelerators for the processing of sparse CNN models, however, is much more challenging compared to the design of dense CNN accelerators. The micro-architecture design, the design of sparse PEs, addressing the load-balancing issues,

and the system-level architectural design issues for processing the entire sparse CNN model are some of the key technical challenges that need to be addressed in order to design a high-performance and energy-efficient sparse CNN accelerator architecture. We break this problem down into two parts.

In the first part, using some of the concepts from the dense NeuroMAX accelerator, we introduce SparsePE, a multi-threaded, and flexible PE, capable of handling both the dense and sparse CNN model computations. The SparsePE core uses the binary mask representation to actively skip ineffective sparse computations involving zeros, and favors valid, non-zero computations, thereby, drastically increasing the effective throughput and the hardware utilization of the core as compared to a dense PE core.

In the second part, we generate a two-dimensional (2D) mesh architecture of the SparsePE cores, which we refer to as the Phantom accelerator. We also propose a novel dataflow that supports processing of **all** layers of a CNN, including unit and non-unit stride convolutions (CONV), and fully-connected (FC) layers. In addition, the Phantom accelerator uses a two-level load balancing strategy to minimize the computational idling, thereby, further improving the hardware utilization, throughput, as well as the energy efficiency of the accelerator. The performance of the dense and the sparse accelerators is evaluated using a custom-built cycle accurate performance simulator and performance is compared against recent works. Logic utilization on hardware is also compared against the prior works. Finally, we conclude by mentioning some more techniques for accelerating CNNs and presenting some other avenues where the proposed work can be applied.

Design of high-performance and energy-efficient accelerators for Convolutional Neural Networks

by

Mahmood Azhar Qureshi

B.Eng., National University of Sciences and Technology, Pakistan, 2013

M.S., University of Engineering and Technology Taxila, Pakistan, 2018

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2021

Approved by:

Major Professor
Dr. Arslan Munir

Copyright

© Mahmood Azhar Qureshi 2021.

Abstract

Deep neural networks (DNNs) have gained significant traction in artificial intelligence (AI) applications over the past decade owing to a drastic increase in their accuracy. This huge leap in accuracy, however, translates into a sizable model and high computational requirements, something which resource-limited mobile platforms struggle against. Embedding AI inference into various real-world applications requires the design of high-performance, area, and energy-efficient accelerator architectures. In this work, we address the problem of the inference accelerator design for dense and sparse convolutional neural networks (CNNs), a type of DNN which forms the backbone of modern vision-based AI systems. We first introduce a fully dense accelerator architecture referred to as the NeuroMAX accelerator. Most traditional dense CNN accelerators rely on single-core, linear processing elements (PEs), in conjunction with 1D dataflows, for accelerating the convolution operations in a CNN. This limits the maximum achievable ratio of peak throughput per PE count to unity. Most of the past works optimize their dataflows to attain close to 100% hardware utilization to reach this ratio. In the NeuroMAX accelerator, we design a high-throughput, multi-threaded, log-based PE core. The designed core provides a 200% increase in peak throughput per PE count while only incurring a 6% increase in the hardware area overhead compared to a single, linear multiplier PE core with the same output bit precision. NeuroMAX accelerator also uses a 2D weight broadcast dataflow which exploits the multi-threaded nature of the PE cores to achieve a high hardware utilization per layer for various dense CNN models.

Sparse convolutional neural network models reduce the massive compute and memory bandwidth requirements inherently present in dense CNNs without a significant loss in accuracy. Designing sparse accelerators for the processing of sparse CNN models, however, is much more challenging compared to the design of dense CNN accelerators. The micro-architecture design, the design of sparse PEs, addressing the load-balancing issues,

and the system-level architectural design issues for processing the entire sparse CNN model are some of the key technical challenges that need to be addressed in order to design a high-performance and energy-efficient sparse CNN accelerator architecture. We break this problem down into two parts.

In the first part, using some of the concepts from the dense NeuroMAX accelerator, we introduce SparsePE, a multi-threaded, and flexible PE, capable of handling both the dense and sparse CNN model computations. The SparsePE core uses the binary mask representation to actively skip ineffective sparse computations involving zeros, and favors valid, non-zero computations, thereby, drastically increasing the effective throughput and the hardware utilization of the core as compared to a dense PE core.

In the second part, we generate a two-dimensional (2D) mesh architecture of the SparsePE cores, which we refer to as the Phantom accelerator. We also propose a novel dataflow that supports processing of **all** layers of a CNN, including unit and non-unit stride convolutions (CONV), and fully-connected (FC) layers. In addition, the Phantom accelerator uses a two-level load balancing strategy to minimize the computational idling, thereby, further improving the hardware utilization, throughput, as well as the energy efficiency of the accelerator. The performance of the dense and the sparse accelerators is evaluated using a custom-built cycle accurate performance simulator and performance is compared against recent works. Logic utilization on hardware is also compared against the prior works. Finally, we conclude by mentioning some more techniques for accelerating CNNs and presenting some other avenues where the proposed work can be applied.

Table of Contents

List of Figures xi

List of Tables xv

Acknowledgements xvi

Dedication xviii

1 Introduction 1

1.1 History and applications 3

1.2 Pitfalls of high-accuracy DNNs 5

1.2.1 Compute and energy bottleneck 5

1.2.2 Sparsity considerations 7

1.3 Contributions 10

1.3.1 Dense accelerator - NeuroMAX 10

1.3.2 Sparse processing core - SparsePE 11

1.3.3 Sparse accelerator - Phantom 12

1.4 Thesis outline 13

2 Background 14

2.1 Neural network architecture 14

2.2 Popular CNN models 18

2.3 Popular CNN datasets 22

2.3.1 MNIST dataset 22

2.3.2 CIFAR 22

2.3.3 ImageNet . . . 23
2.4 CNN processing hardware . . . 23
2.4.1 Kernel transformation for temporal architectures . . . 25
2.4.2 Dataflow design for spatial architectures . . . 25
2.4.3 Near-memory processing . . . 27

3 NeuroMAX accelerator . . . 28
3.1 Introduction . . . 28
3.2 Recent dense CNN accelerators . . . 30
3.3 Log mapping . . . 32
3.4 Hardware architecture . . . 33
3.4.1 Top-level . . . 33
3.4.2 PE matrix . . . 35
3.5 Data flow and processing . . . 39
3.5.1 3 x 3 convolution . . . 39
3.5.2 1 x 1 convolution . . . 44
3.5.3 Higher order convolutions . . . 47
3.6 Implementation and results . . . 48
3.7 Conclusion . . . 55

4 SparsePE core . . . 56
4.1 Introduction . . . 56
4.2 Recent sparse CNN accelerators . . . 57
4.3 Sparsity in CNNs . . . 58
4.4 SparsePE . . . 60
4.4.1 Sparse binary mask . . . 62
4.4.2 Selection . . . 64
4.4.3 Computation . . . 76

4.4.4 Accumulation . . . 82
4.4.5 Output encoding . . . 84
4.5 Implementation and results . . . 86
4.6 Conclusion . . . 90

5 Phantom accelerator . . . 91
5.1 Introduction . . . 91
5.2 Phantom . . . 93
5.2.1 R×C compute matrix . . . 93
5.2.2 Load balancing . . . 94
5.2.3 Regular/depthwise convolution . . . 95
5.2.4 Pointwise convolution . . . 97
5.2.5 FC layers . . . 99
5.2.6 Intra-core balancing . . . 99
5.3 Experiments and results . . . 103
5.3.1 Evaluation methodology . . . 103
5.3.2 Results . . . 104
5.4 Conclusion . . . 114

6 Conclusions and future work . . . 116

Bibliography . . . 119

List of Figures

1.1 DNN processing methodology 2

1.2 Energy cost (relative to 8b Add operation) shown on a log10 scale for a 45nm process technology (Adapted from[1;2]) 6

2.1 A neural network example with 1 input layer, 4 hidden layers, and 1 output layer 15

2.2 Various nonlinear activation functions. (Figure adapted from DNN tutorial survey[2]) 15

2.3 Multi-dimensional convolution operation performed in a CNN CONV layer . 16

2.4 Max and averaging pooling on a 6×6 ifmap 18

2.5 AlexNet[3] architecture 18

2.6 VGG-16[4] architecture 19

2.7 GoogleNet[5] architecture 20

2.8 Inception module[5] 21

2.9 SqueezeNet[6] architecture 21

2.10 Kernel transformation (a) Convolution to matrix multiplication using toeplitz matrix (b) Equivalent FFT computation 24

3.1 Linear vs. log quantization (a) 1.5 bits linear vgg16 net (b) 5.0 bits log vgg16 (c) 5.1 bits log vgg16 (d) 1.5 bits linear squeezeNet (e) 5.0 bits log squeezeNet (f) 5.1 bits log squeezeNet 31

3.2 NeuroMAX system architecture 34

3.3 (a) Compute thread (b) Collection of threads to make a PE (d) 6x3 PE matrix 0 and adder net 0 36

3.4 Adder net 0 psum generation . . . 37
3.5 A convolution example with 12×6 input, 3×3 filter for stride 1 and 2 . . . 38
3.6 State controller operation (a) Input, stride 1 (b) Filter weights (c) Input, stride 2 39
3.7 2D weight broadcast dataflow . . . 40
3.8 Dataflow chart for 3×3 stride 1 Convolution in Figure 3.5 . . . 41
3.9 Adder net 1 configuration (a) Stride 1 (b) Stride 2 . . . 42
3.10 1 × 1 convolution example . . . 42
3.11 State controller load operation . . . 44
3.12 Dataflow chart for 1×1 convolution in Figure 3.10 . . . 45
3.13 (a) Channel-wise accumulation for PE matrix 0 (b) All channel-wise accumulations . . . 46
3.14 (a) 5×5 convolution example (b) Input load operation (c) Weight load operation 47
3.15 Dataflow chart for 5 × 5 convolution in Figure 3.14(a) . . . 48
3.16 Adder configuration for 5 × 5 convolution . . . 49
3.17 Linear vs log PE LUT and FF cost (16-bit) . . . 49
3.18 Breakdown of (a) LUT cost (b) FF cost (c) Power consumption for NeuroMAX 50
3.19 Hardware utilization of NeuroMAX for (a)VGG-16 (b) MobileNet v1 (c) ResNet-34 . . . 52
3.20 PE count vs utilization vs throughput comparison of NeuroMAX with VWA[7] 53

4.1 Sparsity in CNNs (a) Sparsity in AlexNet (b) Sparsity in VGG-16 . . . 59
4.2 Dense convolutions . . . 61
4.3 SparsePE architecture . . . 62
4.4 Sparse data representations (a) Indexed-based representation (b) BM-based representation (c) CSR format . . . 63
4.5 Selection (SL) block diagram . . . 65
4.6 Process of ANDing in the SL block . . . 66
4.7 Selection for maximum multiplier utilization . . . 67

4.8 In-Order selection . . . 68
4.9 In-Order selection algorithm . . . 68
4.10 In-Order selection process for column 1 . . . 69
4.11 Generation of selection matrices . . . 71
4.12 Selection algorithm for maximum multiplier utilization . . . 71
4.13 Selection process for column 1 . . . 72
4.14 Selector implementation . . . 73
4.15 Computation (CM) block diagram . . . 75
4.16 Compute thread matrix . . . 77
4.17 Mapping of compute thread matrix . . . 78
4.18 Thread scheduling for the example in Figure 4.4 . . . 80
4.19 Accumulation for the example in Figure 4.4 . . . 81
4.20 Accumulation stage map . . . 83
4.21 Initial latency of the SparsePE core operations . . . 84
4.22 Output sparse mask generation . . . 84
4.23 1×1 convolution layer transformation into an equivalent 3×3, stride 3 convolution layer . . . 85
4.24 FC layer transformation into an equivalent 3×3, stride 3 convolution layer . . . 87
4.25 Breakdown of resource utilization for (a) LUTs (b) FFs (c) SRAM memory . . . 88
4.26 Sparse binary mask vs. CSC DRAM access for (a) Sparse VGG16 (b) Sparse MobileNet . . . 89

5.1 Phantom architecture . . . 92
5.2 Phantom support for regular/depthwise convolution . . . 96
5.3 Phantom support for pointwise convolution . . . 98
5.4 Phantom support for FC layers . . . 100
5.5 Intra-Core balancing (a) Test example (b) Without balancing (c) With balancing 101

5.6 TDS-IO vs. TDS-OO (a) Per layer comparison with $L_f = 6$ (b) Comparisons with changing L_f 105
5.7 Balanced vs. unbalanced at $L_f = 6$ for (a) Sparse VGG16 (b) Sparse MobileNet 107
5.8 Sensitivity to sparsity and L_f for VGG16 (a) Speedup (b) Average thread utilization 109
5.9 Sensitivity to sparsity and L_f for MobileNet (a) Speedup (b) Average thread utilization 111
5.10 VGG16 speedup comparison with SCNN and SparTen 111
5.11 Speedup comparison with Eyeriss v2 112

List of Tables

1.1 Popular CNN Models 5

3.1 Resource utilization 51

3.2 Comparison of NeuroMAX with previous designs 51

3.3 VGG16 latency comparison 54

4.1 Resource utilization for SparsePE core 88

5.1 Popular CNN layer dimensions 93

5.2 Operation parameters 103

5.3 Accelerator configuration 103

5.4 Comparison of the Phantom accelerator with previous designs 112

5.5 Comparison of the Phantom accelerator with previous designs 113

Acknowledgments

I would like to begin by thanking the Almighty God for giving me the knowledge, patience, perseverance, and critical thinking that helped me succeed in this path. I would then like to thank my Ph.D advisor, Dr. Arslan Munir, for accepting me as his Ph.D student despite my lack of research experience. One thing that I am grateful for is the creative freedom Dr. Munir gave me in choosing my own research direction and working on the problems that I was interested in. Our technical discussions always helped me in formulating some really interesting ideas that I was able to incorporate in my research. I would also like to thank all my committee members for giving me invaluable suggestions during my technical presentations. It helped me in taking a more focused approach towards the research problem. I would also like to thank them for taking the time to review my dissertation and giving helpful feedback. I am also thankful to the current and past members of the Intelligent Systems, Computer Architecture, Analytics, and Security (ISCAAS) lab where I worked during my Ph.D. I am also grateful to the Pakistani community here in Manhattan, Kansas who made K-State feel like a home away from home. Special thanks to Kiran Ambreen Ayub for providing support during each step of the way. This journey would not have been possible without the support of my family and friends in Pakistan and the USA.

I would also like to extend my gratitude towards my managers and mentors at MathWorks and Tesla, where I worked as an intern during the Summer and Fall of 2020, respectively. The invaluable experience I gained in both companies played a critical role in shaping my research. I was involved in solving real-world technical challenges and that broadened my horizon. I was also able to learn some key technical skills that helped me perform some crucial experiments to gather data for this thesis.

Finally, I would like to thank my colleagues at the Center for Advanced Research in Engineering (CARE), Pakistan for helping me become the engineer I am today. Special thanks go to Mr. Hammad A. Khan, my manager, for making me a member of a diverse and

highly competent team of engineers who taught me a lot about digital design and embedded systems.

Dedication

To my Mom and Dad who always believed in me and provided me the comfort and the resources to pursue my dreams. To my elder siblings Wajiha and Umer for providing me strong support and for continuously pushing me for success even during times of low motivation. To my little sister Areej, whose smile brings me joy and whose presence I miss every day during my studies abroad...

Chapter 1

Introduction

Deep neural networks (DNNs) have enabled the deployment of artificial intelligence (AI) in many modern applications including autonomous driving[8], image recognition[9], and speech processing[10]. In many applications, DNNs have achieved close to human-level accuracy and in some, they have exceeded human accuracy[11]. This high accuracy comes from a DNN's unique ability to extract high-level features from a huge quantity of test data using statistical learning and improvement over time. This learning over time provides a DNN with an effective representation of the input space. This is quite different from the earlier approaches where specific features were hand-crafted by domain experts and were subsequently used for feature extraction.

DNN processing is usually carried out in two stages, training and inference, with both of them having their own computational needs. Training is the process where a DNN model is trained using a large application-specific data set. The training time is dependent on the model size and the target accuracy requirements. For high accuracy applications like autonomous driving, training a DNN can take weeks and is usually performed on a cloud. Inference, on the other hand, can be performed either on the cloud or the edge device (mobile device, IoT, autonomous vehicle, etc.). Nowadays, in many applications, it is advantageous to perform the inference process on the edge device, as shown in Figure 1.1. For example, in cellphones, it is desirable to perform image and video processing on the device itself

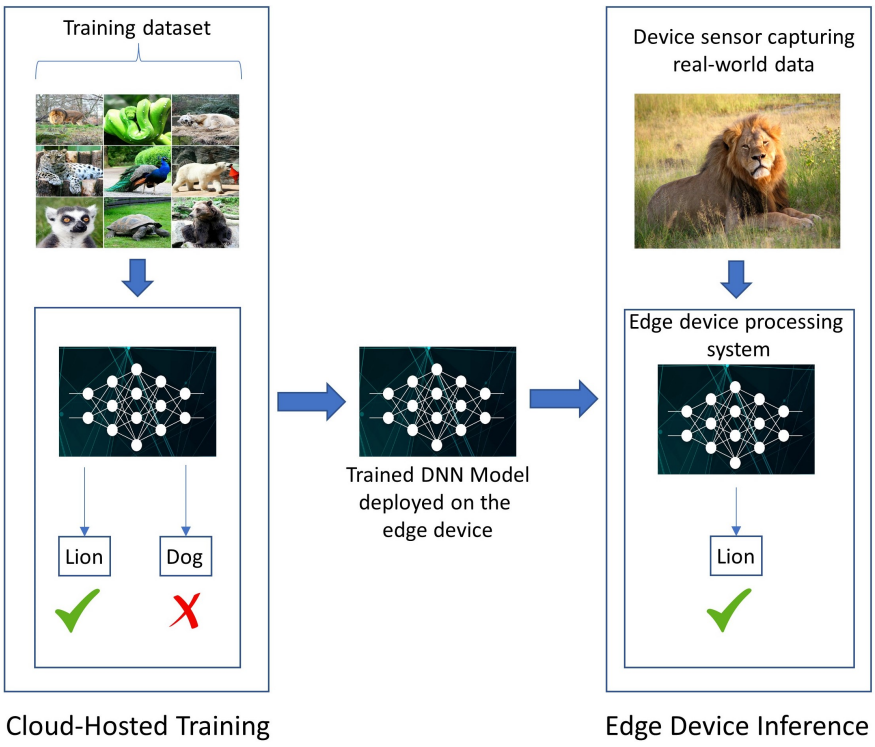


Figure 1.1: *DNN processing methodology*

rather than sending the data over to the cloud for processing. This methodology reduces the communication cost and the latency involved with the data transmission and reception. It also eliminates the risk of losing important device features should there be a network disruption or loss of connectivity. Another motivation for doing inference on the device is the ever-increasing security risk involved with sending personalized data, including images and videos, over to the cloud servers for processing. Autonomous driving systems which require visual data need to deploy solutions to perform inference locally to avoid latency and security issues, both of which can result in a catastrophe, should an undesirable event occurs. Performing DNN inference on the edge presents its own set of challenges. This stems from the fact that the embedded platforms running the edge device have stringent cost limitations which limit their compute capabilities. Running compute and memory-intensive DNN inference in these devices in an efficient manner becomes a matter of prime importance and is a subject of this research.

1.1 History and applications

Neural nets have been around since the 1940s, however, the first practically applicable neural network, referred to as the LeNet[12], was proposed in 1989. This neural network was designed to solve the problem of digit recognition in hand-written numeric digits. It paved the way for the development of neural networks responsible for various applications related to digit recognition like an ATM. The slow growth and a little to no adoption of neural networks in the early days is mainly due to the massive computational requirements involved with their processing which limited their study to theoretical concepts. Over the past decade, there has been an exponential growth in the research on DNNs with many new high accuracy neural networks being deployed for various applications. This has only been possible because of two factors. The first factor is the advancements in the processing power of semiconductor devices and technological breakthroughs in computer architecture. Nowadays, computers have significantly higher computing capability. This enables the processing of a neural network within a reasonable time frame, something that was not achievable in the early days. The second factor is the availability of a large amount of training datasets. As neural networks learn over time, providing huge amounts of training data enables better accuracy. For example, Facebook receives close to a billion user images per day, whereas, YouTube has 300h of video uploaded every minute[2]. This enables the service providers to train their neural networks for targeted ad campaigns bringing in billions of dollars of ad revenue. Apart from their use in social media platforms, DNNs are impacting many other domains and expect to make a huge impact. Some of these areas include:

- Speech Processing: Speech processing algorithms have improved significantly in the past few years. Nowadays, many applications have been developed that use DNNs to perform real-time speech recognition with unprecedented levels of accuracy[10;13–15]. Many technology companies are also using DNNs to perform language translation used in a wide variety of applications. Google, for example, uses Google's Neural Machine Translation system (GNMT)[16] which uses recurrent neural networks (RNNs), a type of DNN, for their language translation applications.

- Autonomous Driving: Autonomous driving has been one of the biggest technological breakthroughs in the auto industry since the invention of the internal combustion engine. It is not a coincidence that the self-driving boom came at the same time when high accuracy DNNs became increasingly popular. Companies like Tesla[17] and Waymo[18] are using various types of self-driving technology including visual feeds and Lidar for their self-driving solutions. One thing which is common in all these solutions is the use of DNNs for visual perception of the road conditions which is the main back-end technology used in advanced driver assistance systems (ADAS).

- Medical AI: Another crucial area where DNNs have become increasingly useful is medicine. Nowadays, doctors can use AI-assisted medical imagery to perform various surgeries. AI systems use DNNs in genomics to gather insights about genetic disorders like autism[19;20]. DNNs are also useful in the detection of various types of cancers like skin and brain cancer[21;22].

- Security: The advent of AI has challenged many traditional security approaches that were previously deemed sufficient. The rollout of 5G technology has caused a massive surge of IoT-based deployments which traditional security approaches are not able to keep up with. Physical unclonablity approaches[23–25] were introduced to protect this massive deployment of IoTs against security attacks with minimum cost overheads. These approaches, however, were also unsuccessful in preventing AI-assisted attacks using DNNs[26;27]. Researchers have now been forced to upgrade the security threat models to incorporate AI-based attacks[28;29]. Because of a massive increase in AI-assisted cyber-attacks on cloud and datacenters, companies have realized that the best way of defeating offensive AI attacks is by incorporating AI-based counterattacks[30;31].

Overall, the use of DNNs in various applications has seen exponential growth over the past decade and this trend has been on the rise for the past many years. The massive increase in DNN deployments on the edge devices requires the development of efficient processing architectures to keep up with the computational requirements for successful DNN inference.

1.2 Pitfalls of high-accuracy DNNs

1.2.1 Compute and energy bottleneck

A common form of DNNs is convolutional neural networks (CNNs). CNNs are composed of multiple convolution layers (CONV) which help in extracting low, mid, and high-level input features for better accuracy. Although CNNs are primarily used in applications related to image and video processing, they are also used in speech processing[10;13], gameplay[32], and robotics[33] applications. We will further discuss the basics of CNNs in chapter 2. In this section, we will explore some of the bottlenecks when it comes to implementing *high accuracy* CNN inference engines in embedded mobile devices.

Table 1.1: *Popular CNN Models*

CNN Model	Layers	Top-1 Accuracy	Top-5 Accuracy	Parameters	MACs
AlexNet[3]	8	63.3%	84.6%	62M	666M
VGG-16[4]	16	74.3%	91.9%	138M	15.3B
GoogleNet[5]	22	68.9%	88%	6.8M	1.5B
MobileNet[5]	28	70.9%	89.9%	4.2M	569M
ResNet-50[34]	50	75.3%	92.2%	25.5M	3.9B

The development of high accuracy CNN models[3;4;34–36] in recent years have strengthened the notion of employing DNNs in various AI applications. The classification accuracy of CNNs for the ImageNet challenge[9] has improved considerably from 63.3% in 2012 (AlexNet[3]) to a staggering 87.3% (EfficientNetV2[11] in 2021). This high jump in accuracy comes with high compute and energy costs for CNN inference. Table 1.1 shows some of the most commonly used CNN models. The models are trained using the ImageNet dataset[9] and the top-1 and top-5 classification accuracy is also given. It can be seen that the addition of more layers results in better accuracy. This addition, however, also corresponds to a greater number of model parameters, requiring more memory. It also results in higher multiply-add (MAC) operations, causing an increase in computational complexity and resource requirements, which in turn, affects the performance of the edge device. Even though some efforts have been made to reduce the size of the high accuracy models, they still require massive amounts of

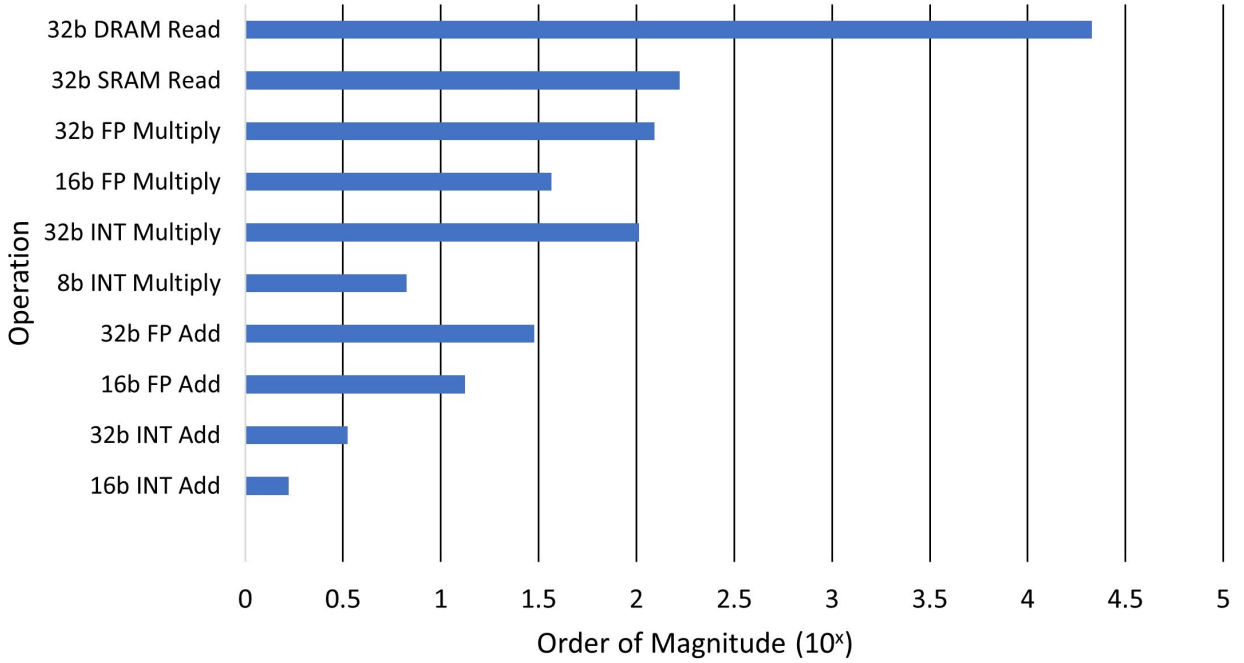


Figure 1.2: *Energy cost (relative to 8b Add operation) shown on a log10 scale for a 45nm process technology (Adapted from[1;2])*

computations over a series of network layers to perform a particular inference task (classification, segmentation, etc.). These tremendous number of computations (typically in tens of millions) present a huge challenge for the **neural network accelerators** (NNAs) running the CNN inference. NNAs are specialized hardware blocks inside a mobile device that speed up the computations of the CNN inference process to maintain the real-time requirements of the system.

Apart from the massive computational requirements, the addition of more layers for higher accuracy drastically increases the CNN model size. This prevents the CNN model from being stored in the limited on-chip SRAM memory of the edge device, and, therefore, requires off-chip DRAM which presents high DRAM access energy cost.

To put this in perspective, the energy cost per fetch for 32b coefficients in an off-chip LPDDR2 DRAM is about 640pJ, which is about 6400× the energy cost of a 32b integer ADD operation[1]. The bigger the model is, the more memory referencing will be performed to access the model data which in turn would expend more energy. Figure 1.2 shows the energy cost of various compute and memory operations relative to an 8b integer add (8b

INT Add) operation. It can be seen that the DRAM Read operation dominates the energy graph with the 32b DRAM Read consuming greater than 4 orders of magnitude higher energy than the 8b INT Add. As a consequence, the energy cost from just the DRAM accesses would be well beyond the limitations of an embedded mobile device with limited battery life. Therefore, in addition to accelerating the total number of compute operations, the NNA also needs to minimize the off-chip memory transactions for decreasing the overall energy consumption.

Many algorithm-level techniques have been developed to minimize the computational requirements of a CNN without incurring a loss in accuracy. Since, the main compute bottleneck in CNN inference is the CONV operation, *Mobilenets*[35;36] were developed to reduce the total number of CONV operations. These DNNs drastically reduce the total number of parameters and multiply-accumulate (MAC) operations by breaking down the standard 2D convolution into depth-wise separable and point-wise convolutions. This results in 8× to 9× reduction in total computations compared to regular CONV operations, with a slight decrease in accuracy. They also eliminate varying filter sizes, and instead, use 3×3 and 1×1 filters for performing convolution operations. This makes them ideal for embedded mobile devices because of their relatively low memory footprint and lower total MAC operations. A widely used approach for decreasing the memory bottleneck is the reduction in the precision of both weights and activations using various quantization strategies[37–39]. This again does not result in a significant loss in accuracy and reduces the model size by a considerable amount. Hardware implementations like Envision[40], UNPU[41], and Stripes[42], show how reduced bit precision, and quantization, translates into better savings in energy.

1.2.2 Sparsity considerations

Non-linear activation functions[2], in addition to deep layers, is one of the key characteristics that improve the accuracy of a CNN model. Typically, non-linearity is added by incorporating activation functions, the most common being the rectified linear unit (ReLU)[2]. The ReLU converts all negative values in a feature map to zeros. Since the output of one layer is

the input to the next layer, many of the computations, within a layer, involve multiplication with zeros. These feature maps containing zeros are referred to as *one-sided* sparse feature maps. The multiplications resulting from this one-sided sparsity wastes compute cycles and decrease the *effective* throughput and hardware utilization, thus, reducing the overall performance of the accelerator. It also results in high energy costs as the transfer of zeros to/from off-chip memory is wasted memory access. In order to reduce the computational and memory access volume, previous works[43–45] have exploited this one-sided sparsity and displayed some performance improvements.

Many pruning techniques have been developed for the compression of DNN models[46–49]. Han et al.[46] iteratively pruned the connections based on parameter threshold, and performed retraining to retain accuracy. This type of pruning is referred to as unstructured pruning. It arbitrarily removes weight connections in a DNN but does little to improve acceleration on temporal architectures like CPUs and GPUs which rely on accelerating matrix multiplications. Another form of pruning, referred to as structured pruning[50;51], reduces the size of weight matrices and maintains a full matrix to support the acceleration of matrix multiplications.

Both pruning strategies resulted in *two-sided* sparsity, i.e., sparsity in both weights and activations, which led to approximately 9× model reduction for AlexNet, and 13× reduction for VGG-16. It also resulted in 4 - 9× *effective* compute reduction (depending on the model). These gains seem very promising, however, designing an accelerator architecture to leverage them is quite challenging because of the following reasons:

- **Data Access Inconsistency:** Computation gating is one of the most common ways by which sparsity is generally exploited. Whenever a zero in the activation or the weight data is read, no operation is performed. This results in energy savings but has no impact on the throughput because of the wastage of compute cycle. Complex read logic needs to be implemented to discard the zeros, and instead, perform effective computations on non-zero data. Some previous works[52;53] use sparse compression formats like compressed sparse column (CSC) or compressed sparse row (CSR) to represent

sparse data. These formats have variable lengths and make *"looking ahead"* difficult if both the weight and the activation sparsity are being considered. Other than that, developing the complex control and read logic to process these formats can be quite challenging.

- **Low Utilization of the Processing Element (PE) Array:** Convolution operations for CNN inference are usually performed using an array of two-dimensional PEs in a CNN accelerator. Different dataflows (input stationary, output stationary, and row stationary) have been proposed that efficiently map the weight data and the activation data onto the PE array to maximize the throughput [2]. Sparsity introduces inconsistency in the scheduling of data thereby reducing hardware utilization. The subset of PEs provided with more sparse data have idle times while those provided with less sparse (or denser) data are fully active. This bounds the throughput of the accelerator to the most active PEs, and therefore, leads to the under utilization of the PE array.

Considering the above-mentioned issues, many accelerators have been proposed in the past that attempt to strike a balance between hardware resource complexity and performance improvements. Cnvlutin [43] attempts to exploit sparsity by skipping the computations during *zero* activation data. It, however, does not avoid the transfer of zeros and only skips cycles for zero-activations but not zero-weights. This results in exploitation of only *one-sided* sparsity. Eyeriss [45] only gates computations for sparse activations. Eyeriss v2 [52] attempts to address the two-sided sparsity by using CSC format for both the activations and weights. It, however, requires complex read logic embedded within a PE that drastically increases the area by $\sim 93\%$ when compared to the original Eyeriss [45]. Cambricon-X [44] does not store activations in the compressed format while Cambricon-S [54] forces regularity by employing coarse grain pruning that affects accuracy. Even though it discards zeros during computation, it still retrieves and stores them. EIE [53] exploits the two-sided sparsity, albeit *only* in fully-connected layers. EIE's performance is equivalent to one-sided sparsity as it discards zeros in the filter but wastes compute cycles due to being idle. Sparse CNN (SCNN) [55] targets two-sided sparsity but suffers heavily from inefficient microarchitecture and systematic load

imbalance as explained in[56]. It, also, cannot handle non-unit stride convolutions and FC layers which are an essential part of CNNs. To address the complexity associated with the CSC compression format, SparTen[56] uses a sparse bitmask to represent the location of zeros and non-zero data values. SparTen, however, needs an offline load balancing strategy, which it refers to as *Greedy Balancing*, to address the systematic load imbalance. This form of balancing adds extra latency and complicates the synchronization of various compute threads. SparTen also employs *Permuter* and *Output Collector Units* for the computation clusters to merge and/or accumulate the outputs from *independently* running compute units. These circuits require rather complex hardware and the complexity grows exponentially as the number of compute units is increased. In addition, SparTen, like SCNN, has no support for FC layers.

1.3 Contributions

In this thesis, we address the compute and energy bottleneck associated with high accuracy CNN inference on the edge device. We present different accelerator architectures that optimize the high compute volume and minimize the off-chip memory transactions for performance improvement and energy efficiency.

1.3.1 Dense accelerator - NeuroMAX

We first explore a dense accelerator design that leverages the efficiency of logarithmic quantization strategy for efficient data mapping and low area cost. The NeuroMAX accelerator[57] has the following features:

- It uses a multi-threaded, low-cost, log-based processing element (PE) core. The PE cores are stacked in a spatial grid of 108 PEs and are capable of performing a wide variety of CONV operations with high hardware utilization. The log quantization ensures minimum hardware area overhead while maximizing the throughput.

- It uses a 2D dataflow that exploits the thread-based PE design to maximize the data reuse and to minimize the off-chip DRAM memory access, consequently reducing the energy consumption.

The NeuroMAX accelerator core is implemented on the Xilinx Zynq 7020 SoC's FPGA fabric. It displays significant performance improvement over other dense accelerator architectures and is more power-efficient than its other FPGA counterparts.

1.3.2 Sparse processing core - SparsePE

The NeuroMAX accelerator, even though, provides good gains in theoretical throughput, does not, however, improves the *effective* throughput. The concept of effective throughput comes after taking the sparsity in a CNN into account. Many high accuracy CNNs are incredibly sparse which means that they have a significant amount of ineffectual computations, i.e., computations involving *zeros*. Here, we introduce SparsePE[58], a high-performance, multi-threaded, generic processing engine core for sparse CNN computations. Unlike many previous works, the micro-architecture design of this PE can exploit full or *two-sided* sparsity and can be used for sparse computations in any layer in a typical CNN model. Apart from CNNs, this core can also be used as a general-purpose, sparse dot product core which could be used for other types of DNNs. In summary, the SparsePE core has the following features:

- The core works by actively skipping a huge number of *ineffective* computations ($zero_w \times zero_a$, $zero_w \times non\text{-}zero_a$, $non\text{-}zero_w \times zero_a$) involving zeros, while only favoring *effective* computations ($non\text{-}zero_w \times non\text{-}zero_a$). This is accomplished by the use of novel selection, computation, and accumulation blocks to dynamically allocate maximum, *non-zero* computations, onto a thread matrix inside the core to drastically improve the hardware utilization. The presented PE core does not target a specific architecture, and thus, can be modified for any accelerator design.

- Unlike previous sparse accelerator approaches that use compressed sparse column (CSC) format for their PEs, the SparsePE core uses bitmask (BM) representation

for sparse computations. We show that, on average, the CSC format has 3× higher DRAM memory accesses compared to the BM representation which directly translates into higher energy requirements for the CSC format.

In short, the SparsePE core shows a significant improvement in performance over a dense PE core and requires fewer memory transactions because of its capability of exploiting the sparsity in a CNN model. It, however, has a higher area overhead compared to a dense core because of additional logic for the selection of non-zero computations.

1.3.3 Sparse accelerator - Phantom

The SparsePE core is a standalone core and cannot process a sparse CNN model on its own. Here, we introduce a sparse accelerator, referred to as the Phantom accelerator[59], that leverages the power of multiple SparsePE cores for complete CNN inference. The Phantom accelerator has the following features:

- Unlike many previously proposed accelerators, the Phantom accelerator is capable of handling all types of convolution operations, including unit and non-unit stride convolution (regular and separable). In addition, unlike the NeuroMAX accelerator, it can also handle FC layers which makes it fully deployable for any CNN processing.
- It addresses the problem of *load balancing* by using a two-tiered, on-the-fly, load balancing strategy. Unlike previous approaches[56], this balancing does not require offline processing or modification of the CNN model.
- Phantom accelerator has a performance gain of 12× over the dense NeuroMAX accelerator. For two-sided sparse accelerators, it provides a performance gain of 4.2×, 2.38×, and 1.98× over SCNN[55], Eyeriss v2[52], and SparTen[56], respectively.

Overall, the motivation behind the NeuroMAX accelerator is to show how an efficient, non-linear, logarithmic quantization can result in an accelerator architecture that can significantly speed up the computational efficiency at a very low hardware area increase. SparsePE

takes the multi-threaded PE core design from the NeuroMAX accelerator and incorporates low-cost circuits that exploit sparsity in a CNN model to drastically improve the performance. SparsePE core also modifies the input data representation to minimize the off-chip memory transactions by using a BM representation, thereby, reducing the energy consumption. The Phantom accelerator uses a spatial grid of SparsePE cores and proposes a dataflow that maximizes the utilization of the SparsePE cores. It also incorporates load balancing strategies to further improve the system-level latency.

1.4 Thesis outline

The outline of the thesis is as follows:

- Chapter 2 gives a more detailed explanation of various CNN architectures. It also gives a literature survey on different techniques for CNN inference processing on the edge.
- Chapter 3 dives deeper into the architectural details of the NeuroMAX accelerator. It provides an overview of the logarithmic quantization strategy and efficient dataflow design for leveraging the multi-threaded log PEs.
- Chapter 4 introduces the SparsePE core design. It gives details of various circuit-level techniques to leverage the sparsity in a CNN model.
- Chapter 5 gives the details of the Phantom accelerator. It also shows the optimized dataflow for performing both the CONV and the FC layer operations in a CNN. Detailed performance, energy, and hardware area comparison is also performed.
- Chapter 6 summarizes the thesis and discusses the possible future work in the domain of deep learning accelerator design.

Chapter 2

Background

In this chapter, we give an overview of the composition of different DNN models, specifically the convolutional neural networks (CNNs), and explore the different layers that these neural networks are made of. We then give a brief description of some of the most popular high accuracy CNN models and the datasets on which these CNNs operate on. Finally, we review some of the architectural and algorithmic techniques for efficient processing of high accuracy CNN models on edge devices.

2.1 Neural network architecture

DNNs are a manifestation of the notion of deep learning, which comes under the umbrella of AI. The main inspiration behind DNNs is the way different neurons work in a brain to process and communicate information. The raw sensory input is transformed into a high-level abstraction in order to extract meaningful information and make decisions. This transformation, referred to as inference, comprises many stages of non-linear processing, with each stage called a layer.

Figure 2.1 shows a simple DNN with 4 hidden layers and 1 input and output layer. The DNN layers receive a weighted sum of input values ($w_i x_i$) and compute outputs using a non-linear function, referred to as the activation function. Many activation functions have

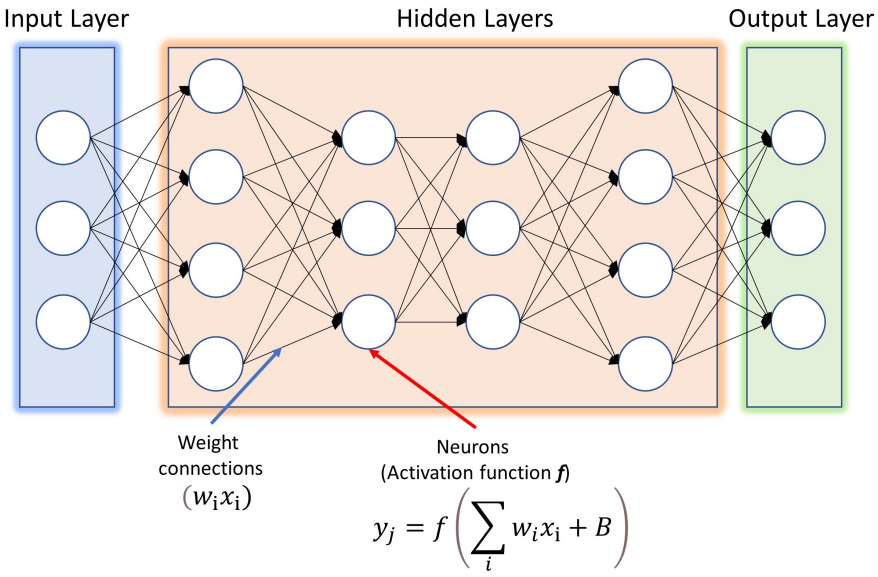


Figure 2.1: *A neural network example with 1 input layer, 4 hidden layers, and 1 output layer*

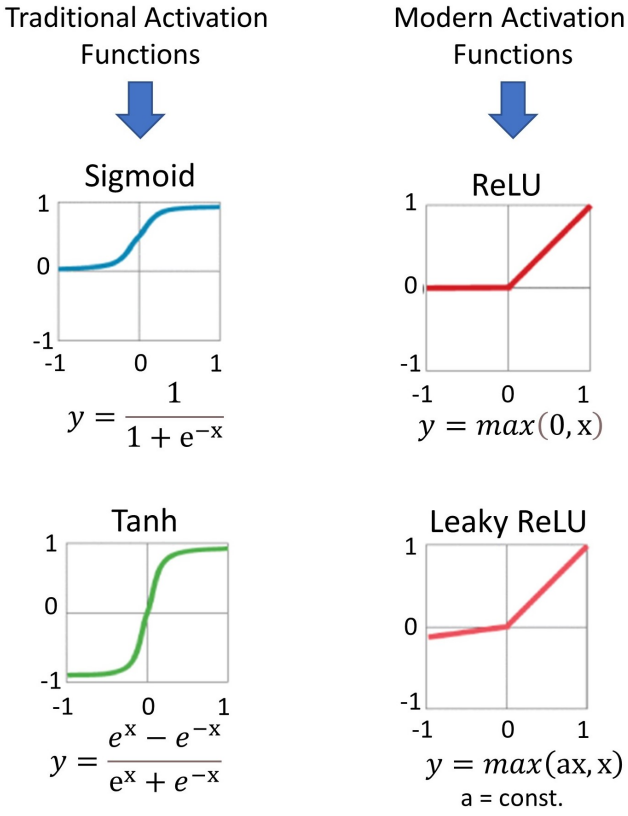


Figure 2.2: *Various nonlinear activation functions. (Figure adapted from DNN tutorial survey[2])*

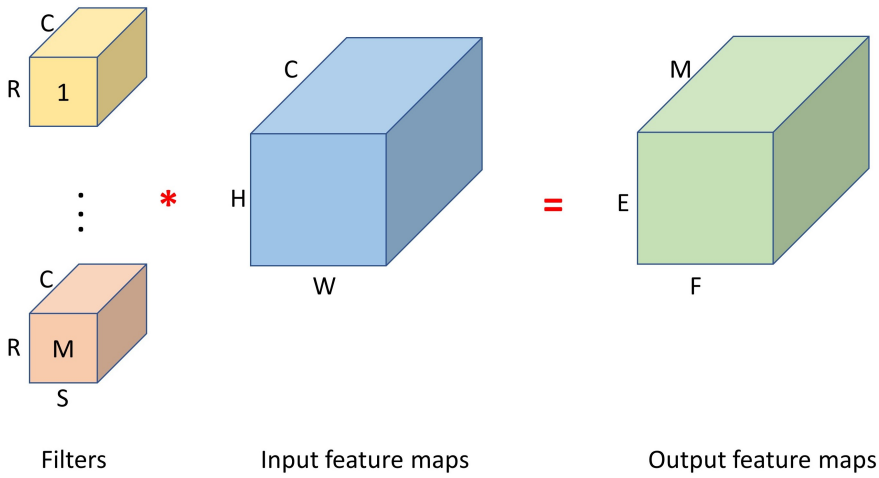


Figure 2.3: *Multi-dimensional convolution operation performed in a CNN CONV layer*

been proposed in the literature, some of which are shown in Figure 2.2. Rectified linear unit (ReLU) is one of the most commonly used activation function with many modern state-of-the-art DNNs using ReLU as the activation function. The weights in each layer are determined through the process of training. Once the training is complete after meeting a desired accuracy, the trained model is deployed on the edge device where inference is performed. In this thesis, the main focus is on the efficient inference of the trained model on the edge device.

DNNs come in various shapes and sizes depending on the target application. Multi-layer perceptrons (MLPs) are DNNs that consist of many fully connected (FC) layers, with each layer followed by a non-linear activation function. In MLPs, each neuron in layer_i is connected to every neuron in layer_{i+1}. In this way, the FC layer computations can be generalized as matrix-vector multiplications followed by the activation function. The FC layer computation can be represented as:

$$y = f(Wi + b) \tag{2.1}$$

Where i is the input vector, W is the weight matrix, b is the bias, and f is the activation function. y represents the output activation vector. It can be seen that the DNN in Figure 2.1 is an example of an MLP network.

CNNs, in addition to the FC layers, also contain multiple convolutional layers (CONV). In CNNs, each successive layer extracts features that are at a higher level of abstraction compared to the previous layers. Modern CNNs are capable of achieving superior performance by employing a deep hierarchy of layers. The CONV layers in a CNN are composed of multi-dimensional convolution operations, as shown in Figure 2.3. The input to the CONV layer is an $H \times W \times C$ matrix, called an input feature map (ifmap). Here, H and W are the height and width, respectively, of the ifmap and C is the total number of channels. For an input layer of a CNN, this ifmap is usually a $224 \times 224 \times 3$ image. Each channel of the ifmap is convolved with a unique 2D filter (weight) from a set of filters, one for each channel. The dimensions of the output feature maps are dependent on the size of the input and the filters. Based on Figure 2.3, the following equation governs the multi-dimensional convolution operation in a CONV layer of a CNN:

$$O[z][u][x][y] = B[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} I[z][k][Ux+i][Uy+j] \times W[u][k][i][j],$$
$$0 \leq z < N, 0 \leq u < M, 0 \leq x < F, 0 \leq y < E, E = (H - R + U)/U, F = (W - S + U)/U \tag{2.2}$$

Here O, B, I, and W are the ofmap, bias, ifmap, and filter matrices, respectively, whereas U represents the convolution stride. Typically, a CNN also comprises of a few FC layers after CONV layers for classification purposes.

Apart from the CONV and the FC layer, CNNs also contain pooling layers. Pooling is performed to reduce the dimensions, and variations to small distortions, making the CNN less compute-intensive and more robust. Pooling layers generally use max or averaging operations on non-overlapping blocks of ifmaps. A stride of non-unit size is applied to reduce the dimensionality of the ifmaps. Figure 2.4 shows an example of max and average pooling applied on a 6×6 ifmap to produce 3×3 outputs.

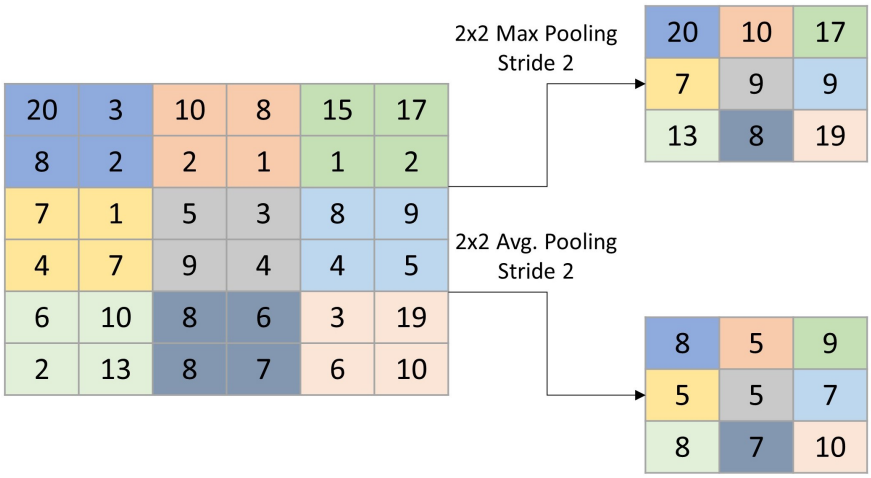


Figure 2.4: *Max and averaging pooling on a* 6×6 *ifmap*

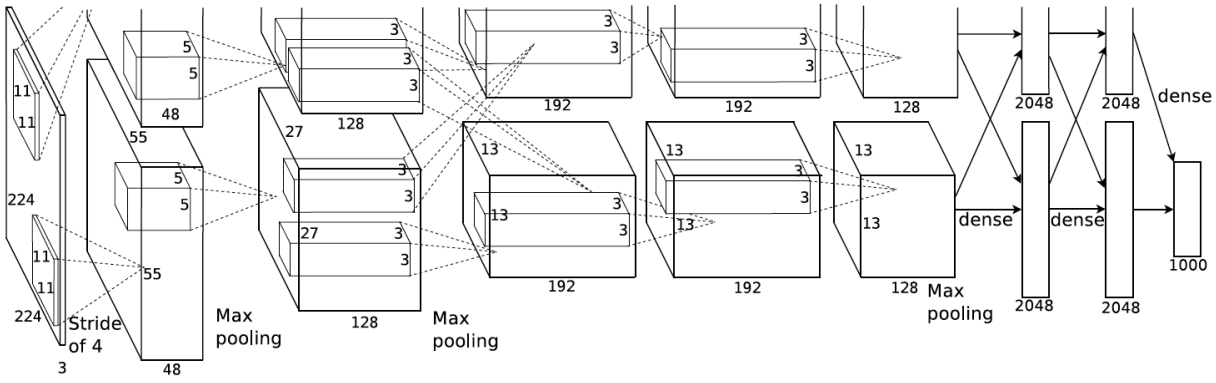


Figure 2.5: *AlexNet*[3] *architecture*

2.2 Popular CNN models

Many CNN models have been developed by researchers in the past. These models have different architectures in terms of layer dimensions, the total number of layers, layer types, and layer connections. In this section, we will give an overview of some of the most popular CNN models. We start off by showing the AlexNet[3] model which was proposed in 2012. It is the first CNN to win the ImageNet challenge[9]. Figure 2.5 shows the architecture of AlexNet. It comprises five CONV layers and three FC layers. Max pooling is performed after every CONV layer to reduce the layer dimensions. AlexNet has a total of 61 million parameters and achieves a top-1 accuracy of 57.2% and a top-5 accuracy of 80.3% on ImageNet dataset.



Figure 2.6: *VGG-16[4] architecture*

The CONV layers of AlexNet have three kernel (filter sizes): 11×11, 5×5, and 3×3.

VGGNet[4] architecture was proposed in 2014 and it comprises different versions, with each version having a different number of layers. The version shown in Figure 2.6 is the 16 layer version. It comprises of a total of 13 CONV layers and 3 FC layers. VGG-16 has a total of 138 million parameters and achieves a top-1 and top-5 accuracy of 68.5% and 88.7%, respectively. The accuracy jump in the VGG-16 net can be accounted towards the use of a greater number of CONV layers for finer feature extraction. This, however, also results in a significant increase in the model parameters which makes VGG-16 much slower than the AlexNet. AlexNet, however, uses variable filter dimensions which makes it much more difficult to implement in hardware. VGG-16, on the other hand, only uses filter dimensions of 3×3 making it more hardware friendly.

GoogleNet[5] was proposed in 2014 and is a very parameter-efficient CNN architecture. Even though it has a total of 58 layers (57 CONV and 1 FC), it only has 7 million parameters. GoogleNet comprises of specialized modules, referred to as the Inception modules, as shown in Figure 2.8. The Inception modules comprise of 4 branches of variable length filters (1×1, 3×3, 5×5). The GoogleNet achieves a top-1 and top-5 accuracy of 68.9% and 89.0%, respectively.

SqueezeNet (Figure 2.9), proposed in 2016, targets mobile applications by having a relatively smaller number of model parameters (1.2 million), but with accuracy comparable to the AlexNet. It comprises of a total of 26 CONV layers and no FC layer. SqueezeNet has 8 unique modules, which it refers to as *Fire* modules. The Fire modules are an improvement over GoogleNet's Inception module and use 1×1 convolution and a pair of 1×1 and 3×3 convolutions. It has a top-1 and top-5 accuracy of 57.4% and 80.5%, respectively.

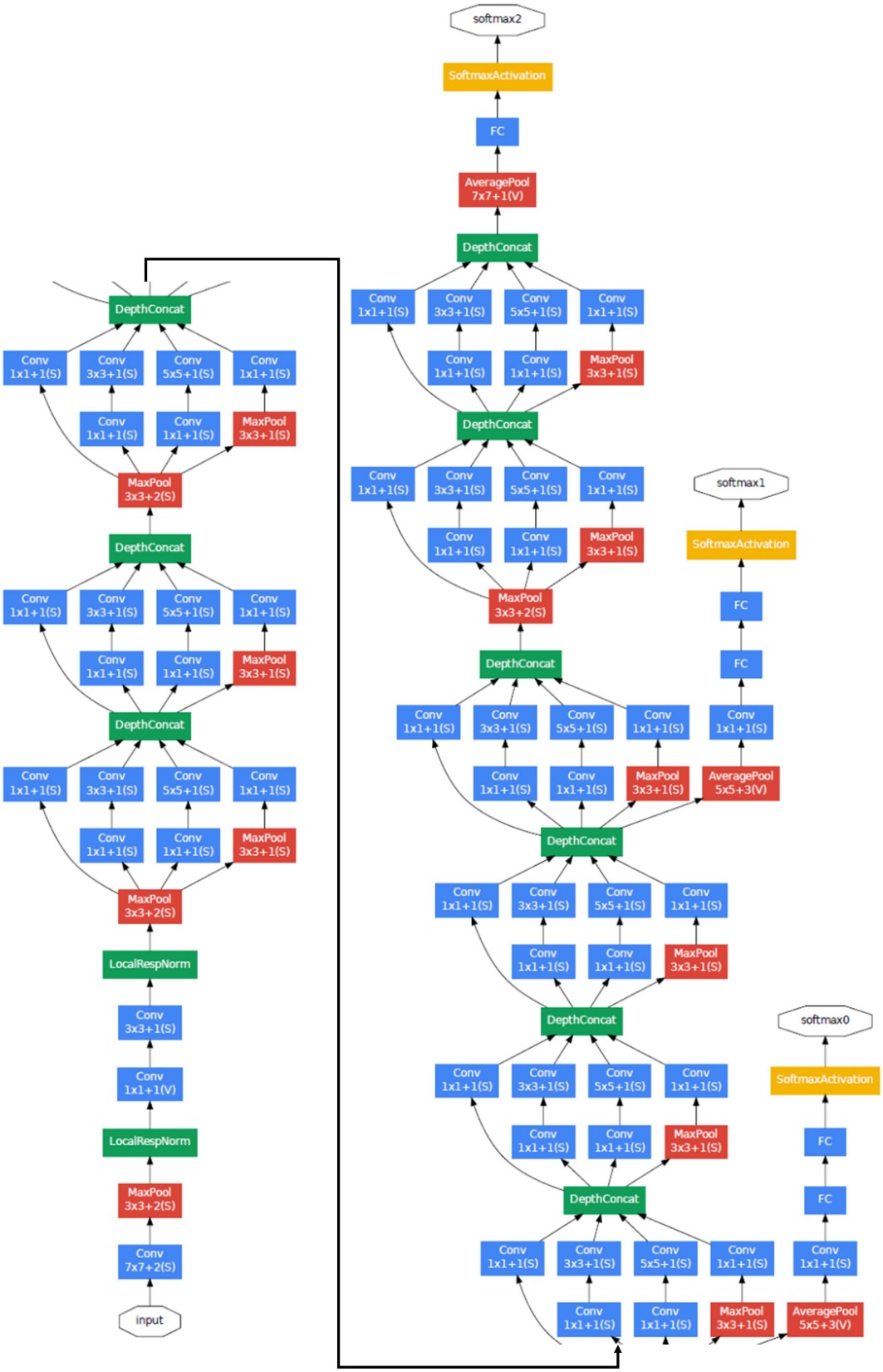


Figure 2.7: *GoogleNet*[5] *architecture*

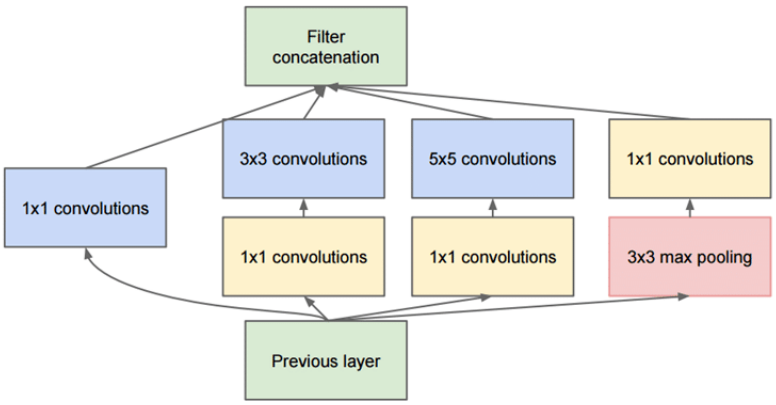


Figure 2.8: *Inception module*[5]

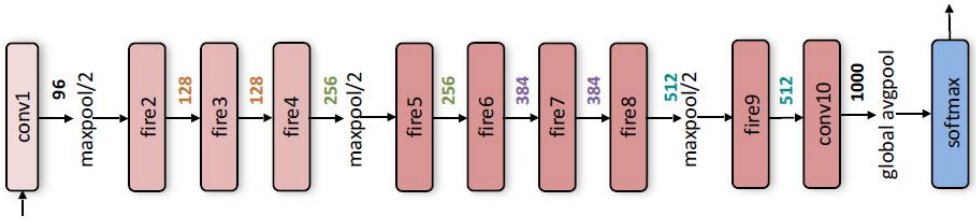


Figure 2.9: *SqueezeNet*[6] *architecture*

SqueezeNet's primary purpose is to provide CNN support for low-cost embedded platforms where high accuracy is not a huge concern.

Another class of CNNs that significantly reduce the total number of MAC computations, resulting from CONV operations, are the binary neural networks. BinaryConnect[60] modified the weights to only have two possible values (-1 and +1). These reduced the MAC operations to simple additions and subtractions. Binary Neural Network (BNN)[61] further extended the concept to activations as well which reduced the MAC operations to much more hardware-friendly XNORs. This, however, severely impacted the accuracy of the BNN[62].

Recently, another class of CNNs, referred to as *EfficientNets*, have been proposed[11;63]. EfficientNet[63], proposed in 2019, uses a dynamic scaling approach to scale all the dimensions of depth, width, and resolution, using simple yet effective *compound coefficient*. A family of neural nets, referred to as EfficientNets, are developed based on this dynamic scaling approach. The EfficientNet variant EfficientNet-B7 has a top-1 accuracy of 84.3% on ImageNet while having a parameter count of 43 million. EfficientNetv2[11] builds upon EfficientNet and drastically reduces the total number of parameters to 24 million with a slight decrease in

accuracy to 83.9%.

Overall, it can be seen that by applying various algorithmic approaches, the size of CNNs continues to decrease while the accuracy keeps on increasing with time.

2.3 Popular CNN datasets

The complexity of a DNN is generally determined by the level of difficulty of the task the DNN is used for. A simple task of digit recognition requires a much simpler DNN than a DNN that is used for classifying an object into one of the 1000 classes of objects. LeNet[12], which is a much smaller DNN, is designed for the task of digit classification, whereas, more complex CNNs like AlexNet[3], GoogleNet[5], ResNet[34], and mobilenets[35;36] are built for 1000-class image classification task.

2.3.1 MNIST dataset

The accuracy of modern DNNs is determined using a wide variety of publicly available datasets. The MNIST dataset[64], introduced in 1998, is used for digit classification. The dataset comprises of images of handwritten digits. The images are 28×28 in dimensions and are in grayscale format. There are a total of 10 classes, one for each digit from 0-9. The dataset consists of 60,000 training images and 10,000 test images. The LeNet-5[12] DNN was able to achieve an accuracy of 99.05% on the MNIST dataset. This accuracy has further increased to 99.79% using regularization techniques. The MNIST has become a fairly simple dataset and is, therefore, rarely used as a metric for DNN performance.

2.3.2 CIFAR

CIFAR dataset[65] was introduced in 2009 and consists of 32×32 colored images of objects. CIFAR-10 comprises of 10 classes of objects and has a total of 50,000 training images and 10,000 test images. Most modern DNNs are able to acheive a high level of accuracy ($\approx 96.5\%$) on the CIFAR dataset.

2.3.3 ImageNet

The ImageNet dataset[9], introduced in 2010, is by far the most widely used dataset for the accuracy modelling of modern DNNs. It contains color images of dimensions 256×256 in 1000 classes. There are multiple categories of classes with no overlap between each category. The ImageNet dataset has a total of 1.3 million training images, 100,000 test images, and 50,000 validation images. The ImageNet uses two measures to report a DNN accuracy: Top-1 and Top-5 error percentage. Top-1 requires the top-scoring category, determined by the DNN, to be correct, whereas, Top-5 refers to the object being in the top-5 classifications of the DNN. CNNs came into the picture, when, in 2012, the AlexNet[3] was able to take the top spot when it achieved a Top-1 and Top-5 accuracy of 61.9% and 83.6%, respectively, on the ImageNet dataset. The Top-5 accuracy of AlexNet displayed a significant 13.2% increase in accuracy than the second place approach, which did not use a CNN. Since then, the accuracy of CNNs has significantly improved with the latest CNNs having the top-5 accuracy as high as 97%.

2.4 CNN processing hardware

The popularity of CNNs has led many researchers and companies to design specialized hardware for accelerating the CNN processing on edge devices. Qualcomm's neural processing engine (NPE)[66] embedded Snapdragon processors enable accelerated processing of 8-bit quantized DNN models on mobile devices. Xilinx provides a deep learning processing unit (DPU)[67] for its Zynq Ultrascale+ MPSoC devices. The Xilinx DPU specifically targets the acceleration of CNNs in FPGA devices. Samsung Exynos 2100 mobile processor, powering Samsung Galaxy S21 smartphones, uses triple-core neural processing unit (NPU)[68] and can perform 26 TOPS for accelerated video processing applications. It is apparent that the trend of shifting CNN inference on the edge has become increasingly popular and the efficient design of hardware accelerators for CNN processing is of paramount importance. In this section, we will explore some of the techniques for high-performance, energy-efficient

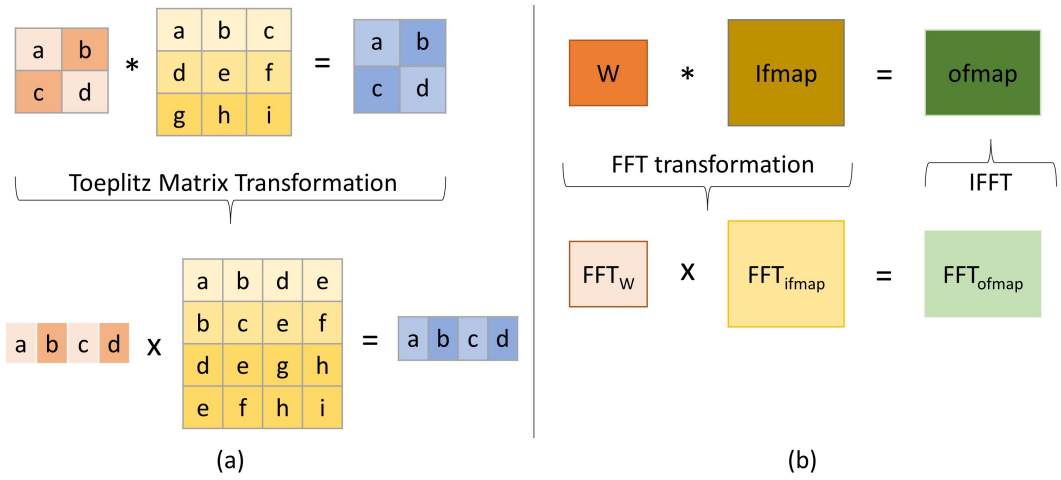


Figure 2.10: *Kernel transformation (a) Convolution to matrix multiplication using toeplitz matrix (b) Equivalent FFT computation*

hardware acceleration of CNNs on edge devices.

Looking deeper into the structure of CNNs, we can see that the basic component in a CNN is a multiply-accumulate (MAC) operation. Both the CONV and the FC layers use MAC operation for the generation of ofmaps from ifmaps. Therefore, the main problem of CNN acceleration boils down to the acceleration of MAC operations. The three most commonly used computation paradigms for CNN processing are the temporal architectures, spatial architectures, and near-memory processing.

The temporal architectures are equivalent to CPU and GPU architectures. They contain a large number of ALUs, all of which can communicate directly with the memory, but, cannot communicate with one another. These architectures use single instruction multiple data (SIMD) processing for improving parallelism. They use computational transforms on the kernel matrices to reduce the total number of computations for increasing the throughput. The spatial architectures, used for CNN acceleration in ASIC and FPGAs, use different dataflows for passing data among different ALUs (also called PEs) in an efficient manner to increase the throughput and energy efficiency. Near-memory processing techniques reduce the energy consumption by minimizing the off-chip data movement and performing computations in or near the memory.

2.4.1 Kernel transformation for temporal architectures

Many CPU and GPU architectures are optimized for matrix multiplication operations. These platforms typically implement CONV and FC layers using matrix multiplications. Figure 2.10 shows two of the most widely used convolution transformations for reducing the computational workload. Figure 2.10(a) uses Toeplitz transformation for converting convolution operations to a general matrix multiplication which can be carried out using efficient specialized hardware of a processor or a GPU. The disadvantage of using such transformation is the additional memory requirement of storing redundant data which can lead to memory inefficiencies.

Fast Fourier Transform (FFT) based approaches[69;70] reduce the computational workload from $O(N^2M^2)$ to $O(N^2log_2N)$, where the ofmap size is $N \times N$ and the filter size is $M \times M$[2]. In FFT-based transformation, the input and the weight matrices are transformed into their frequency domain counterparts and perform multiplication in the frequency domain. Inverse FFT (IFFT) transformation is applied to the output to bring the result back in the spatial domain, as shown in Figure 2.10(b). Even though FFT-based approaches reduce the total number of computations, they require more memory and bandwidth as the size of the FFT is determined by the size of the output feature map which is typically much larger than the filter size. The FFT also produces complex coefficients which require more storage.

Many other algorithms have been proposed for reducing the total number of MACs required for convolution by applying different transformation techniques. Cong et. al.[71] uses Strassen's algorithm[72] for reducing the total number of multiplications from $O(N^3)$ to $O(N^{2.81})$. Similarly, Lavin et. al.[73] uses Winograd transformation[74] for accelerating convolution operations involving 3×3 filters.

2.4.2 Dataflow design for spatial architectures

Many previously proposed accelerator architectures focus mainly on performance improvement, however, one of the major bottlenecks that limit a CNN accelerator's performance is the data movement. CNN inference generates a significant amount of data that needs

to be moved to and from the off-chip memory. There are 4 memory transactions during a MAC operation in a CNN: filter read, ifmap read, partial sum (psum) read, and psum write. Partial sums are intermediate outputs that are generated as a result of a MAC operation. Because of big layer sizes and a limited number of parallel multipliers in hardware, intermediate sums (psums) need to be stored in the memory, and then used later to generate the ofmaps. Instead of using the DRAM, which results in higher energy consumption, spatial architectures use local memory hierarchy for storing the intermediate results on the less energy-consuming on-chip SRAM memory. Since the on-chip SRAM memory is very limited, specialized dataflows are designed to maximize the *reuse* of the data brought on-chip from the DRAM.

Many dataflows have been proposed to limit the movement of the data off-chip for minimizing the energy consumption. The weight stationary dataflow (Figure 25(a) in[2]) minimizes the energy consumption of reading weights by storing them in a local register file (RF) of the PE. Since, in a CONV operation, the same weight matrix is convolved with the input feature map, it makes sense to store the weight locally instead of reading it from DRAM for every MAC operation. The global buffer present on-chip stores the generated psums and accumulates them across the PE array. Accelerator architectures presented in[75] and[76] use weight stationary dataflow for data reuse. Output stationary dataflow (Figure 25(b) in[2]) stores the psums in the local RF of the PEs. It, therefore, minimizes the energy consumption of psum movement. The psums for a particular ofmap are accumulated locally. No Local Reuse dataflow (Figure 25(c) in[2]) eliminates the local memory (RF) inside the PEs, and instead, uses a bigger global buffer. This is because, even though the RF memory is energy efficient, it is not area efficient. Thus, eliminating it translates into having a larger on-chip global memory for more local data storage. DianNao[77] and DaDianNao[78] use no local reuse dataflow in their accelerator architectures. Another dataflow, referred to as the row stationary (RS) dataflow, was proposed in Eyeriss accelerator[45]. It aims to maximize the reuse of all data types (weights, ifmaps, and psums) for overall better energy efficiency. Comparison of different dataflows shows that the RS dataflow is the most energy-efficient owing to its minimum off-chip data movement[45].

2.4.3 Near-memory processing

The previous sections show how energy consumption is minimized by the reuse of data locally instead of going off-chip. There have been several enhancements towards bringing data and processing much closer to accelerate the processing and minimize energy consumption. Embedded DRAM (eDRAM) technology places high-density DRAM on the accelerator chip for faster execution and minimum memory delays. It is approximately 2.85× denser than a standard SRAM and is 321× more energy efficient than a DDR3 memory[78]. A major downside of eDRAM is its higher cost compared to standard DRAM memory. 3D memory technology, referred to as hybrid memory cube (HMC), stacks DRAM on top of the chip to minimize the data movement delays for higher performance. Works like Neurocube[79] and Tetris[80] use HMC technology for faster and more energy-efficient CNN inference.

Another approach found in the literature for faster and more energy efficient CNN inference is processing-in-memory. Since CNN inference requires MAC operations, bringing trivial computations involving MAC into the memory itself would eliminate the need for data movement, which in turn, would decrease the energy consumption significantly. Zhang et.al[81] integrated MAC operations directly in the SRAM array by using binary weights (+1 and -1). This gives an energy saving of 12× compared to a standard 1b read from the SRAM cell. It, however, hampers the CNN accuracy on the account of using binary weights.

Chapter 3

NeuroMAX accelerator

In this chapter, we introduce NeuroMAX[1], a dense CNN accelerator capable of accelerating CONV operations. Section 3.1 provides the motivation behind NeuroMAX and also summarizes the main contributions. Section 3.2 gives an overview of some of the recently proposed dense CNN accelerators. Section 3.3 gives the architectural details of the NeuroMAX accelerator. Section 3.4-3.5 details the proposed dataflow on some use cases. Section 3.6 gives the implementation results and finally, Section 3.7 summarizes and concludes the chapter.

3.1 Introduction

Design of resource and compute efficient processing engine is of paramount importance when it comes to processing CNN on the edge devices. Since the edge devices have limited resources, the accelerator should be capable of providing the required throughput with minimum resources and area. The design of an efficient dataflow for scheduling the data into the accelerator is equally important. An inefficient dataflow results in reduced hardware utilization which causes a decrease in throughput. Dataflow should also promote the reusability of data, since, in most cases, the same kernels are being applied on the entire input feature map to produce the output.

[1]This work has been published in the 39th IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2020[57], DOI: https://doi.org/10.1145/3400302.3415638

Section 1.2 gives an overview of the energy cost of various compute and memory operations. In summary, the movement of data to/from the DRAM memory is 200× more costly in terms of energy consumption than a standard MAC operation[1]. Thus, the dataflow design should not only optimize the throughput and the area, but also the data movement, in order to ensure reduced energy expenditure. Log-based accelerators have recently gained quite a lot of traction because of their simpler structure as compared to traditional accelerators with linear processing elements (PEs). Each PE in a traditional CNN accelerator is essentially responsible for one multiplication in a convolution operation. Log PEs replace the bulky multiplier cores with low-cost barrel shifters without incurring a significant loss in accuracy. We clarify that cost here primarily refers to the logic utilization cost, which is determined by the number of Lookup tables (LUTs) for field-programmable gate arrays (FPGAs) and gate count for application-specific integrated circuits (ASICs). This logic utilization cost is important because there are limited resources on-chip, and thus, this cost also translates to the monetary cost of the system-on-chip (SoC).

Many past approaches have designed log-based PE elements but have not exploited the low-cost and overhead of such PEs. They instead rely on already established spatial architectures and 1D dataflows used for linear PEs. The **NeuroMAX** accelerator comprises of 108 PEs arranged in a $6 \times 3 \times 6$, 3D spatial grid. The accelerator optimizes the most commonly used 3×3 and 1×1 kernel sizes to achieve high throughput and utilization. It can also be used for larger kernel sizes because of its grid structure and configurable 2D dataflow. The NeuroMAX accelerator has the following contributions:

- Design of a multi-threaded, low-cost, log-based PE core. Using this core, we generate a spatial grid of $6 \times 3 \times 6 = 108$ PEs, capable of performing a wide variety of convolution operations commonly used in many CNNs.

- NeuroMAX utilizes a 2D dataflow that exploits the thread-based PE design to maximize the throughput and enhance the data reuse to minimize the off-chip DRAM memory accesses.

- The NeuroMAX architecture is implemented in software and its perfomance is com-

pared against recent dense accelerator architectures. It is also implemented on an FPGA to show improvement in terms of resource utilization and static power consumption.

3.2 Recent dense CNN accelerators

Many dense architectures have been introduced in the past for the acceleration of CNN inference. The proposed accelerators either optimize the compute[39;82–86] or the memory bandwidth[87;88]. Accelerators proposed in[89;90] use Booth encoding to avoid the use of zeros and reduce the total computations. They, however, still transfer zeros to and from the memory which incurs SRAM area and energy. Block circulant matrices for weights were introduced in CirCNN[91]. CirCNN, however, requires complex fast Fourier transform (FFT) operations in its PE design which significantly increases the area overhead.

In-memory accelerators[92;93] have also been presented that use analog logic design to perform matrix multiplications within memory. Analog circuits, however, are impacted by noise and variations during the manufacturing process which can significantly impact the CNN model accuracy during inference.

Chen et al. proposed Eyeriss[45], a non-systolic array, reconfigurable spatial architecture along with a new dataflow scheme called *row stationary* to maximize the data reuse. This design, however, incurs high PE costs owing to local storage and control in PE. It also has low hardware utilization which results in low throughput per PE. Liu et al.[94] proposed an FPGA-based CNN accelerator with an integrated depth-wise separable mode of operation. This accelerator, however, has low throughput because of the usage of a 32-bit floating point format. Bai et al.[95] proposed an FPGA-based CNN accelerator having a dedicated matrix multiplication engine (MME) on Arria 10 SoC. It achieves a frame rate of 266 fps, however, its MME engine has a huge digital signal processing (DSP) cost of 1200+ DSP blocks. Miyashita et al.[37] introduced the concept of logarithmic data representation for neural network accelerator designs. It also gives accuracy comparison between linear and log quantization. Vogel et al.[38] proposed an accelerator design using an arbitrary log base.

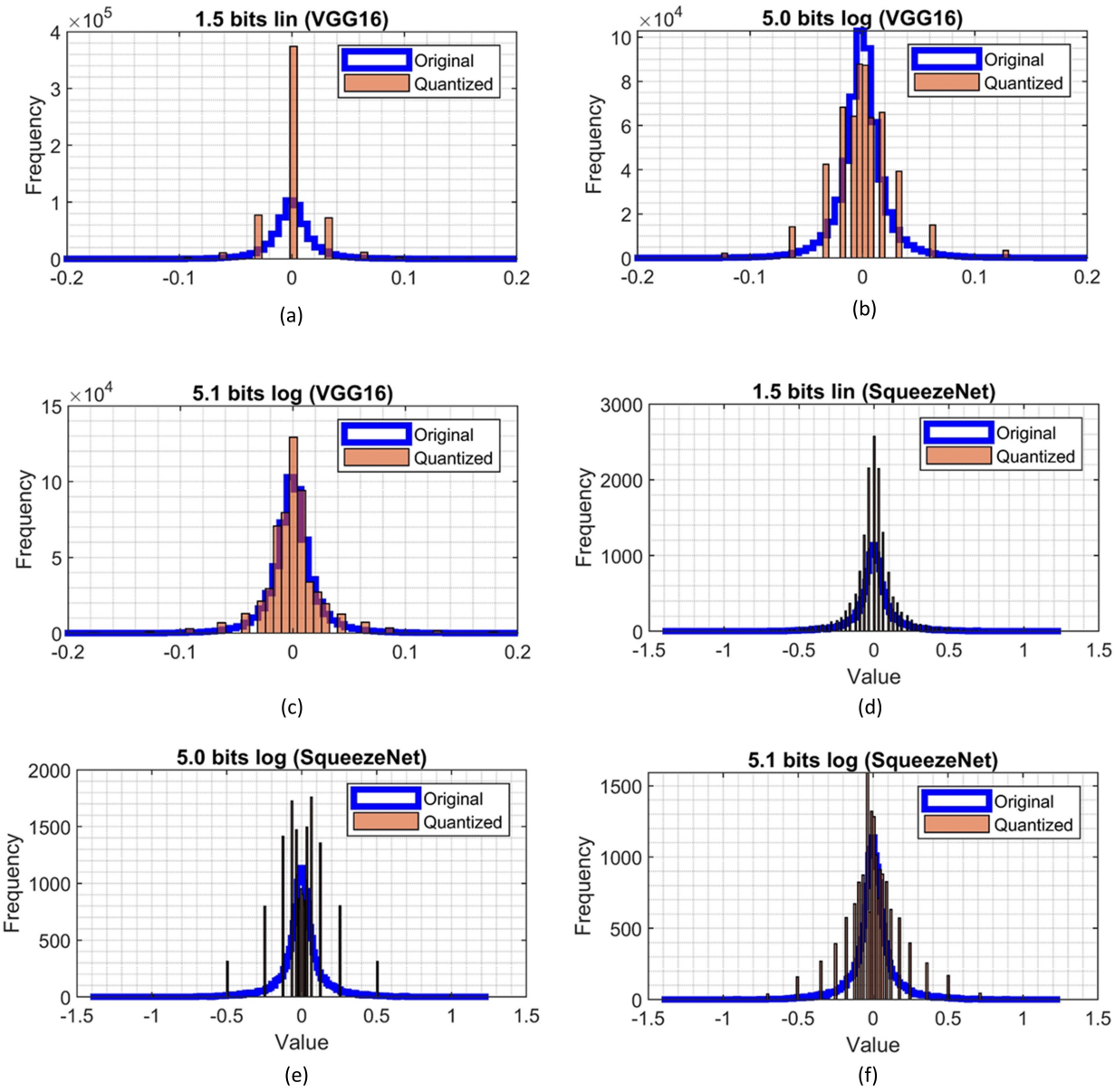


Figure 3.1: *Linear vs. log quantization (a) 1.5 bits linear vgg16 net (b) 5.0 bits log vgg16 (c) 5.1 bits log vgg16 (d) 1.5 bits linear squeezeNet (e) 5.0 bits log squeezeNet (f) 5.1 bits log squeezeNet*

It, however, does not utilize the low hardware overhead of the log-based PE and instead rely on linear PE arrangements. Huan et al.[96] proposes a reconfigurable design for various convolution kernels. It uses a propagated input data flow scheme but incurs high latency and low hardware utilization. Jo et al.[97] proposed a rescheduled dataflow for convolution to optimize the energy efficiency. Chang et al.[7] proposed VWA, a vectorwise accelerator architecture, with the goal of maximizing hardware utilization. It supports various kernel sizes from 1×1 to 5×5.

Although some of the recent designs achieve high hardware utilization, they are not able to increase the peak throughput per PE count beyond unity owing to the use of single-core, linear PEs with high area cost. NeuroMAX overcomes the limitations of prior works by leveraging *log* PEs with multiple low-cost threads within each log PE and designing a 2D dataflow that promises high throughput by exploiting multi-level parallelism.

3.3 Log mapping

Log mapping or log quantization maps an input value x to a logarithmically quantized value x'. Many trained neural nets have weights w and input activations a which are non-uniformly distributed. Mapping these 32-bit floating-point (fp32), non-uniformly distributed values over fixed-point, linearly-quantized values introduces a significant amount of quantization noise for small bit width. Most hardware platforms use fixed-point arithmetic for data manipulation where the fixed point number is represented in signed $Qm.n$ format. Here, m represents the integer part whereas n represents the fractional part. The range of values which can be represented are $range_{lin} = [-2^{m-1}, 2^{m-1} - \epsilon]$ where, $\epsilon = 2^{-n}$, is the step size.

A linear quantizer rounds the fp32 value to the nearest multiple of ϵ and then clips it as follows:

$$x_q = clip\left[\left(round(\frac{x}{\epsilon})\right) \cdot \epsilon, -2^{m-1}, 2^{m-1} - \epsilon\right] \tag{3.1}$$

where,

$$clip(x, min, max) = \begin{cases} \max, & x \geq max \\ x, & min < x < max \\ \min, & \text{otherwise} \end{cases} \tag{3.2}$$

A log quantizer takes as input, x, and the quantization parameters $< m, n, b >$, where b is the logarithmic base, and produces a log-quantized value x' as output. The quantization process can be written as:

$$x' = clip\left[(round(log_b(|x|))), -2^{m-1}, 2^{m-1} - \epsilon\right] \tag{3.3}$$

$$x_q = \begin{cases} 0, & x = 0 \\ sign(x) \cdot b^{x'}, & otherwise \end{cases} \tag{3.4}$$

Figure 3.1 shows some of the quantization results for the first five convolution layers of VGG16[4] and SqueezeNet[6]. Instead of using log base-2 (log_2) for quantization, we use log base-$\sqrt{2}$ ($log_{\sqrt{2}}$) for more accurate mapping, as shown in Figure 3.1(c) and (f). In fact, we observe that VGG16, pretrained on ImageNet dataset, with fp32 data, after $log_{\sqrt{2}}$ quantization, has top-1 accuracy decrement by only ≈3.5% from 67.5% to 63.8%. This is opposed to log base-2 quantization which decreases the accuracy by ≈10%.

3.4 Hardware architecture

3.4.1 Top-level

Figure 3.2 shows the top level hardware architecture of the proposed NeuroMAX CNN accelerator on Zynq-7020 SoC. The CONV core is the accelerator module containing a memory block, a state controller, PE grid, adder stages and post processing module. The memory block contains the weight, input and output SRAMs with a total cumulative size of 3.8Mb.

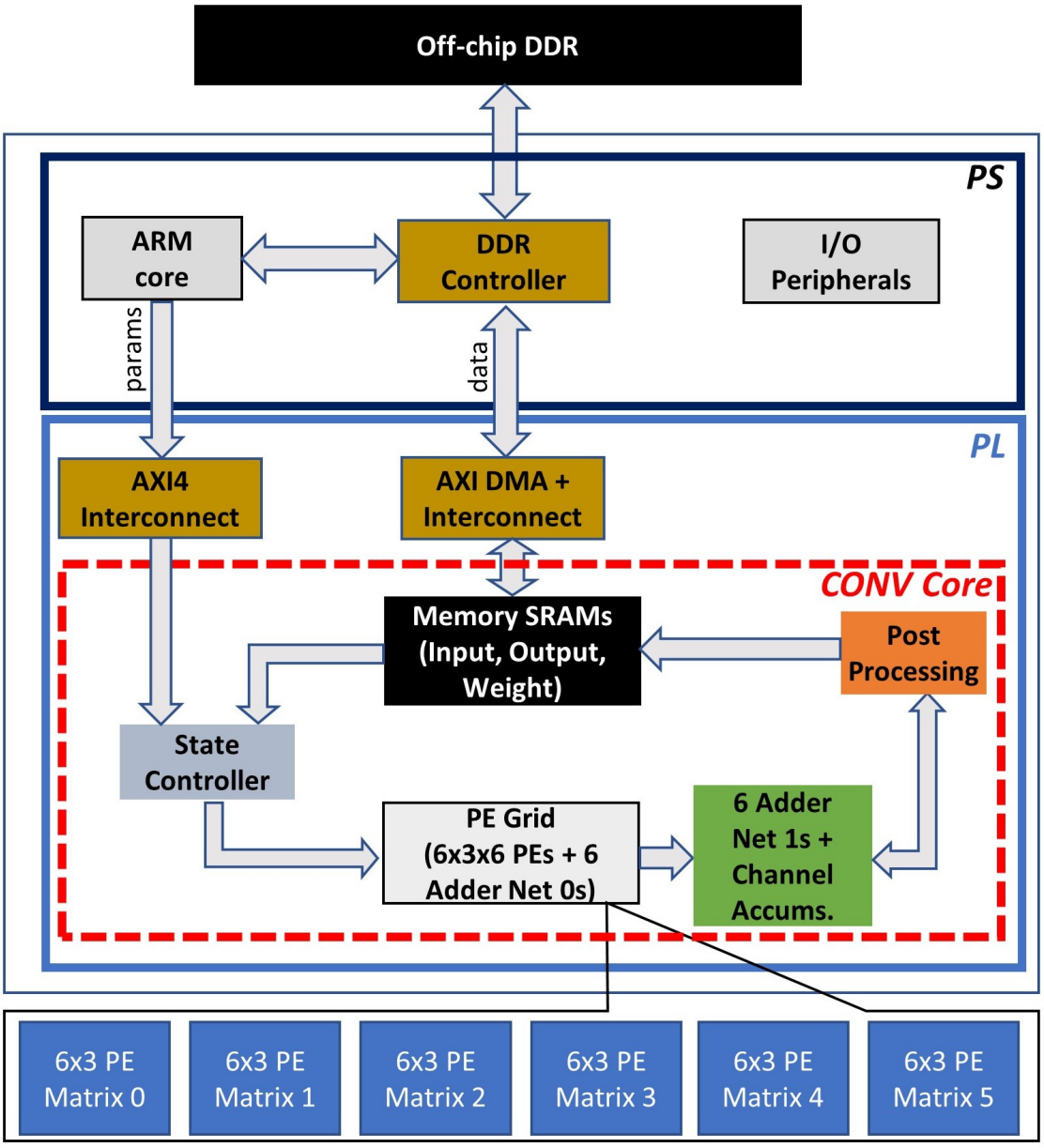


Figure 3.2: *NeuroMAX system architecture*

The PE grid consists of 108 PEs arranged in a 6×3×6 3D array. Figure 3.2 also shows the internal structure of the PE grid containing PE matrices, numbered from 0 to 5. The PE matrices are all connected to their respective input, weight, and output SRAM blocks. Each PE matrix processes independent channels in parallel for standard and separable convolutions for maximizing the throughput. The outputs from the PE matrices are provided to their respective adder nets within the PE grid. A total of six adder net 0s are present corresponding to six PE matrices. The configuration of these adder nets remain constant regardless of the type of convolution used or the filter size. The output from the adder net 0 is provided to six configurable two-stage adders whose input connections change based on the filter size and the convolution type. The first adder stage is referred to as adder net 1 and the second stage is the channel accumulation stage.

To perform a convolution operation, a tile of log-quantized input fmap and weight data is loaded from the off-chip DRAM memory into the SRAMs in the CONV core by AXI DMA and interconnect. The processor also sends the parameter information containing the values for filter size, input width, input height, output width, output height and total channels to the state controller inside the CONV core. The state controller modifies the configurable adders and determines the dataflow to be used for the convolution operation. The linear convolution outputs are sent to the post-processing block which performs ReLU operation and quantizes the results back into log values using pre-computed log table. These output log values are loaded into the output SRAMs and sent back to the off-chip DRAM memory to be used for processing the next layer. No intermediate outputs or partial sums are stored in the DRAM memory and all the intermediate processing is done within the CONV core to minimize the off-chip traffic.

3.4.2 PE matrix

Figure 3.3 shows the hierarchical design of a single PE matrix (PE matrix 0) in a bottom up view. Each PE receives a 1D vector of weight values and one input value. It should be noted that both the input (*i0'*) and the weight values ($w0_{0\text{-}2}$') are log quantized. The output

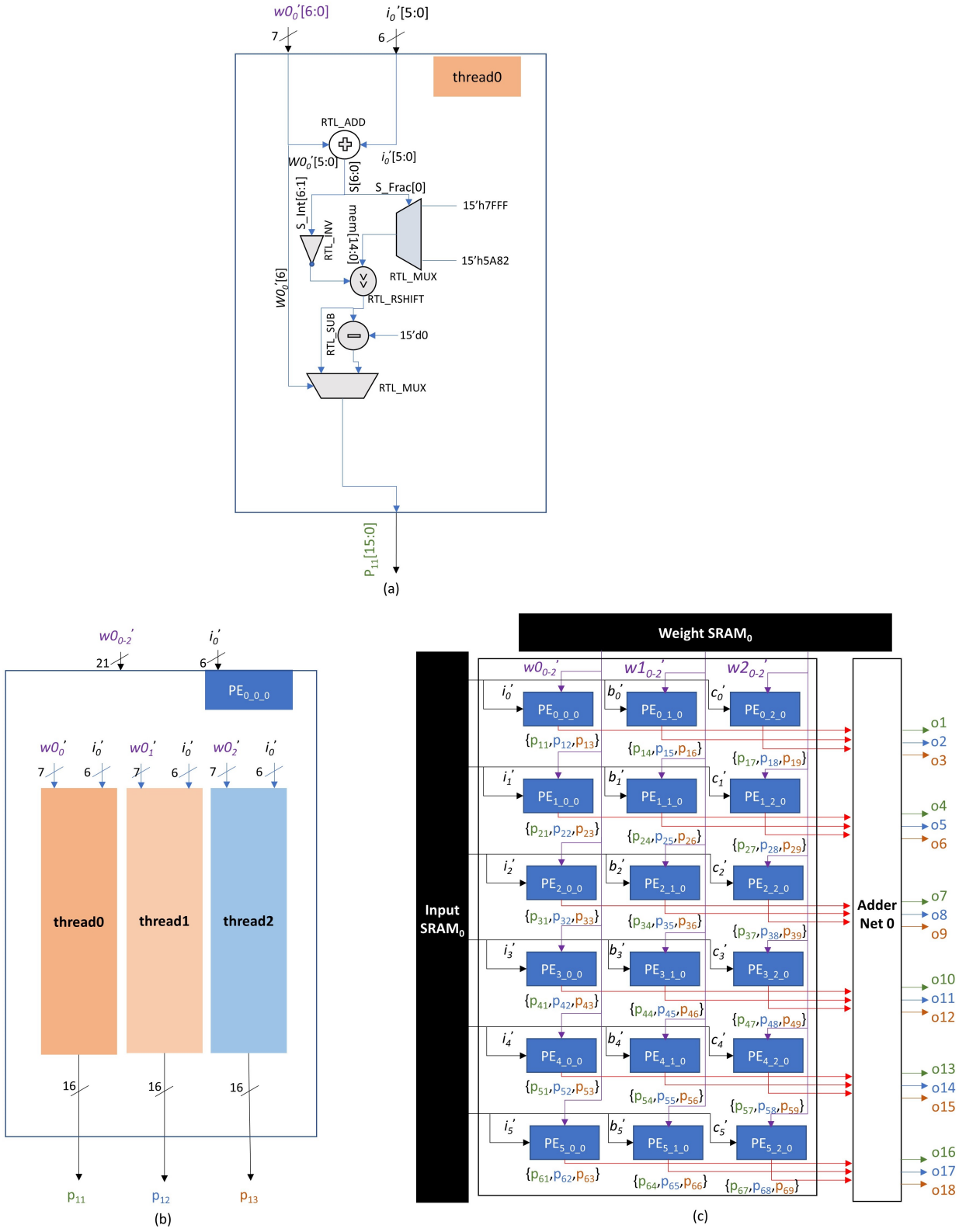


Figure 3.3: *(a) Compute thread (b) Collection of threads to make a PE (d) 6x3 PE matrix 0 and adder net 0*

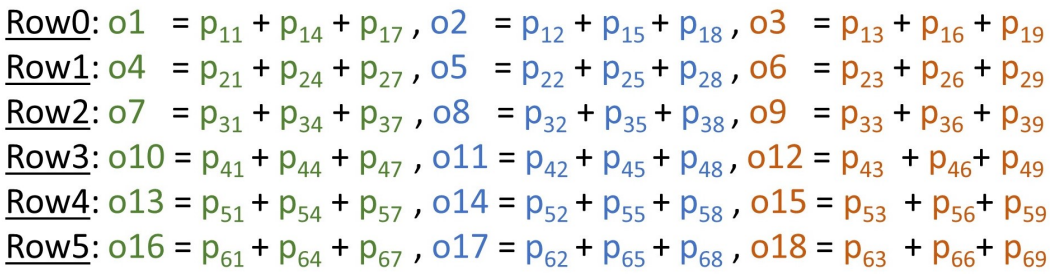


Figure 3.4: *Adder net 0 psum generation*

vectors from PEs are provided to the adder net 0 which generates 18 psums (o1-o18). This adder net works by summing the same color coded values generated within a row of PEs, as shown in Figure 3.4.

Figure 3.3(b) shows the internal structure of a single PE element ($PE_{0_0_0}$). There are three compute cores or threads, each processing a single weight data and the input value, and in turn, produce three outputs (p_{11},p_{12},p_{13}). The lowest level of the PE matrix is a thread within an individual PE, as shown in Figure 3.3(a). Basic log-based multiplication operation is performed in a single thread. Assuming we have two log-quantized values, w_q' and a_q', representing the original weight (w_q) and the activation input (a_q) respectively, the multiplication of these values in log domain can be carried out as:

$$w_q a_q = sign(w_q) \cdot 2^{g_q{}'} \tag{3.5}$$

where,

$$g_q^{'} = w_q^{'} + a_q^{'} \tag{3.6}$$

Equation (3.5) can be implemented in hardware by decomposing the exponent into its integer and fractional part as:

$$w_q a_q = sign(w_q) \cdot 2^{INT(g_q{}')} \cdot 2^{FRAC(g_q{}')} \tag{3.7}$$

The integer part $2^{INT(g_q{}')}$ can be implemented by a shift operation, whereas, the fractional part can be pre-computed and stored within the thread. The total number of fractional

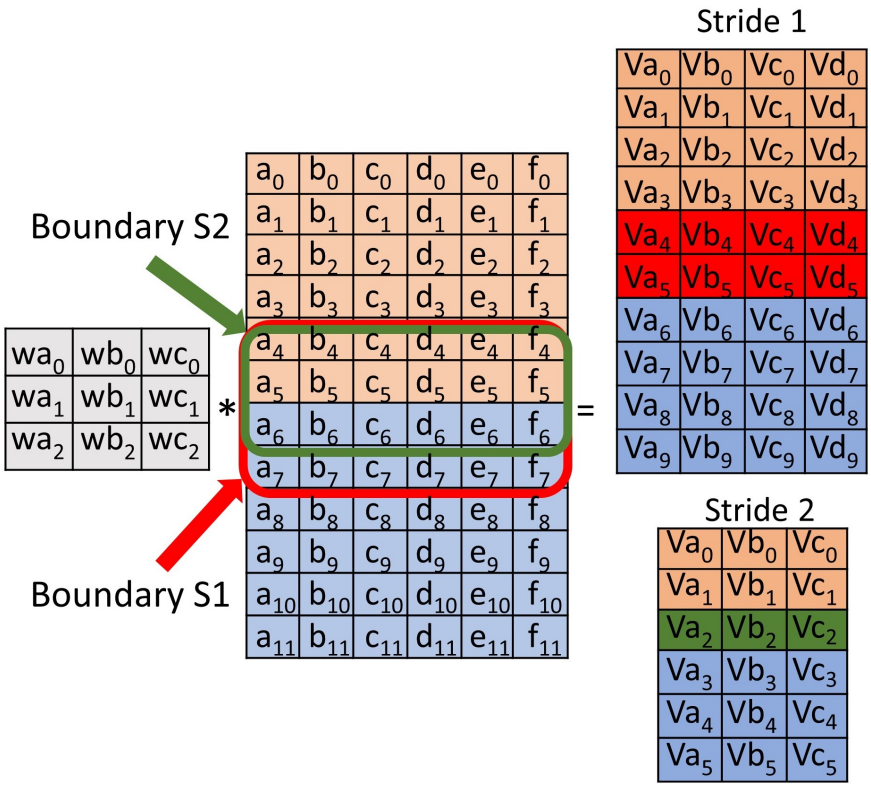


Figure 3.5: *A convolution example with 12×6 input, 3×3 filter for stride 1 and 2*

computations depends on the total number of fractional bits (n) used. In our case, we have $n = 1$ and thus store $2^n = 2$ values in the thread memory. Equation (3.7) can now be rewritten as:

$$w_q a_q = sign(w_q) \cdot (LUT(FRAC(g_q^{'})) >> \neg INT((g_q^{'}))) \tag{3.8}$$

The hardware implementation of equation (3.8) is shown in Figure 3.3(a). Since weights can have negative values, which is not accounted for in the log computations, we use an additional bit to represent the log weight data with the most significant bit $w_q'[6]$ representing the sign of the weight before quantization. This is not required for the input fmap values since most modern CNNs use ReLU activations which eliminate the negative outputs.

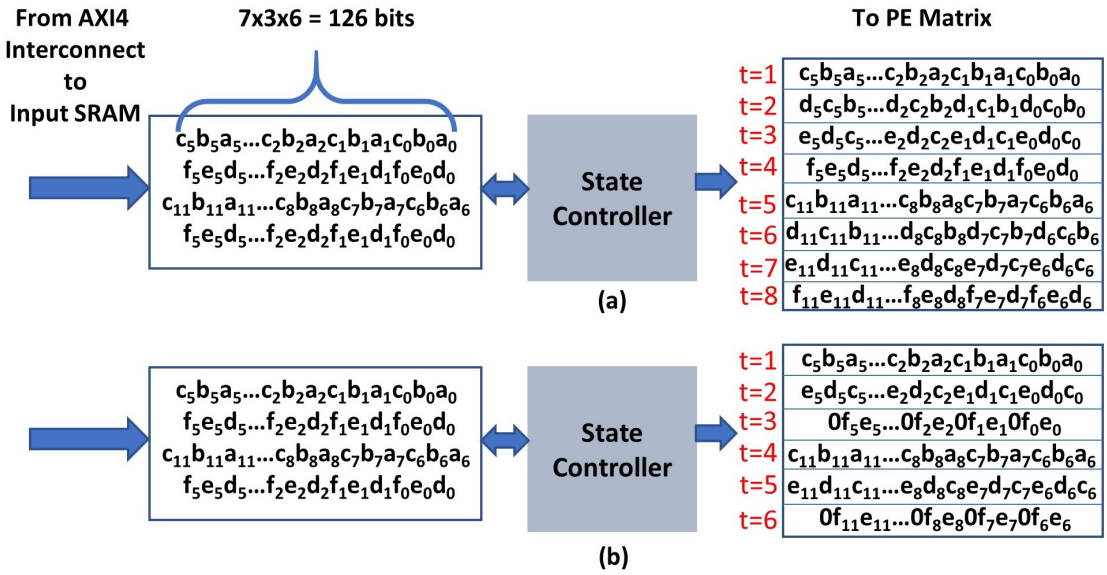


Figure 3.6: *State controller operation (a) Input, stride 1 (b) Filter weights (c) Input, stride 2*

3.5 Data flow and processing

The main idea behind designing an efficient dataflow is to minimize the data movement to/from the off-chip DRAM memory. One MAC operation typically requires three memory reads corresponding to weight, ifmap, psum, and one memory write, corresponding to the updated psum. A neural net like AlexNet, with 724M MACs, will need ≈3000M DRAM memory accesses. Many efficient dataflows have been presented in literature to minimize this data movement. Some of these include *output stationary*, *weight stationary* and, *row stationary*[2]. Since convolution operation requires the reuse of filter weights, input, and psums, in successive operations, the dataflows are designed to optimize the re-usability without accessing the DRAM memory. We introduce a 2D weight broadcast dataflow for maximizing the re-usability of the weights, input, and psums.

3.5.1 3 x 3 convolution

Figure 3.5 shows a 3×3 convolution example. Here, a 12×6 input is convolved with a 3×3 filter to produce a 10×4 output for stride 1 and a 6×3 output for stride 2. A total of 108 bits, corresponding to the 6×3 input tile, are received from the AXI4 interconnect and

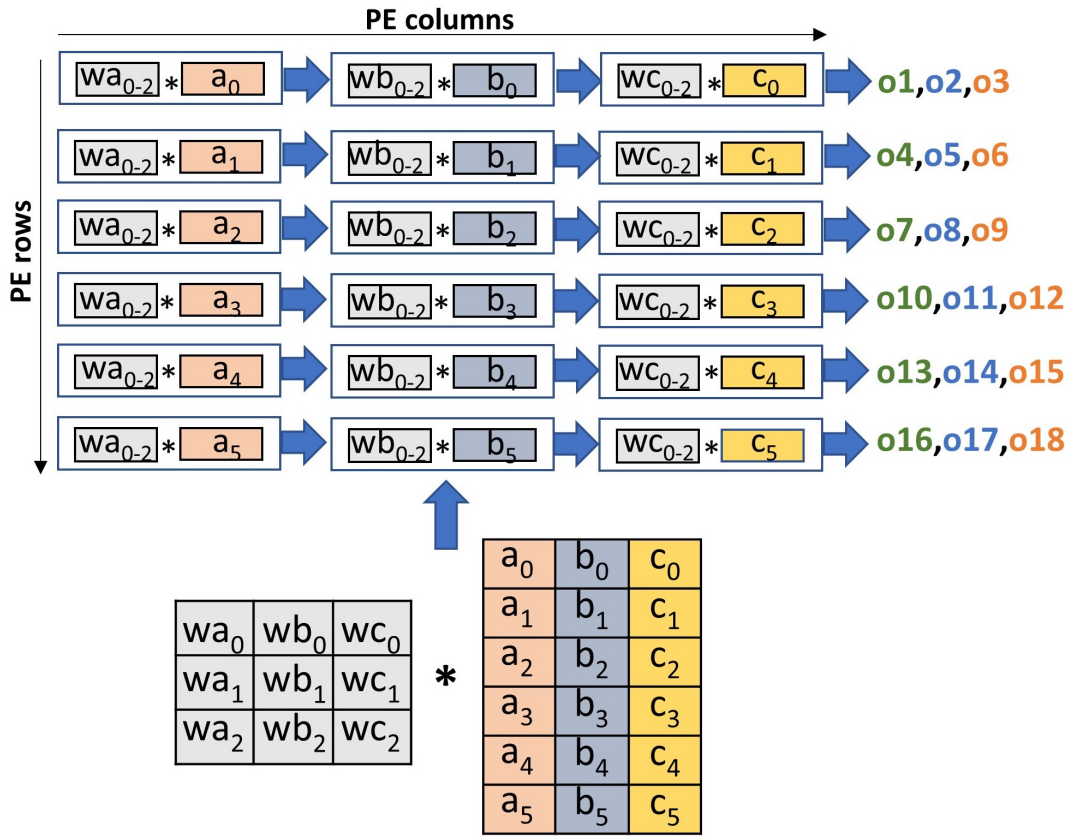


Figure 3.7: *2D weight broadcast dataflow*

stored in the input SRAM. This input tile is modified by the state controller and provided to the PE matrix in a row shifted pattern as shown in Figure 3.6(a) and (c) for stride 1 and stride 2, respectively. We also acquire a 2D weight array and broadcast it to the PE matrix as shown in Figure 3.6(b). Figure 3.7 shows the dataflow and the operation of the PE matrix for the first 6×3 input tile and the weight matrix at time stamp t = 1. The entire input tile and the 2D weight array is loaded into the PE matrix simultaneously. Because of the multi-threaded structure of PEs, each PE performs three multiplication operations using three threads and the outputs are row-wise summed to generate the psums (o1-o18) using adder net 0 (Figure 3.4). The dataflow chart and the processing of the entire 12×6 input is shown in Figure 3.8. The output a_0wa_{012}, in Figure 3.8, represents the three outputs a_0wa_0, a_0wa_1, a_0wa_2 generated by three threads within a PE. The adder net 0 computes the partial sum outputs (o1-o18) the same way as shown in Figure 3.4, where $p_{11} = a_0wa_0$, $p_{14} = b_0wb_0$ and $p_{17} = c_0wc_0$ are the same colored (green) outputs along the row.

t=1	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$c_0b_0a_0$	a_0wa_{012}	b_0wb_{012}	c_0wc_{012}
$c_1b_1a_1$	a_1wa_{012}	b_1wb_{012}	c_1wc_{012}
$c_2b_2a_2$	a_2wa_{012}	b_2wb_{012}	c_2wc_{012}
$c_3b_3a_3$	a_3wa_{012}	b_3wb_{012}	c_3wc_{012}
$c_4b_4a_4$	a_4wa_{012}	b_4wb_{012}	c_4wc_{012}
$c_5b_5a_5$	a_5wa_{012}	b_5wb_{012}	c_5wc_{012}

t=2	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$d_0c_0b_0$	b_0wa_{012}	c_0wb_{012}	d_0wc_{012}
$d_1c_1b_1$	b_1wa_{012}	c_1wb_{012}	d_1wc_{012}
$d_2c_2b_2$	b_2wa_{012}	c_2wb_{012}	d_2wc_{012}
$d_3c_3b_3$	b_3wa_{012}	c_3wb_{012}	d_3wc_{012}
$d_4c_4b_4$	b_4wa_{012}	c_4wb_{012}	d_4wc_{012}
$d_5c_5b_5$	b_5wa_{012}	c_5wb_{012}	d_5wc_{012}

t=3	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$e_0d_0c_0$	c_0wa_{012}	d_0wb_{012}	e_0wc_{012}
$e_1d_1c_1$	c_1wa_{012}	d_1wb_{012}	e_1wc_{012}
$e_2d_2c_2$	c_2wa_{012}	d_2wb_{012}	e_2wc_{012}
$e_3d_3c_3$	c_3wa_{012}	d_3wb_{012}	e_3wc_{012}
$e_4d_4c_4$	c_4wa_{012}	d_4wb_{012}	e_4wc_{012}
$e_5d_5c_5$	c_5wa_{012}	d_5wb_{012}	e_5wc_{012}

t=4	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$f_0e_0d_0$	d_0wa_{012}	e_0wb_{012}	f_0wc_{012}
$f_1e_1d_1$	d_1wa_{012}	e_1wb_{012}	f_1wc_{012}
$f_2e_2d_2$	d_2wa_{012}	e_2wb_{012}	f_2wc_{012}
$f_3e_3d_3$	d_3wa_{012}	e_3wb_{012}	f_3wc_{012}
$f_4e_4d_4$	d_4wa_{012}	e_4wb_{012}	f_4wc_{012}
$f_5e_5d_5$	d_5wa_{012}	e_5wb_{012}	f_5wc_{012}

t=5	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$c_6b_6a_6$	a_6wa_{012}	b_6wb_{012}	c_6wc_{012}
$c_7b_7a_7$	a_7wa_{012}	b_7wb_{012}	c_7wc_{012}
$c_8b_8a_8$	a_8wa_{012}	b_8wb_{012}	c_8wc_{012}
$c_9b_9a_9$	a_9wa_{012}	b_9wb_{012}	c_9wc_{012}
$c_{10}b_{10}a_{10}$	$a_{10}wa_{012}$	$b_{10}wb_{012}$	$c_{10}wc_{012}$
$c_{11}b_{11}a_{11}$	$a_{11}wa_{012}$	$b_{11}wb_{012}$	$c_{11}wc_{012}$

t=6	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$d_6c_6b_6$	b_6wa_{012}	c_6wb_{012}	d_6wc_{012}
$d_7c_7b_7$	b_7wa_{012}	c_7wb_{012}	d_7wc_{012}
$d_8c_8b_8$	b_8wa_{012}	c_8wb_{012}	d_8wc_{012}
$d_9c_9b_9$	b_9wa_{012}	c_9wb_{012}	d_9wc_{012}
$d_{10}c_{10}b_{10}$	$b_{10}wa_{012}$	$c_{10}wb_{012}$	$d_{10}wc_{012}$
$d_{11}c_{11}b_{11}$	$b_{11}wa_{012}$	$c_{11}wb_{012}$	$d_{11}wc_{012}$

t=7	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$e_6d_6c_6$	c_6wa_{012}	d_6wb_{012}	e_6wc_{012}
$e_7d_7c_7$	c_7wa_{012}	d_7wb_{012}	e_7wc_{012}
$e_8d_8c_8$	c_8wa_{012}	d_8wb_{012}	e_8wc_{012}
$e_9d_9c_9$	c_9wa_{012}	d_9wb_{012}	e_9wc_{012}
$e_{10}d_{10}c_{10}$	$c_{10}wa_{012}$	$d_{10}wb_{012}$	$e_{10}wc_{012}$
$e_{11}d_{11}c_{11}$	$c_{11}wa_{012}$	$d_{11}wb_{012}$	$e_{11}wc_{012}$

t=8	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$f_6e_6d_6$	d_6wa_{012}	e_6wb_{012}	f_6wc_{012}
$f_7e_7d_7$	d_7wa_{012}	e_7wb_{012}	f_7wc_{012}
$f_8e_8d_8$	d_8wa_{012}	e_8wb_{012}	f_8wc_{012}
$f_9e_9d_9$	d_9wa_{012}	e_9wb_{012}	f_9wc_{012}
$F_{10}e_{10}d_{10}$	$d_{10}wa_{012}$	$e_{10}wb_{012}$	$f_{10}wc_{012}$
$F_{11}e_{11}d_{11}$	$d_{11}wa_{012}$	$e_{11}wb_{012}$	$f_{11}wc_{012}$

Figure 3.8: *Dataflow chart for 3×3 stride 1 Convolution in Figure 3.5*

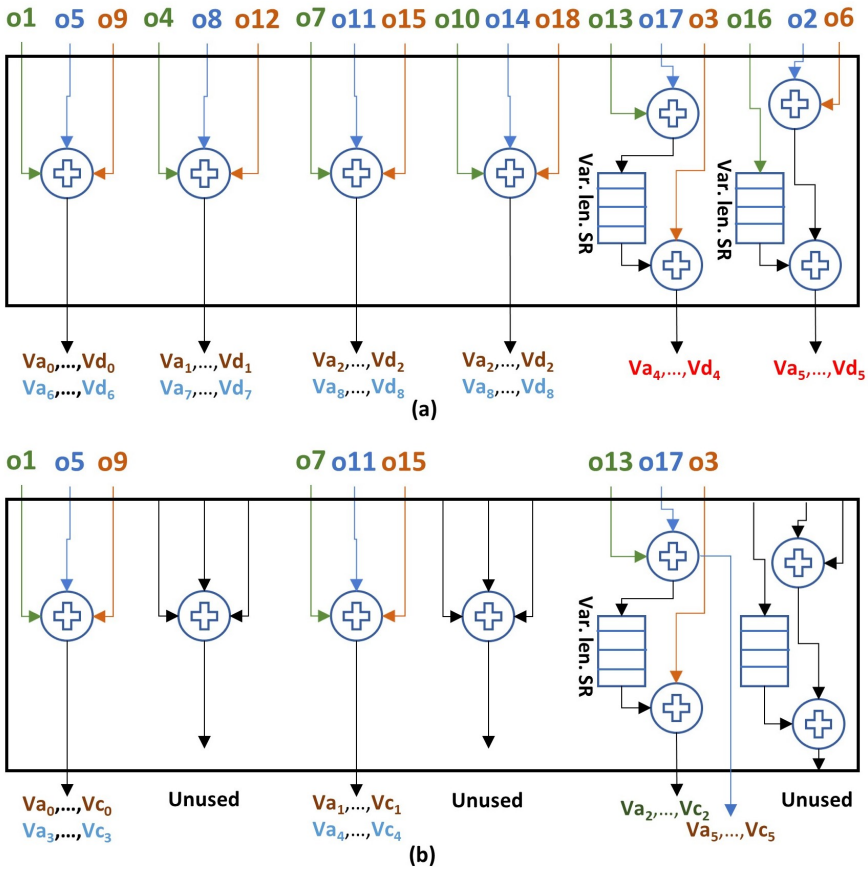


Figure 3.9: *Adder net 1 configuration (a) Stride 1 (b) Stride 2*

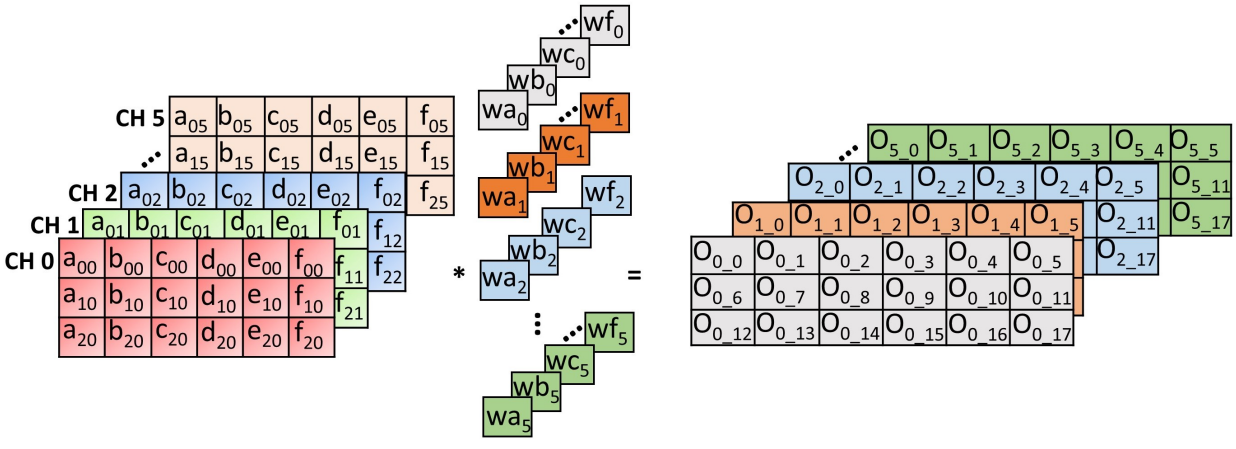


Figure 3.10: 1×1 *convolution example*

The dark red outputs (row 5 and 6) in Figure 3.5 for stride 1 and green outputs (row 3) for stride 2 represent the boundary outputs. The boundary condition occurs when the filter overlaps two different column-wise input tile sectors. For clarity, we assume that the first input tile at t = 1 is processed by the PE matrix. This corresponds to the first six rows and the first three columns of the input. The PE matrix will process the last row-wise input tile at t = 4 which corresponds to the first six rows and the last three columns of the input as shown in Figure 3.6(a). The input tile will then jump to the next column-wise 6x3 input tile which corresponds to the last six rows and the first three columns at t = 5 as shown in Figure 3.6(a). However, it can be seen that the row 5 and 6 in the output are dependent on the overlapping results from the two concurrent column-wise input tile sectors (e.g. at t = 1 and t = 5, t = 2 and t = 6 and so on). To resolve this, the three dependent psums (o13, o17 and o16), generated from row 5 and row 6, of first column-wise tile sector of the 12x6 input, are passed through a variable length shift register with the maximum length equal to the width of the input. These psums are subsequently utilized when the next column-wise 6x3 input tile (a_6 to a_{11}) is being processed. Thus, the rows 1 to 4 in the output are generated during the time intervals t = 1 to t = 4, whereas the rows 5 to 10 are generated during the time interval t = 5 to t= 8.

The output in Figure 3.5, for stride 1, is generated by alternate colored, column-wise summation of the psums in the adder net 1 as shown in Figure 3.9(a). Figure 3.9(b) shows the output generation for stride 2 case. The shift registers (VAR Len SR) for generating the boundary outputs are also shown in Figure 3.9. It can be observed that because of the optimized dataflow, only 2 out of 18 or 11% psums require local storage as opposed to ¿50% psums requiring storage (local or off-chip) in previously proposed dataflows. The throughput for the above example is 45 OPS/cycle (total OPS/total cycles = 360/8 = 45), which results in an 83.3% overall thread utilization $(45/(3 \times 6 \times 3)) \times 100$. We will simply use thread utilization as hardware utilization in this context.

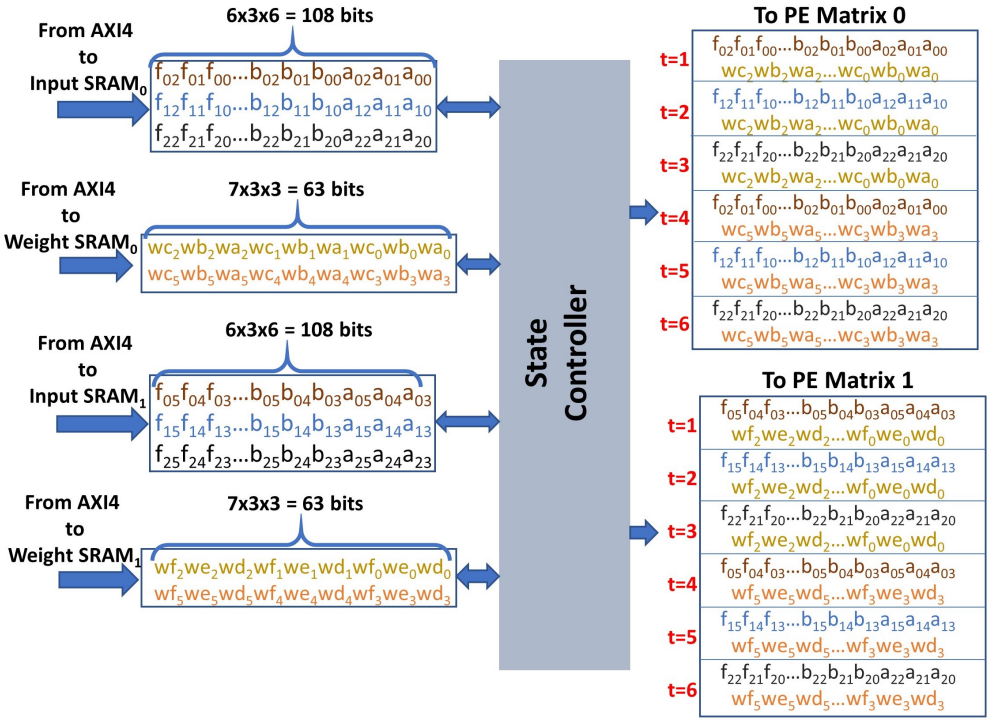


Figure 3.11: *State controller load operation*

3.5.2 1 x 1 convolution

1×1 convolutions are very popular in modern CNNs. These convolutions, along with the depth-wise separable, are replacing the normal 2-D convolutions because of the less number of MAC operations[35]. The 1×1 CONV operation convolves $1 \times 1 \times C \times P$ filters with a $M \times N \times C$ input to produce $M \times N \times P$ outputs. Here, C is the number of channels, P is the number of filters, M is the input width and N is the input height.

Figure 3.10 shows a 1×1 CONV example where a $3 \times 6 \times 6$ input is convolved with 6, $1 \times 1 \times 6$ filters to produce a $3 \times 6 \times 6$ output. Since this convolution generates the psums by channel accumulation, the outputs from the multiple PE matrices are utilized. For the example in Figure 3.10, the state controller data scheduling for PE matrix 0 and 1 is shown in Figure 3.11. It can be seen that the first three channels of the input are convolved with the first three channels of all the filters in PE matrix 0, whereas, the last three channels of the input are convolved with the last three channels of all the filters in PE matrix 1. The time stamps during specific processing of input and weights in PE matrices are also shown

t=1	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$a_{00\text{-}02}$	$a_{00}wa_{012}$	$a_{01}wb_{012}$	$a_{02}wc_{012}$
$b_{00\text{-}02}$	$b_{00}wa_{012}$	$b_{01}wb_{012}$	$b_{02}wc_{012}$
$c_{00\text{-}02}$	$c_{00}wa_{012}$	$c_{01}wb_{012}$	$c_{02}wc_{012}$
$d_{00\text{-}02}$	$d_{00}wa_{012}$	$d_{01}wb_{012}$	$d_{02}wc_{012}$
$e_{00\text{-}02}$	$e_{00}wa_{012}$	$e_{01}wb_{012}$	$e_{02}wc_{012}$
$f_{00\text{-}02}$	$f_{00}wa_{012}$	$f_{01}wb_{012}$	$f_{02}wc_{012}$

t=2	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$a_{10\text{-}12}$	$a_{10}wa_{012}$	$a_{11}wb_{012}$	$a_{12}wc_{012}$
$b_{10\text{-}12}$	$b_{10}wa_{012}$	$b_{11}wb_{012}$	$b_{12}wc_{012}$
$c_{10\text{-}12}$	$c_{10}wa_{012}$	$c_{11}wb_{012}$	$c_{12}wc_{012}$
$d_{10\text{-}12}$	$d_{10}wa_{012}$	$d_{11}wb_{012}$	$d_{12}wc_{012}$
$e_{10\text{-}12}$	$e_{10}wa_{012}$	$e_{11}wb_{012}$	$e_{12}wc_{012}$
$f_{10\text{-}12}$	$f_{10}wa_{012}$	$f_{11}wb_{012}$	$f_{12}wc_{012}$

t=3	$wa_{0\text{-}2}$	$wb_{0\text{-}2}$	$wc_{0\text{-}2}$
$a_{20\text{-}22}$	$a_{20}wa_{012}$	$a_{21}wb_{012}$	$a_{22}wc_{012}$
$b_{20\text{-}22}$	$b_{20}wa_{012}$	$b_{21}wb_{012}$	$b_{22}wc_{012}$
$c_{20\text{-}22}$	$c_{20}wa_{012}$	$c_{21}wb_{012}$	$c_{22}wc_{012}$
$d_{20\text{-}22}$	$d_{20}wa_{012}$	$d_{21}wb_{012}$	$d_{22}wc_{012}$
$e_{20\text{-}22}$	$e_{20}wa_{012}$	$e_{21}wb_{012}$	$e_{22}wc_{012}$
$f_{20\text{-}22}$	$f_{20}wa_{012}$	$f_{21}wb_{012}$	$f_{22}wc_{012}$

t=4	$wa_{3\text{-}5}$	$wb_{3\text{-}5}$	$wc_{3\text{-}5}$
$a_{00\text{-}02}$	$a_{00}wa_{345}$	$a_{01}wb_{345}$	$a_{02}wc_{345}$
$b_{00\text{-}02}$	$b_{00}wa_{345}$	$b_{01}wb_{345}$	$b_{02}wc_{345}$
$c_{00\text{-}02}$	$c_{00}wa_{345}$	$c_{01}wb_{345}$	$c_{02}wc_{345}$
$d_{00\text{-}02}$	$d_{00}wa_{345}$	$d_{01}wb_{345}$	$d_{02}wc_{345}$
$e_{00\text{-}02}$	$e_{00}wa_{345}$	$e_{01}wb_{345}$	$e_{02}wc_{345}$
$f_{00\text{-}02}$	$f_{00}wa_{345}$	$f_{01}wb_{345}$	$f_{02}wc_{345}$

t=5	$wa_{3\text{-}5}$	$wb_{3\text{-}5}$	$wc_{3\text{-}5}$
$a_{10\text{-}12}$	$a_{10}wa_{345}$	$a_{11}wb_{345}$	$a_{12}wc_{345}$
$b_{10\text{-}12}$	$b_{10}wa_{345}$	$b_{11}wb_{345}$	$b_{12}wc_{345}$
$c_{10\text{-}12}$	$c_{10}wa_{345}$	$c_{11}wb_{345}$	$c_{12}wc_{345}$
$d_{10\text{-}12}$	$d_{10}wa_{345}$	$d_{11}wb_{345}$	$d_{12}wc_{345}$
$e_{10\text{-}12}$	$e_{10}wa_{345}$	$e_{11}wb_{345}$	$e_{12}wc_{345}$
$f_{10\text{-}12}$	$f_{10}wa_{345}$	$f_{11}wb_{345}$	$f_{12}wc_{345}$

t=6	$wa_{3\text{-}5}$	$wb_{3\text{-}5}$	$wc_{3\text{-}5}$
$a_{20\text{-}22}$	$a_{20}wa_{345}$	$a_{21}wb_{345}$	$a_{22}wc_{345}$
$b_{20\text{-}22}$	$b_{20}wa_{345}$	$b_{21}wb_{345}$	$b_{22}wc_{345}$
$c_{20\text{-}22}$	$c_{20}wa_{345}$	$c_{21}wb_{345}$	$c_{22}wc_{345}$
$d_{20\text{-}22}$	$d_{20}wa_{345}$	$d_{21}wb_{345}$	$d_{22}wc_{345}$
$e_{20\text{-}22}$	$e_{20}wa_{345}$	$e_{21}wb_{345}$	$e_{22}wc_{345}$
$f_{20\text{-}22}$	$f_{20}wa_{345}$	$f_{21}wb_{345}$	$f_{22}wc_{345}$

Figure 3.12: *Dataflow chart for 1×1 convolution in Figure 3.10*

in Figure 3.11. It should be noted that for an input with more channels, the rest of the PE matrices will also be used. Thus, by using the dataflow in Figure 3.11, the architecture can process 18 channels concurrently by using the 6 PE matrices, with each PE matrix processing 3 input and filter channels.

The dataflow chart for the PE matrix 0 for the example in Figure 3.10 is shown in Figure 3.12. The same dataflow chart can also be generated for the PE matrix 1. As mentioned earlier, the psums in 1×1 convolution are calculated using channel-wise accumulation. The eighteen outputs (o1-o18) generated by the individual PE matrices are summed in their respective adder net 1s. The input connections for adder net 1 (AN 1_0) and the channel

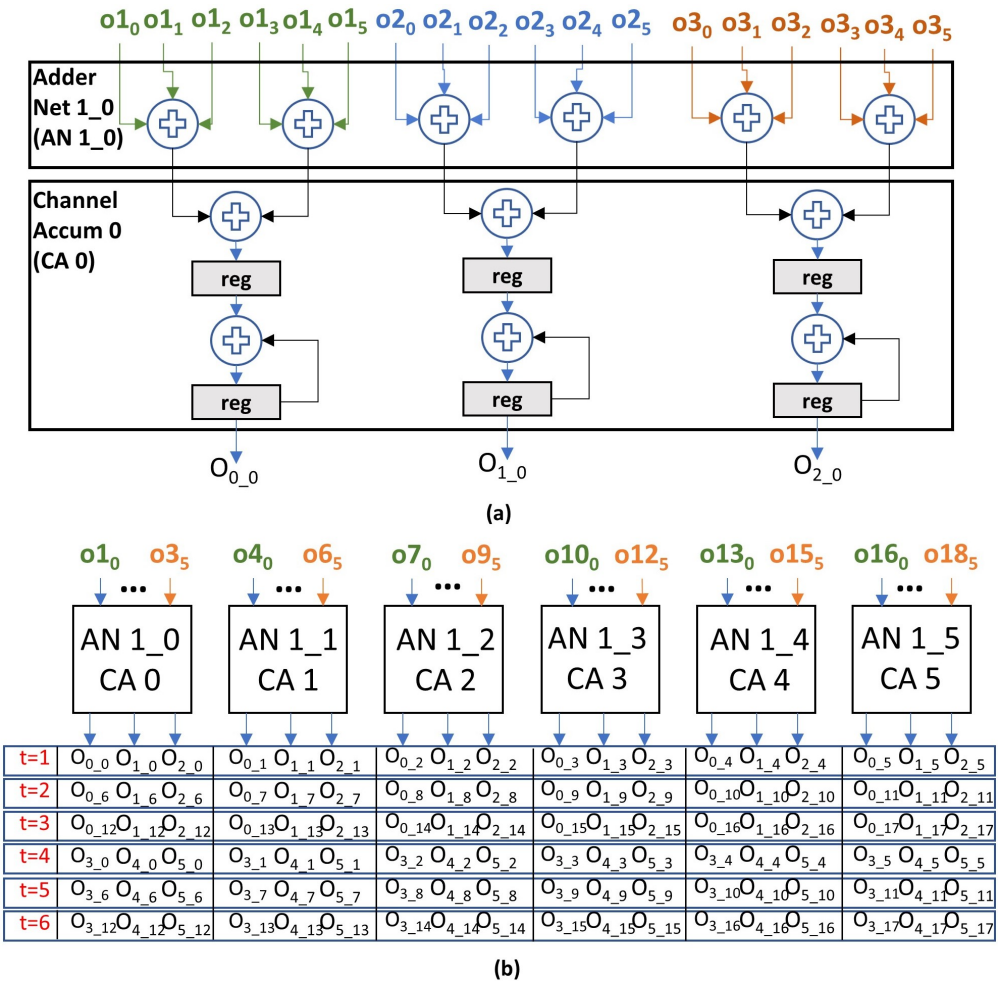


Figure 3.13: *(a) Channel-wise accumulation for PE matrix 0 (b) All channel-wise accumulations*

accumulator (CA 0) of PE matrix 0 are shown in Figure 3.13(a). Here, $o1_0$ is the psum output from the PE matrix 0 and $o1_5$ is the psum output from the PE matrix 5. Since the example in Figure 3.10 is small, it only requires the first two PE matrices and their outputs, that is, only $o1_0$-18_0 and $o1_1$-18_1 are active. The output in Figure 3.10 is generated by using all six adder net 1s and the channel accumulators as shown in Figure 3.13(b). The throughput for the above example is 108 OPS/cycle (total OPS/total cycles = $(6 \times 6 \times 3 \times 6/6 = 108)$, which results in a 100% overall thread utilization $(108/(3 \times 6 \times 3 \times 2))\times 100$.

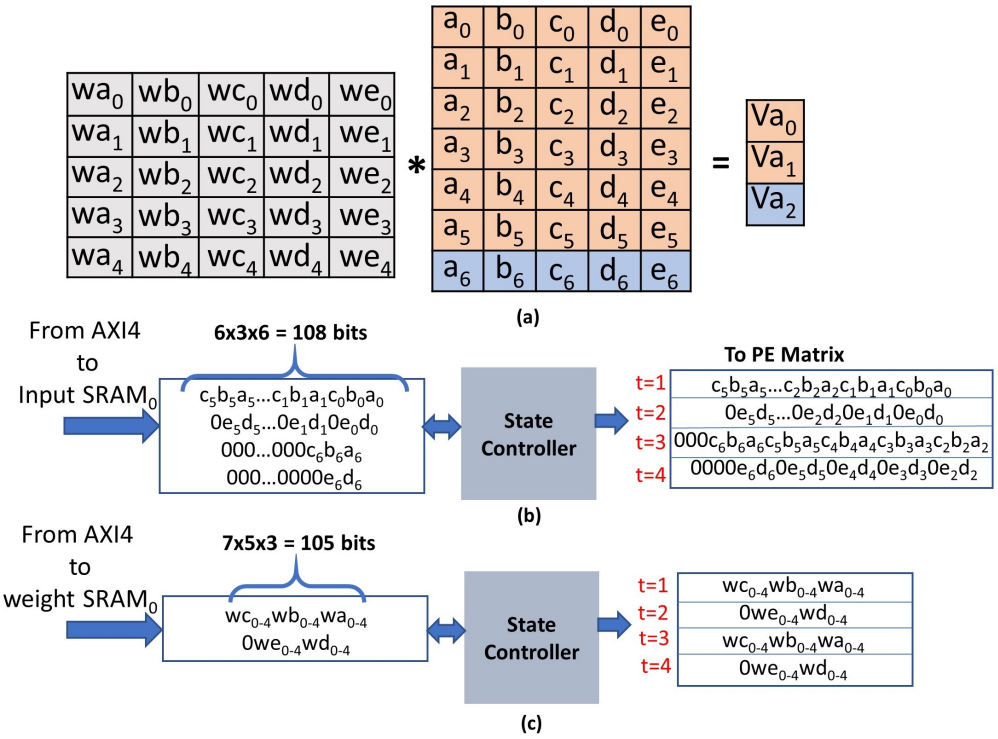


Figure 3.14: *(a) 5×5 convolution example (b) Input load operation (c) Weight load operation*

3.5.3 Higher order convolutions

The proposed NeuroMAX accelerator is designed to optimize 3×3 and 1×1 convolutions. It can, however, also be used to accelerate larger kernel sizes.[98] proposed a kernel decomposition method such that an additional support for 4×4 and 5×5 filter is needed to implement any filter size. Figure 3.14(a) gives an example of 5×5 convolution. As the size of the PE matrix is 6×3, a filter of width greater than 3 and height greater than 6 needs multiple cycles to calculate the output value. This can be seen in Figure 3.14(b) and (c) where the last two columns of the input matrix and the weight matrix are loaded at time stamp t = 2. Figure 3.15 shows the dataflow chart which accounts for this configuration. The generated psums (o1-o18) are provided to the adder net 1 as shown in Figure 3.16. For this convolution, the output values are calculated as:

$$Va_0, Va_2 = ((o1 + o5 + o9) + (o10 + o14))_{old} + (o1 + o5 + o9)_{new} \tag{3.9}$$

t=1	wa_{0-4}	wb_{0-4}	wc_{0-4}
$c_0b_0a_0$	a_0wa_{012}	b_0wb_{012}	c_0wc_{012}
$c_1b_1a_1$	a_1wa_{012}	b_1wb_{012}	c_1wc_{012}
$c_2b_2a_2$	a_2wa_{312}	b_2wb_{312}	c_2wc_{312}
$c_3b_3a_3$	a_3wa_{342}	b_3wb_{342}	c_3wc_{342}
$c_4b_4a_4$	a_4wa_{342}	b_4wb_{342}	c_4wc_{342}
$c_5b_5a_5$	a_5wa_{342}	b_5wb_{342}	c_5wc_{342}

t=2	wd_{0-4}	we_{0-4}	0
$0e_0d_0$	d_0wd_{012}	e_0we_{012}	0
$0e_1d_1$	d_1wd_{012}	e_1we_{012}	0
$0e_2d_2$	d_2wd_{312}	e_2we_{312}	0
$0e_3d_3$	d_3wd_{342}	e_3we_{342}	0
$0e_4d_4$	d_4wd_{342}	e_4we_{342}	0
$0e_5d_5$	d_5wd_{342}	e_5we_{342}	0

t=3	wa_{0-4}	wb_{0-4}	wc_{0-4}
$c_2b_2a_2$	a_2wa_{012}	b_2wb_{012}	c_2wc_{012}
$c_3b_3a_3$	a_3wa_{012}	b_3wb_{012}	c_3wc_{012}
$c_4b_4a_4$	a_4wa_{312}	b_4wb_{312}	c_4wc_{312}
$c_5b_5a_5$	a_5wa_{342}	b_5wb_{342}	c_5wc_{342}
$c_6b_6a_6$	a_6wa_{342}	b_6wb_{342}	c_6wc_{342}
000	0	0	0

t=4	wd_{0-4}	we_{0-4}	0
$0e_2d_2$	d_2wd_{012}	e_2we_{012}	0
$0e_3d_3$	d_3wd_{012}	e_3we_{012}	0
$0e_4d_4$	d_4wd_{312}	e_4we_{312}	0
$0e_5d_5$	d_5wd_{342}	e_5we_{342}	0
$0e_6d_6$	d_6wd_{342}	e_6we_{342}	0
000	0	0	0

Figure 3.15: *Dataflow chart for* 5×5 *convolution in Figure 3.14(a)*

$$Va_1 = ((o4 + o8 + o12) + (o13 + o17))_{old} + (o4 + o8 + o12)_{new} \tag{3.10}$$

In equations (3.9) and (3.10), the *old* value corresponds to the convolution output from the first three columns of the input and the weight matrix at $t = 1$, whereas, the *new* value corresponds to the last two columns at $t = 2$. The adder net 1 and the channel accumulator configuration for this convolution is shown in Figure 3.16. A similar configuration and dataflow chart is used for implementing a 4×4 convolution. In addition to this, the CONV core can also perform pooling operation by choosing the appropriate stride and kernel.

3.6 Implementation and results

This section discusses the implementation of the proposed NeuroMAX accelerator and presents the area cost, power consumption, performance, throughput, and hardware utilization results. The accelerator has been implemented in software and also on hardware (PL side of Xilinx Zynq-7020 SoC operating at 200MHz). Figure 3.17 shows cost comparison between

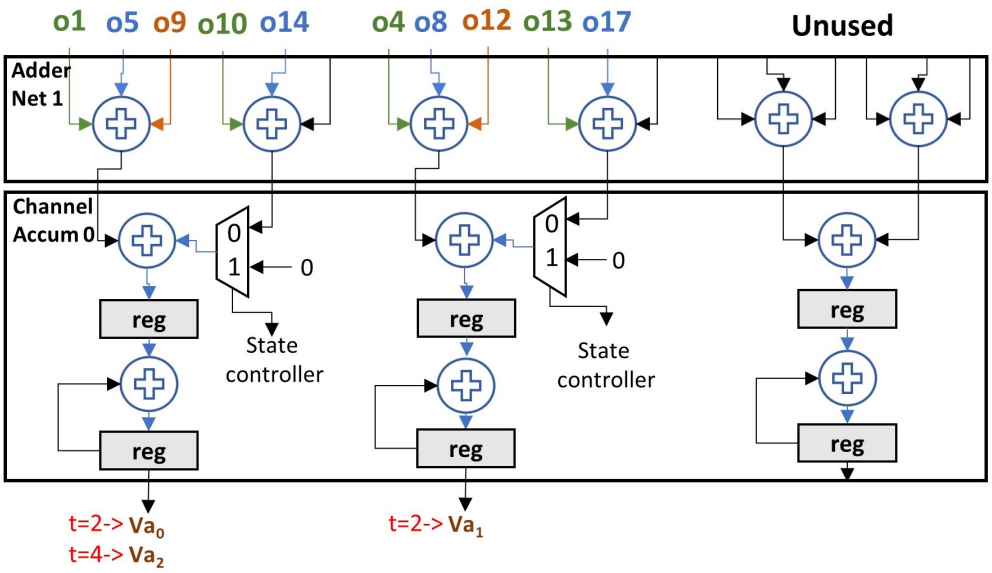


Figure 3.16: *Adder configuration for* 5×5 *convolution*

our multi-threaded log PE core and an area-optimized linear multiplier core with equal output bit precision and latency. It can be seen that by choosing a thread count of 3 (shown as log (3) in Figure 3.17), the LUT and FF cost is only 1.05× and 1.14× that of the linear PE. Thus, a total of 108 linear PEs would be equivalent, in cost, to ≈122 multi-threaded log PEs. For fairness, we will use the cost adjusted PE number for performance comparison.

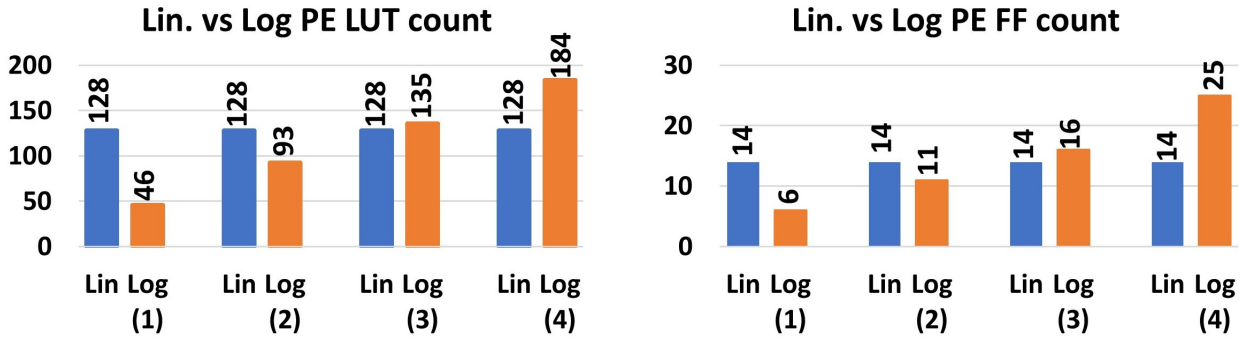


Figure 3.17: *Linear vs log PE LUT and FF cost (16-bit)*

Table 4.1 shows the resource utilization of the implemented accelerator core as well as the power consumption. Figure 3.18 (a), (b), (c) shows the breakdown of LUT cost, FF cost and power consumption among different modules of the accelerator. The PE grid and the adder net 0 combined have the highest LUT and FF count (81% and 91%, respectively). The post processing block consumes negligible resources. The processing system (ARM core)

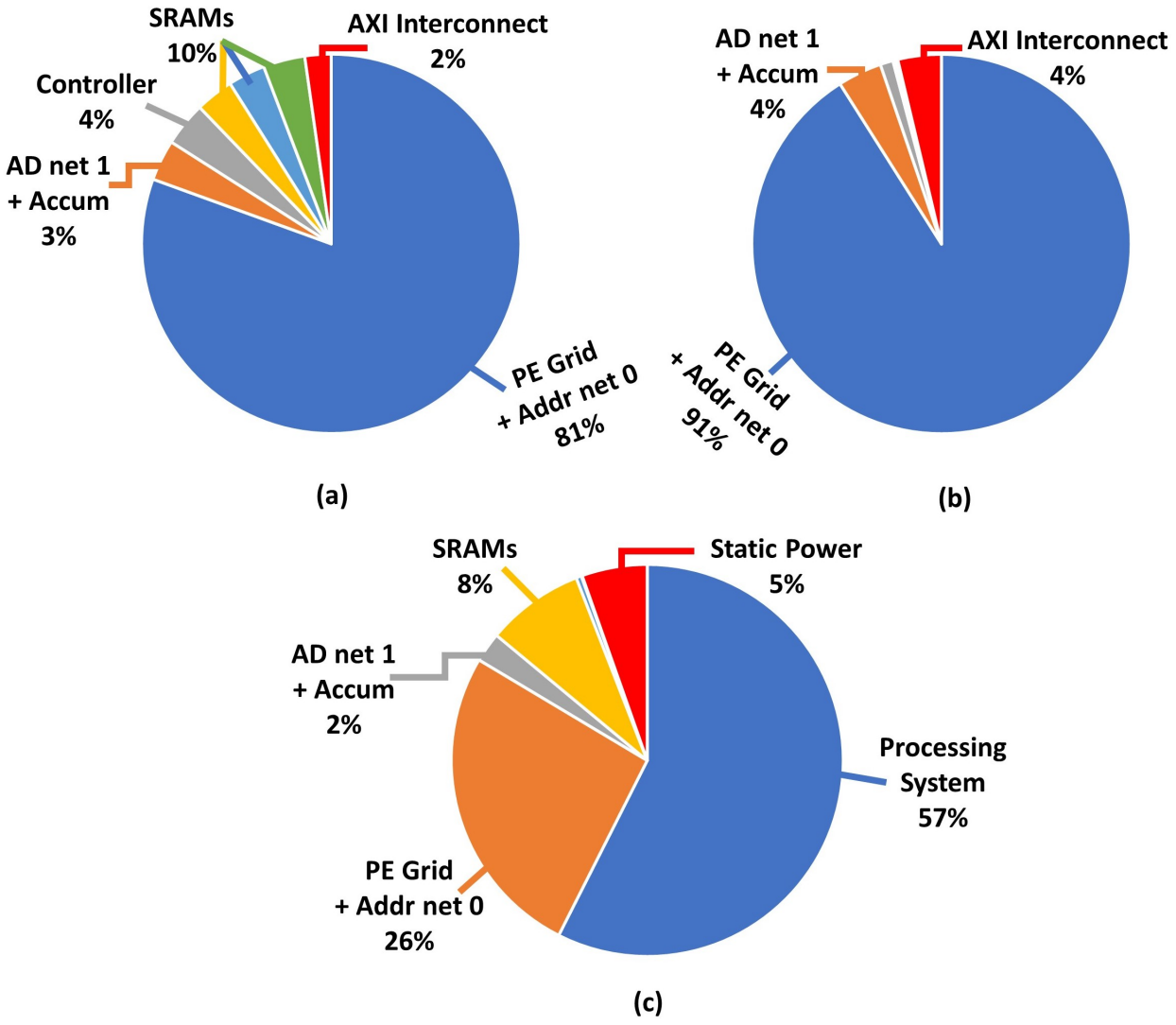


Figure 3.18: *Breakdown of (a) LUT cost (b) FF cost (c) Power consumption for NeuroMAX*

Table 3.1: *Resource utilization*

Property	Accelerator	Utilization
LUTs	20680	38%
FFs	17207	16%
36kB BRAMs	108	77%
Power	2.727 W	NA

Table 3.2: *Comparison of NeuroMAX with previous designs*

Property	NeuroMAX	Eyeriss[45]	Liu et al.[94]	Vogel et al.[38]	VWA[7]
Technology	Zynq-7020 SoC	65nm	Zynq-7100	Arria 10 SoC	Virtex-7
Precision(bits)	6-bit log	16-bit	32fp	16-bit	5-bit log
PE number	122(adjusted)	168	1926	1278	256
Processing clock (MHz)	200	200	100	133	Unreported
Peak Throughput (GOPS)	324	84	17.11	170.6	Unreported
Peak Throughput/PE	2.7(adjusted)	0.5	0.008	0.13	Unreported
Cost (LUTs(a),gates(b))	20.6k(a)	1176k(b)	142k(a)	66k(a)	29k(a)
Power (W)	2.72	0.278	4.083	Unreported	3.756

dominates the power consumption (57%), while the PE grid and adder net 0 have the second highest consumption (26%) of the total.

Figure 3.19 shows layer by layer hardware utilization for various CNN architectures. NeuroMAX achieves an average utilization of 95%, 84% and, 86% for VGG-16, MobileNet v1 and, ResNet-34, respectively. The dip in hardware utilization in some layers of mobilenet and ResNet-34 is because of stride 2 convolutions which utilize only 50% of the available PE cores. The low utilization in the first layer of VGG16 is because it only has 3 channels and since each PE matrix processes one channel, the last 3 PE matrices remain idle which gives an exact utilization of 50%.

Chang et al.[7] recently presented an accelerator, referred to as VWA, with 1D broadcast dataflow which promises higher utilization and throughput (GOPS) than all the previous designs. NeuroMAX is, therefore, compared against VWA[7] in Figure 3.20 for various CNNs. VWA uses a total of 168 PE cores and provides a utilization of 99% with throughput 166.32 GOPS, 93.4% with throughput 156.91 GOPS and, 90.2% with throughput 151.54 GOPS for VGG16, ResNet-34 and mobilenet, respectively. We use 122 PE cores (cost adjusted),

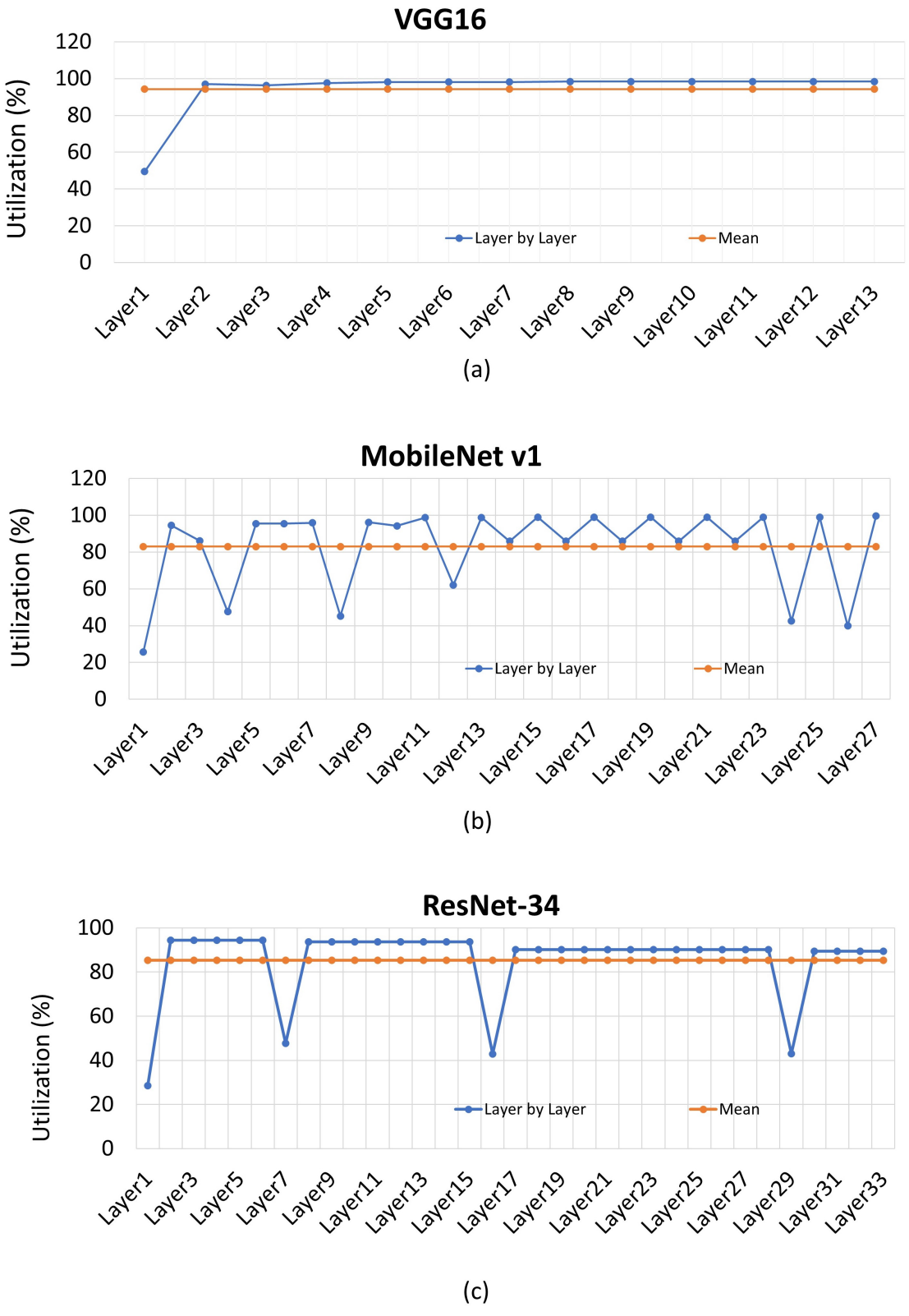


Figure 3.19: *Hardware utilization of NeuroMAX for (a)VGG-16 (b) MobileNet v1 (c) ResNet-34*

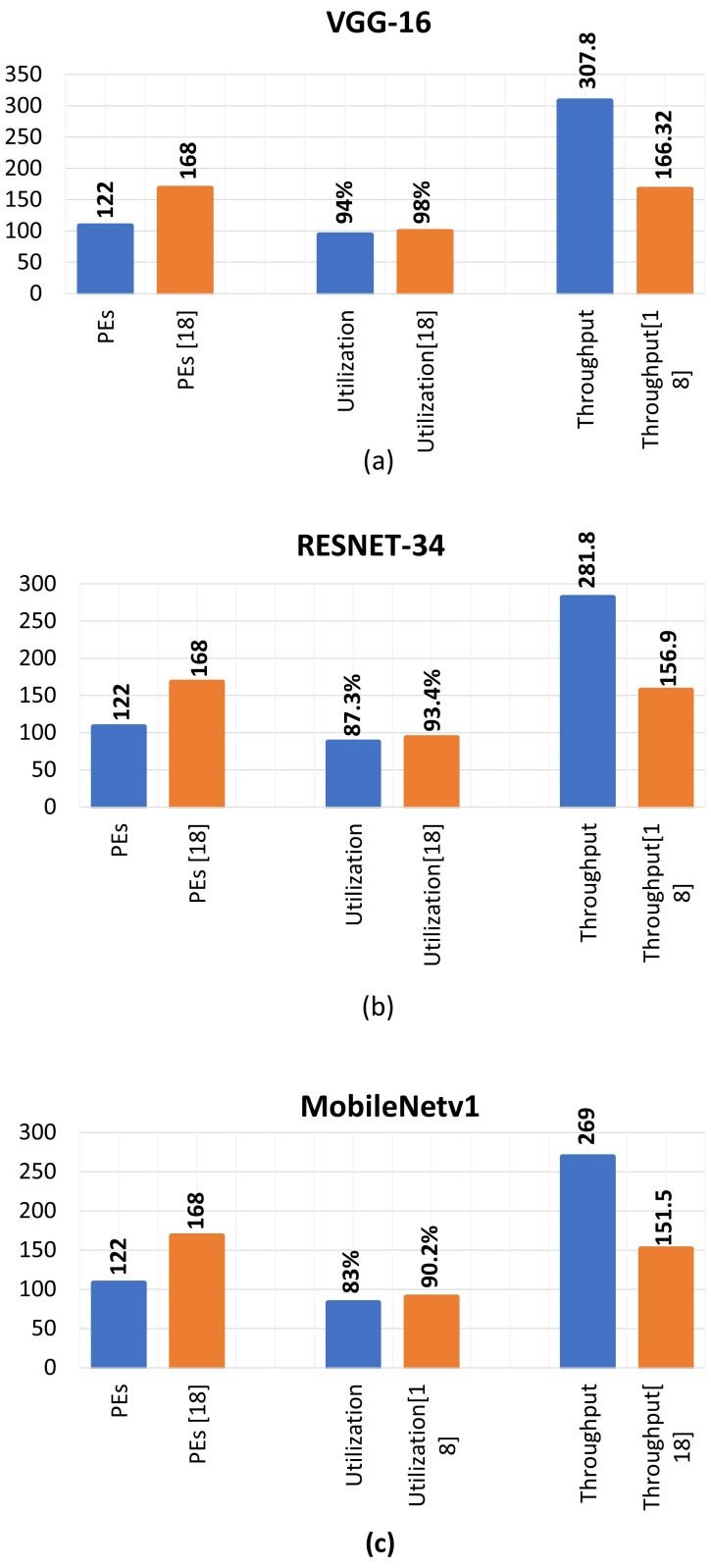


Figure 3.20: *PE count vs utilization vs throughput comparison of NeuroMAX with VWA*[7]

Table 3.3: *VGG16 latency comparison*

Layer	**NeuroMAX**	Eyeriss[45]	VWA[7]
CONV1_1(ms)	1.35	38.0	2.57
CONV1_2(ms)	28.9	810.6	55.04
CONV2_1(ms)	14.4	405.3	27.43
CONV2_2(ms)	29.26	810.8	55.7
CONV3_1(ms)	14.54	204	27.7
CONV3_2(ms)	28.6	408.1	54.5
CONV3_3(ms)	28.7	408.1	54.6
CONV4_1(ms)	14.4	105.1	27.42
CONV4_2(ms)	29	210.0	55.23
CONV4_3(ms)	29.5	210.0	56.19
CONV5_1(ms)	7.24	48.3	13.79
CONV5_2(ms)	7.23	48.5	13.77
CONV5_3(ms)	7.11	48.5	13.54
Total(ms)	**240.23**	**3755.3**	**457.5**

a 28% decrease from VWA, and provide a throughput of 307.8 GOPS, an 85% increase, 281.8 GOPS, a 79.4% increase, and 268.92 GOPS, a 77.4% increase, for the three CNNs, respectively. This increase in throughput with lower PE count is attributed towards our low-cost, multi-threaded PE core design and an efficient 2D dataflow. We also achieve somewhat similar hardware utilization, that is, 94% for VGG16, 87.3% for ResNet-34 and, 83% for mobilenet. It should be noted that VWA implements the accelerator on an ASIC, whereas, we use an FPGA, thus, an accurate comparison in LUT count, FF count and, power consumption cannot be made. It is, however, evident that the design in VWA when ported into FPGA will have ≈31% more LUTs and FFs owing to more number of PEs used. Table 5.4 shows the comparison of our accelerator with previous state of the art ASIC and FPGA designs. We see an improved performance in terms of PE number, peak throughput and peak throughput/PE ratio. Only VWA has a peak throughput/PE ratio equal to unity with average around 0.85. Our peak throughput/PE is 3 with average around 2.7 after cost adjustment. The power comparison reveals that the FPGA-based designs inherently consume more power compared to ASICs. We can, however, see that NeuroMAX consumes significantly less power and has lower cost in terms of LUT count compared to other FPGA designs. Table 4.21 gives a layer-by-layer processing latency comparison for VGG16. Both

Eyeriss[45] and VWA[7] benchmark the latency of their accelerators on this CNN, therefore, we also evaluate and compare NeuroMAX's performance on VGG16. It should be noted however that VWA uses 500MHz processing clock in their design. For fair comparison, we make suitable adjustments in their reported values. Our proposed NeuroMAX accelerator has 93% and, 47% decrease in latency, when compared to Eyeriss and VWA, respectively, at 200 MHz clock.

3.7 Conclusion

This chapter explains the architectural details and working of NeuroMAX. NeuroMAX is a dense accelerator which uses multi-threaded, log-based PE cores. The designed PE cores are capable of providing a 200% increase in peak throughput while only increasing the area overhead by 6%, when compared to a standard multiplier-based PE core. NeuroMAX uses an efficient 2D weight broadcast dataflow scheme which exploits the multi-level parallelism of the processing engine and enables hardware utilization close to a 100%. The accelerator is capable of performing a wide variety of convolutions including standard and separable 3×3 stride 1 and 2, 4×4, 5×5 and 1×1 depthwise, required in modern CNN architectures. NeuroMAX gives a throughput increase of 77.4% and a latency decrease of 47% with a 28% decrease in PE count against recently proposed accelerator designs for modern CNNs. NeuroMAX also provides at least a 27% and a 29% decrease in power consumption and LUT count, respectively, against prior FPGA-based CNN accelerators.

Chapter 4

SparsePE core

In this chapter, we introduce SparsePE[1], a processing element capable of accelerating sparse computations. Section 4.1 introduces the SparsePE core and also summarizes the main contributions. Section 4.2 gives an overview of some of the recently proposed sparse CNN accelerators. Section 4.3 gives the motivation behind sparse CNN accelerator design. Section 4.4 details the working of the SparsePE core on some use cases. Section 4.5 gives the implementation details and Section 4.6 summarizes and concludes the chapter.

4.1 Introduction

Modern CNNs owe their high accuracy to deep layers and the non-linear activation functions like ReLU[2]. The feature maps generated by ReLU comprises of zeros and are referred to as *one-sided* sparse feature maps. *Two-sided* sparsity is introduced when, in addition to the feature maps, the weight matrices also comprises of zeros. Han et al.[46] presented an approach based on iterative pruning which introduced sparsity in the weight matrices. This led to approximately 9× model reduction for AlexNet, and 13× reduction for VGG-16. It also resulted in 4 - 9× *effective* compute reduction. Computation reduction through sparsity, although very promising, introduces significant challenges in terms of implementation. Section 1.2.2 gives a detailed overview of some of these challenges. The SparsePE core is

[1]This work appears in the IEEE Access Journal[58], DOI: https://doi.org/10.1109/ACCESS.2021.3126708

designed to overcome these challenges and provides the following contributions:

- Like the NeuroMAX PE, the SparsePE core is a multi-threaded, high-performance, processing engine core which exploits two-sided sparsity to maximize the effective throughput. The core's micro-architecture is designed in such a way that it actively skipping a huge number of *ineffective* computations ($zero_w \times zero_a$, $zero_w \times non\text{-}zero_a$, $non\text{-}zero_w \times zero_a$) involving zeros, while only favoring *effective* computations ($non\text{-}zero_w \times non\text{-}zero_a$). The core's micro-architecture includes selection, computation, and accumulation blocks to dynamically allocate maximum, *non-zero* computations, on to a thread matrix inside the core to drastically improve the hardware utilization.

- Unlike previous approaches that use compressed sparse column (CSC) format for their PEs, the SparsePE core uses bitmask (BM) representation for sparse computations. We show that, on average, the CSC format has 3× higher DRAM memory accesses compared to the BM representation which directly translates into higher energy requirements for the CSC format.

4.2 Recent sparse CNN accelerators

Sparse architectures reduce the compute and memory access volume by exploiting the zeros in activations (one-sided), or both activations and weights (two-sided). Cnvlutin[43] and Cambricon-X[44] exploit one-sided sparsity by ignoring zeros in weights or input maps, but not both. Cnvlutin also does not avoid transferring zeros and only skips cycles for activations. Tensaurus[99] accelerates sparse and dense tensor factorizations by introducing compressed interleaved sparse slice (CISS) dataflow. It, however, only supports one-sided sparsity. Recent sparse GEMM (SpGEMM) accelerators[100–104] target generalized sparse-matrix, sparse-matrix multiplications. Sigma[102] and ExTensor[101] use inner-product (output stationary) dataflow for sparse matrix multiplications. Inner product, however, is inefficient for highly sparse matrices because every element of the rows and the columns must be traversed even though there are fewer effectual computations (non-zero × non-zero). This leads to a sig-

nificant amount of wasted computations. SpArch[103] and OuterSPACE[104] use outer-product (or input stationary) dataflow to avoid the inefficiencies associated with the traversals inherent in the inner-product dataflow. Outer-product, however, gives poor output reuse as the partial outputs generated are much more than the final outputs causing significant memory traffic. Finally, MatRaptor introduces a modified version of the CSR format referred to as channel cyclic sparse row (C^2SR) for better reuse and memory efficiency but requires complex encoding for output matrices. Eyeriss v2[52] uses the CSC format for both weights and activations to address the two-sided sparsity. It, however, suffers from systematic load imbalance due to variations in the density of the sparse matrices. The PE design of Eyeriss v2 also requires complex buffering logic that drastically increases the area by $\sim 93\%$ when compared to the original Eyeriss[45]. EIE[53] exploits the two-sided sparsity in FC layers and does not address the CONV layers. EIE essentially discards zeros in weights but remains idle, thus, wasting compute cycles. Sparse CNN (SCNN)[55] targets two-sided sparsity but its PEs suffer from inefficient microarchitecture and system-level load imbalance (also pointed out in[56]). SCNN, also, is incapable of handling non-unit stride convolutions and FC layers. SparTen[56] attempts to strike a balance between data reuse and load imbalance by employing an offline load balancing strategy. It, however, also cannot handle FC layers and requires complex architecture-level circuits to enable synchronization between different PEs.

Although some previous accelerator architectures attempt to exploit sparsity in CNNs, they do not address the issues related to high PE cost, inefficient micro-architecture, and dependence of PE on accelerator design. The SparsePE core not only addresses the issues present in the previous designs but also, can carry out general sparse dot product computations for any application.

4.3 Sparsity in CNNs

Sparsity refers to the fraction of zeros in a CNN layer's weight and input activation matrices. Weight sparsity is static and is introduced while pruning a network during training. Han et al.[46] developed an iterative scheme for pruning a network and retraining to retain the

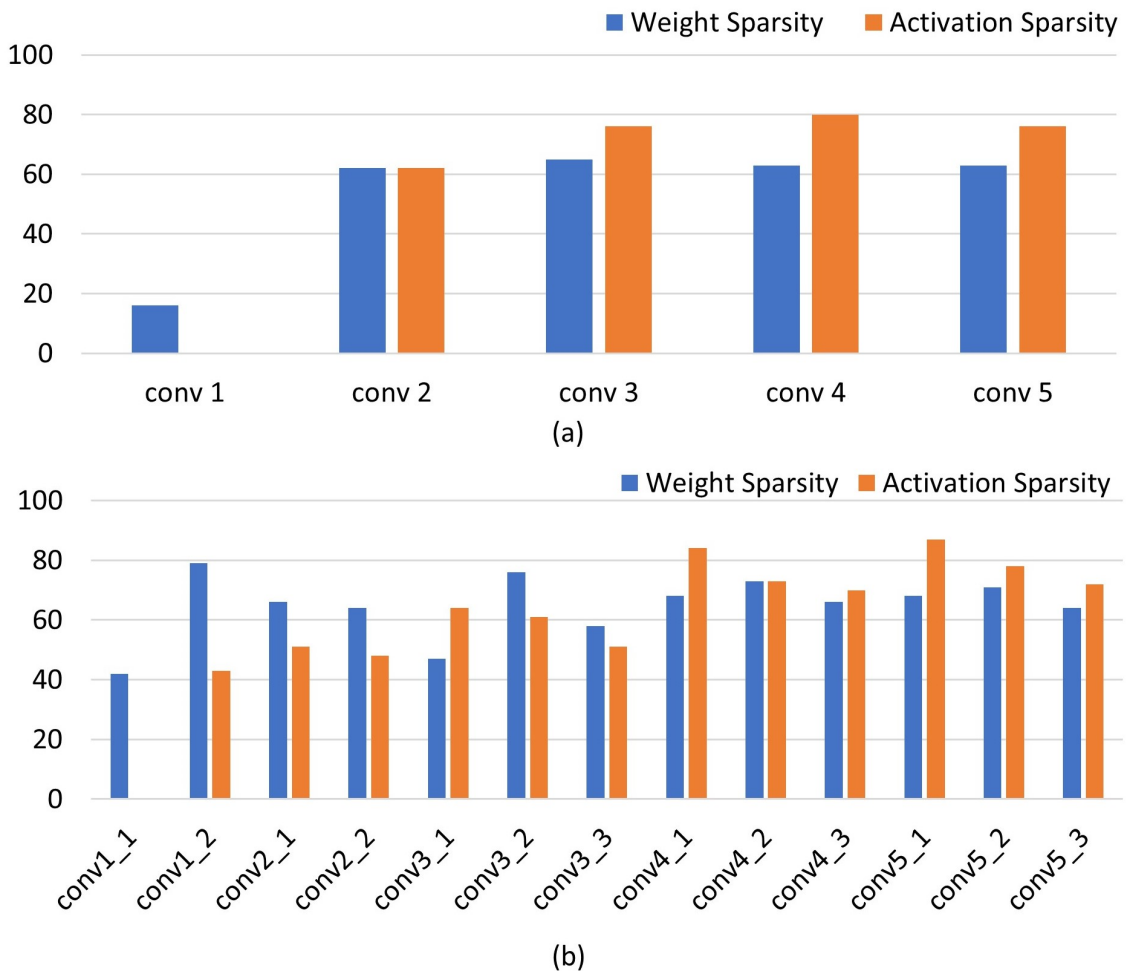


Figure 4.1: *Sparsity in CNNs (a) Sparsity in AlexNet (b) Sparsity in VGG-16*

network's accuracy. Activation sparsity is introduced dynamically during the inference phase and is highly dependent on the input being processed. This sparsity occurs because of the ReLU activation function, most commonly found in many CNNs, which converts all the negative outputs of a layer to zero.

Figure 4.1 shows the weight and activation sparsity among two of the most commonly used CNNs. We randomly select 50,000 images from the ImageNet dataset[9] and run pre-trained sparse AlexNet and VGG16. The weight sparsity values in Figure 4.1 were obtained directly from the sparse CNNs, whereas, the activation sparsity values were obtained by averaging out the layer-by-layer activation sparsity during the processing of the input images. The whole process was done using the Keras framework[105]. It can be seen that the weight sparsity for AlexNet and VGG-16 can reach as high as 70% and 80%, respectively, for some layers. Activation sparsity tends to be lower in the initial layers but rises considerably in later layers with some layers of VGG-16 having activation sparsity as high as 85%. This shows that many neural nets, though seemingly compute and memory bandwidth-intensive, are incredibly sparse with huge amounts of redundant computations. A PE design that efficiently exploits this redundancy can provide immense gains in both performance and energy efficiency.

4.4 SparsePE

Figure 4.2 shows a typical convolution operation in a CNN. Here, a 3×8 sparse input is convolved with a 3×3 sparse weight to generate a 1×6 output. The input and weight matrices have sparsity of 42% and 45%, respectively. The convolution operation can be broken down into 6 smaller convolution chunks (C0-C5), as shown in Figure 4.2. Each of the 9 multiplications in a single convolution chunk is performed by a compute thread within a 3×3 compute thread matrix. The multiplications in red are ineffective computations which means that either one or both the multiplication operands are zeros, resulting in a wasted computation, whereas, the multiplications in black are effective. It can be seen that, on average, 66% multiplications in a convolution chunk are ineffective (Output Sparsity OS =

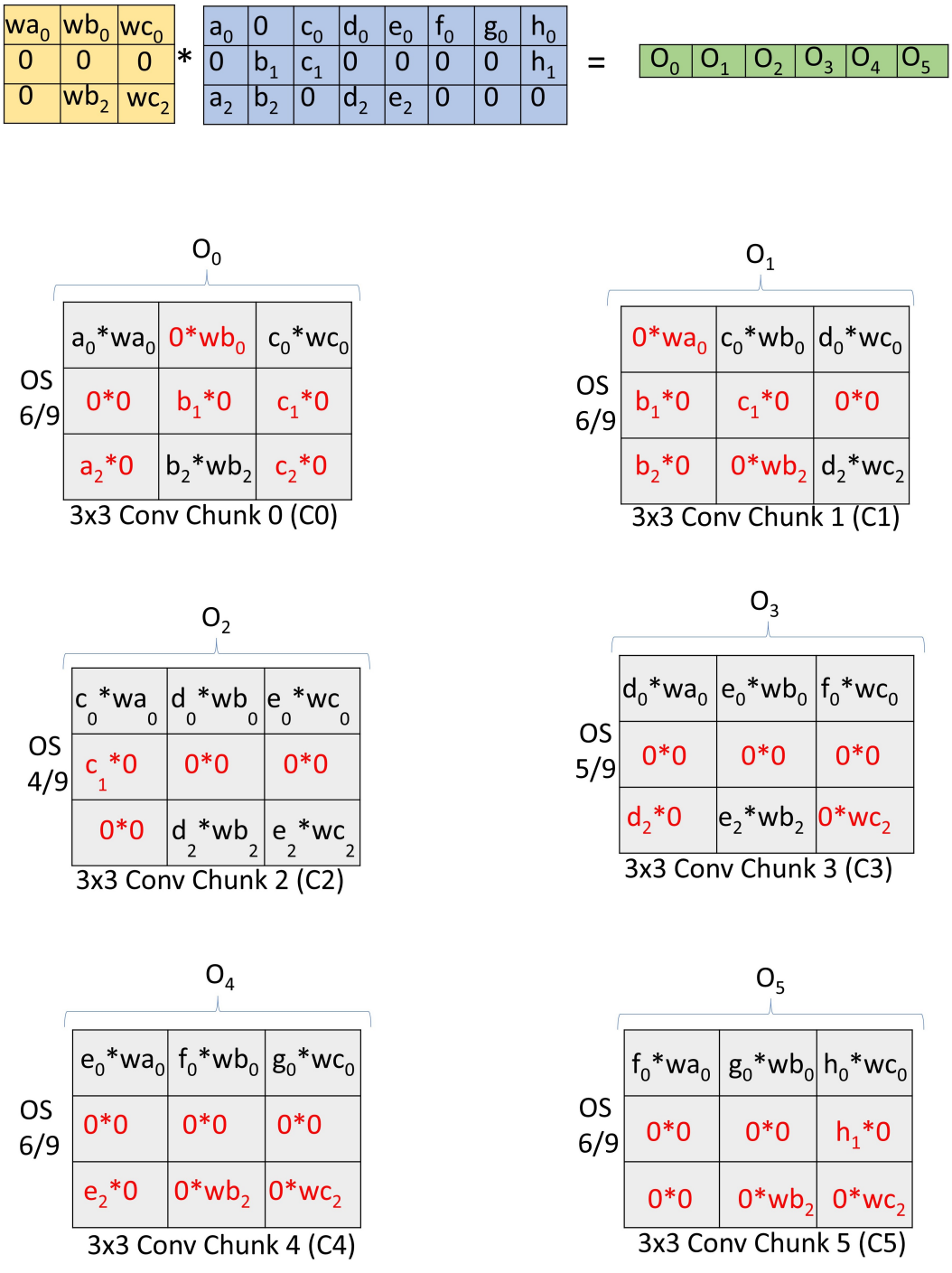


Figure 4.2: *Dense convolutions*

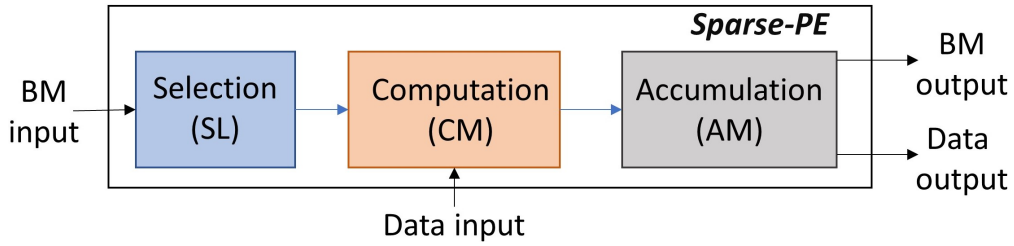


Figure 4.3: *SparsePE architecture*

6/9) which corresponds to an *effective* hardware utilization of only 33%. This represents a significant loss in computational efficiency as most of the compute cycles are wasted on ineffective computations. The SparsePE core addresses this issue and increases the hardware utilization, consequently the throughput, by minimizing the total number of ineffective computations performed by the 3×3 compute matrix. It does this by *looking-ahead* into the computations beforehand and scheduling only the valid computations to minimize the total compute cycles.

Figure 4.3 shows the high-level block diagram of the SparsePE core. The core takes the binary mask (BM) and data input and performs sparse computations to generate output data and binary mask. The SparsePE core consists of three main components: Selection (SL), Computation (CM), and Accumulation (AM). The SL block uses the sparse binary masks of input data/feature maps and weights to perform selection of valid computations ($non\text{-}zero_w \times non\text{-}zero_a$). These valid computations are represented by a set of binary matrices referred to as select matrices. The CM block uses the select matrices to map the sparse input and weight data onto a 3×3 matrix of compute threads. The mapping is performed in such a way as to maximize the utilization of individual compute threads. The AM block accumulates the CM outputs to produce valid output results. The output sparse binary mask (BM) is also generated which will be used for the next CNN layer.

4.4.1 Sparse binary mask

Many previous approaches use compressed sparse row (CSR) or column (CSC) formats to represent sparse data[43;44;53]. We, instead, use a binary representation referred to as sparse

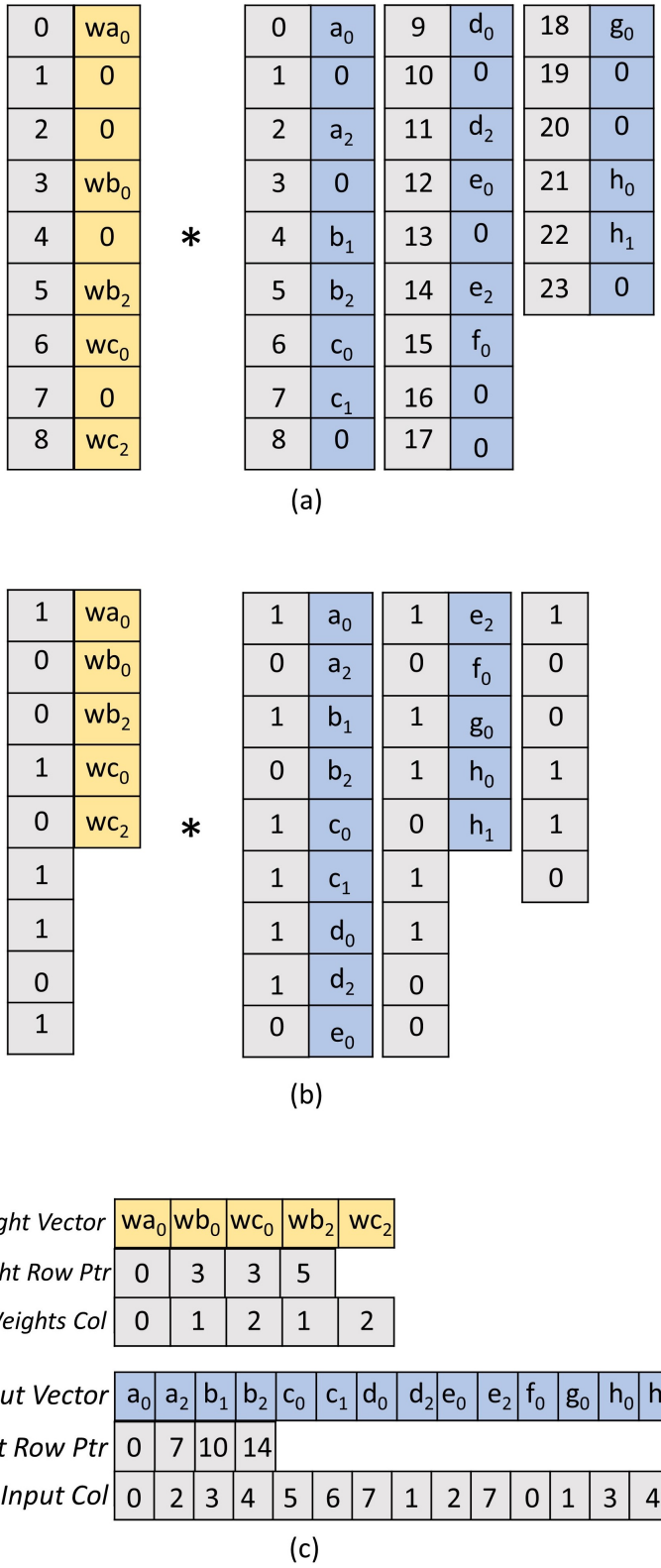


Figure 4.4: *Sparse data representations (a) Indexed-based representation (b) BM-based representation (c) CSR format*

binary mask (BM) for representing both weights and activations. The BM representation provides a simplistic, and a more convenient method for representing the *unstored* zero data and the *stored*, non-zero data. Unlike the CSR/CSC formats, this representation does not require storage of *count* and *data pointers* which significantly decreases the memory footprint of the BM representation. Figure 4.4 shows the dense, BM and CSR format representation of the 3×3 weight matrix and the 3×8 input data/feature map given in Figure 4.2. In the dense representation, both the zero and non-zero data is stored in the memory along with the indices, as shown in Figure 4.4(a). Figure 4.4(b) shows the equivalent BM representation where only the non-zero data is stored. The BM representation represents non-zero (stored) data with the binary 1, and zero (unstored) data with the binary 0. Finally, Figure 4.4(c) shows the CSR format representation, where the relative locations of the non-zero data are represented by the row and col pointers. We will further analyze the superiority of the BM representation over the CSR format in Section 4.5.

To process the input sparse data, the SparsePE core is provided the BM and the associated data in the form of chunks for processing of a particular CNN layer. Although the SparsePE core can work on any type of convolution or FC layer, for convenience and ease of understanding, we will show the working of the core using the input and the weight matrix in Figure 4.2. We also assume that the input and the weight data are 8 bits wide.

4.4.2 Selection

Lookahead masking

The convolution operation works by performing dot product between two vectors. In a *two-sided* sparse CNN model, the dot product can result in four possible multiplication outputs.

i) $zero_o = zero_w \times zero_a$

ii) $zero_o = zero_w \times non\text{-}zero_a$

iii) $zero_o = non\text{-}zero_w \times zero_a$

iv) $non\text{-}zero_o = non\text{-}zero_w \times non\text{-}zero_a$

It can be seen that the only valid multiplication is the $non\text{-}zero_o$ which results when a

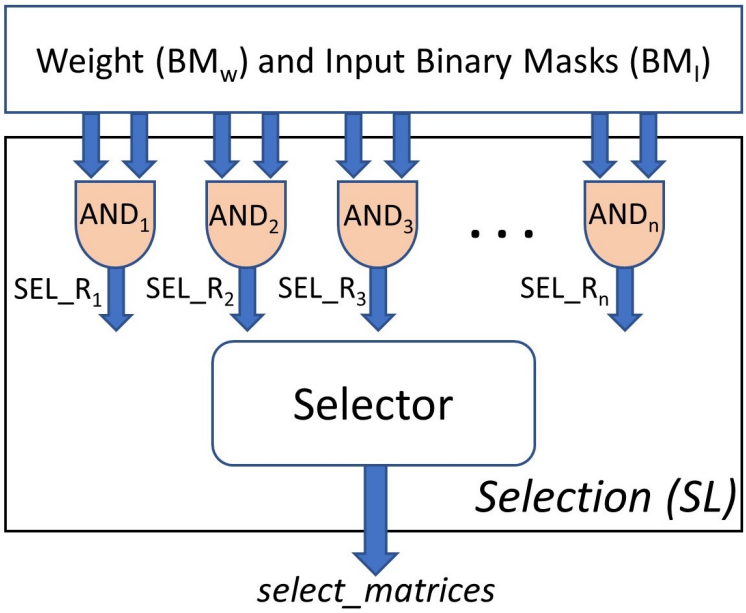


Figure 4.5: *Selection (SL) block diagram*

non-zero weight ($non\text{-}zero_w$) is multiplied by a non-zero input/activation ($non\text{-}zero_a$). The SL block has two user-defined parameters, n and k. The main purpose of the SL block is to determine k, $non\text{-}zero_o$ computations in a set of n convolution chunks. The value of k represents the total number of multiplier threads in the CM block. In this design, we have a 3×3 matrix of multipliers, making the value of $k = 9$. We define n as the *lookahead factor*, which represents the number of convolution chunks the core looks into to determine k multiplications. There are a total of 6 convolution chunks (C0-C5) in the example input, as shown in Figure 4.2. For this design, we consider the value of n to be 3. This means that during one cycle, the core looks into $k = 9$ valid multiplications in a set of $n = 3$, 3×3 convolution chunks. The SL block does this by using a series of n, $2k$-input AND gates followed by a selector, as shown in Figure 4.5. Figure 4.6 shows the process of ANDing. During the first cycle, the sparse masks of the weight and $n = 3$ chunks of the input matrix are loaded into the core. Bit by bit ANDing is performed between the sparse masks of the weight and the input to generate SEL_R1, SEL_R2, and SEL_R3, representing the ANDed output of the first, second, and third, convolution chunks, respectively. The ones in the SEL_R outputs represent the location of valid non-zero multiplications, whereas, zeros

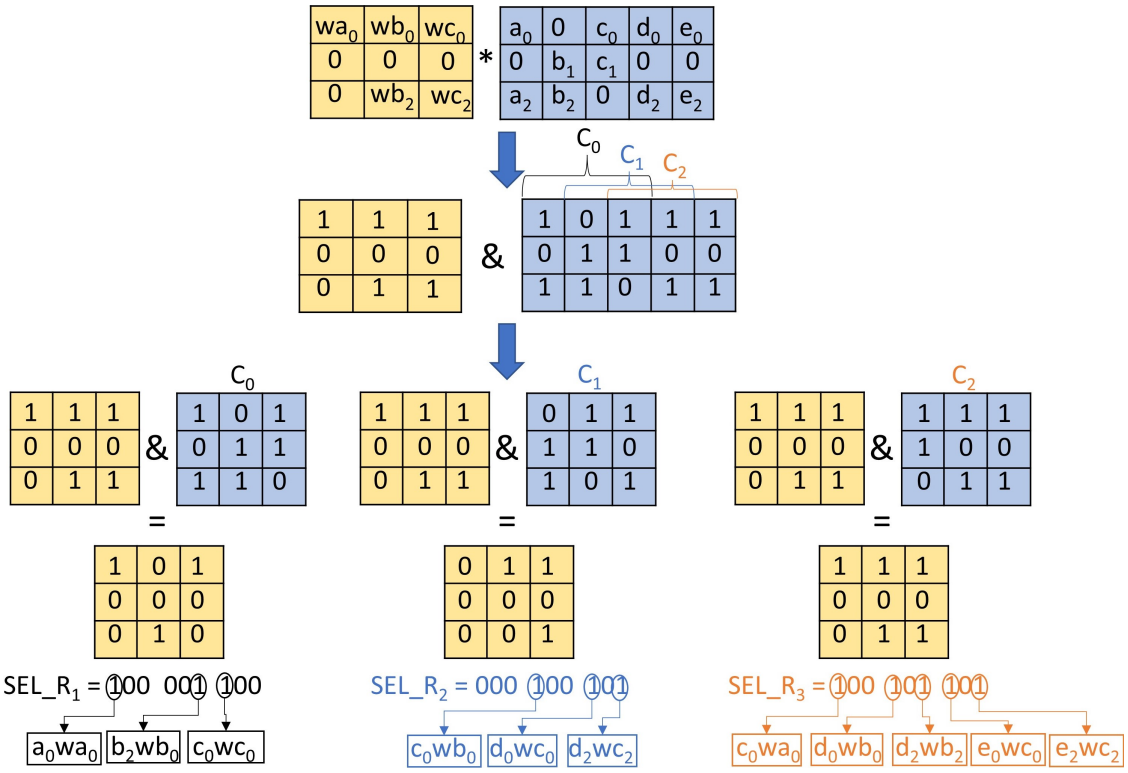


Figure 4.6: *Process of ANDing in the SL block*

represent in-effective computations involving zero operands. To process the six convolution chunks, two cycles are needed and the final SEL_R outputs are shown in Figure 4.7. The first cycle generates the ANDed outputs for the first three convolution chunks (C0, C1, C2), whereas, the second cycle generates the ANDed outputs for the last three convolution chunks (C3, C4, C5).

Top-down selector

The ANDed outputs are provided to the top-down selector (TDS) which selects the valid multiplications in a column-major format. There are a total of n selectors each processing a particular *comma-separated* column. For this design, the three columns Col_0, Col_1, Col_2 (shown in Figure 4.7) are processed in parallel by the three selectors. The purpose of the selector is to schedule the effective computations in each of the input columns (Col_0, Col_1, Col_2) on to the respective columns of the multiplier matrix. Since, at any point, the maximum number of scheduled multiplications per column cannot exceed the total number

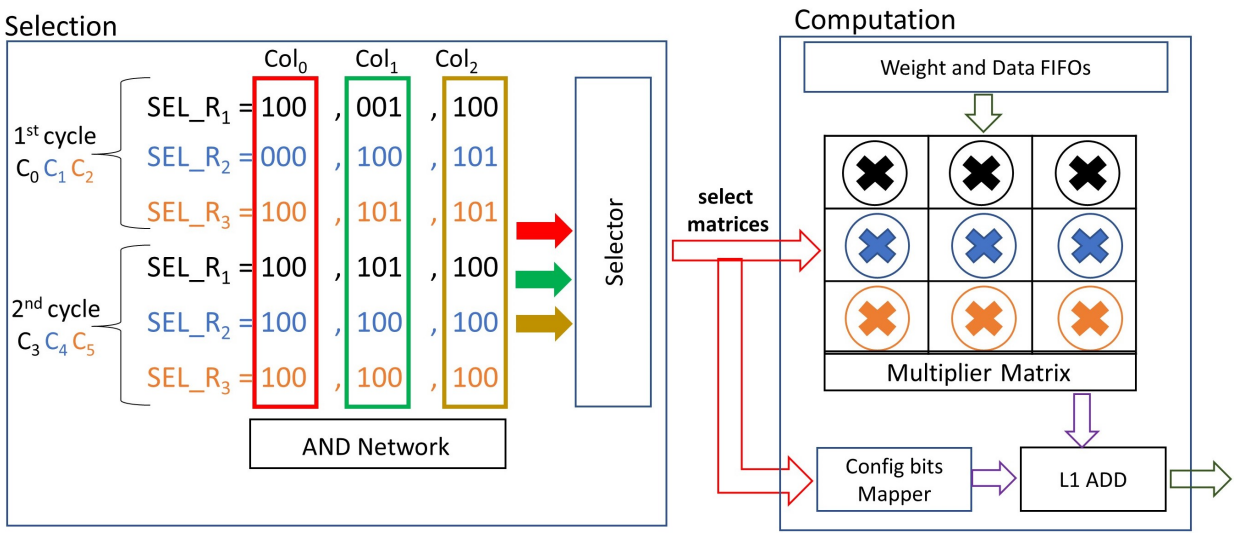


Figure 4.7: *Selection for maximum multiplier utilization*

of multipliers per column of the multiplication matrix, the selector has to make sure that during any cycle the total number of selected multiplications are equal to the multiplier threads per column of the multiplier matrix ($(k = 9)/3 = 3$) to ensure maximum hardware utilization. Since every 1 in the input columns (Col_0, Col_1, Col_2) represents a valid computation, the selection algorithm works by counting the number of ones in the individual entries and selecting the entries that maximize the utilization of the multiplier matrix. We develop two selection algorithms for the TDS namely, *in-order* selection (TDS-IO) and *out-of-order* selection (TDS-OO).

In-order selection: The TDS-IO process occurs iteratively in a linear fashion. Consider the second column (Col_1) in Figure 4.7 on which the second selector operates. The selection algorithm is shown in Figure 4.9. The selector iterates over the first $n = 3$ values (val_1, val_2, val_3) and generates two accumulation values (accum_1, accum_2). **#ones** function calculates the total number of ones in a particular val. The first iteration comprises of the first three values 001 (val_1), 100 (val_2), 101 (val_3). The first value, i.e., 001 is assigned the highest priority and the selector counts the total number of ones in this entry and stores the result, which in this case is 1, in the *init* variable. It then computes the accumulator variables (accum_1, accum_2) using the next two values (val_2, val_3). The selector then selects the values based on whether the total number of ones in accumulated values exceed the total number of

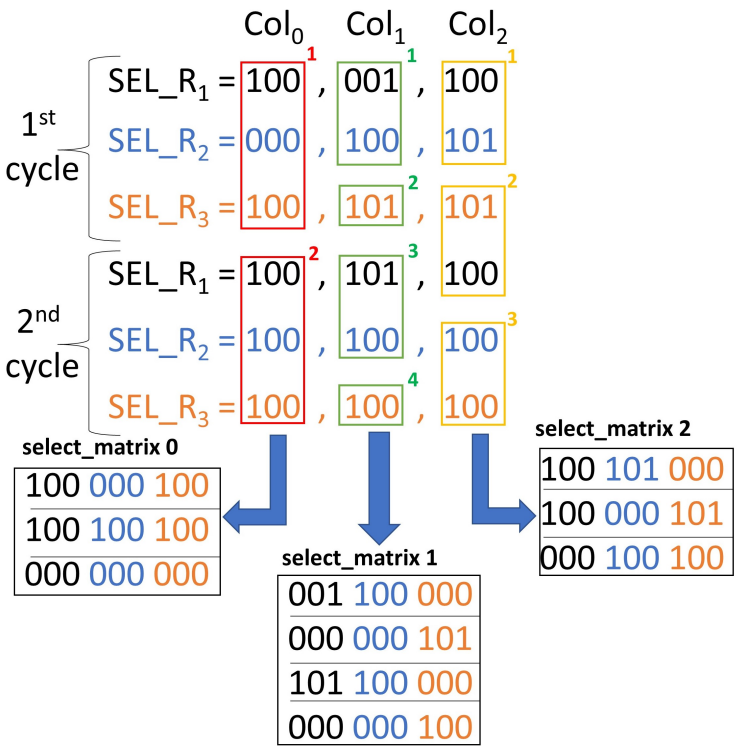


Figure 4.8: *In-Order selection*

```
init = #ones(val1)
accum1 = init + #ones(val2)
accum2 = init + #ones(val2) + #ones(val3)
if(accum1) <= k/3
        select val1, val2
        init = val3
elseif(accum2) <= k/3
        select val1, val2, val3
else
        select val1
        init = val2
```

Figure 4.9: *In-Order selection algorithm*

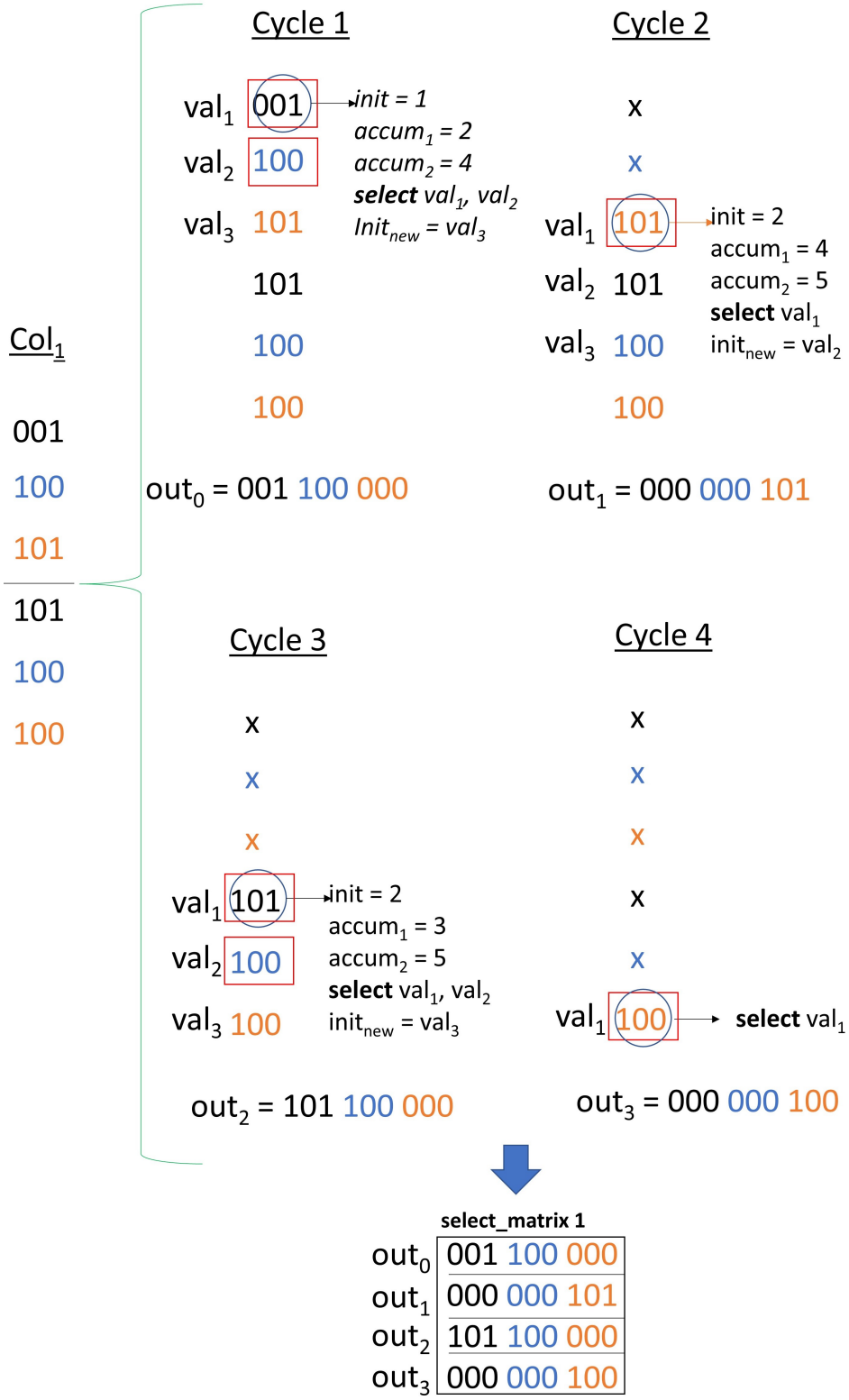


Figure 4.10: *In-Order selection process for column 1*

compute threads in one column of the thread matrix (i.e. 3). The working of the algorithm for Col_1 is shown in Figure 4.10. In the first iteration (cycle 1), the selector selects val_1 and val_2, based on the algorithm in Figure 4.9. The selector generates the output (out_0) by creating a single row of the selected values. Since a total of three values were considered, the selector replaces the last unselected value (val_3) by zeros in the out_0 output. In the next iteration (cycle 2), the selector prioritizes the unselected value (val_3) from the previous cycle and repeats the same process to generate out_1. It takes a total of four cycles for the selector to process the entire input column. At the end, a total of three select matrices are generated by the three selectors operating in parallel, as shown in Figure 4.8. Algorithm 1 (implemented on MATLAB R2020a) shows the In-Order selection process. LAM_col is a matrix containing the input columns and the select_matrix is the selection output.

Algorithm 1 In-Order Selection

Input: SEL_R Cols
Output: select_matrix

```
1:  procedure TDS_inOrder(LAM_col)
2:   Initialize (select_matrix)
3:   Accum = 0, jump = 0, row = 0;
4:      for i ← 0 to length(LAM_col) - 1 do
5:          Accum = 0;
6:          col = 0;
7:          for j ← i to numThreads + i do
8:              Accum = Accum + count_ones(LAM_col[j])
9:              if (Accum ≤ 3) then
10:                 select_matrix[row][col] = LAM_col[j];
11:                 jump = jump + 1;
12:             else
13:                 select_array[row][col] = '000';
14:             end if
15:             col++
16:         end for
17:         i = i + jump;
18:         row++;
19:     end for
20: end procedure
```

Out-of-order selection: In-Order selection is highly dependent on the order in which

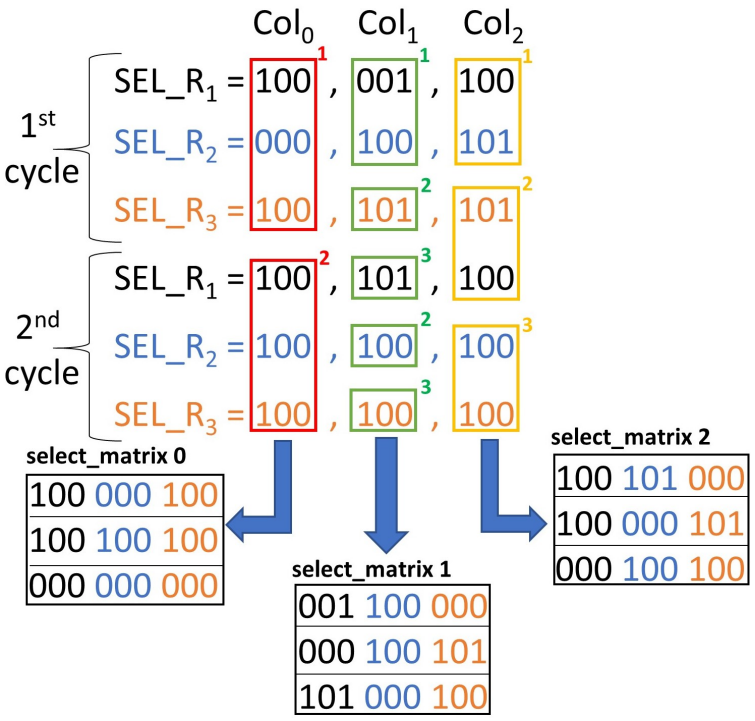


Figure 4.11: *Generation of selection matrices*

```
init = #ones(val1)
accum1 = init + #ones(val2)
accum2 = init + #ones(val3)
accum3 = init + #ones(val2)  + #ones(val3)
if (accum3) <= k/3
        select val1, val2, val3
elseif (accum1) <= k/3
        select val1, val2
        init = val3
elseif (accum2) <= k/3
        select val1, val3
        init = val2
else
        select val1
        init = val2
```

Figure 4.12: *Selection algorithm for maximum multiplier utilization*

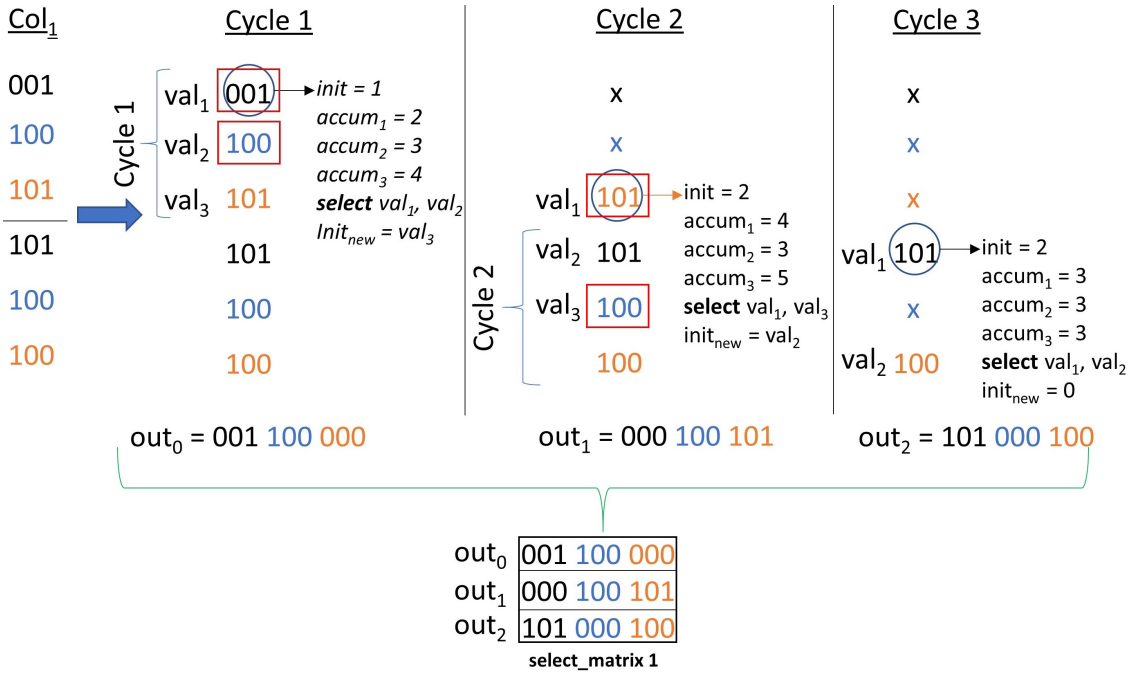


Figure 4.13: *Selection process for column 1*

the inputs appear. This can lead to under-utilization of the threads in the PE, as shown in Figure 4.8. Col_1 in Figure 4.8 takes a total of four cycles, whereas, both col_0 and col_2 take three cycles. Therefore, the selectors for col_0 and col_2 have to wait for 1 cycle for the col_1 selector to process its own column, To minimize this effect, the TDS-OO process occurs iteratively in a non-linear (or out-of-order) fashion. Figure 4.12 shows the algorithm for TDS-OO. It includes one additional accumulator (accum_3) and an extra conditional statement. In this case, the selection does not need to happen in a linear fashion, as shown in Figure 4.11. Consider the second column (Col_1) in Figure 4.7 on which the second selector operates. The selector iterates over the first $n = 3$ values (val_1, val_2, val_3) and generates three accumulation values (accum_1, accum_2, accum_3). #**ones** function calculates the total number of ones in a particular val. The first iteration comprises of the first three values 001 (val_1), 100 (val_2), 101 (val_3). The first value, i.e., 001 is assigned the highest priority and the selector counts the total number of ones in this entry and stores the result, which in this case is 1, in the *init* variable. It then computes the accumulator variables (accum_1, accum_2, accum_3) using the next two values (val_2, val_3). The selector then selects the values based on whether the

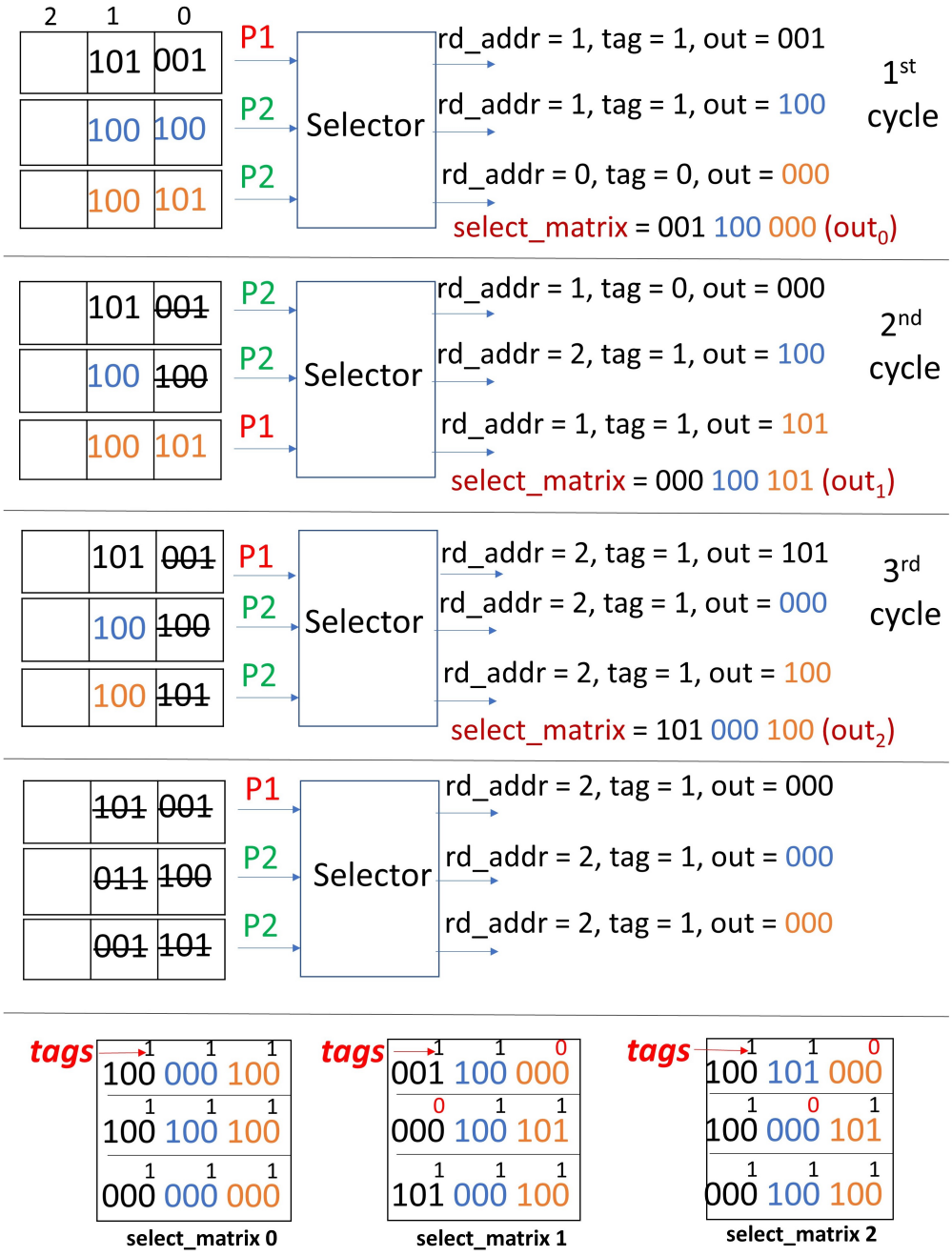


Figure 4.14: *Selector implementation*

total number of ones in accumulated values exceed the total number of compute threads in one column of the thread matrix (i.e. 3). The working of the algorithm for Col_1 is shown in Figure 4.13. In the first iteration (cycle 1), the selector selects val_1 and val_2, based on the algorithm in Figure 4.12. The selector generates the output (out_0) by creating a single row of the selected values. Since a total of three values were considered, the selector replaces the last unselected value (val_3) by zeros in the out_0 output. In the next iteration (cycle 2), the selector prioritizes the unselected value (val_3) from the previous cycle and repeats the same process to generate out_1. It takes a total of three cycles for the selector to process the entire input column, one less than the case of in-order selection. At the end, a total of three select matrices are generated by the three selectors operating in parallel, as shown in Figure 4.11. The hardware overhead of the out-of-order selector is roughly 1.03× that of the in-order selector variant, but can increase the hardware utilization by as much as 60%. SparsePE, therefore, employs the out-of-order selection variant of the TDS for higher thread utilization and efficient mapping.

Figure 4.14 shows the implementation of the selection process for Col_1 in Figure 4.7. The outputs from the AND network are stored in a local SRAM memory. On every cycle, a new value is written into the memory and values are read based on the priority of selection, with P1 being the higher priority. The selector block implements the algorithm presented in Figure 4.12. After every iteration, the read address (rd_addr) gets incremented to read the next value in the memory and the priorities are reversed. The outputs get stored into the selection matrix based on the row numbers. The tag bits are a crucial part of this process since they are used by the CM and the AM block for accumulation of data (explained in subsequent sections). Whenever a particular value is not selected during a selection run, the tag bit associated to that value is resetted to 0. The selection process ends when no more values need to be read from the memory and read addresses for all the memories are the same and the tag bits set to 1. The final select matrices, along with the associated tag bits for every value are shown in Figure 4.14.

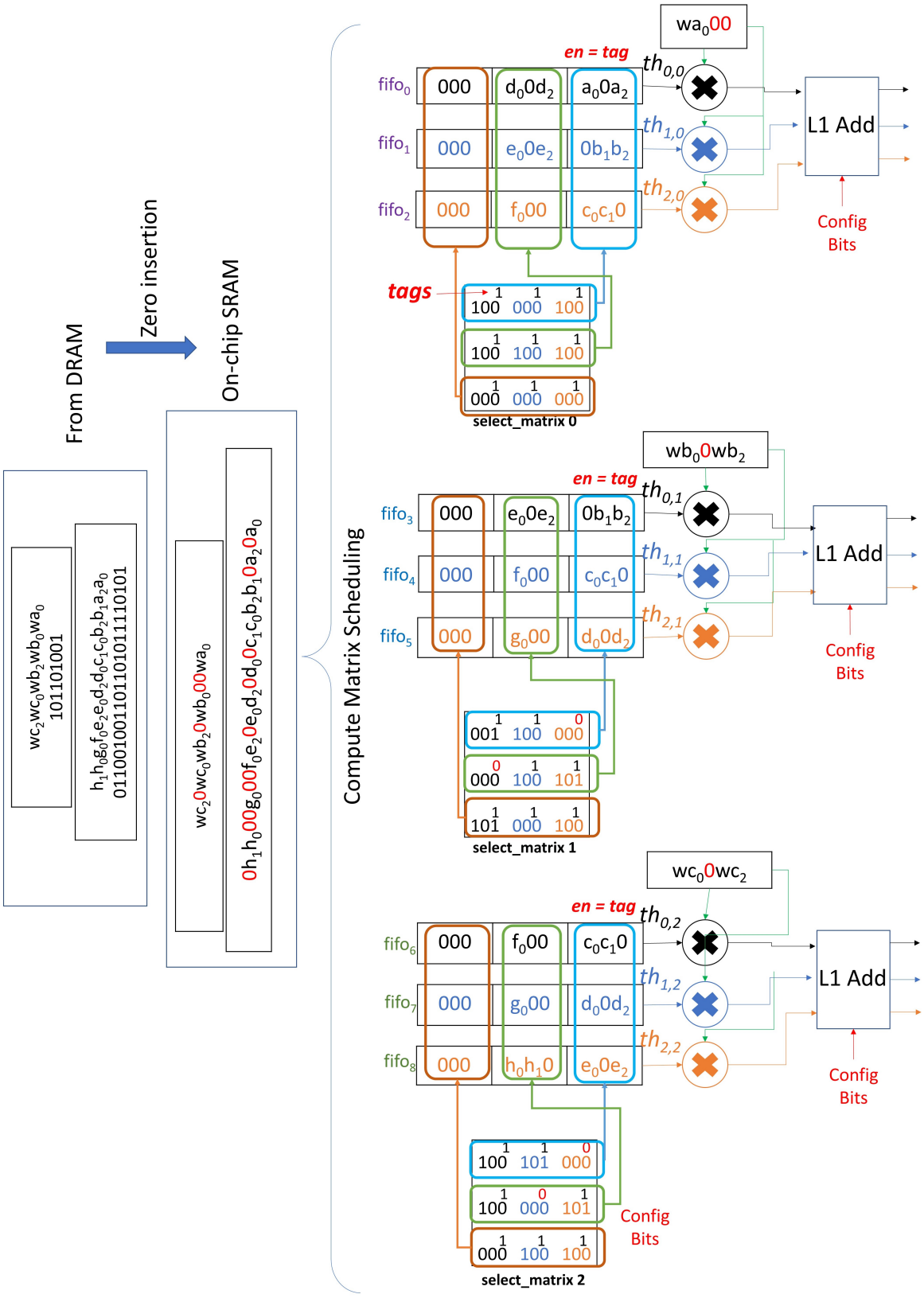


Figure 4.15: *Computation (CM) block diagram*

4.4.3 Computation

Figure 4.15 shows the process of computation in the CM block. The actual sparse data and its BM representation for a particular layer are loaded into the on-chip SRAM. Based on the BM of the input and the weight matrix, zeros are inserted at various locations for length equalization, as shown in Figure 4.15. The CM block consists of a series of fifos connected to the multiplier threads of the thread matrix. Since the size of multiplier matrix is 3×3, a total of 9 fifos ($fifo_0$-$fifo_8$) are connected to the 9 individual multiplier threads. Each column of the multiplier matrix gets a portion of the zero-inserted weight matrix. This can be seen in Figure 4.15, where the first column of multiplier threads ($th_{0,0}$, $th_{1,0}$, $th_{2,0}$) gets the weight vector $wa_0 00$, the second column ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) gets $wb_0 0 wb_2$ and the third column ($th_{0,2}$, $th_{1,2}$, $th_{2,2}$) gets $wc_0 0 wc_2$. Zero-inserted data is also scheduled to the threads from the fifos as shown in Figure 4.15. The enable pin of the fifos is connected to the tag bits of the select matrices. Whenever the tag bit for a particular entry is 1, the fifo is enabled and a new value from the fifo is passed to the multiplier which performs the computation. Figure 4.16 shows the process of computation for the second column ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) of the thread matrix. In the first cycle, the first three values in the three fifos ($fifo_3$, $fifo_4$, $fifo_5$) are controlled by the first row of the select_matrix 1 (001 100 000). The first three bits (001) along with the associated tag bit (1) controls $fifo_3$. Similarly the next 2 sets of the three bits control $fifo_4$ and $fifo_5$, respectively. Since the tag bits for the first two values (001 and 100) are set to 1, the two corresponding fifos are enabled and the data is moved to the corresponding multiplier threads. The multiplier determines the valid non-zero computation based on the values from the select matrix. For the first value (001), the multiplier extracts the last 8 bits corresponding to the weight value (wb_2) and the data (b_2) and performs the multiplication to generate the output value $b_2 wb_2$. Similarly the second multiplier thread ($th_{1,1}$) generates the value $c_0 wb_0$. Since the last three bits of the first row of the select matrix 1 have a tag 0 associated with it, the $fifo_5$ enable is off and the multiplier $th_{2,1}$ does not perform a valid multiplication. The same process repeats for the next two rows of the select matrix for cycles 2 and 3. The green rectangles on the fifo entries in Figure 4.16

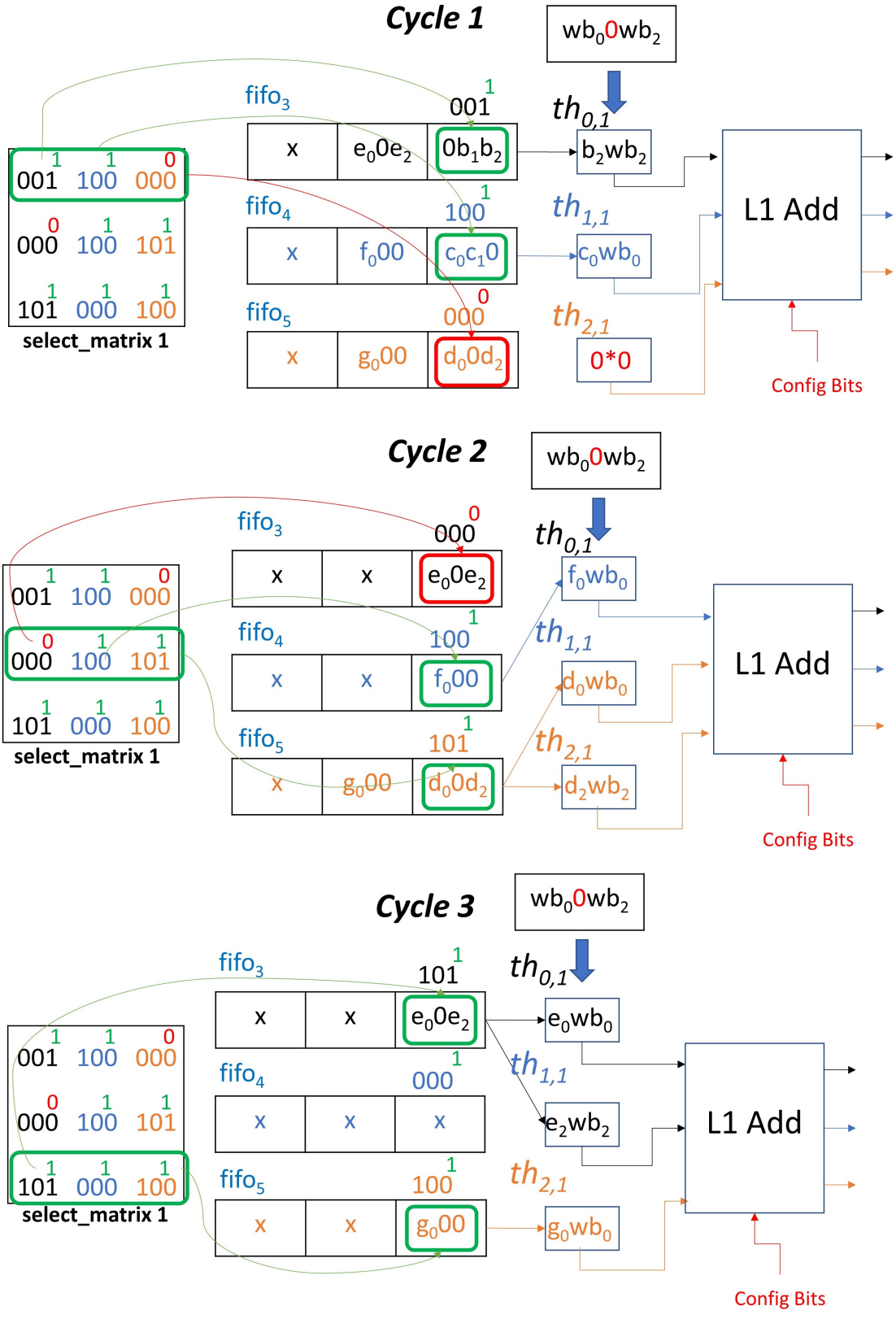


Figure 4.16: *Compute thread matrix*

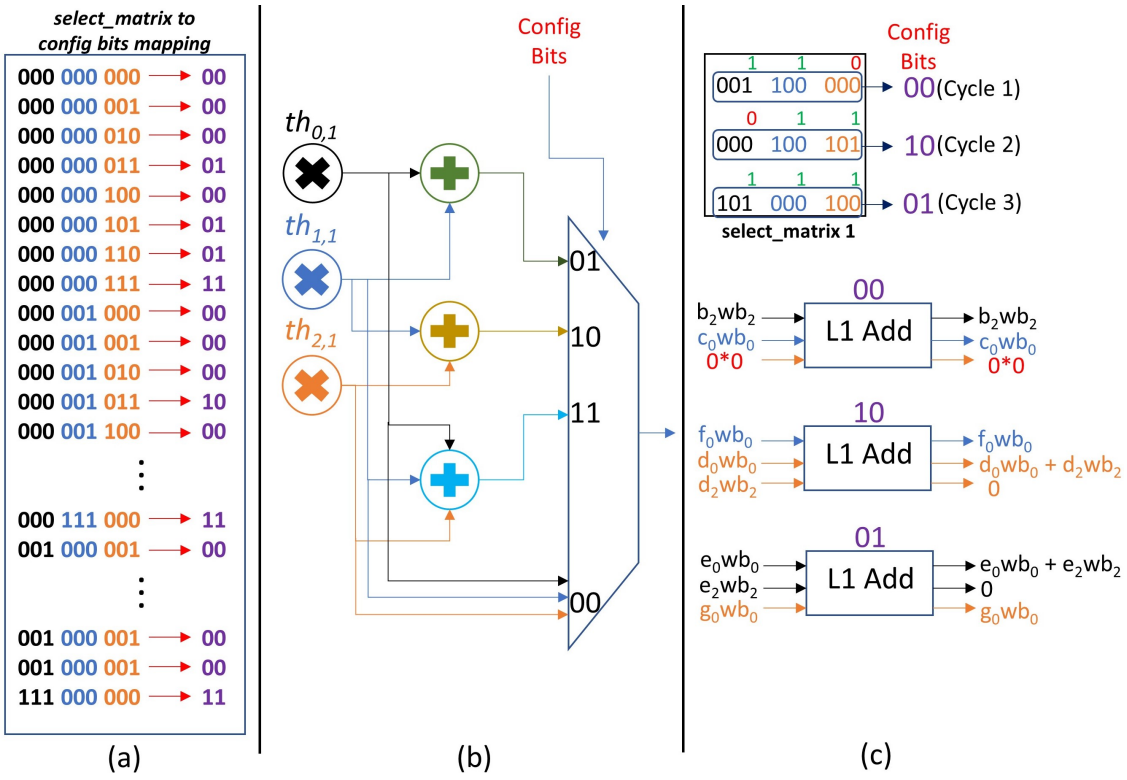


Figure 4.17: *Mapping of compute thread matrix*

represent the fifo entries that have tag bits of 1 and are thus utilized in the current cycle, whereas, the red rectangles indicate the entries that have tag bit of 0, and thus, are not utilized in the current cycle. Parallel to this multiplier column, the first ($th_{0,0}$, $th_{1,0}$, $th_{2,0}$) and the third ($th_{0,2}$, $th_{1,2}$, $th_{2,2}$) multiplier columns process the data in their own respective fifos using their own select matrices to generate the outputs.

From Figure 4.15, it can be seen that the output of the multiplier threads is fed into the level 1 (L1) Adder circuit. The purpose of this adder is to accumulate the outputs which belong to the same convolution chunk and to bypass the addition if the outputs belong to different convolution chunks. To further illustrate this, consider Cycle 2 in Figure 4.16. The outputs from $th_{1,1}$ (d_0wb_0) and $th_{2,1}$ (d_2wb_2) belong to the same convolution chunk (C2 in Figure 4.2) and therefore need to accumulated, whereas, the output from $th_{0,1}$ (f_0wb_0) belongs to a completely different convolution chunk (C4 in Figure 4.2) and has no relevance to the outputs from $th_{1,1}$ and $th_{2,1}$. This happens because the selector in the previous step is selecting the valid non-zero computations in a non-linear fashion to maximize the multiplier

utilization and does not necessarily care about maintaining a proper flow of multiplications. To circumvent this issue and to determine which multiplier outputs need to be accumulated, the L1 adder uses a bit mapper that encodes the select matrix rows to appropriate values, as shown in Figure 4.17(a). The two bits at the output encode the following information:

00: The thread outputs within the multiplier column are not added and passed as is.

01: The outputs of $th_{0,x}$ and $th_{1,x}$ are added, whereas, the $th_{2,x}$ output is passed as is.

10: The outputs of $th_{1,x}$ and $th_{2,x}$ are added, whereas, the $th_{0,x}$ output is passed as is.

11: The outputs of all the threads within a multiplier column are added.

The mapping from the 9 bits of the select matrix to 2-bit output is straightforward. Since every color-coded set of three bits represent a multiplication within a particular convolution chunk, whenever there are multiple ones within the same set (black, blue, or orange), the thread outputs associated with those values need to be added together. This can be seen from entry 4 of Figure 4.17(a) (000 000 011 -¿ 01). The orange set has value 011, so according to the above mapping, the first two thread outputs are added together because they belong to the same convolution chunk. Similarly, the last entry (111 000 000) is mapped to 11, meaning that all three thread outputs within a multiplier column belong to the same convolution chunk, and therefore, need to be accumulated. Using this mapping, the L1 adder can keep track of which thread outputs need to accumulate and which do not. Figure 4.17(b) shows the internal structure of the L1 Adder. It comprises of three adders that add different combinations of the thread outputs. It also comprises of an output multiplexer whose select line is connected to the mapper output (2 bits). The L1 Adder operation for the second column of the thread matrix ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) is shown in Figure 4.17(c). The mapper maps the individual rows of the select matrix to 2 configuration bits. Using the configuration bits, the L1 Adder generates the outputs. In the first cycle, all three outputs (b_2wb_2, c_0wb_0, 0) belong to different convolution chunks and therefore the config bits are 00, which passes the inputs to outputs without accumulation. In the second cycle, the select matrix row maps to 10, which adds the last two thread outputs (d_0wb_0, d_2wb_2) together because they belong to the same convolution chunk and forwards the first thread output (f_0wb_0) as is because it belongs to a different convolution chunk. In the last cycle, the first two thread outputs are

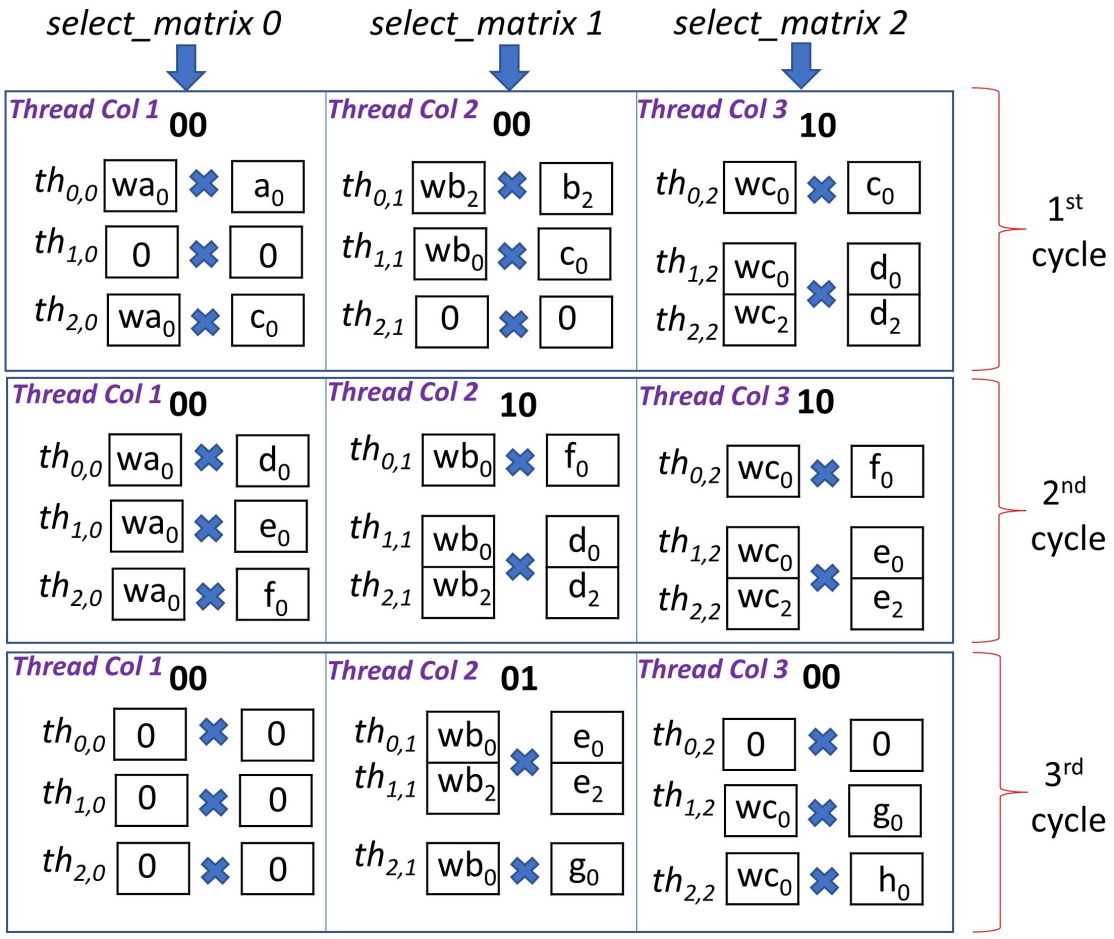


Figure 4.18: *Thread scheduling for the example in Figure 4.4*

added and the last is passed as is. It should be noted that there are a total of three identical mappers (Figure 4.18(a)), each belonging to a particular L1 adder in Figure 4.15.

Even though the row length of the select matrix is 9 which produces $2^9 = 512$ combinations, the mapper only needs to store those combinations for which the total number of ones in the select matrix rows are less than or equal to the multiplier threads per column of the multiplier matrix (3 in this case). Therefore, the total combinations that need to be stored are only 130. Generally, the total combinations that need to be stored can be found using the following equation:

$$Combinations = \binom{k}{0} + \binom{k}{1} + \binom{k}{2} + ... + \binom{k}{n} \tag{4.1}$$

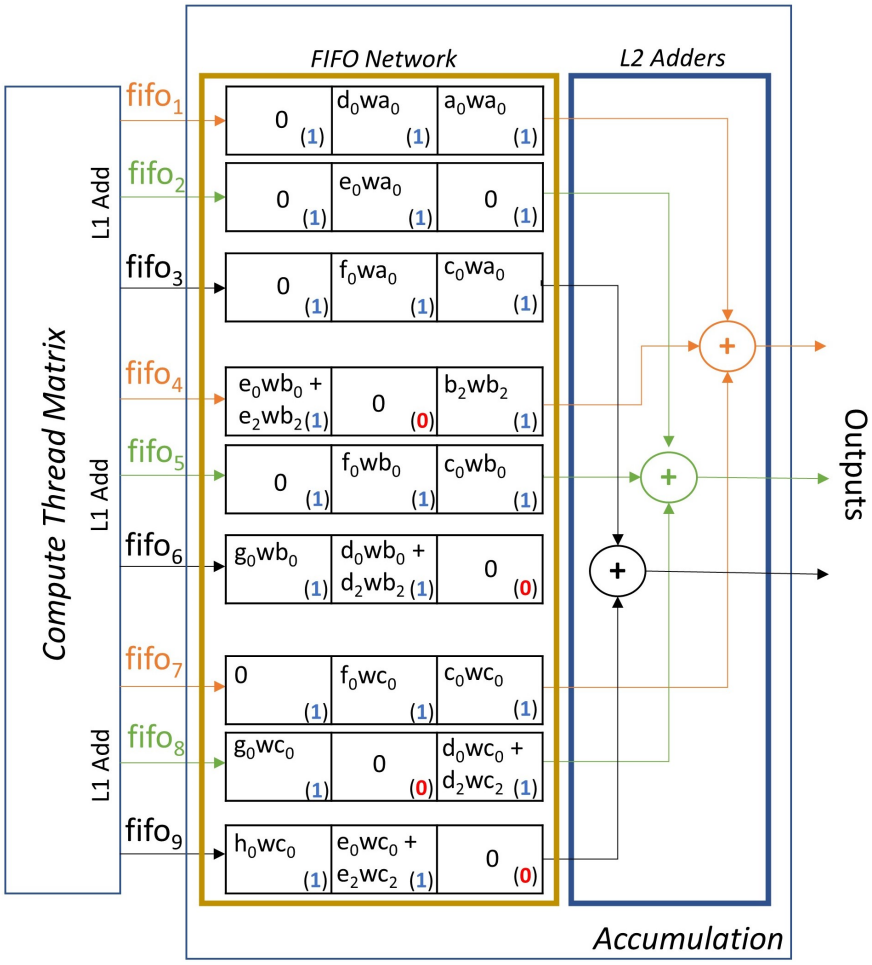


Figure 4.19: *Accumulation for the example in Figure 4.4*

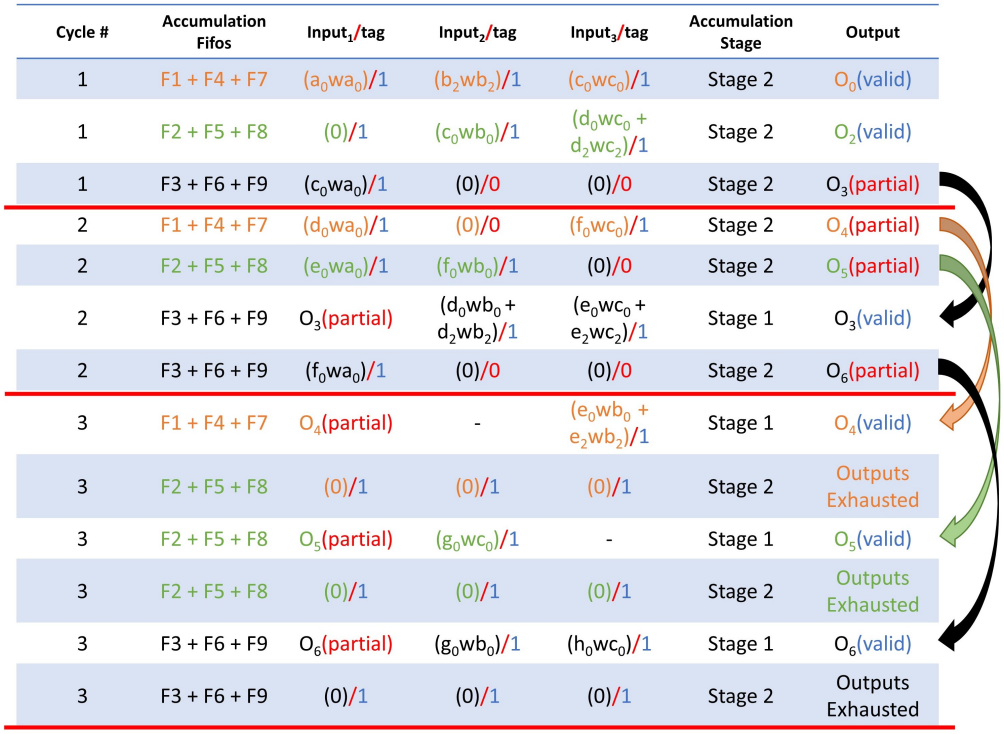
Where, k represents the length of a select matrix row (9 in this case) and n represents the total multiplier threads per column of the thread matrix (3 in this case). Therefore, the total memory required for storing the three mappers is only 780 bits ($130 \times 2 \times 3$).

Figure 4.18 shows the cycle by cycle scheduling of the data onto the multiplier thread matrix for the example in Figure 4.2. All the threads are connected to their respective fifos and receive the data. The select matrices are used to schedule the data and for generating the 2-bit L1 Adder configuration bits (also shown in Figure 4.18). To compare the performance of a dense design that does not exploit sparsity, we can see from Figure 4.2 that a dense approach would take a total of 6 cycles to generate the output using the same number of multipliers (9 arranged in a 3×3 matrix). The approach presented here utilizes just three cycles to process the entire output by exploiting the sparsity and maximizing the scheduling of valid,

non-zero multiplications during the majority of the processing cycles. This represents a 50% increase in hardware utilization which consequently represents a 50% increase in throughput. It should also be noted that the hardware utilization increases further when the input sparse matrix is larger. It can be seen from Figure 4.18 that the last three multiplications in the Thread Col 1 are all zeros because the input matrix has been exhausted. For a larger input matrix, the average hardware utilization and throughput would be higher. We will further explore the impact of the size of the input on the hardware utilization and throughput in Section 5.3.

4.4.4 Accumulation

The accumulation (AM) block buffers and accumulates the partial dot product outputs from the CM block to produce the final output results. The AM block consists of a series of fifos ($fifo_1$-$fifo_9$) connected at the output of the 3 L1 Adder circuits, as shown in Figure 4.19. The output of the fifos is provided to a level 2 (L2) adder which accumulates the dot products for a particular convolution chunk to generate the final output. The CM block outputs the partial dot products as well as the tag values associated with every product. The tag bits are shown in parenthesis in Figure 4.19. The process of accumulation occurs in two stages by the same color-coded fifos ($fifo_1$ + $fifo_4$ + $fifo_7$), ($fifo_2$ + $fifo_5$ + $fifo_8$), and ($fifo_3$ + $fifo_6$ + $fifo_9$). The outputs are either valid or partial, based on the tag values associated with each accumulation stage. If the tag values for all the inputs are equal to **1**, the output is considered *valid*, otherwise, it is considered *partial*. Figure 4.20 shows the accumulation stage map for the example in Figure 4.2. On every cycle, there are two accumulation stages. In the first stage, the previously generated partial outputs are added to the new entries to make the output *valid*. To do this, the AM block checks the tag bits in the partial output and adds the missing tag 1 input to make the partial output complete/valid. In the second stage, the AM replaces the already used tag 1 values with zeros and generates new partial outputs by summing the unused tag 1 values. This process can be seen in Figure 4.20. The O_3 partial value is generated by summing the fifos F3, F6, F9 in cycle 1. The output is



Cycle #	Accumulation Fifos	Input$_1$/tag	Input$_2$/tag	Input$_3$/tag	Accumulation Stage	Output
1	F1 + F4 + F7	($a_0$wa$_0$)/1	($b_2$wb$_2$)/1	($c_0$wc$_0$)/1	Stage 2	O_0(valid)
1	F2 + F5 + F8	(0)/1	($c_0$wb$_0$)/1	($d_0$wc$_0$ + $d_2$wc$_2$)/1	Stage 2	O_2(valid)
1	F3 + F6 + F9	($c_0$wa$_0$)/1	(0)/0	(0)/0	Stage 2	O_3(partial)
2	F1 + F4 + F7	($d_0$wa$_0$)/1	(0)/0	($f_0$wc$_0$)/1	Stage 2	O_4(partial)
2	F2 + F5 + F8	($e_0$wa$_0$)/1	($f_0$wb$_0$)/1	(0)/0	Stage 2	O_5(partial)
2	F3 + F6 + F9	O_3(partial)	($d_0$wb$_0$ + $d_2$wb$_2$)/1	($e_0$wc$_0$ + $e_2$wc$_2$)/1	Stage 1	O_3(valid)
2	F3 + F6 + F9	($f_0$wa$_0$)/1	(0)/0	(0)/0	Stage 2	O_6(partial)
3	F1 + F4 + F7	O_4(partial)	-	($e_0$wb$_0$ + $e_2$wb$_2$)/1	Stage 1	O_4(valid)
3	F2 + F5 + F8	(0)/1	(0)/1	(0)/1	Stage 2	Outputs Exhausted
3	F2 + F5 + F8	O_5(partial)	($g_0$wc$_0$)/1	-	Stage 1	O_5(valid)
3	F2 + F5 + F8	(0)/1	(0)/1	(0)/1	Stage 2	Outputs Exhausted
3	F3 + F6 + F9	O_6(partial)	($g_0$wb$_0$)/1	($h_0$wc$_0$)/1	Stage 1	O_6(valid)
3	F3 + F6 + F9	(0)/1	(0)/1	(0)/1	Stage 2	Outputs Exhausted

Figure 4.20: *Accumulation stage map*

partial because the Input$_2$ and Input$_3$ tag values are 0. O_3 is made valid by summing the O_3(partial) with the now available Input$_2$ and Input$_3$ tag 1 values in the stage 1 during the 2nd cycle. In the second stage, during the same cycle, the already used tag 1 inputs are replaced by 0 and the new partial product (O_6) is generated. This process is repeated in subsequent cycles until all the inputs are utilized and all the outputs are *valid*.

Figure 4.21 shows the cycle-by-cycle processing latency of the various blocks of the SparsePE core for the example in Figure 4.2. The process of ANDing takes a total of two cycles for the generation of a pair of 3-tuples (SEL_R$_1$, SEL_R$_2$, SEL_R$_3$). After the generation of the first tuple of the SEL_R$_1$, SEL_R$_2$, and SEL_R$_3$ values, the selection block starts processing and takes a total of 3 cycles to generate the select_matrices and the associated tag values (Figure 4.14). The computation block gets triggered after the first row of the select_matrices is generated and takes a total of three cycles to process the three rows of the select_matrices (Figure 4.18). Finally, the accumulation block takes a total of three cycles to process the partial product outputs from the computation block (Figure 4.20) to generate

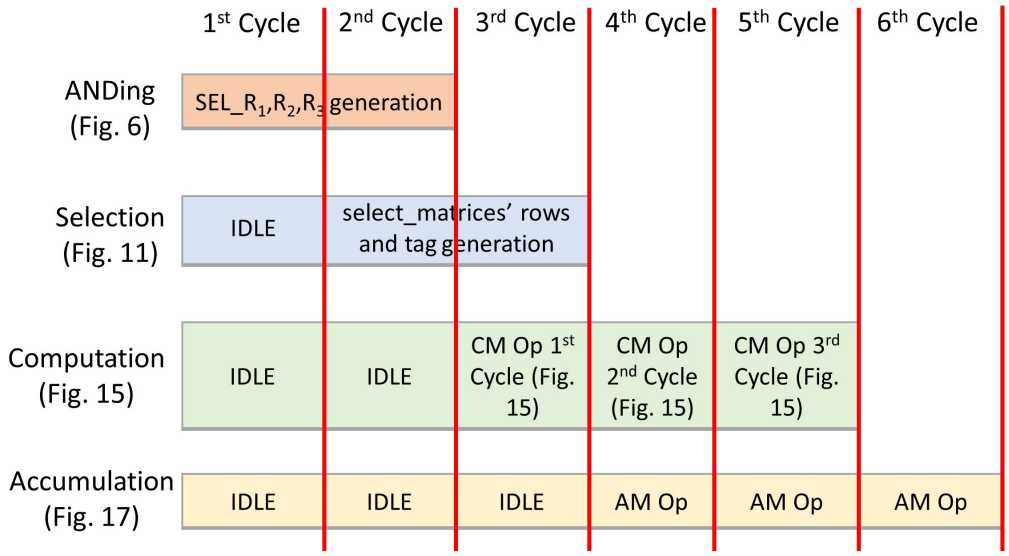


Figure 4.21: *Initial latency of the SparsePE core operations*

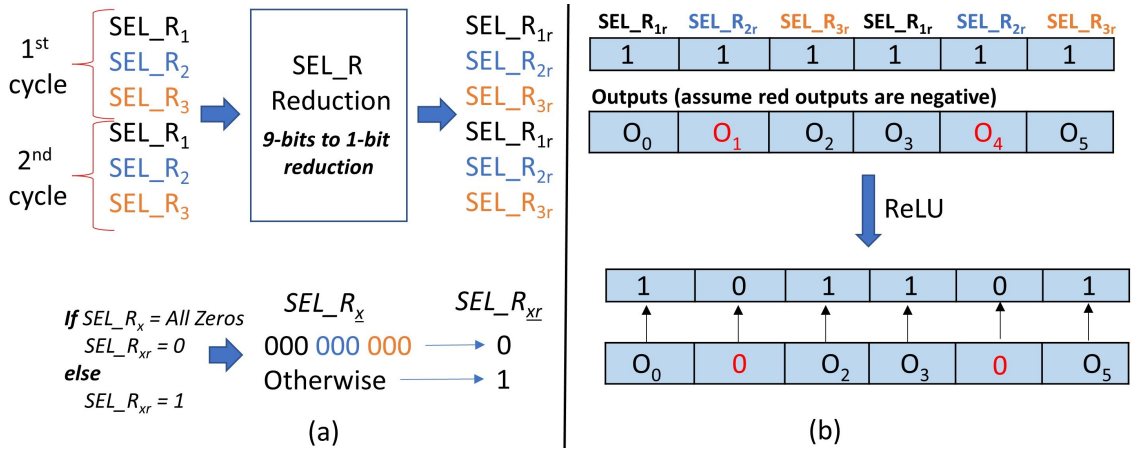


Figure 4.22: *Output sparse mask generation*

the final outputs. Therefore, the initial processing latency to generate the first output is six cycles. This initial latency, however, is amortized over processing over a larger input.

4.4.5 Output encoding

Figure 4.22 shows the process of output sparse binary mask encoding. Unlike the weight masks, the output binary mask needs to be generated on the fly because of its dynamic nature. From Figure 4.6, we can see that the SEL_R outputs show the presence of the non-

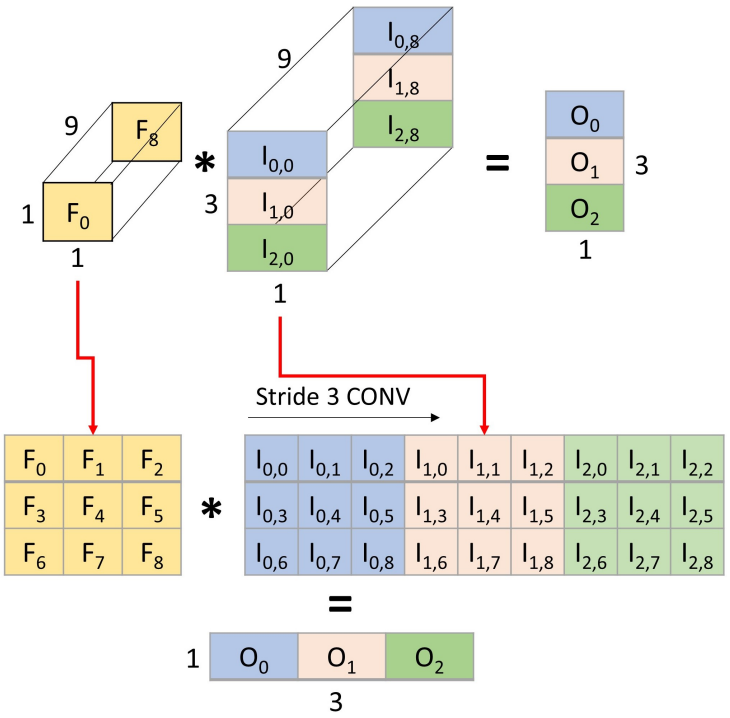


Figure 4.23: 1×1 *convolution layer transformation into an equivalent* 3×3*, stride 3 convolution layer*

zero partial products for a particular convolution chunk. The presence of even a single *one* in the SEL_R outputs show that the output is non-zero. To determine the output binary mask, we can use the same metadata. The first step involves reduction of the individual SEL_R outputs to a single bit (SEL_R_{xr}), based on an *all-zero* check, as shown in Figure 4.22(a). This generates the sparse binary mask for the outputs before ReLU. Note that the SEL_R values are taken from the test example (Figure 4.2). Figure 4.22(b) shows the second step after ReLU, where the negative outputs, and their corresponding sparse mask locations, are converted into zeros. This final sparse mask is stored as-is, whereas, the output is shifted first to omit zero data entries, and then stored.

This concludes the processing that goes on in a single SparsePE core when convolution is performed using a 3×3 filter. For a filter of size 5×5, 7×7, or higher, the kernel factorization[106] is employed to convert larger filters into a set of smaller 3×3 filters. This factorization makes it possible to keep the design of PEs relatively simple and not incorporate complex flow control to support larger filters. SparsePE, however, also supports 5×5 filters

by using a multiplier matrix of 5×5, instead of the current 3×3.

1×1 convolution, used in many modern CNN models, is performed the same way as the 3×3 convolution. Figure 5.2 shows an example of 1×1 convolution. Here, a $1 \times 1 \times 9$ filter is convolved with a $3 \times 1 \times 9$ input matrix to produce a 3×1 output. Figure 5.2 also shows the transformation employed by the architecture to transform the 1×1 convolution operation into an equivalent 3×3 stride 3 convolution. Here, individual channels of filter and input matrix are transformed into a 3×3 matrix and scheduled to the core for processing. The core then performs the selection, computation, and accumulation in the same manner as explained previously.

SparsePE architecture also allows the processing of FC layers. Figure 5.3 shows an example of FC layer processing. Here, a 9×1 input array and a 9×3 filter matrix produces a 1×3 output. The input and the filter matrices are again transformed into an equivalent 3×3 stride 3 convolution. SparsePE then processes the transformed input to generate the final outputs.

This concludes the processing that goes on in a single SparsePE core. In the next section, we will show the performance improvement offered by an accelerator that uses a system of SparsePE cores to accelerate the CNN inference process.

4.5 Implementation and results

An individual SparsePE core works by computing dot products between a subsection of the sparse input data and the sparse weight data to produce a subsection of the output data. To process multiple subsections, we envision a 7×4 matrix architecture, referred to as the *Phantom* architecture, consisting of a total of $7 \times 4 = 28$ SparsePE cores. The Phantom architecture will be explored in Chapter 5. We use the Xilinx Z-7100 SoC to implement the SparsePE core design. The SoC is divided into two parts, the programmable logic (PL), containing the FPGA fabric, and the processing system (PS), containing ARM cores. The two are connected using an AXI on-chip communication subsystem. We implement the SparsePE core on the PL and use the PS to transfer data to/from a desktop computer to

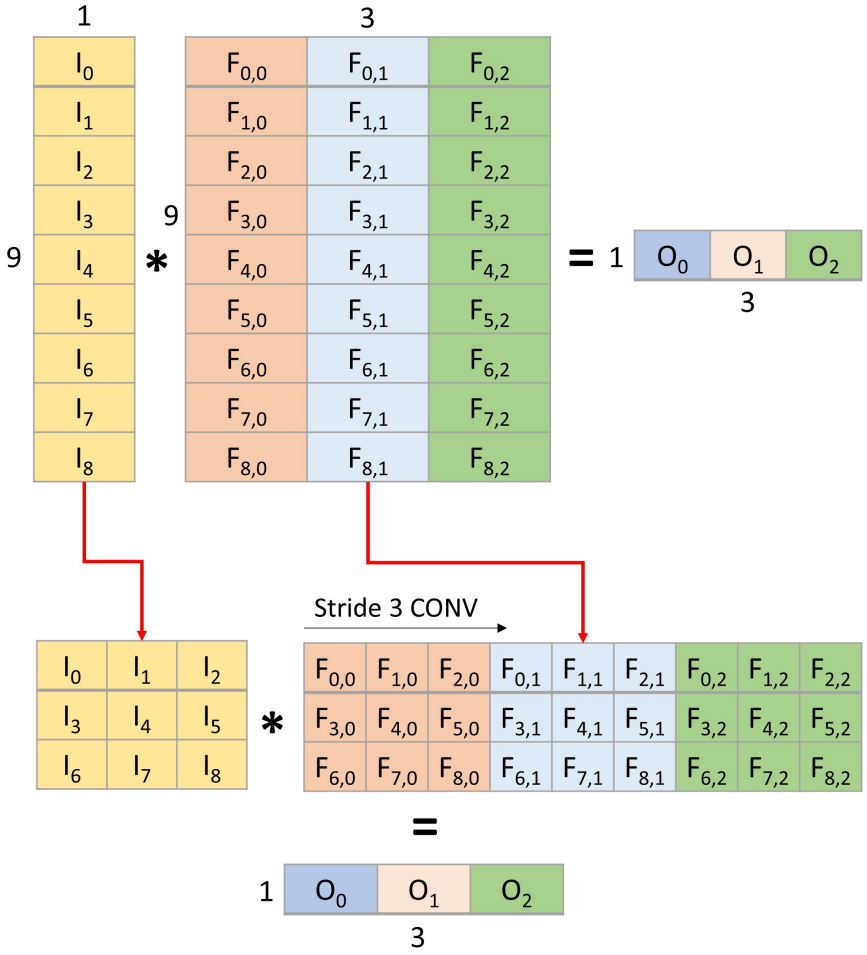


Figure 4.24: *FC layer transformation into an equivalent* 3×3*, stride 3 convolution layer*

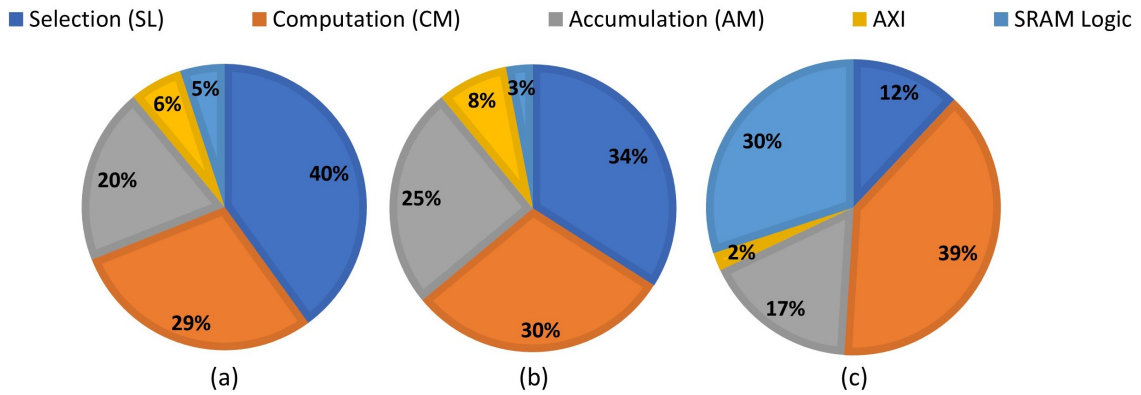


Figure 4.25: *Breakdown of resource utilization for (a) LUTs (b) FFs (c) SRAM memory*

Table 4.1: *Resource utilization for SparsePE core*

Property	Available	Used	Utilization (%)
LUTs	277k	3.4k	1.23%
FFs	554k	6k	1.1%
On-chip SRAM	26.5Mb	2.1kB	0.01%

PL. The test design is implemented for the SparsePE having lookahead factor (l_f) = 27. Table 4.1 shows the resource utilization results and Figure 4.25 shows the breakdown of the utilization among various sub-blocks of the SparsePE core. The LUT and the FF cost are dominated by the selection (SL) block with SL taking almost 40% and 34% of the overall LUTs and FFs used, respectively. The SRAM utilization is dominated by the computation (CM) block because of the bit mapping and the buffering fifos (Figure 4.16). The design has a power consumption of 2.48W with the PS dominating the consumption (55%). Eyeriss v2[52], implemented on 65nm ASIC, does not provide individual PE cost, which makes it difficult to compare against SparsePE. It, however, uses a total of 2695k gates which represents a 108% increase in area cost when compared to the original Eyeriss[45].

Energy comparison among different cores is somewhat challenging as it requires working RTL implementations. Since the energy consumption is dominated by the total number of DRAM accesses, therefore, by estimating the DRAM accesses, energy difference among the cores can be approximated. Many of the recent works rely on the CSC/CSR formats for the storage of non-zero data. We, therefore, compare the accessed memory for the CSC format against the sparse binary mask format. Figure 4.26 shows the intermediate

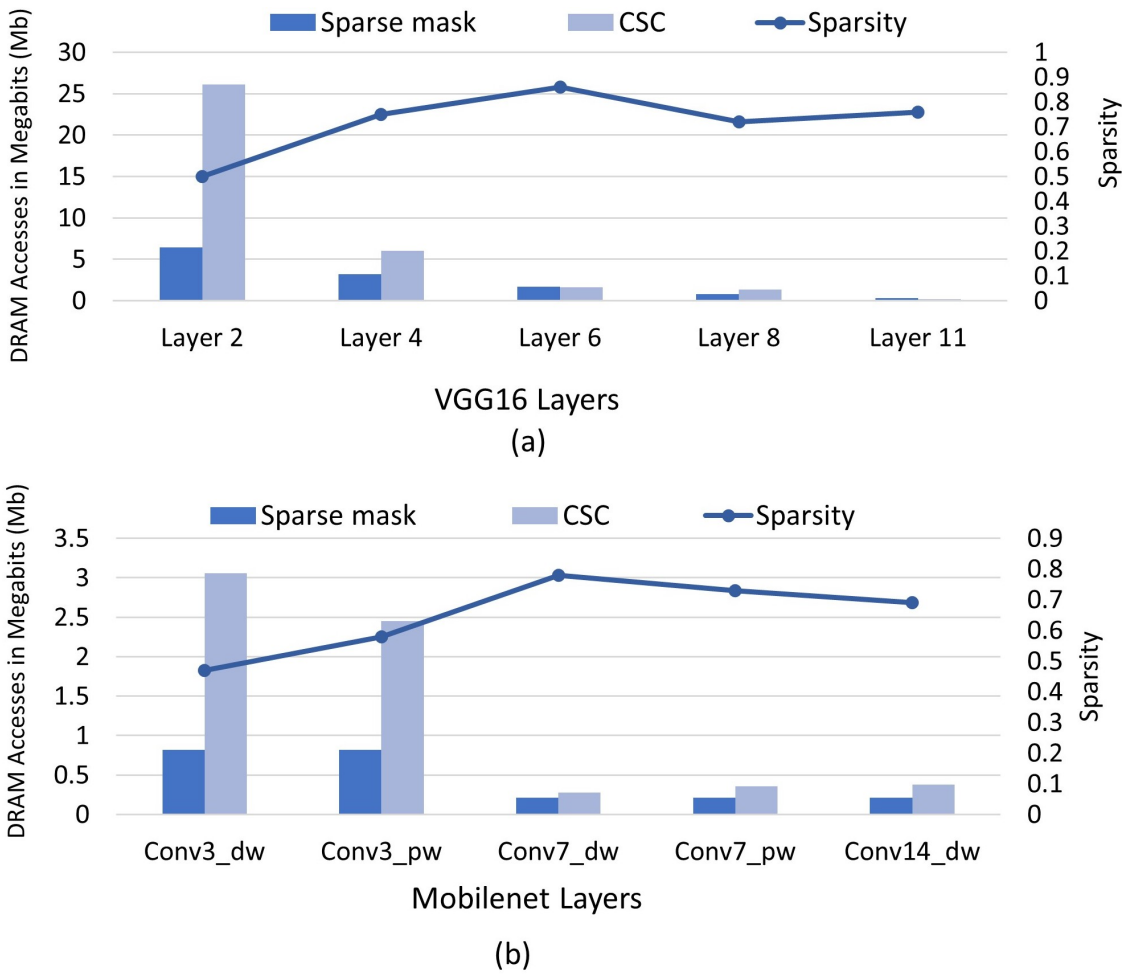


Figure 4.26: *Sparse binary mask vs. CSC DRAM access for (a) Sparse VGG16 (b) Sparse MobileNet*

activations' memory access comparison for selected sparse VGG 16 and MobileNet layers [2]. The activation sparsity for different layers is also shown. In the initial layers with low activation sparsity, the CSC format has approximately 4× and 3.7× higher DRAM memory accesses than the sparse mask for sparse VGG16 and Mobilenet, respectively. In the deeper layers with moderate to high sparsity, the memory requirement for the CSC format is around 1.7× that of the sparse mask.

This shows that the sparse binary mask format not only needs less encoding/decoding logic but is also efficient when it comes to memory requirements when compared against the CSC format. This translates directly to higher energy, area, and compute savings for our SparsePE core which employs the sparse binary mask.

4.6 Conclusion

In this chapter, we gave a detailed architectural description of the SparsePE core. The SparsePE is a multi-threaded, general-purpose, dot product processing core for sparse CNNs. The designed core is capable of exploiting *two-sided* sparsity, that is, sparsity in both the weights and activations, to maximize the throughput and hardware utilization. Unlike contemporary approaches that use the CSC format and the associated complex PE design, the SparsePE core uses the sparse binary mask format and has relatively low complexity. The core uses novel, low-cost circuits, including selection, computation, and accumulation, which, when used in conjunction, allows the core to skip a huge number of computations involving *zero* data and only favor computations involving *non-zero* data to maximize the throughput. The core has a relatively low area overhead when compared against other cores with similar functionality. In the next chapter, we introduce the *Phantom* accelerator which uses multiple SparsePE cores. We also introduce a new dataflow that is capable of performing a wide variety of operations used in high accuracy CNNs.

[2]The memory requirement for the non-zero data is not shown since it is the same for both the CSC format and the sparse binary mask. The accessed memory comparison is made for the sparse binary mask and the location vectors (column, index) of the CSC format.

Chapter 5

Phantom accelerator

In this chapter, we introduce an accelerator architecture that uses a number of SparsePE cores, arranged in a spatial grid, to process a wide variety of sparse CNN models. We refer to this accelerator architecture as the Phantom accelerator. Section 5.1 introduces the Phantom accelerator and summarizes the main contributions. Section 5.2 gives the architectural details of the accelerator. Section 5.3 provides the simulation results of the accelerator and performs comparison against some recent approaches. Section 5.4 summarizes and concludes the chapter.

5.1 Introduction

Chapter 4 describes the working of the various blocks in the SparsePE core. The individual core works by computing a dot product between a sparse weight matrix and a subsection of the sparse input feature map to compute a subsection of the output feature map. To compute the *full* output feature map, corresponding to a particular layer, we design a two-dimensional accelerator comprising of the SparsePE cores, which we refer to as the Phantom accelerator. The main contributions of the accelerator are as follows:

- The Phantom accelerator is capable of handling all types of convolution operations, including strided convolution (regular and separable). Also, unlike the NeuroMAX

accelerator, it can handle the FC layers which makes it fully deployable for any CNN processing.

- It addresses the problem of *load balancing* by using a two-tiered, on-the-fly, load balancing strategy. This balancing does not require any modification of the original CNN model.

- Phantom accelerator has a performance gain of 12× over the dense NeuroMAX accelerator. For two-sided sparse accelerators, it provides a performance gain of 4.2×, 2.38×, and 1.98× over SCNN[55], Eyeriss v2[52], and SparTen[56], respectively, which are some of the most cited recent sparse CNN accelerators.

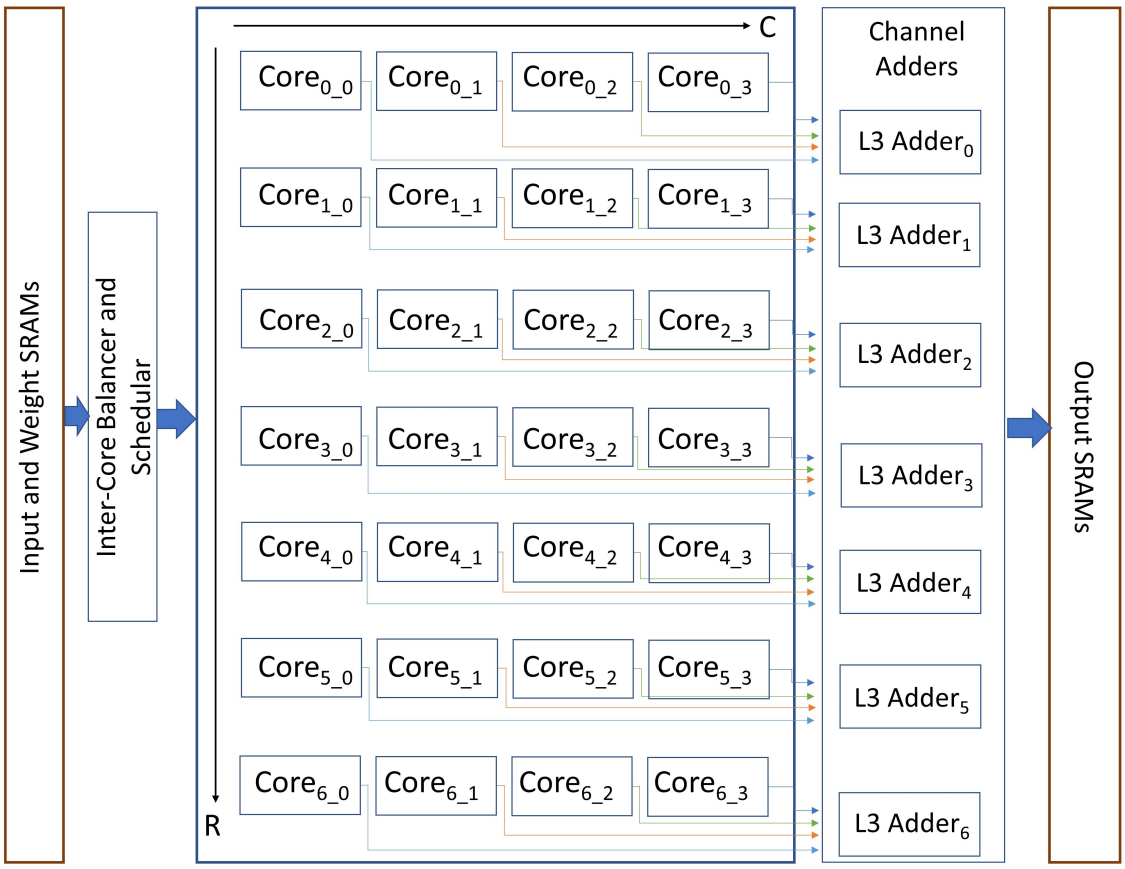


Figure 5.1: *Phantom architecture*

5.2 Phantom

Figure 5.1 shows the architectural overview of the Phantom accelerator. We envision that the Phantom accelerator connects to a memory bus and accepts data and instructions from a CPU. The accelerator contains on-chip SRAMs (weight, input, and output), the *inter-Core* balancer and scheduler, the $R \times C$ compute matrix, comprising of the SparsePE cores, and the accumulator circuits.

Table 5.1: *Popular CNN layer dimensions*

CNN	Layer Dims	Channel count
VGG-16	$3 \times [224 \times 224]$ $2 \times [112 \times 112]$ $3 \times [56 \times 56]$ $3 \times [28 \times 28]$ $3 \times [14 \times 14]$ $4 \times [7 \times 7]$	[3] $2 \times [64]$ $2 \times [128]$ $3 \times [256]$ $3 \times [512]$ $3 \times [512]$
MobileNet	$1 \times [224 \times 224]$ $3 \times [112 \times 112]$ $4 \times [56 \times 56]$ $4 \times [28 \times 28]$ $4 \times [14 \times 14]$ $4 \times [7 \times 7]$	[3] $2 \times [32]$ $2 \times [64]$ $4 \times [128]$ $4 \times [256]$ $4 \times [512]$ $2 \times [1024]$

5.2.1 R×C compute matrix

The compute unit consists of an $R \times C$ matrix of the SparsePE cores and R adders for channel accumulation. Here, R and C refers to the rows and columns, respectively. The design choice for R and C, and the dataflow associated with the transfer of data across various SparsePE cores, is based on the following design goals:

G1: To maximize the data reuse (weights and input) across multiple input subsections.

G2: To optimize the data scheduling across various SparsePE cores to maximize the *theoretical* hardware utilization.

G3: To support **all** layers of a CNN. This includes support for a variety of CONV layers and the FC layers.

Looking deeper into the dimensions and configurations of various popular CNN models (VGGnet[4], Resnet[34], MobileNets[35;36], Googlenet[5] etc), we observe that the channel count for various CNN layers is almost always a multiple of 4 (Table 5.1). In the Phantom architecture, the channels in a CNN layer are broken along the columns. Therefore, to ensure that the cores are engaged *most* of the time, and thus, satisfy G2, we set the C value equal to 4. Similar rationale applies for the choice of $R = 7$. The input subsections are broken along the vertical axis and distributed along the rows. Most of the popular CNNs have an input layer size (width or height) S, a multiple of 7, thus, having $R = 7$, ensures equal distribution of chunks of data among all the cores.

5.2.2 Load balancing

The design choice for $R \times C$ matrix is highly dependent on the layer dimensions of various CNNs. It, however, has no relevance to the static and dynamic sparsity in weights and inputs, respectively. Efficient reuse of data is one of the key requirements to minimize the memory accesses, repeatedly, for the same data. This reuse, however, amplifies the computational imbalance among different PEs. If the same filter is held in the local memory of the PEs and the input subsections are swept across the PEs, the subsections with higher density would inevitably take more cycles to compute the output, compared to the PEs receiving the subsections with lower density. This varying sparsity of the input maps would create a *system-level* load imbalance among the PEs which would be exposed during the next filter broadcast. Holding the input maps and sweeping the filters would also have similar results, as would the buffering of data. In order to address this *system-level* load imbalance, we incorporate a two-level load balancing strategy in the *Phantom* architecture. The **Inter-Core** balancer (Figure 5.1), balances the computational load dynamically using the density of the weight matrix such that each phantom core works on the weight matrices with the same/similar density over the CNN layers (as described in Section 5.2.3 with an example).

This balancing is only performed when the weight data is actively being reused (e.g. in regular and depthwise seperable CONV). The second balancer, referred to as **Intra-Core** balancer, is local to each core, and performs column-wise balancing of the weight matrix. This balancer is always enabled, regardless of the layer, and significantly improves the individual thread scheduling performed by the TDS (Section 4.4.2).

In the subsequent sections, we demonstrate, with examples, the designed dataflow for various CONV and FC layers. We choose the input sizes that fit well with the layer sizes of actual CNNs. We also show how the balancing is performed to prevent idling of the cores, which in turn, maximizes the throughput, all the while ensuring high data reuse.

5.2.3 Regular/depthwise convolution

Figure 5.2 shows a 3×3 depthwise separable convolution example, where 8, 3×3 filters are convolved with 8, 9×5 inputs to generate 8, 7×3 outputs. We choose these size parameters as an example because they fit well when considering the layer sizes of the actual CNNs. Figure 5.2 also shows the scheduling and mapping of data on to the $R \times C$ matrix. The input is broken down into n chunks of size 3×3 along the rows, where, n is the number of rows of the output. These n chunks are then scheduled along the rows of the $R \times C$ matrix. Each SparsePE core processes 3 rows until all input columns are exhausted. The filters F0 to F7 represent the channel wise filters, with each column of the $R \times C$ matrix processing the same set of filters (F_R), thereby, achieving filter reuse (G1). The non-unit stride convolutions follow the same dataflow. The L3 adder circuits perform accumulation along the rows to generate the final outputs. Because of the efficient dataflow, and choice of the $R \times C$ dimensions, all SparsePE cores are provided data for a particular processing chunk, thereby, achieving a 100% inter-core utilization (G2).

Inter-core balancing

The inter-core balancing is performed for the layers that support filter reuse because of the static nature of the weights. In the proposed dataflow, the layers that support filter reuse

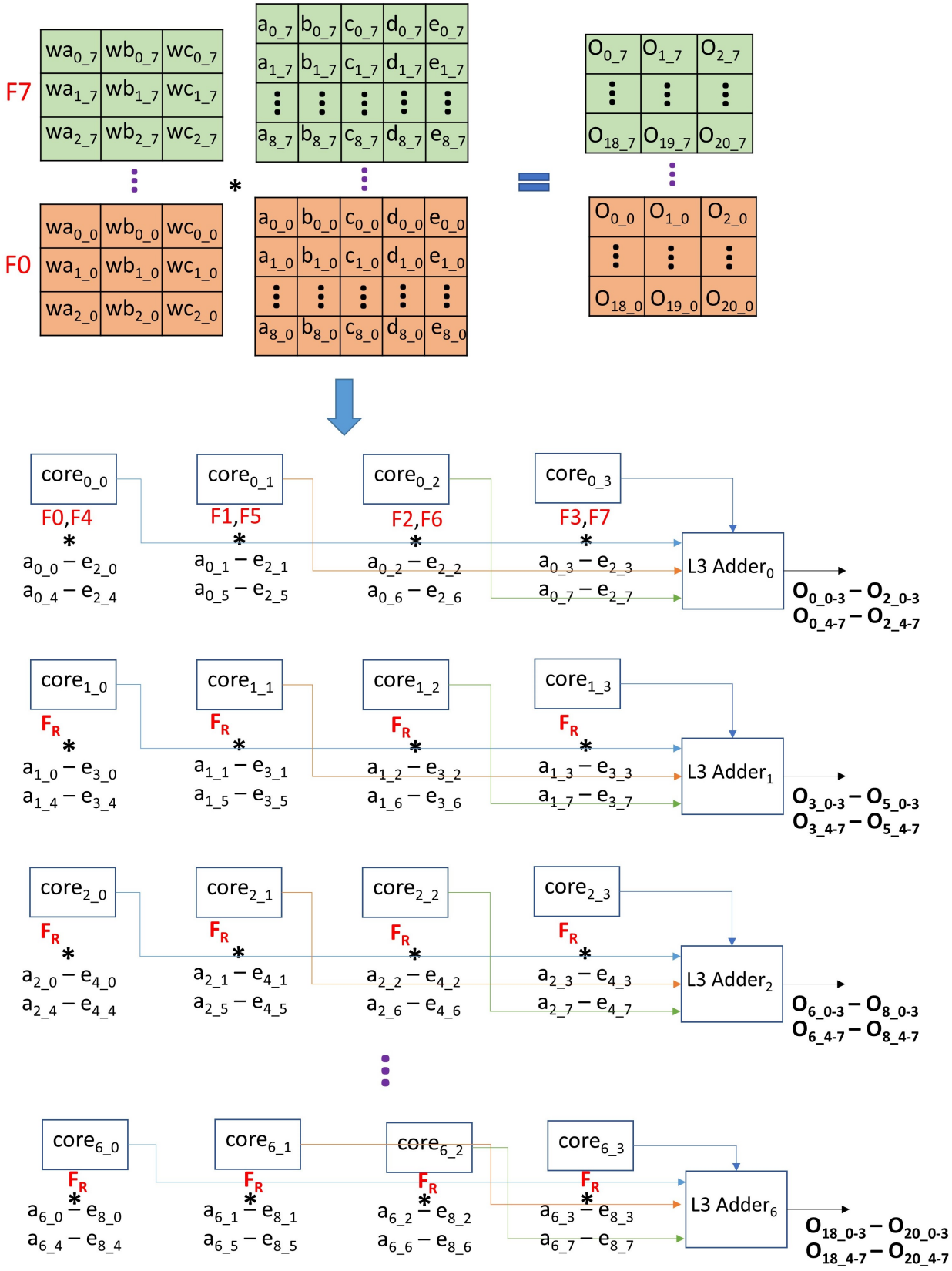


Figure 5.2: *Phantom support for regular/depthwise convolution*

are the regular and the depthwise separable convolution layers. As shown in Figure 5.2, each $R \times C$ matrix column, at any given time, processes the same filter. The filters are scheduled in a *low latency, more dense* and *high latency, less dense* approach. Assuming that during the first filters (F0, F1, F2, F3) broadcast, the first column's processing latency is the lowest. The scheduler schedules the filter with the highest density (assume F4) and the associated input channel to the first column, as shown in Figure 5.2. In a similar fashion, as the columns proceed to completion, the next filters are scheduled based on the order of their completion. This ensures that the processing completes uniformly across all the columns so that the idle time on SparsePE cores is minimized. Computation of density is trivial as it involves finding the total number of ones in the sparse mask of the filters, and thus, requires minimal resources.

5.2.4 Pointwise convolution

Figure 5.3 shows a pointwise convolution example where a set of $1 \times 1 \times 36 \times 7$ sparse filters convolve with a $3 \times 4 \times 36$ sparse input to produce a $3 \times 4 \times 7$ output. These dimensional parameters are a good representative of the layer dimensions of actual CNNs. The right side of Figure 5.3 shows the dataflow and the mapping of the various computations involved for this example. It can be seen that the 7 filters (wa - wg) are scheduled along the 7 rows of SparsePE cores with each row processing equal number of channels along the columns. The channels are equally divided based on the number of combined compute threads in each SparsePE core. Since, in this design, each SparsePE core consists of 3 PEs, with each PE containing 3 threads, the channels are divided equally into batches of 9 and scheduled along the columns to maximize the hardware utilization (G2). To enhance data reuse (G1), each weight matrix is held locally in a particular core while the input is swept across it. The input is scheduled in a *channel-first* manner, followed by rows, and then columns. From Figure 5.3, the input $a_{0\text{-}2_0\text{-}8}$ is a $3 \times 1 \times 9$ matrix. The 9 values $a_{0_0\text{-}8}$, in the first row are provided first, followed by the 9 values in the second row. After all the rows have been exhausted, the next column ($b_{0\text{-}2_0\text{-}8}$) is evaluated in a similar fashion till all the columns have been

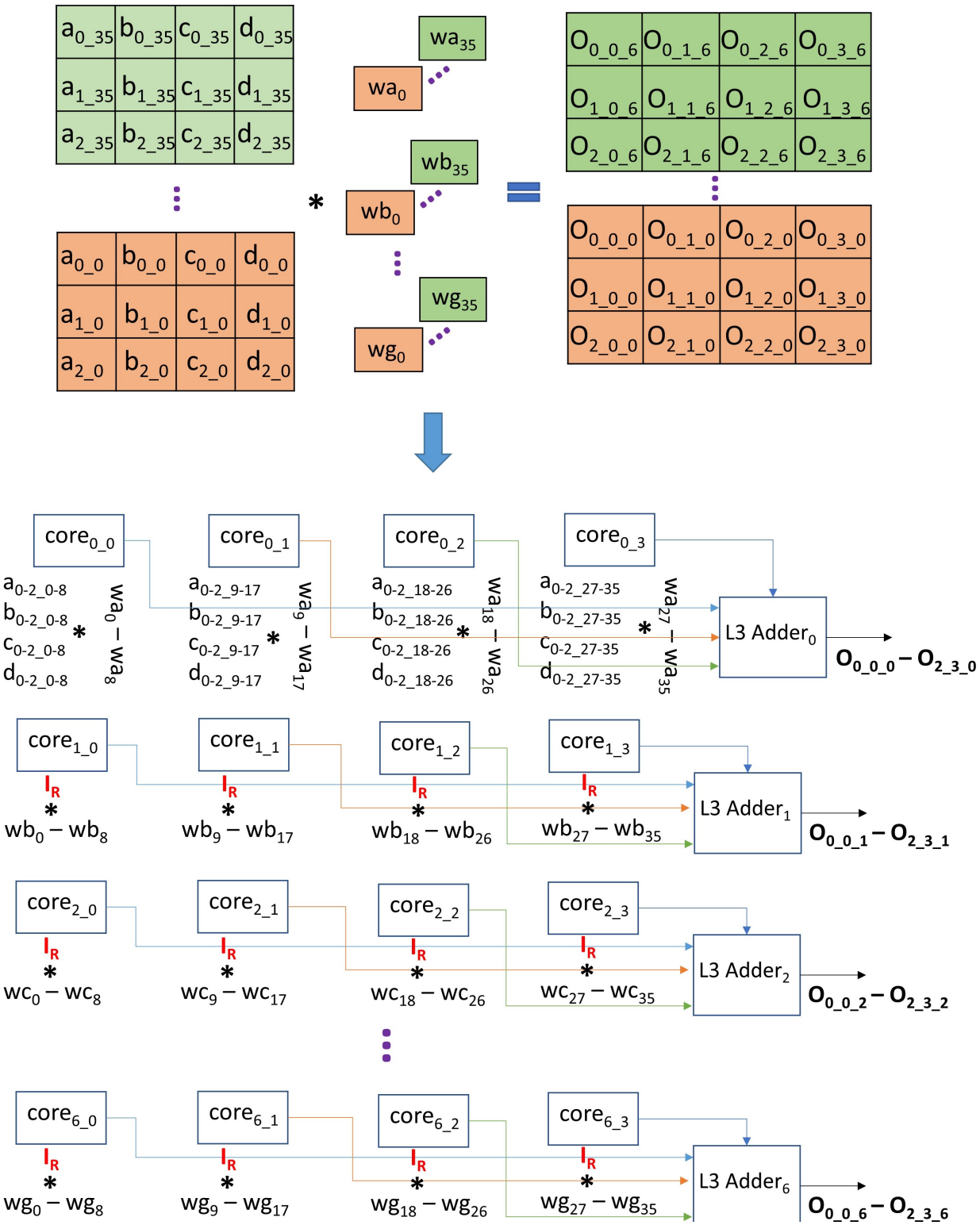


Figure 5.3: *Phantom support for pointwise convolution*

swept through. The L3 adder circuit gathers all the partial outputs from the cores along the columns to generate the required outputs.

5.2.5 FC layers

FC layers are an inherent part of modern CNN designs and, therefore, need to be accounted for in CNN accelerator designs (G3). AlexNet and VGG-16 both have FC layers with activation vectors that are 4K long and weight matrix that is 4K $\times$ 4K long. Similarly, Mobilenet has an FC layer with activation vector that is 1K long and weight matrix that is 1K $\times$ 1K long. The FC parameters comprise of a total of 24.3% of all the parameters in the Mobilenet. FC computations, therefore, are crucial and need to be accounted for in CNN accelerator designs (G3).

Figure 5.4 shows an FC computation example where a length 36 sparse input vector (I_0-I_{35}) is element-wise convolved with a 36 $\times$ 49 sparse weight matrix to generate a length 49 output vector. Figure 5.4 also shows the dataflow and the computational mapping of the example onto the SparsePE cores in the Phantom architecture. Similar to the pointwise convolution, the input and the weight channels are broken into 4 batches of length 9 and scheduled across the columns. The input vector, is held stationary (input stationary) across the rows and the individual weight vectors are swept over the input vector to generate the partial outputs. Similar accumulation is performed by the L3 adders to generate the final outputs.

5.2.6 Intra-core balancing

The *Intra-Core* balancer performs thread-level balancing inside every SparsePE core. This is opposed to the *Inter-Core* balancer which performs balancing across all the SparsePE cores in the Phantom architecture. Recall that the TDS (Section 4.4.2) operates in a column-wise manner where each column of the SEL_R outputs are evaluated concurrently. Also, recall that the total number of ones selected by the TDS per column are less than or equal to the number of threads per PE (3 in this case). Because of the column-wise selection, the

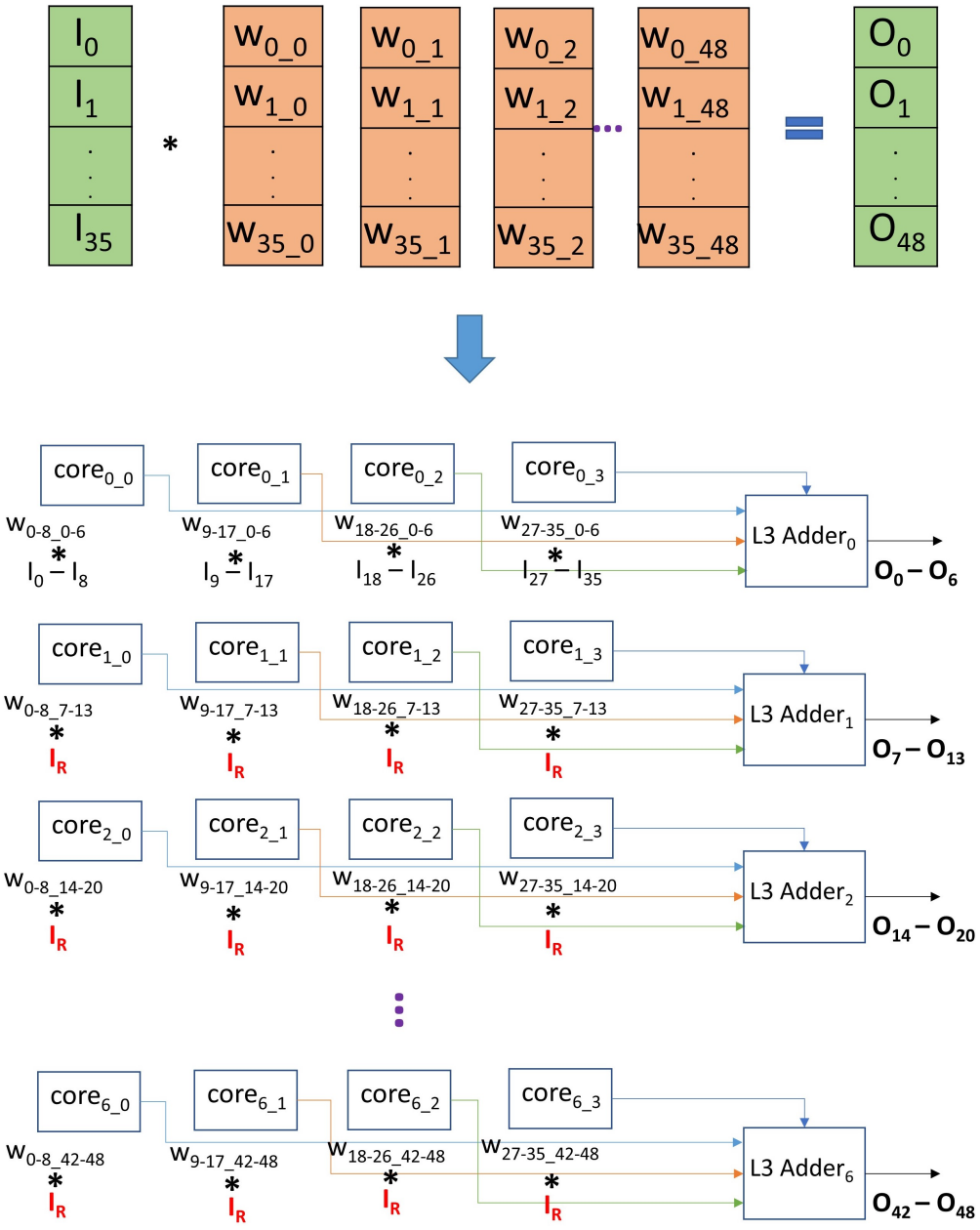


Figure 5.4: *Phantom support for FC layers*

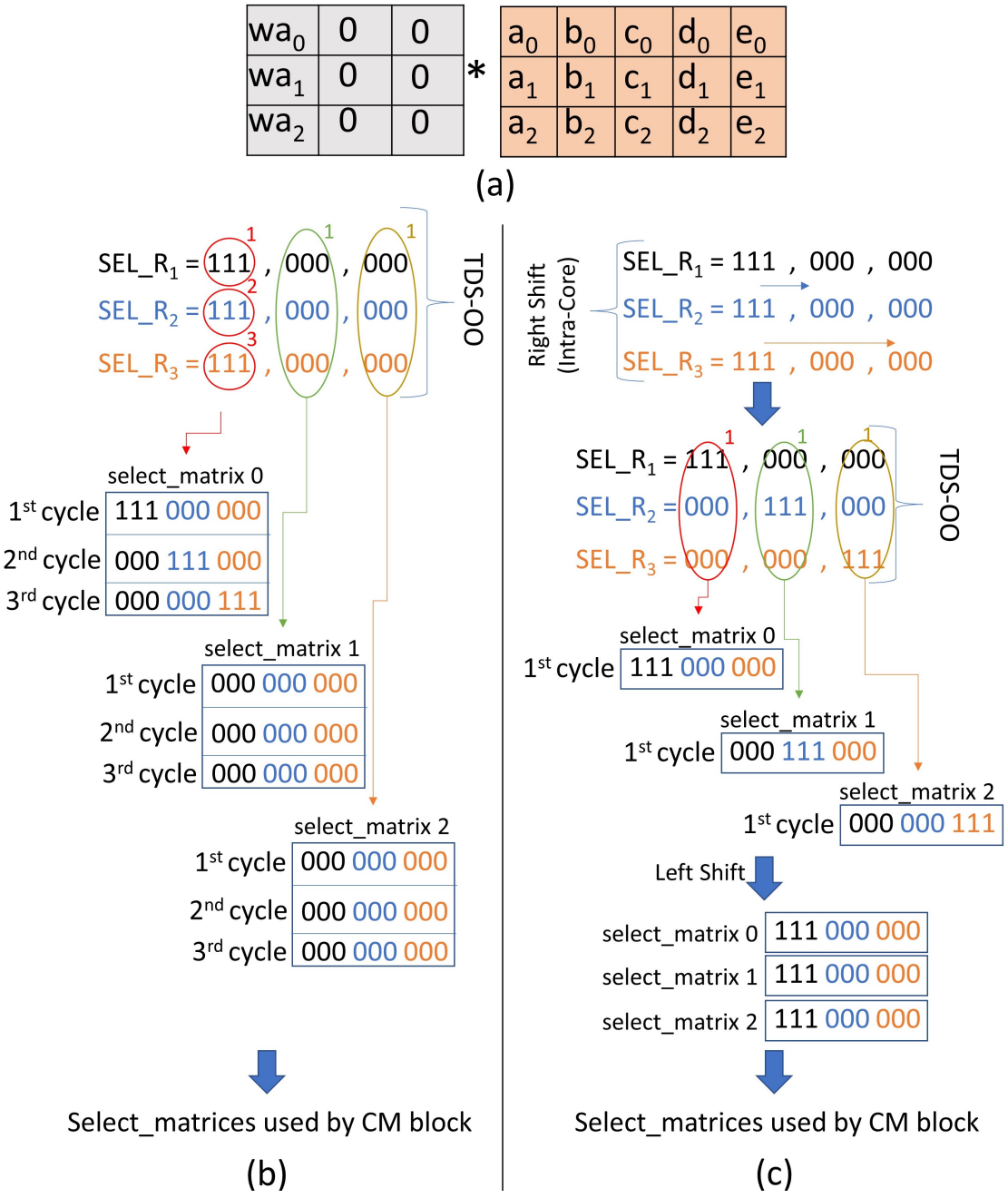


Figure 5.5: *Intra-Core balancing (a) Test example (b) Without balancing (c) With balancing*

TDS latency is bounded by the column with the highest density. Figure 5.5(a) shows a test example to explain the intra-core balancing and its significance. Figure 5.5(b) shows the core operation without any balancing. The first column of the weight matrix has the highest density. This is also reflected in the generated SEL_R values. This uneven distribution results in the first column taking the highest number of cycles, whereas, the second and the third column selection completing in only one cycle, as shown in Figure 5.5(b). This stalls the core as the core must wait for the first column to complete all 3 cycles before processing the next block which significantly decreases the thread utilization of the core (Valid Computations/(Cycles $\times$ PEs $\times$ ThreadsPerPE) = $9/(3 \times 3 \times 3) = 33\%$).

Figure 5.5(c) shows how the intra-core balancer mitigates this issue by efficient distribution of the load, prior to the TDS operation, in a relatively simplistic manner. A right circular shift is performed on SEL_R_2 and SEL_R_3 values, as shown in Figure 5.5(c). This modifies the load distribution among the three SEL_R columns. Operation of the TDS this time, ensures the selection of all three columns in one cycle. A circular left shift is performed on the generated select_matrices in the same manner as the circular right shift in the first step, to ensure a valid mapping by the computation block in the SparsePE. The updated matrices, after shifting, are shown in Figure 5.5(c). The updated matrices are then used in the computation block for computation mapping. The thread utilization of the core, in this case, increases drastically (Valid Computations/(Cycles $\times$ PEs $\times$ ThreadsPerPE) = $9/(1 \times 3 \times 3) = 100\%$). Finally, it should be noted that the evaluation of the example in Figure 5.5, is performed with a lookahead factor (n in Section 4.4.2) of 3. In this section, for clarity, we will use L_f to define the lookahead factor. We will further explore the combined affect of both, the inter-core, and the intra-core balancing, on performance, in Section 5.3.

Table 5.2: *Operation parameters*

Operation	Level	Parameters
TDS	SparsePE	TDS_inOrder TDS_outOrder
Balancing	SparsePE Phantom	Unbalanced intra_core inter_core Full(inter + intra)
Lookahead Factor (L_f)	SparsePE	$3 \leq L_f \leq 27$
CNN models	Phantom	Dense Sparse

Table 5.3: *Accelerator configuration*

Configuration Parameter	Value
Compute Matrix Size	28 (7×4)
PEs per SparsePE core	3
Threads per PE	3
Total Compute Threads/MACs	252

5.3 Experiments and results

5.3.1 Evaluation methodology

Cycle-Accurate Simulator: To accurately model the performance of an individual SparsePE core and the Phantom accelerator, as a whole, we design a cycle-level performance simulator. The simulator is parameterizable across various core and architecture level design parameters to capture their effect on the system wide performance. Table 5.2 shows the modifiable design parameters. The operations are categorized as core level and architecture level. Table 5.3 shows the arrangement and configuration of the Phantom accelerator's compute matrix containing SparsePE cores.

The simulator has a set of 5 built-in test scenarios which covers the sweeping of all the parameters shown in Table 5.2. During the processing of each layer, every core outputs 7 values which includes the total cycles by a dense design, the cycles of TDS in-order and TDS out-of-order, coupled with two-level load balancing, and the average thread utilization for various configurations. The evaluation files in the simulator use the data provided by

the individual cores and schedulers to generate the throughput and speedup results for various dense and sparse CNN models. The simulator also contains routines for NeuroMAX, SparTen[56], SCNN[55], and Eyeriss v2[52] for performing comparisons.

Simulated Models: Although we test our design on many CNN models including Alexnet[3], VGG16[4], MobileNet[35], MobileNet v2[36], and GoogleNet[5], we present the results for sparse versions of VGG16 and MobileNet for comparison purposes. The reason for our selection of these models is because they cover the different types of layers usually present in all CNN models. These layers include regular and depthwise separable, unit and non-unit stride convolutions and FC layers. We use the approach presented in[46] to prune these networks using the MATLAB's *Deep Learning Toolbox*, and ensure that we achieve the same level of weight sparsity as previous approaches, for fair comparisons. The activation sparsity is highly dependent on the input and changes dynamically during the inference process. We, therefore, average out the input sparsity for a batch of 100 randomly selected inputs. After pruning of the network, we generate the sparse binary masks for every layer and generate a network containing only the sparse masks, since only this information is needed to efficiently represent the MAC operations needed per layer for the Phantom accelerator. The sparse masks are fed into the simulator in the model dimensions (i.e., mapped to the dimension of each CNN layer in the model). The simulator's scheduler uses the dataflow for various layers, presented in Figures 5.2 - 5.4, to break down the dimensions, and schedule the individual binary masks to different SparsePE cores.

5.3.2 Results

We capture the simulator's results in an incremental approach, starting from the basic core-level configurations, and moving towards system-level configurations. All the comparisons are made with an equivalent dense architecture, having an equal number of MAC units, but without leveraging the sparse optimizations. At the end, we compare different versions of Phantom accelerator against previously proposed two-sided sparse architectures.

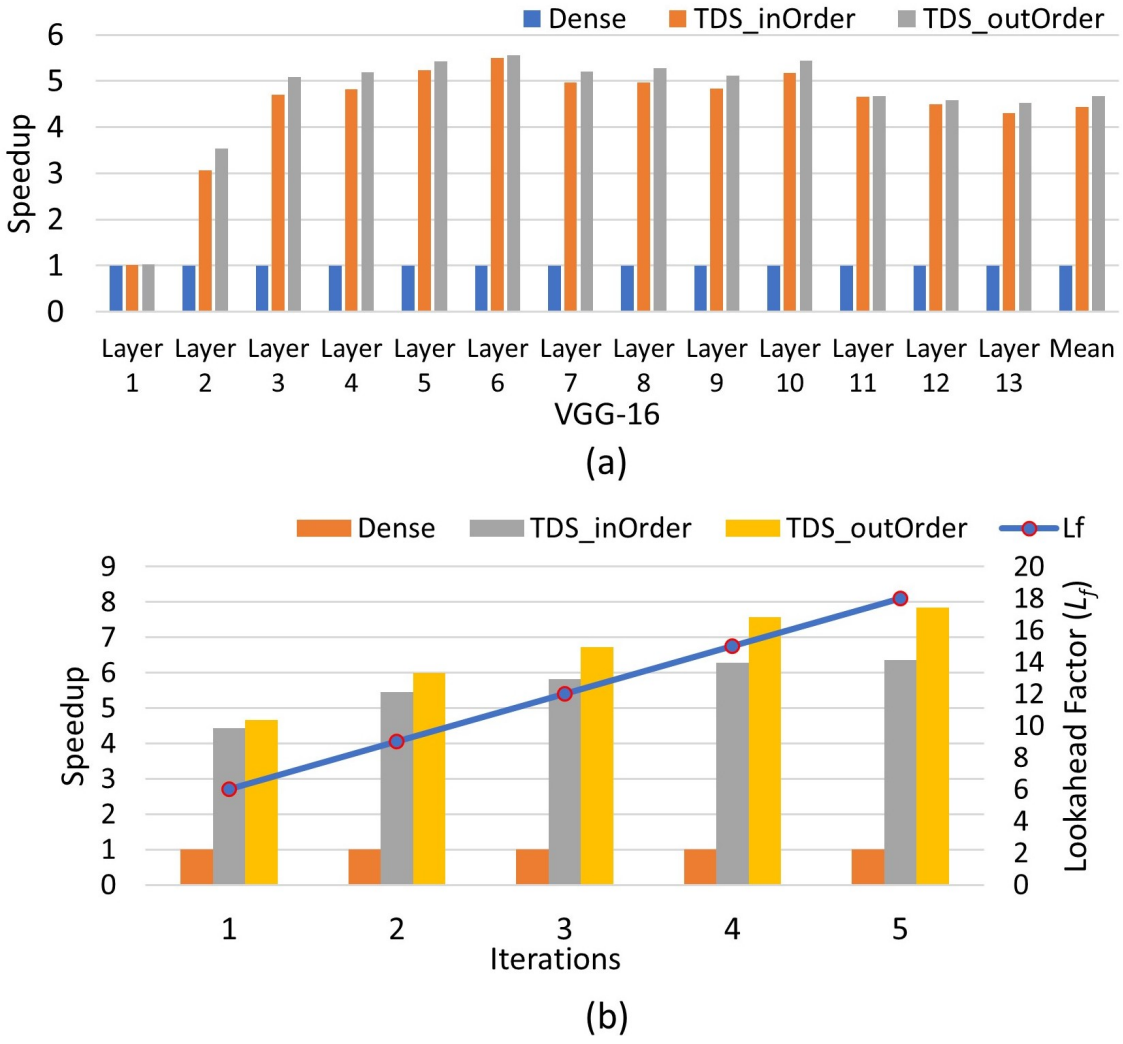


Figure 5.6: *TDS-IO vs. TDS-OO (a) Per layer comparison with $L_f = 6$ (b) Comparisons with changing L_f*

TDS Variants Comparison

Figure 5.6(a) shows the performance comparison of the two TDS variants, namely, the in-order TDS (TDS-IO) and the out-of-order TDS (TDS-OO), evaluated on a sparse VGG16 net. We set the lookahead factor (L_f) (Section 4.4.2) for this test to 6 for both the variants. The performance of the dense architecture can be modeled by setting the L_f value equal to 1. This would ensure that no future computations are observed by the TDS, thus, replicating a dense accelerator like NeuroMAX. It can be seen that the first layer does not have any significant performance gains over a dense architecture because of very low sparsity. The performance gains increase drastically in subsequent layers with TDS-IO variant, on average, being 4.5×, and the TDS-OO variant, being 4.8× faster, than the equivalent dense architecture. This represents a 1.07× performance improvement of TDS-OO over TDS-IO. The performance difference, however, improves substantially as L_f is increased, as shown in Figure 5.6(b). We run the VGG16 net a total of five times, and average out the speedups, while sweeping L_f from 6 to 18, with a jump of three in every iteration. For $L_f = 18$, we observe a 6.35× and a 7.9× performance gain of TDS-IO and TDS-OO, respectively, over dense architecture, representing a 43% and a 68% increase, when switching from $L_f = 3$ to $L_f = 18$, respectively. TDS-OO, for $L_f = 18$, gives a 1.24× performance gain over TDS-IO, a jump of 16% from $L_f = 6$. These results are inline with our preliminary observations in Section 2.4. The improvement in the core's thread utilization from TDS-OO directly correlates to the increase in performance. For the next experiments, we will always use the TDS-OO variant.

Impact of Load Balancing

Figure 5.7 shows the impact of load balancing (*inter-core* + *intra-core*) on the performance for sparse VGG16 and MobileNet with $L_f = 6$. During the initial layers for both the CNNs, we observe a drastic improvement in performance (as much as 1.5× for VGG16 and 1.3× for MobileNet). On average, we observe a performance difference of 1.1× and 1.08× for VGG16 and MobileNet, respectively. From our experiments, we also observe that the

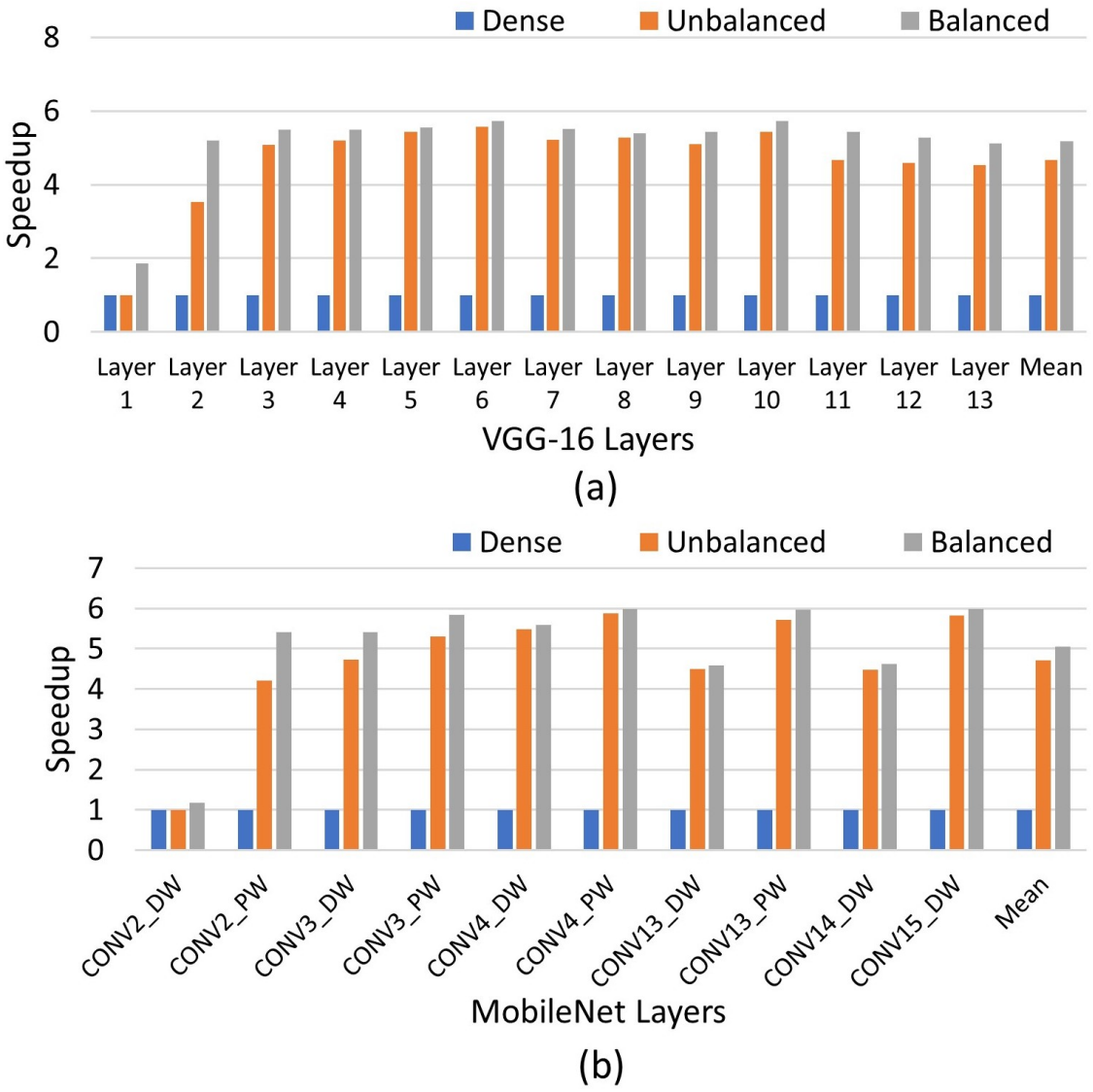


Figure 5.7: *Balanced vs. unbalanced at $L_f = 6$ for (a) Sparse VGG16 (b) Sparse MobileNet*

performance difference due to *intra-core* balancing increases drastically at high sparsity and greater value of L_f. This conclusion is inline with the example in Figure 5.5, where the increase in thread utilization from *intra-core* balancing increases the speedup by 3×. The *inter-core* balancing is dominant in later layers where there are a large number of channels. For reduction in simulation times, we only use approximately 25% of the channel filters for our simulations in the case of regular (and depthwise seperable) convolutions, which prevents us from exploiting the full power of the *inter-core* balancing. Hence, Figures 5.7(a) and (b) do not show a significant improvement in layers with a large number of channels. Based on our experiments, we observe, on average, a 7% increase in speedup by having 15% more filters in our simulations.

Sensitivity to Sparsity and L_f

The performance simulator has the capability to sweep the weight/activation sparsity from low 0.1/0.1 (10%) to high 1.0/1.0 (100%). This is shown in Figures 5.8 and 5.9. We also plot the average thread utilization at different levels of sparsity for VGG16 and MobileNet. For a dense architecture, we observe a higher thread utilization at low sparsity and lower thread utilization at a higher sparsity, as shown in Figures 5.8(b) and 5.9(b). Phantom, however, exhibits significantly higher thread utilization compared to a dense architecture, even at high levels of sparsity. For VGG16, Phantom consistently keeps the thread utilization higher than 90% even at 60% sparsity for both weights and activations, whereas, as expected, the utilization for the dense architecture decreases by $25-30\%$ and decreases by almost 50% at higher sparsity levels. This thread utilization difference directly correlates to greater speedups at mid to high levels of sparsity, as shown in Figures 5.8(a) and 5.9(a). At higher sparsity levels (0.8/0.8 - 1.0/1.0), we observe a massive speedup over dense architecture even when the thread utilization starts decreasing. This is because the Phantom accelerator starts actively skipping all the *zero* computations without wasting compute cycles.

For convenience in comparing different versions of the Phantom accelerators, we rename them as *Phantom-CV* (conservative, with $L_f = 9$, balanced), *Phantom-MD* (moderate, with

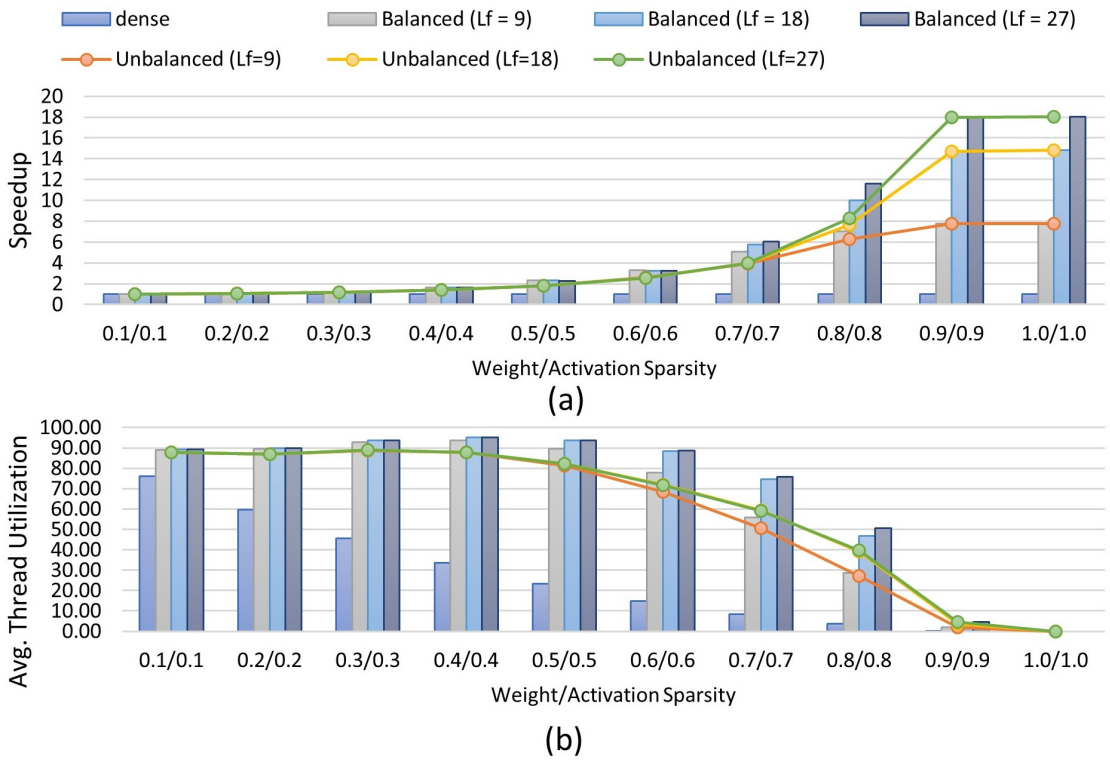


Figure 5.8: *Sensitivity to sparsity and L_f for VGG16 (a) Speedup (b) Average thread utilization*

$L_f = 18$, balanced), and *Phantom-HP* (high-performance, with $L_f = 27$, balanced). At low sparsity, all three versions exhibit similar speedups and thread utilization. This difference increases at higher sparsity levels with *Phantom-MD* and *Phantom-HP* being 1.43× and 1.65× faster, respectively, than *Phantom-CV*, at 80% sparsity. The unbalanced configurations display a similar trend among each other. Comparison between the *Phantom-HP*, balanced and unbalanced configurations, show a 1.4× speedup of balanced over unbalanced at 80% sparsity. Analysis of hardware resources show that the *Phantom-HP* requires only 1.05× more LUTs than *Phantom-CV*. This is because the higher values of L_f only increase the LUT count of LAM and TDS blocks, whereas, the Mapper, CE, and the OB blocks' LUT count remains the same.

Comparison Against Past Approaches

Figure 5.10 shows the speedup comparison of the different versions of the Phantom accelerators over dense architecture, SCNN, and SparTen, for sparse VGG16 net. Note that both the SCNN, and SparTen, do not support FC layers, whereas, Phantom does. In addition, SCNN also does not support non-unit stride convolutions present in Alexnet and MobileNet. Therefore, for fair comparison, we omit the last three FC layers in our results and use sparse VGG16 which does not contain any non-unit stride convolution. We observe that the *Phantom-CV* version, on average, performs 1.05×, 2.56×, and 6.4×, better than SparTen, SCNN, and dense architecture, respectively. The speedup increases to approximately 1.57×, 3.8×, and 9.9× for *Phantom-MD*, and 1.98×, 4.1×, and 11×, for *Phantom-HP* over SparTen, SCNN, and dense architecture, respectively. Upon comparing different versions of Phantom, we observe that, on average, *Phantom-HP* has a 67% and a 14% improvement in performance, compared to the *Phantom-CV* and *Phantom-MD* versions, respectively. Lastly, we would like to point out the impact of FC layers on the Phantom versions. Our experiments show, on average, a speedup of 13×, 11.4×, and 8.6× for *Phantom-HP*, *Phantom-MD*, and *Phantom-CV*, respectively, over dense architecture after inclusion of FC14, FC15, and FC16 layers of the VGG16 net. This improvement corresponds to our efficient dataflow and scheduling for the FC layers.

Figure 5.11 shows the performance comparison of the Phantom versions against Eyeriss v2 on sparse MobileNet. Authors of Eyeriss v2 performed their comparisons against their previous approach (Eyeriss[45]). Eyeriss v2 uses twice the number of MACs as Eyeriss, which doubles their final speedups. For fair comparison, we adjust the speedup values of Eyeriss v2 for comparison against a dense accelerator having the same number of MACs and only use the layers used by Eyeriss v2. We observe that, on average, *Phantom-CV* performs 1.04× better than Eyeriss v2, whereas, the *MD* and *HP* versions, on average perform 1.71× and 2.86×, respectively, better than Eyeriss v2. Doing a layer by layer comparison, we observe that Eyeriss v2 performs better than Phantom in CONV2-DW (DW indicates depthwise convolution) because of its efficient hierarchical mesh network-on-chip (NoC), however,

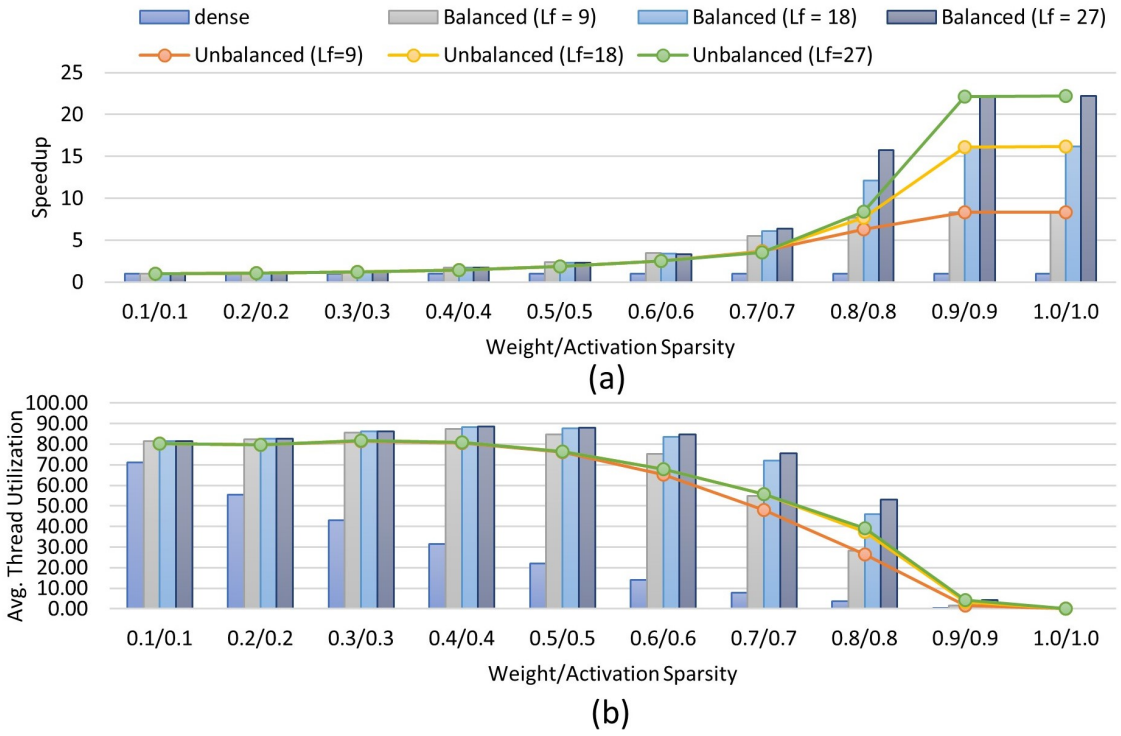


Figure 5.9: *Sensitivity to sparsity and L_f for MobileNet (a) Speedup (b) Average thread utilization*

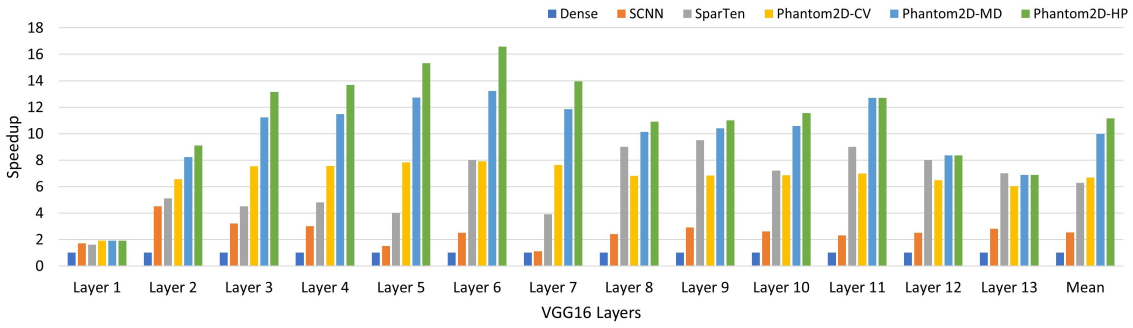


Figure 5.10: *VGG16 speedup comparison with SCNN and SparTen*

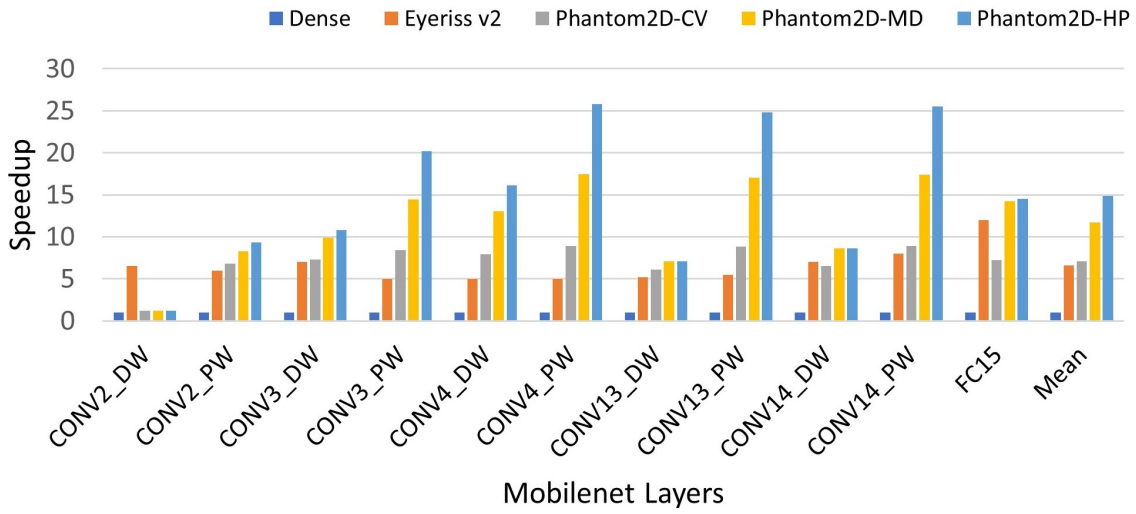


Figure 5.11: *Speedup comparison with Eyeriss v2*

Table 5.4: *Comparison of the Phantom accelerator with previous designs*

Property	Phantom accelerator	Eyerissv2[52]	SparTen[56]
Technology	Z-7100	65nm	Cyclone IV
Precision(bits)	8-bit	8-20 bits	8-bit
PE number	252	192	256
Frequency (MHz)	200	200	50
Accelerator type	$\text{sparse}_{\text{wa}}$	$\text{sparse}_{\text{wa}}$	$\text{sparse}_{\text{wa}}$
Resources (LUTs(a),Gates(b))	112k(a)	2695k(b)	unreported
Core only Power (W)	3.7	0.460	unreported
Supported layers	CONV + FC	CONV + FC	CONV only

Phantom-HP catches up in deeper layers and almost always provides an improvement. Point-wise layers are especially faster in the case of *Phantom-HP*, with the average speedup of 4.5× over Eyeriss v2, and 25× over a dense architecture. This is because of the efficient channel wise breakdown offered by our dataflow for these layers. Comparing different versions of the Phantom accelerators, *Phantom-HP*, on average, offers 108.9% and 27.4% increase in speedup over *Phantom-CV* and *Phantom-MD*, respectively, for sparse MobileNet.

Table 5.4 and 5.5 shows the implementation details of some recently proposed sparse CNN accelerators. Eyeriss v2, implemented on a 65nm ASIC, supports two-sided sparsity ($\text{sparse}_{\text{wa}}$), i.e., sparsity in both weights and activations. Eyeriss v2, however, uses a total of 2695k gates which represents a 108% increase in area cost when compared to the original Eyeriss[45]. This is because of the relatively complex design of the PEs of Eyeriss v2. SparTen

Table 5.5: *Comparison of the Phantom accelerator with previous designs*

Property	ZASCA [107]	Zhu et al. [108]	Xie et al. [109]	Lu et al. [110]
Technology	65nm	ZCU102	Aria 10	ZCU102
Precision(bits)	16-bit	16-bit	8-bit	16-bit
PE number	192	192	512	288
Frequency (MHz)	200	200	170	200
Accelerator type	sparse_a	sparse_w	sparse_w	sparse_w
Resources (LUTs(a),Gates(b))	1036k(b)	390k(a)	102.6k(a)	132k(a)
Core only Power (W)	0.301	unreported	4.6	unreported
Supported layers	CONV + FC	CONV + FC	CONV only	CONV only

accelerator[56] is implemented on Intel Cyclone IV FPGA and operates at 50 MHz. It also supports two-sided sparsity but has no support for FC layers. Although the implementation cost of SparTen is not reported, the architecture of SparTen requires complex inter-PE synchronization circuits which would greatly increase its cost. Zero-Activation-Skipping Convolutional Accelerator (ZASCA)[107], implemented on a 65nm ASIC, uses a total of 192 PEs and runs at 200 MHz. Just like Phantom, ZASCA accommodates both the CONV and the FC layers in its architecture. However, unlike Phantom, ZASCA only exploits activation sparsity (sparse_a). In addition, because of architectural limitations, ZASCA cannot fully exploit it's resources for 1×1 convolution, resulting in a significant decrease in hardware utilization for such convolutions. Zhu et al.[108] proposed a structured sparse CNN accelerator, implemented on Xilinx ZCU102 FPGA. The proposed accelerator uses structured pruning to reduce irregularities in sparse weights and employs a sparse wise dataflow scheme for high data reuse. The accelerator proposed in[108], however, only exploits sparsity in weights (sparse_w) and cannot exploit activation sparsity. Because of the complex design and the dataflow scheme, the accelerator proposed in[108] has a huge logic utilization cost (390k LUTs). Phantom accelerator, even though, employs 32% more PEs than[108], it, however, utilizes 71% lesser resources. Xie et al.[109] proposed a flexible accelerator architecture that supports both structured and unstructured weight sparsity. The accelerator is implemented on Intel Aria 10 SoC and uses a hybrid parallel (HP) dataflow. Even though, the resource count of the accelerator is lower than Phantom, the dataflow employed in the accelerator does not support FC layers. It, also, can only exploit weight sparsity (sparse_w) and has no support

for activation sparsity. The accelerator proposed in[110] uses a weight-oriented dataflow that performs element-matrix multiplication. It also uses a tile lookup table (TLUT) to match the sparse weight with the input pixel to exploit weight sparsity ($sparse_w$). Similar to[109], the accelerator proposed in[110] is also a CONV only accelerator and has no support for FC layers. It also, cannot exploit activation sparsity ($sparse_a$).

Comparing energy among different accelerators is quite challenging as it requires working RTLs. Also, many FPGA synthesis tools do not provide the DRAM energy for comparison against the accelerator's energy. We breakdown the energy into compute and memory energy, which can be further broken down into *zero* and *non-zero* access energy. Intuitively, Phantom's energy savings would be much closer to SparTen's, since SparTen also uses the sparse binary masks instead of a more traditional CSC format, used by Eyeriss v2. We have already established the superiority of using the sparse binary mask against the CSC format, when it comes to both compute and memory savings (Section 4.4). Authors of Sparten[56] have also confirmed this finding (Section 3.1 of Sparten[56]). The memory savings can be estimated by normalizing the *Phantom's* (non-zero) access with the dense architecture's (zero + non-zero) access. On average, a 35% and a 38% reduction in overall DRAM access for sparse VGG16 and MobileNet, respectively, was observed. Eyeriss v2 uses the CSC format that requires more DRAM accesses compared to the sparse binary mask, as already establish in Section 4.4. It's PEs also contain more complex logic to handle the CSC compressed, weight and activation data, compared to the relatively simple SparsePE core design. It is, therefore, established that the Phantom accelerator has superiority over Eyeriss v2 when it comes to memory and compute energy savings.

5.4 Conclusion

Designing a CNN accelerator to leverage the two-sided sparsity is quite challenging owing to the varying layer shapes and sizes, associated with a sparse model. In this chapter, the Phantom accelerator has been introduced. Phantom comprises of a system of SparsePE cores, arranged in a spatial matrix, and uses a novel dataflow that efficiently uses the capabilities

of the SparsePE cores to accelerate the sparse computations. As opposed to many previous approaches, the Phantom accelerator supports **all** layers of a CNN, including unit and non-unit stride convolutions, and FC layers. In addition, the Phantom accelerator leverages a two-level load balancing strategy that efficiently balances the load across the architecture level (*inter-core*), and at the thread level (*intra-core*) to minimize idling of the compute threads, thereby, further increasing the throughput. Comparisons have been made with many state-of-the-art two-sided sparse CNN accelerators. The simulations show that, on average, the Phantom accelerator performs 12×, 4.1×, 1.98×, and 2.36×, better than NeuroMAX, SCNN, SparTen, and Eyeriss v2, respectively, while retaining the energy efficiency of SparTen.

Chapter 6

Conclusions and future work

The pace of research in the development of high accuracy CNN models has greatly increased over the past few years. This has led researchers to develop high-performance and energy-efficient CNN inference accelerators for deploying CNN models on edge devices. The proposed accelerator designs are broadly classified as dense and sparse CNN accelerators. The dense accelerators are relatively low-cost but experience performance issues owing to the vast amount of computational requirements of a typical dense CNN model. The sparse CNN models decrease the computational requirements at the cost of complex accelerator architectures.

In this thesis, we first focus on accelerating the inference of the dense CNN models. Since the CONV layers of a CNN model account for more than 95% of the total inference processing, we focus on devising ways to accelerate the CONV layers. Towards this end, we introduce NeuroMAX, a multi-thread, log-based accelerator for accelerating the processing of dense CNN models. Unlike other accelerator designs that use linear quantization techniques, the NeuroMAX accelerator uses $\log_{\sqrt{2}}$ quantization. This type of quantization has two advantages; 1) Because of the companding nature of the log operation, the $\log_{\sqrt{2}}$ quantization allows more efficient mapping of the weight and the activation data using a lower number of bits as compared to linear quantization. This allows lower memory requirements for storing data and decreases the total off-chip DRAM accesses, which improves the energy efficiency

of the accelerator. 2) Performing multiply-accumulate (MAC), which forms the basis of the CONV layers, is reduced to a smaller and more area and resource-efficient shift and add operations. This allows the packing of more log processing element (PE) cores in the accelerator within the same area as the linear PE cores. The NeuroMAX accelerator design takes advantage of the above two points and uses a spatial grid of multi-threaded, log-based PEs for processing the log-quantized weight and activation values. This gives a significant performance improvement over other dense CNN accelerator designs with a minimal increase in area.

A major bottleneck in performing CNN inference on the edge is the limited compute and energy bandwidth of the edge device. Dense accelerators, even though resource-efficient, cannot decrease the total computations involved in dense CNN models. This has led researchers to come up with techniques, such as pruning, for reducing the total number of parameters in a CNN model without decreasing the model's accuracy. These models are commonly referred to as sparse CNN models. Sparse model processing in hardware, however, is complicated because of the variation in shapes and sizes of the weight matrices. Many accelerator architectures have been proposed that use compressed sparse row or column (CSR or CSC) for representing the sparse data, and then use specialized PEs for sparse data processing. These PEs are very complex and consume a lot of resources in the underlying hardware. To circumvent this problem, this thesis presents SparsePE, a high-performance, and area-efficient multi-threaded core, capable of processing both dense as well as sparse data. The SparsePE core uses a sparse mask for the representation of the sparse data. It uses simple logical blocks for scheduling effective sparse computations in order to maximize the hardware utilization and the throughput with a small area overhead. The use of a sparse mask for sparse data representation also proves to be more energy-efficient than its CSR and CSC counter part.

In the last part of the thesis, we introduce Phantom, a sparse CNN accelerator that leverages the advantages of the SparsePE core for processing sparse CNN models. The Phantom accelerator employs a spatial grid of the SparsePE cores and uses a novel dataflow that enables all-layer processing of a CNN model, including the strided CONV and the FC layers. This gives Phantom a major advantage over other accelerator architectures that

allow either the CONV layer or the FC layer processing, but not both. The Phantom accelerator also introduces a two-level load balancing strategy for equalizing the uneven distribution of the computational load across different SparsePE cores. This enables high gains in performance when compared against other sparse CNN accelerator architectures.

The research on the development of efficient CNN inference accelerator architectures is ongoing and many new techniques are being developed for improving the performance and energy efficiency of CNN accelerators. One technique that can be explored in the future is the use of approximate computing. Since sparse CNN accelerators decrease the total number of computations in an inference pass by exploiting the sparsity, the computations can be further decreased by *pre-approximation* of small intermediate results. Unlike pruning, the approximation can be performed during inference in real-time to speed up the computations by allocating resources for computations with a higher impact on the result.

Another possible avenue to explore in the future is to bring training on the device itself. This thesis focuses primarily on inference at the edge. However, in many applications, like self-driving vehicles, new data is constantly available during inference. This data can be used to simultaneously improve the CNN model's accuracy. This would require architectural support for training in accelerator design. Lastly, the thesis focuses mainly on CNN inference. Another avenue to explore is using the design techniques presented in this thesis to accelerate the inference of other popular DNNs like recurrent neural networks (RNNs), long short-term memory (LSTM) models, modular neural networks, and many more.

Bibliography

[1] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). volume 57, pages 10–14, 02 2014. ISBN 978-1-4799-0920-9. doi: 10.1109/ISSCC.2014.6757323.

[2] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015. URL http://arxiv.org/abs/1409.4842.

[6] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[7] K. Chang and T. Chang. Vwa: Hardware efficient vectorwise accelerator for convolutional neural network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(1):145–154, 2020.

[8] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730, 2015. doi: 10.1109/ICCV.2015.312.

[9] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015. ISSN 0920-5691. doi: 10.1007/s11263-015-0816-y. URL https://doi.org/10.1007/s11263-015-0816-y.

[10] li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, Yifan Gong, and Alex Acero. Recent advances in deep learning for speech research at microsoft. pages 8604–8608, 10 2013.

[11] Mingxing Tan and Quoc V. Le. Efficientnetv2: Smaller models and faster training. *CoRR*, abs/2104.00298, 2021. URL https://arxiv.org/abs/2104.00298.

[12] Y. Le Cun, L.D. Jackel, B. Boser, J.S. Denker, H.P. Graf, I. Guyon, D. Henderson, R.E. Howard, and W. Hubbard. Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989. doi: 10.1109/35.41400.

[13] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. 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. doi: 10.1109/MSP.2012.2205597.

[14] Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu,

and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 02 2011.

[15] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016. URL http://arxiv.org/abs/1609.03499.

[16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

[17] Tesla AI. URL https://www.tesla.com/AI.

[18] Waymo. URL https://waymo.com/waymo-driver/.

[19] Xia-an Bi, Yingchao Liu, Qin Jiang, Qing Shu, Qi Sun, and Jianhua Dai. The diagnosis of autism spectrum disorder based on the random neural network cluster. *Frontiers in Human Neuroscience*, 12:257, 2018. ISSN 1662-5161. doi: 10.3389/fnhum.2018.00257. URL https://www.frontiersin.org/article/10.3389/fnhum.2018.00257.

[20] Xinyu Guo, Kelli C Dominick, A. Minai, Hailong Li, Craig A. Erickson, and Long Lu. Diagnosing autism spectrum disorder from brain resting-state functional connectivity patterns using a deep neural network with a novel feature selection method. *Frontiers in Neuroscience*, 11, 2017.

[21] Alper Baştürk, Mehmet Emin Yüksei, Hasan Badem, and Abdullah Çalışkan. Deep neural network based diagnosis system for melanoma skin cancer. In *2017 25th Signal*

Processing and Communications Applications Conference (SIU), pages 1–4, 2017. doi: 10.1109/SIU.2017.7960563.

[22] Masoumeh Siar and Mohammad Teshnehlab. Brain tumor detection using deep neural network and machine learning algorithm. In *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 363–368, 2019. doi: 10.1109/ICCKE48569.2019.8964846.

[23] Mahmood Azhar Qureshi and Arslan Munir. Puf-ipa: A puf-based identity preserving protocol for internet of things authentication. In *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–7, 2020. doi: 10.1109/CCNC46108.2020.9045264.

[24] Mahmood Azhar Qureshi and Arslan Munir. Puf-rla: A puf-based reliable and lightweight authentication protocol employing binary string shuffling. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 576–584, 2019. doi: 10.1109/ICCD46524.2019.00084.

[25] Mahmood Azhar Qureshi and Arslan Munir. Puf-rake: A puf-based robust and lightweight authentication and key establishment protocol. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2021. doi: 10.1109/TDSC.2021.3059454.

[26] Pranesh Santikellur, Aritra Bhattacharyay, and Rajat Subhra Chakraborty. Deep learning based model building attacks on arbiter puf compositions. *IACR Cryptol. ePrint Arch.*, 2019:566, 2019.

[27] Khalid T. Mursi, Bipana Thapaliya, Yu Zhuang, Ahmad O. Aseeri, and Mohammed Saeed Alkatheiri. A fast deep learning method for security vulnerability study of xor pufs. *Electronics*, 9(10), 2020. ISSN 2079-9292. doi: 10.3390/electronics9101715. URL https://www.mdpi.com/2079-9292/9/10/1715.

[28] Yisroel Mirsky, Ambra Demontis, Jaidip Kotak, Ram Shankar, Deng Gelei, Liu Yang,

Xiangyu Zhang, Wenke Lee, Yuval Elovici, and Battista Biggio. The threat of offensive ai to organizations, 2021.

[29] Yuantian Miao, Chao Chen, Lei Pan, Qing-Long Han, Jun Zhang, and Yang Xiang. Machine learning based cyber attacks targeting on controlled information: A survey, 2021.

[30] Hooman Alavizadeh, Julian Jang-Jaccard, Tansu Alpcan, and Seyit A. Camtepe. A markov game model for ai-based cyber security attack mitigation, 2021.

[31] Dattaraj Rao and Shraddha Mane. Zero-shot learning approach to adaptive cybersecurity using explainable ai, 2021.

[32] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

[33] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies, 2016.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[35] 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, 2017.

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2018.

[37] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation, 2016.

[38] Sebastian Vogel, Mengyu Liang, Andre Guntoro, Walter Stechele, and Gerd Ascheid. Efficient hardware acceleration of cnns using logarithmic data representation with arbitrary log-base. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359504. doi: 10.1145/3240765.3240803. URL https://doi.org/10.1145/3240765.3240803.

[39] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks, 2016.

[40] 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 *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247. IEEE, 2017.

[41] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2018.

[42] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[43] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016. doi: 10.1109/ISCA.2016.11.

[44] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. doi: 10.1109/MICRO.2016.7783723.

[45] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[46] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[47] Morteza Mousa Pasandi, Mohsen Hajabdollahi, Nader Karimi, and Shadrokh Samavi. Modeling of pruning techniques for deep neural networks simplification, 2020.

[48] Xiaolong Ma, Sheng Lin, Shaokai Ye, Zhezhi He, Linfeng Zhang, Geng Yuan, Sia Huat Tan, Zhengang Li, Deliang Fan, Xuehai Qian, Xue Lin, Kaisheng Ma, and Yanzhi Wang. Non-structured dnn weight pruning – is it beneficial in any platform?, 2020.

[49] 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, 2019.

[50] Chuhan Min, Aosen Wang, Yiran Chen, Wenyao Xu, and Xin Chen. 2pfpce: Two-phase filter pruning based on conditional entropy, 2018.

[51] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks, 2016.

[52] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices, 2018.

[53] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE*

43rd Annual International Symposium on Computer Architecture (ISCA), pages 243–254, 2016. doi: 10.1109/ISCA.2016.30.

[54] 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 *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 15–28. IEEE Press, 2018. ISBN 9781538662403. doi: 10.1109/MICRO.2018.00011. URL https://doi.org/10.1109/MICRO.2018.00011.

[55] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, 2017. doi: 10.1145/3079856.3080254.

[56] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 151–165, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358291. URL https://doi.org/10.1145/3352460.3358291.

[57] M. A. Qureshi and A. Munir. Neuromax: A high throughput, multi-threaded, log-based accelerator for convolutional neural networks. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[58] Mahmood Azhar Qureshi and Arslan Munir. Sparse-pe: A performance-efficient processing engine core for sparse convolutional neural networks. *IEEE Access*, pages 1–1, 2021. doi: 10.1109/ACCESS.2021.3126708.

[59] Mahmood Azhar Qureshi and Arslan Munir. Phantom: A high-performance computational core for sparse convolutional neural networks, 2021.

[60] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations, 2016.

[61] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[62] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks, 2016.

[63] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[64] Tensorflow-MNIST. URL https://www.tensorflow.org/datasets/catalog/mnist.

[65] Tensorflow-CIFAR. URL https://www.tensorflow.org/datasets/catalog/cifar10.

[66] Qualcomm NPE Reference Guide, . URL https://developer.qualcomm.com/sites/default/files/docs/snpe/.

[67] Xilinx DPU Reference Guide, . URL https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/dpu/v3_3/pg338-dpu.pdf.

[68] Samsung NPU Reference, . URL https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-2100/.

[69] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017. doi: 10.1145/3061639.3062228.

[70] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts, 2014.

[71] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In Stefan Wermter, Cornelius Weber, Włodzisław Duch, Timo Honkela, Petia Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa, editors, *Artificial Neural Networks and Machine Learning – ICANN 2014*, pages 281–290, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11179-7.

[72] V. STRASSEN. Gaussian elimination is not optimal. *Numerische Mathematik*, 13: 354–356, 1969. URL http://eudml.org/doc/131927.

[73] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks, 2015.

[74] Shmuel Winograd. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980. doi: 10.1137/1.9781611970364. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611970364.

[75] Murugan Sankaradass, Venkata Jakkula, Srihari Cadambi, Srimat T. Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60, 2009.

[76] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *SIGARCH Comput. Archit. News*, 38(3):247–257, June 2010. ISSN 0163-5964. doi: 10.1145/1816038.1815993. URL https://doi.org/10.1145/1816038.1815993.

[77] 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. ASPLOS '14, page 269–284, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323055. doi: 10.1145/2541940.2541967. URL https://doi.org/10.1145/2541940.2541967.

[78] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014. doi: 10.1109/MICRO.2014.58.

[79] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. *SIGARCH Comput. Archit. News*, 44(3):380–392, June 2016. ISSN 0163-5964. doi: 10.1145/3007787.3001178. URL https://doi.org/10.1145/3007787.3001178.

[80] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. *SIGARCH Comput. Archit. News*, 45(1):751–764, April 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037702. URL https://doi.org/10.1145/3093337.3037702.

[81] Jintao Zhang, Zhuo Wang, and Naveen Verma. A machine-learning classifier implemented in a standard 6t sram array. *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2, 2016.

[82] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017. doi: 10.1109/ISCAS.2017.8050809.

[83] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu

Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit, 2017.

[84] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. 12(1), March 2019. ISSN 1936-7406. doi: 10.1145/3289185. URL https://doi.org/10.1145/3289185.

[85] Jeonghun Lee, Jiayuan He, and Ke Wang. Fpga-based neural network accelerators for millimeter-wave radio-over-fiber systems. *Opt. Express*, 28(9):13384–13400, Apr 2020. doi: 10.1364/OE.391050. URL http://www.osapublishing.org/oe/abstract.cfm?URI=oe-28-9-13384.

[86] Teng Wang, Chao Wang, Xuehai Zhou, and Huaping Chen. An overview of fpga based deep learning accelerators: Challenges and opportunities. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1674–1681, 2019. doi: 10.1109/HPCC/SmartCity/DSS.2019.00229.

[87] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015. doi: 10.1145/2749469.2750389.

[88] 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. *SIGPLAN Not.*, 50(4):369–381, March 2015. ISSN 0362-1340. doi: 10.1145/2775054.2694358. URL https://doi.org/10.1145/2775054.2694358.

[89] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, and Andreas Moshovos. Laconic deep learning inference acceleration. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 304–317, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366694. doi: 10.1145/3307650.3322255. URL https://doi.org/10.1145/3307650.3322255.

[90] Alberto Delmas, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, and Andreas Moshovos. Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how, 2018.

[91] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan. Circnn: Accelerating and compressing deep neural networks using block-circulant weight matrices. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 395–408, 2017.

[92] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen mei Hwu, John Paul Strachan, Kaushik Roy, and Dejan S Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference, 2019.

[93] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016. doi: 10.1109/ISCA.2016.12.

[94] Bing Liu, Danyin Zou, Lei Feng, Shou Feng, Ping Fu, and Junbao Li. An fpga-based

cnn accelerator integrating depthwise separable convolution. *Electronics*, 8(3):281, 2019.

[95] L. Bai, Y. Zhao, and X. Huang. A cnn accelerator on fpga using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(10): 1415–1419, 2018.

[96] Y. Huan, J. Xu, L. Zheng, H. Tenhunen, and Z. Zou. A 3d tiled low power accelerator for convolutional neural network. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

[97] J. Jo, S. Kim, and I. Park. Energy-efficient convolution architecture based on rescheduled dataflow. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(12): 4196–4207, 2018.

[98] Y. Lin and T. S. Chang. Data and hardware efficient design for convolutional neural network. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(5):1642–1651, 2018.

[99] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702, 2020. doi: 10.1109/HPCA47549.2020.00062.

[100] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780, 2020. doi: 10.1109/MICRO50266.2020.00068.

[101] 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 *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 319–333, New York, NY, USA,

2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358275. URL https://doi.org/10.1145/3352460.3358275.

[102] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70, 2020. doi: 10.1109/HPCA47549.2020.00015.

[103] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274, 2020. doi: 10.1109/HPCA47549.2020.00030.

[104] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, 2018. doi: 10.1109/HPCA.2018.00067.

[105] Keras API Reference, . URL https://keras.io/.

[106] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[107] Arash Ardakani, Carlo Condo, and Warren J. Gross. Fast and efficient convolutional accelerator for edge computing. *IEEE Transactions on Computers*, 69(1):138–152, 2020. doi: 10.1109/TC.2019.2941875.

[108] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas. 28(9):1953–1965, September 2020. ISSN 1063-8210. doi: 10.1109/TVLSI.2020.3002779. URL https://doi.org/10.1109/TVLSI.2020.3002779.

[109] Xiaoru Xie, Jun Lin, Zhongfeng Wang, and Jinghe Wei. An efficient and flexible accelerator design for sparse convolutional neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(7):2936–2949, 2021. doi: 10.1109/TCSI.2021.3074300.

[110] 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 *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25, 2019. doi: 10.1109/FCCM.2019.00013.