Hardware Acceleration of Deep Convolutional Neural Networks on FPGA

by

Yufei Ma

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2018 by the
Graduate Supervisory Committee:

Sarma Vrudhula, Co-Chair
Jae-sun Seo, Co-Chair
Yu Cao
Hugh Barnaby

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

The rapid improvement in computation capability has made deep convolutional neural networks (CNNs) a great success in recent years on many computer vision tasks with significantly improved accuracy. During the inference phase, many applications demand low latency processing of one image with strict power consumption requirement, which reduces the efficiency of GPU and other general-purpose platform, bringing opportunities for specific acceleration hardware, e.g. FPGA, by customizing the digital circuit specific for the deep learning algorithm inference. However, deploying CNNs on portable and embedded systems is still challenging due to large data volume, intensive computation, varying algorithm structures, and frequent memory accesses. This dissertation proposes a complete design methodology and framework to accelerate the inference process of various CNN algorithms on FPGA hardware with high performance, efficiency and flexibility.

As convolution contributes most operations in CNNs, the convolution acceleration scheme significantly affects the efficiency and performance of a hardware CNN accelerator. Convolution involves multiply and accumulate (MAC) operations with four levels of loops. Without fully studying the convolution loop optimization before the hardware design phase, the resulting accelerator can hardly exploit the data reuse and manage data movement efficiently. This work overcomes these barriers by quantitatively analyzing and optimizing the design objectives (e.g. memory access) of the CNN accelerator based on multiple design variables. An efficient dataflow and hardware architecture of CNN acceleration are proposed to minimize the data communication while maximizing the resource utilization to achieve high performance.

Although great performance and efficiency can be achieved by customizing the

FPGA hardware for each CNN model, significant efforts and expertise are required leading to long development time, which makes it difficult to catch up with the rapid development of CNN algorithms. In this work, we present an RTL-level CNN compiler that automatically generates customized FPGA hardware for the inference tasks of various CNNs, in order to enable high-level fast prototyping of CNNs from software to FPGA and still keep the benefits of low-level hardware optimization. First, a general-purpose library of RTL modules is developed to model different operations at each layer. The integration and dataflow of physical modules are predefined in the top-level system template and reconfigured during compilation for a given CNN algorithm. The runtime control of layer-by-layer sequential computation is managed by the proposed execution schedule so that even highly irregular and complex network topology, e.g. GoogLeNet and ResNet, can be compiled. The proposed methodology is demonstrated with various CNN algorithms, e.g. NiN, VGG, GoogLeNet and ResNet, on two different standalone FPGAs achieving state-of-the art performance.

Based on the optimized acceleration strategy, there are still a lot of design options, e.g. the degree and dimension of computation parallelism, the size of on-chip buffers, and the external memory bandwidth, which impact the utilization of computation resources and data communication efficiency, and finally affect the performance and energy consumption of the accelerator. The large design space of the accelerator makes it impractical to explore the optimal design choice during the real implementation phase. Therefore, a performance model is proposed in this work to quantitatively estimate the accelerator performance and resource utilization. By this means, the performance bottleneck and design bound can be identified and the optimal design option can be explored early in the design phase.

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor, Dr. Sarma Vrudhula for the opportunity, guidance and consistent encouragement he has offered throughout my doctorate study and research. I would also like to express my great appreciation to Dr. Jae-sun Seo and Dr. Yu Cao, for their great support and insightful suggestions on my academic career. My Ph.D. work would be impossible without their invaluable helps.

I am thankful to Dr. Hugh Barnaby for taking out time reviewing my work and being my Ph.D. committee member.

I am also thankful for frequent helps from my colleagues and friends at ASU. Especially, thank to Naveen Suda, Minkyu Kim, Jinghua Yang, Abinash Mohanty, Zihan Xu, Shihui Yin, Ming Sun and Niranjan Kulkarni.

I owe my deepest gratitude to my family, my lovely wife Lusen Shi, my parents Jungui Ma and Meiyun Dai, for their unconditional love, support, understanding and encouragement to my academic pursuit, through good and hard times, while living on the other side of the world.

This dissertation is based on works supported in part by the NSF I/UCRC Center for Embedded Systems through NSF grants, Intel Labs, Samsung Advanced Institute of Technology, and C-BRIC, one of six centers in JUMP, a SRC program sponsored by DARPA.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

In recent years, Deep Neural Networks (DNNs) have demonstrated a great success on many Artificial Intelligence (AI) applications such as Convolutional Neural Networks (CNNs) on computer vision Krizhevsky *et al.* (2012) and Recurrent Neural Networks (RNNs) on natural language processing Sutskever *et al.* (2014). Instead of using hand-crafted features in traditional machine learning algorithms, DNNs take advantage of the rapidly improved computation capability to learn from much larger training datasets, leading to exceptional recognition accuracy close to or even better than human-level perception. With the significantly improved accuracy and expanded application domains, however, the computation complexity and memory requirement of deep learning algorithms have dramatically increased, which still challenge the state-of-art computing platforms to achieve real-time performance with high energy efficiency.

To realize high throughput, high performance GPUs are often used to accelerate the training and inference tasks of DNNs, as they can take advantage of the thousands of parallel cores, operating at high clock frequencies at GHz level, and achieve hundreds of GB/s memory bandwidth. However, their power consumption is too high (>150W) for power and energy constrained platforms. Furthermore, GPUs are best suited for achieving high throughput when processing large batches of images. However, for applications that require very low latency for processing a single image, as in autonomous drive and surveillance, the completion of detection must be done at the speed of incoming data stream, which degrades GPUs' performance and energy-efficiency substantially.

On the other hand, various deep learning hardware accelerators have been recently proposed based on application specific integrated circuits (ASICs) **?** Shin *et al.* (2017), system on chips (SoCs) Gokhale *et al.* (2014) and field-programmable gate arrays (FPGAs) Zhang *et al.* (2015) targeting at high performance and energy efficiency. FPGAs have gained increasing interests and popularity in particular to accelerate the inference tasks, due to their (1) high degree of reconfigurability, (2) faster development time compared to ASICs to catch up with the rapid evolving of DNNs, (3) good performance, and (4) superior energy efficiency compared to GPUs Aydonat *et al.* (2017) Wei *et al.* (2017) Ma *et al.* (2017a). The high performance and efficiency of an FPGA can be realized by synthesizing a circuit that is customized for a specific computation to directly process billions of operations with the customized memory systems. For instance, hundreds to thousands of digital signal processing (DSP) blocks on modern FPGAs support the core DNN operations, e.g. multiplication and addition, with high parallelism. Dedicated data buffers between external off-chip memory and on-chip processing engines (PEs) can be designed to realize the preferred dataflow by configuring tens of MByte on-chip block random access memories (BRAM) on the FPGA chip.

The goal of this dissertation is to deploy DNN inference tasks on FPGA-based hardware accelerators with high performance, efficiency and flexibility, especially for CNNs on image classification and object detection tasks.

1.1 Motivation: Challenges and Opportunities

1.1.1 Performance and Efficiency: Loop Optimization

The state-of-the-art CNNs require a large number (> 1 billion) of computationally intensive task (e.g. matrix multiplications on large numbers), involving a very large number of weights (> 50 million) Simonyan and Zisserman (2014) He *et al.* (2016a).

Deep CNN algorithms have tens to hundreds of layers, with significant differences between layers in terms of sizes and configurations. The limited computational resources and storage capacity on FPGA make the task of optimal mapping of CNNs (e.g. minimizing latency subject to energy constraints or vice versa) a complex and multi-dimensional optimization problem. The high cost of off-chip communication is another major impediment to achieving higher performance and lower energy. In fact, the energy cost associated with the large amount of data movements and memory accesses often exceeds the energy consumption of the computations Chen *et al.* (2016) Zhang *et al.* (2016b). For these reasons, energy-efficient hardware acceleration of CNNs on a FPGA requires simultaneous maximization of resource utilization and data reuse, and minimization of data communication.

More than 90% of the operations in a CNN algorithm involve convolutions Krizhevsky *et al.* (2012) Simonyan and Zisserman (2014) He *et al.* (2016a). Therefore, it stands to reason that acceleration schemes should focus on the management of parallel computations and the organization of data storage and access across multiple levels of memories, e.g. off-chip dynamic random-access memory (DRAM), on-chip memory and local registers. In CNNs, convolutions are performed by four levels of loops that slide along both kernel and feature maps. This gives rise to a large design space consisting of various choices for implementing parallelism, sequencing of computations, and partitioning the large data set into smaller chunks to fit into on-chip memory. These problems can be handled by the existing loop optimization techniques Bacon *et al.* (1994) Zhang *et al.* (2015), such as loop unrolling, tiling and interchange. Although some CNN accelerators have adopted these techniques Zhang *et al.* (2015) Suda *et al.* (2016) Guo *et al.* (2018) Motamedi *et al.* (2016), the impact of these techniques on design efficiency and performance has not been systematically and sufficiently studied. Instead, most prior works only explore the design space after hardware architecture

or parallelism scheme has been determined, and optimize their implementation by tuning the design variables only within their architecture. Without fully studying the loop operations of convolutions, it is difficult to efficiently customize the dataflow and architecture for high-throughput CNN implementations.

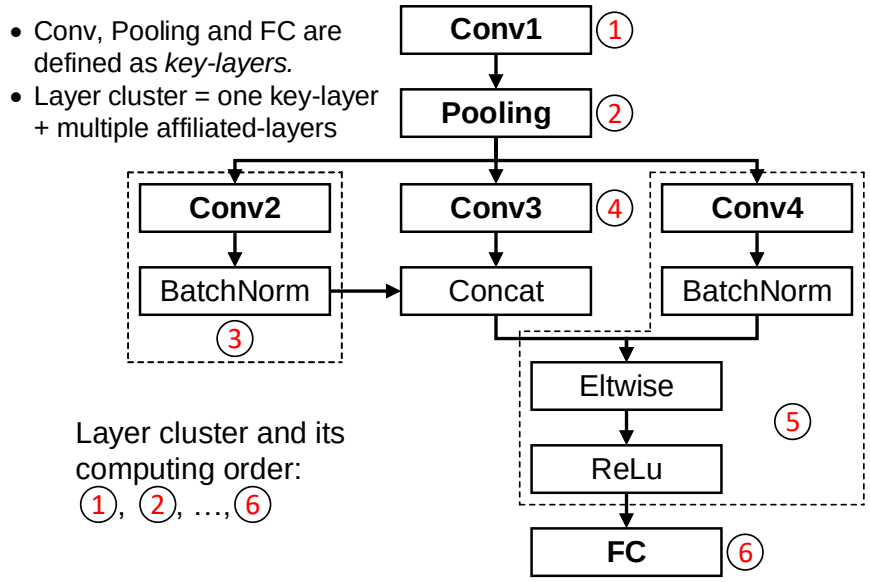

Figure 1.1: Example of directed acyclic graph (DAG) form layer connections in recent CNN algorithms with multiple parallel branches involving different types of layers.

1.1.2 Flexibility: Automated CNN Mapping

To pursue higher accuracy and enable various intelligent applications, CNNs with greater depth, new types of layers and more complex networks are being proposed. For example, the deep residual networks (ResNets) He *et al.* (2016a) Szegedy *et al.* (2017) He *et al.* (2016b) can achieve substantially greater accuracy at the cost of having more than 1,000 convolution layers (Conv) with widely differing dimensions and kernel sizes, as well as many other various types of layers. Unlike earlier CNNs such as AlexNet Krizhevsky *et al.* (2012), NiN Lin *et al.* (2013) and VGG Simonyan

and Zisserman (2014), in which the layers are strung out in a sequence, the layers in the more recent CNN algorithms such as ResNet, GoogLeNet Szegedy *et al.* (2015) and Inception Szegedy *et al.* (2017), form a directed acyclic graph (DAG), as shown in Figure 1.1. They have multiple parallel branches and include feedforward connections between non-adjacent layers.

All these trends, which have increased the complexity of CNN algorithms, have made it more difficult to design a general-purpose CNN hardware accelerator to efficiently map a diverse range of CNN algorithms. Previous FPGA implementations based on high-level synthesis (HLS) tools Zhang *et al.* (2015) Suda *et al.* (2016) Venieris and Bouganis (2016) have achieved good flexibility, easy programmability and short design time, but their hardware and memory utilization is inefficient and may not allow exploitation of low-level hardware structures to achieve higher performance and throughput. Another approach is to undertake custom hardware design at the register-transfer level (RTL) for each specific CNN with fine-grained hardware level optimization. Experience has shown that such an approach requires detailed knowledge of both the CNN algorithm and the FPGA system architecture, and many months of design effort involving numerous design iterations, which makes it difficult to catch up with the rapidly evolving CNN algorithms and diverse emerging applications.

On the software side, machine learning researchers have been able to efficiently develop deep learning algorithms through flexible frameworks, e.g. Caffe Jia *et al.* (2014), which run on CPUs or GPUs. These software frameworks have simple expression and modularity, which allow researchers to efficiently explore various algorithms and network structures. Unfortunately, the hardware design community does not yet have such a flexible modular framework for hardware implementation of CNN and other deep learning algorithms, inevitably spreading out the hardware research efforts

instead of coalescing them.

In this context, there is a timely need to reform the strategy to automatically map CNN algorithms onto physical hardware, and to support modular and scalable hardware customization without sacrificing design performance and flexibility.

1.1.3 Design Space Exploration: Performance Modeling

With the intervals of computation and off-chip communication overlapped using dual buffering (or ping-pong buffering) technique, the performance of the CNN accelerator will be limited by either the computation delay or the DRAM transfer delay, and the actual bound will be determined by the values of the associated design parameters, as described by the *roofline model* in Zhang *et al.* (2015) Zhang *et al.* (2016a). The computation delay is determined by the number of parallel processing engines (PEs), their utilization, and the operating frequency. The DRAM transfer latency is mainly affected by the external memory bandwidth and the number of DRAM accesses, and the latter is strongly affected by the size of the on-chip buffers. With regard to the energy efficiency (i.e. performance per watt), the main components that determine the dynamic power consumption are the computation logic and the memory traffic, the latter requiring efficient data movement and high data reuse. All these considerations show that there are numerous design parameters that determine the performance and energy efficiency of a CNN accelerator, making it impractical to find their optimal values during the implementation phase, as the synthesis of one FPGA design may take several hours. Robust and parametric models become a necessity for efficient design space exploration and selection of the optimal values of the design parameters. The architectural design space must be numerically characterized by design variables to control the accelerator performance and efficiency. For instance, loop optimization techniques Zhang *et al.* (2015) Ma *et al.* (2018a), such

as loop unrolling and tiling, are employed to customize the acceleration strategy of parallel computation and data communication for convolution loops, whose variables in turn affect the resource utilization and memory access.

1.1.4 Efficiency: Algorithm-Hardware Co-Design

The rapid improvement in computation capability has made CNN algorithms a great success in recent years on image classification tasks, which has also prospered the development of objection detection algorithms with significantly improved accuracy. The Single Shot Detector (SSD) Liu *et al.* (2016) algorithm uses VGG-16 CNN model as the base feature extractor to predict the locations of bounding boxes and the classification probability of objects, and then uses additional convolution layers at the end to predict objects from multi-scale feature maps. However, it is a great challenge to directly implement SSD on mobile hardware, e.g. embedded systems and edge devices, to achieve real-time detection with high energy efficiency, because of (1) the large volume of data and operations, (2) the use of complex nonlinear functions, and (3) the highly varying layer sizes and configurations. Directly implementing the original SSD algorithm onto an FPGA may cause low utilization of the available computation resources and consequently result in low performance and efficiency. Therefore, it is essential to tailor the original software-orientated algorithm for efficient hardware implementation.

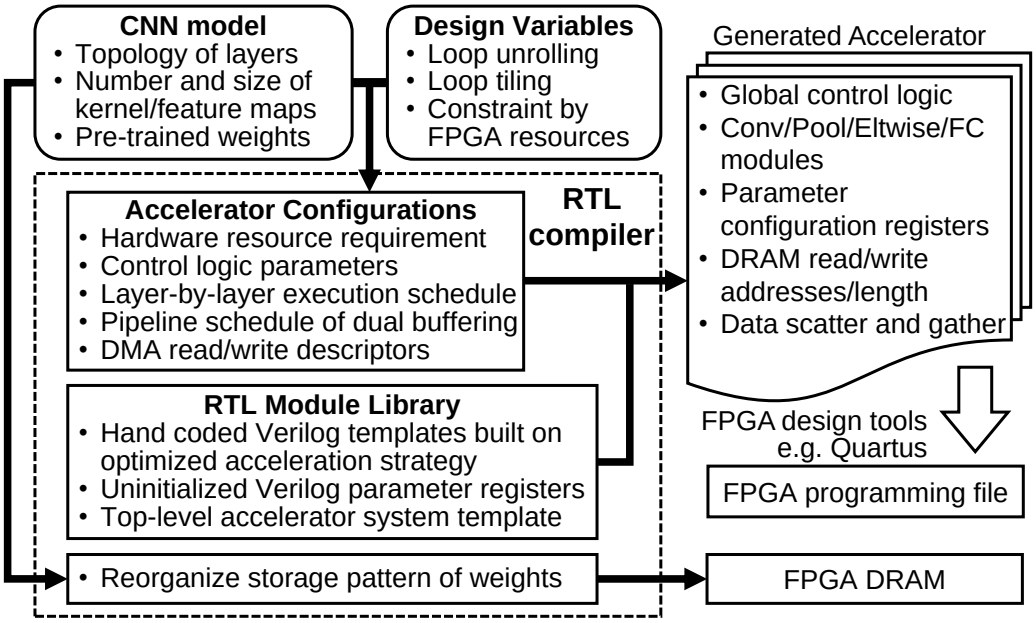

Figure 1.2: The overall compilation flow of the proposed CNN RTL compiler: the hardware resource usage and the execution schedule of the FPGA accelerator are configured for the given CNN model; the RTL module library defines the computation pattern and dataflow of different types of layers with parameterized Verilog templates.

1.2 Contribution and Dissertation Outline

In this dissertation, a complete framework is proposed to automatically map the inference process of various deep CNN algorithms onto the high-performance FPGA accelerator, where an efficient dataflow and hardware architecture are designed based on the convolution loop optimization and the design space is explored through the proposed performance model. The main contributions of this dissertation include the following:

- Chapter 3: We provide an in-depth analysis of the convolution loop optimization and use corresponding design variables to numerically characterize the ac-

celeration scheme. An efficient convolution acceleration strategy and dataflow is proposed aimed at minimizing data communication and memory access. A data router is designed to handle different settings for convolution sliding operations, e.g. strides and zero paddings, especially for highly irregular CNNs. A corresponding hardware architecture is designed that fully utilizes the computing resources for high performance and efficiency, which is uniform and reusable for all the layers.

- Chapter 4: A user-friendly and high-level compiler is proposed as in Figure 1.2 to automatically configures the FPGA-based accelerator for various large-scale CNN algorithms with user-specified hardware resource constraints, such as computing parallelism and buffer usage, targeting FPGA platforms with different amount of hardware resources. It exploits the reconfigurability of FPGAs and the fine-grain optimization that is possible with an RTL description. An RTL module library is designed to accommodate different types of layers with manually coded Verilog templates, which has been designed to allow incorporation of new layers or operations for future deep learning algorithms. The flexibility of the proposed compilation methodology is validated by implementing the inference task of both conventional CNNs: NiN and VGG-16; and the more complex DAG networks: GoogLeNet and ResNets with 50 and 152 convolution layers, respectively.

- Chapter 5: A high-level performance model is proposed to estimate the accelerator throughput, on-chip buffer size and the number of external and on-chip memory accesses. The accelerator design objectives and resource costs are formulated using the hardware design variables of loop unrolling and tiling. Design space exploration is performed to identify the performance bottleneck and ob-

tain the optimal design option. The performance model is validated across a variety of CNN algorithms comparing with the on-board test results on two different FPGAs.

- Chapter 6: The algorithm-hardware co-design is proposed to tailor the CNN-based SSD object detection algorithm for efficient hardware realization, and the low precision fixed-point data with dynamic quantization is employed for inference to reduce the resource requirements of logic and memory at the cost of marginal accuracy degradation.

The outline of the dissertation is organized as follows. Chapter 2 overviews the operations and structures of the recent representative CNN algorithms as well as the related works on FPGA-based CNN acceleration. Chapter 3 quantitatively analyzes the convolution loop optimization strategy for high performance and efficient accelerator dataflow and hardware architecture. Chapter 4 presents the RTL compiler that enables fast and automatic mapping of various deep CNN algorithms from software deep learning frameworks, e.g. Caffe, onto FPGA hardware. Chapter 5 describes the proposed high-level performance model to estimate the throughput and resource utilization of the CNN accelerators allowing design space exploration at early design stage. Chapter 6 customizes the SSD object detection algorithm to benefit its hardware implementation with low data precision.

Chapter 2

BACKGROUND

2.1 Overview of CNN Operations

Convolutional neural networks as illustrated in Figure 2.1 typically incorporate multiple layers of convolution, pooling/subsampling, and normalization that extract low-level to high-level features from the input during the feed-forward inference process. These features can be categorized into a finite number of output classes by the final classification layers such as the multi-layer perceptron or fully-connected layers.

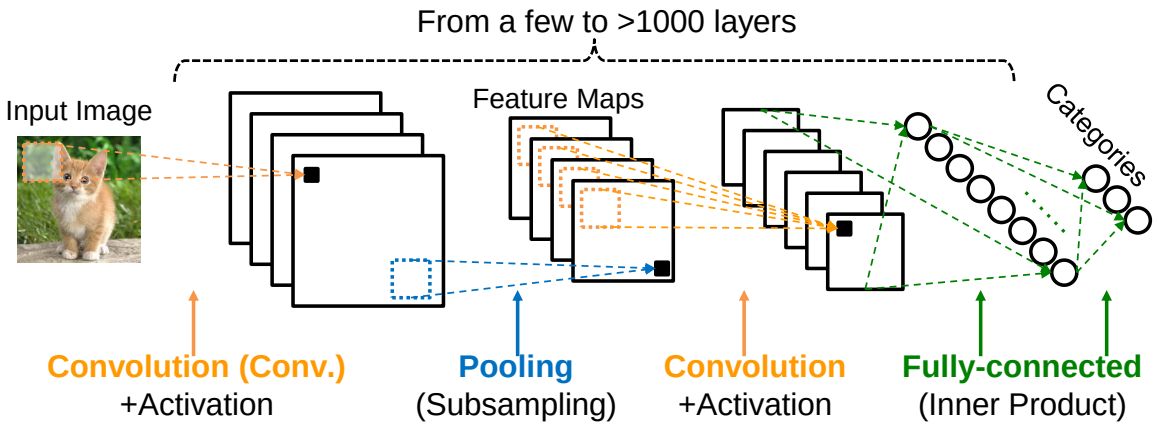

Figure 2.1: Convolutional neural networks incorporate multiple layers to extract features for classification during feed-forward inference process.

Convolution (Conv) is the main operation in CNN algorithms, which involves three-dimensional multiply and accumulate (MAC) operations of input pixels (features, neurons, or activations) and kernel weights. As shown in Figure 2.2, multiple dimensions are used to describe the sizes of the feature and kernel maps of each convolution layer for a given CNN. The width and height of one kernel (or filter) window

is described by (Nkx, Nky). (Nix, Niy) and (Nox, Noy) define the width and height of one input and output feature map, respectively. Nif and Nof denote the number of input and output feature maps (or channels), respectively. The detailed convolution operation is depicted as below:

$$pixel_L(no; x, y) =$$

$$\sum_{ni=1}^{\text{Nif}} \sum_{ky=1}^{\text{Nky}} \sum_{kx=1}^{\text{Nkx}} pixel_{L-1}(ni; \text{S} \times x + kx, \text{S} \times y + ky) \times weight(ni, no; kx, ky) \quad (2.1)$$

$$+ bias(no);$$

where S is the sliding stride, $x \in \{1, 2, \ldots, \text{Nox}\}$, $y \in \{1, 2, \ldots, \text{Noy}\}$, $no \in \{1, 2, \ldots, \text{Nof}\}$, $L \in \{1, 2, \ldots, \#CONVs\}$, and $\#CONVs$ is the number of convolution layers.

Convolution is implemented by four levels of loops as illustrated in Figure 2.2 and shown in the pseudo codes in Figure 2.3. Loop-1 computes the MAC of pixels

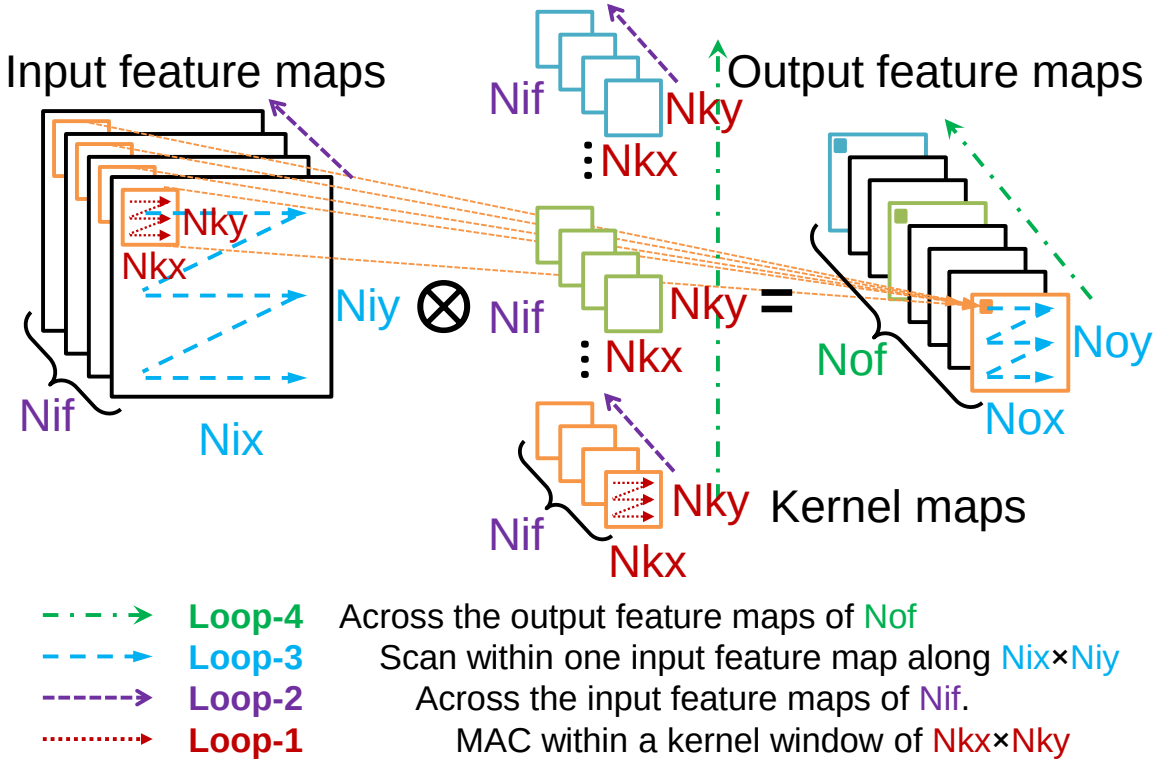

Figure 2.2: Four levels of convolution loops and their dimensions.

and weights within a kernel window of dimension Nkx × Nky. Loop-2 accumulates the sum of products of MAC across different input feature maps of dimension of Nif. After finishing Loop-1 and Loop-2, we can obtain one final output pixel by adding the bias. Loop-3 slides the kernel window within an input feature map of dimension Nix × Niy. Loop-4 generates different output feature maps with dimension of Nof.

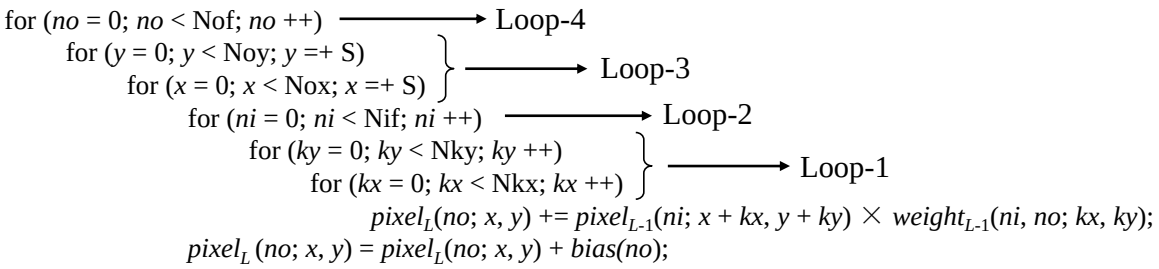

Figure 2.3: Four levels of convolution loops, where L denotes the index of convolution layer and S denotes the sliding stride

The relationship of input and output feature maps is described by Equation 2.2,

$$\begin{aligned} \text{Nix} &= (\text{Nox} - 1)\text{S} + \text{Nkx}, \\ \text{Niy} &= (\text{Noy} - 1)\text{S} + \text{Nky}. \end{aligned} \tag{2.2}$$

where S is the stride of the sliding step. To control the spatial size of output feature maps (Nox and Noy), sometimes zeros are padded around the border of input feature maps, and the size of zero padding is included in Nix and Niy.

Pooling (Pool) or subsampling layer is commonly employed after convolution to reduce the dimensionality of the input features while preserving key information. This is done by replacing input neurons inside the kernel window (e.g. 2×2 or 3×3) with their maximum or average value as shown in Figure 2.1, depending on the model definition.

Normalization or local response normalization (LRN) layer implements a form of lateral inhibition Krizhevsky *et al.* (2012) by normalizing the neuron value by factors

of α and β depending on its K neighboring features at the same (x, y) location, as shown in Equation 2.3:

$$pixel_L(no; x, y) = \frac{pixel_{L-1}(no; x, y)}{\left(1 + \frac{\alpha}{K} \sum_{ni=no-K/2}^{no+K/2} pixel_{L-1}^2(ni; x, y)\right)^{\beta}} \tag{2.3}$$

Batch normalization followed by scale (Bnorm) has been commonly used in recent CNN models He *et al.* (2016a) Szegedy *et al.* (2017), which is applied after each training batch and normalizes the distribution of the outputs to be uniform across different layers, enabling high learning rate and fast training convergence. Their operations are depicted in (2.4) and (2.5):

$$y = \frac{x - bn0}{\sqrt{bn1}}, \tag{2.4}$$

$$z = sc0 \times y + sc1. \tag{2.5}$$

During the inference process, $bn0$, $bn1$, $sc0$ and $sc1$ of Bnorm are all constants for each output feature map along Nof, which provides us the opportunity to simplify its implementation on hardware.

Element Wise (Eltwise) layer performs element-wise addition or multiplication of two input layers as shown in Figure 1.1, where the input layers must have the same size and shape of input feature maps (Nix $\times$ Niy $\times$ Nif), and ResNet CNNs use addition in the Eltwise layers.

Concat layer is used to concatenate the outputs of multiple layers together as shown in Figure 1.1. If the concatenation is along multiple channels and all the input layers must have the same feature map sizes (Nix $\times$ Niy), which is the case for GoogLeNet Szegedy *et al.* (2015) and Inception Szegedy *et al.* (2017).

Fully-connected (FC) or inner-product layers as shown in Figure 2.1 are final classification layers where the output features are computed as matrix-vector multi-

plications of the fully-connected weight matrix and the input features. As the kernel weights of FC are not spatially reused by different input features, the data volume of FC weights are usually larger than that of convolution layers, even though FC layers have much less operations, which makes FC layers to be memory intensive.

Activation layers perform non-linear activation functions such as tanh, sigmoid or Rectified Linear Unit (ReLU) after convolution and fully-connected layers. ReLU, defined as $y = max(x, 0)$, has become a popular choice for the activation function, due to faster convergence in training Krizhevsky *et al.* (2012) as well as compact hardware implementation.

2.2 CNN Structures

Figure 2.4 shows the structures of several representative CNN algorithms in recent years for image classification task.

AlexNet Krizhevsky *et al.* (2012) significantly increases the classification accuracy compared with traditional machine learning methods based on hand crafted features and is the first CNN model winning the ILSVRC contest Russakovsky *et al.* (2015) in 2012. AlexNet is comprised of 5 convolution layers and 3 FC layers, has

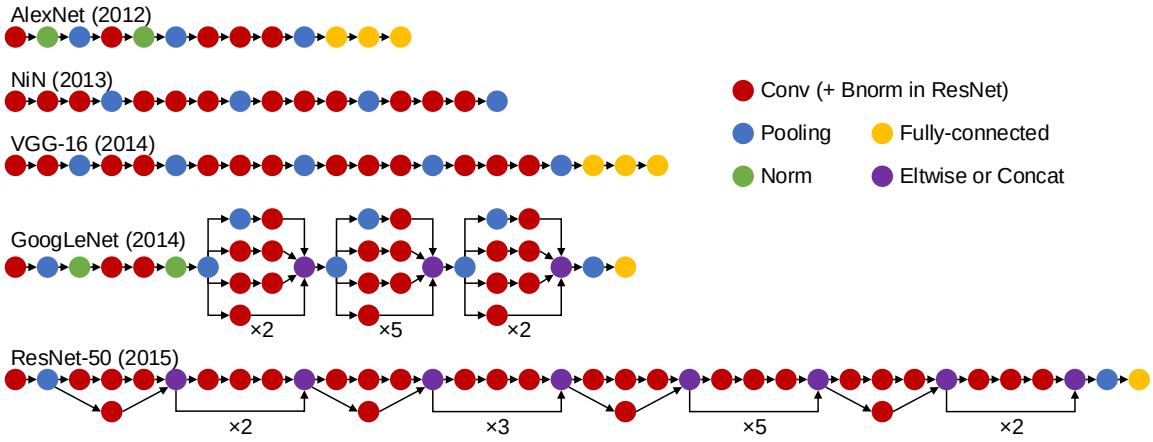

Figure 2.4: The structures of the representative CNN algorithms in recent years.

61 million kernel weights (parameters), and requires about 1.4 billion operations of multiplication and addition. The top-5 accuracy of AlexNet Caffe model on ImageNet Russakovsky *et al.* (2015) validation dataset is 80.2%. There are three different kernel sizes (Nkx × Nky) in AlexNet convolution layers: 11×11, 5×5, and 3×3.

NiN Lin *et al.* (2013) (network-in-network) replaces the FC layers with global average pooling layer at the end to save a large number of weights, while it can still reach the same accuracy level as AlexNet on ImageNet. NiN consists of small and stacked convolution layers (e.g. cccp) with kernel size 1×1 for better local feature abstraction. Like AlexNet, NiN also has kernel sizes of: 11×11, 5×5, and 3×3.

VGG Simonyan and Zisserman (2014) has a very regular structure, with only 3×3 convolution layers and 2×2 max pooling layers. However, the three FC layers need about 123 million weights and the convolution layers in VGG-16 also require about 15 million weights. The overall VGG-16 needs about 31 billion operations to process one image in the feed forward process. The top-5 accuracy of VGG-16 Caffe model on ImageNet validation dataset is 88.7%.

GoogLeNet Szegedy *et al.* (2015) reaches the same accuracy-level as VGG-16 with 89.0% top-5 accuracy on ImageNet, while demanding only 6.1 million weights and 3.2 billion operations, which is achieved by a more complex architecture with inception module as shown in Figure 2.4. One inception module consists of six convolution layers with kernel sizes 1×1, 3×3 and 5×5 and one max pooling layer in four parallel branches. To save space, the inception modules with the same structure are not exhibited and instead their repeat times are shown, e.g. $\times 2$, in Figure 2.4.

ResNet He *et al.* (2016a) solves the problem of training very deep CNNs with hundreds of layers and can still obtain compelling accuracy with top-5 accuracy higher than 92.2% on ImageNet. Very deep networks are hard to train because the gradients are vanishing after back-propagated through too many layers. ResNet incorporates

skip connections ("shortcut") between two non-adjacent layers so that the gradients can flow back to the earlier layers more easily during training. ResNet-50 has 53 convolution layers, one max pooling and one FC layer, where each convolution layer is followed by one Bnorm layer. The kernel sizes used in ResNet are 7×7, 3×3, and 1×1. The number of weights in ResNet-50 is about 26 million and the number of operations is approximately 7.7 billion. The repeated residual sub-structures are represented by their repeat times, e.g. $\times 2$, in Figure 2.4.

2.3 FPGA Hardware System

From the aforementioned discussion, recently reported CNN algorithms involve a large amount of data and weights. For them, the block memory (BRAM) on the FPGA chip, which is normally smaller than 8 MByte, is insufficient to store all the data, requiring gigabytes of external off-chip memory (DRAM). Therefore, a typical CNN accelerator consists of three levels of hierarchy: 1) external memory, 2) on-chip

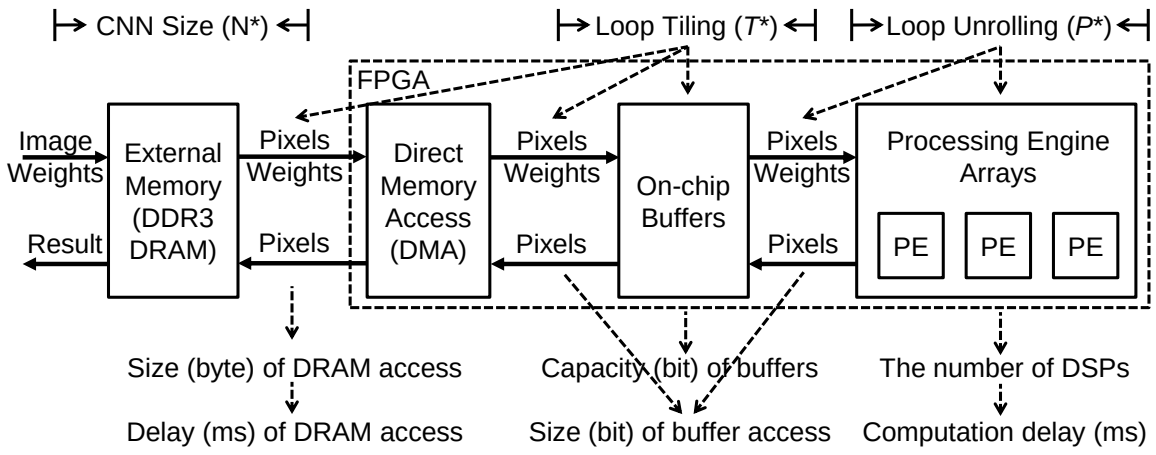

Figure 2.5: A general CNN hardware accelerator with three levels of hierarchy, where the loop design variables determine the key accelerator metrics, e.g. delay, resource usage and memory access.

buffers, and 3) registers and processing engines (PEs) as shown in Figure 2.5. The basic flow is to fetch data from external memory to on-chip buffers by the direct memory access (DMA) engine, and then feed them into registers and PEs. After the PE computation completes, the results are transferred back to on-chip buffers and to the external memory if necessary, which will be used as input to the subsequent layer.

2.4 Related Work

The great success of CNNs on computer vision applications has dramatically prospered the research and development of FPGA based CNN inference accelerators, and these related works are briefly summarized in this section. More comprehensive comparisons and discussions will be presented in the following chapters with detailed experimental results.

Recent FPGA works on hardware acceleration of CNN inference have demonstrated throughput improvement from 62 GOPS Zhang *et al.* (2015) to 1,382 GOPS Aydonat *et al.* (2017), while also significantly improving the energy efficiency when compared to GPU based implementations as presented in Zhang *et al.* (2016a) Guan *et al.* (2017). Fast algorithm, e.g. Winograd transform and Fourier transform, has been applied on FPGA to further boost the computation speed by reducing the multiplication operations in Aydonat *et al.* (2017) and Zeng *et al.* (2018). Efforts have been made to reduce the gap between the rapid development of deep learning algorithms and the long design time for FPGA hardware implementation. Several FPGA-based frameworks or compilers have been proposed to automatically map different CNN algorithms, e.g. AlexNet, VGG and ResNet, onto FPGA hardware Venieris and Bouganis (2016) Zhang *et al.* (2016a) Ma *et al.* (2017a) Ma *et al.* (2018b) Guan *et al.* (2017) Zeng *et al.* (2018). To speed up the FPGA implementation, approaches based

on high-level synthesis (HLS) techniques and the use of OpenCL, are becoming increasingly popular, due to their easy programmablity and reduced design time Zhang *et al.* (2015) Suda *et al.* (2016) Wei *et al.* (2017) Aydonat *et al.* (2017). However, the conventional design methodology that relies on manual register-transfer level (RTL) allows much finer level of optimization of the hardware, resulting in higher performance and energy efficiency Qiu *et al.* (2016) Ma *et al.* (2017b).

Chapter 3

OPTIMIZE CONVOLUTION LOOP OPERATIONS ON FPGA

3.1 Acceleration of Convolution Loops

3.1.1 Convolution Loops

As the main operation in CNN algorithms, convoluiton involves three-dimensional multiply and accumulate (MAC) operations of input feature maps and kernel weights as illustrated in Figure 2.2. To efficiently map and perform the convolution loops, three loop optimization techniques Bacon *et al.* (1994) Zhang *et al.* (2015), namely, loop unrolling, loop tiling and loop interchange, are employed to customize the computation and communication patterns of the accelerator with three levels of memory hierarchy.

Loop unrolling determines the parallelism scheme of certain convolution loops, and thus the required size of registers and PEs. Loop tiling determines the required capacity of on-chip buffers. It divides the loops into multiple blocks, and the data of the execution blocks are read from external memory and stored in on-chip buffers. Loop interchange determines the computation order of the four loops and thus affects the dataflow between the adjacent levels of memory hierarchy.

3.1.2 Loop Optimization and Design Variables

In this section, we describe how the design of a CNN is parameterized. These parameters are numerical quantities that determine the extent to which loop optimization can be done within the limitations of the hardware. They also determine the size of the functional units that must be synthesized. Subsequently, the quantities

Table 3.1: Convolution Loop Dimensions and Hardware Design Variables

	Kernel Window (width/height)	Input Feature Map (width/height)	Output Feature Map (width/height)	# of Input Feature Maps	# of Output Feature Maps
Convolution Loops	Loop-1	Loop-3	Loop-3	Loop-2	Loop-4
Convolution Dimensions (N*)	Nkx, Nky	Nix, Niy	Nox, Noy	Nif	Nof
Loop Tiling (T*)	Tkx, Tky	Tix, Tiy	Tox, Toy	Tif	Tof
Loop Unrolling (P*)	Pkx, Pky	Pix, Piy	Pox, Poy	Pif	Pof

to be optimize, e.g. latency or memory access, will be expressed as functions of these parameters, providing a means to explore the design tradeoffs.

As shown in Figure 2.2, multiple dimensions are used to describe the sizes of the feature and kernel maps of each convolution layer for a given CNN. The hardware design variables of loop unrolling and loop tiling will determine the acceleration factor and hardware footprint. All dimensions and variables used in this work are listed in Table 3.1.

The loop unrolling design variables are (Pkx, Pky), Pif, (Pox, Poy), and Pof, which denote the number of parallel computations along different feature or kernel map dimensions. The loop tiling design variables are (Tkx, Tky), Tif, (Tox, Toy), and Tof, which represent the portion of data of the four loops stored in on-chip buffers. The constraints of these dimension and variables are given by $1 \leqslant P* \leqslant T* \leqslant$ N*, where N*, $T*$ and $P*$ denote any dimension or variable that has a prefix of capital N, T and P, respectively. For instance, $1 \leqslant Pkx \leqslant Tkx \leqslant$ Nkx. By default, $P*$, $T*$ and N* are applied to all convolution layers.

Similar to Equation 2.2, the relationship of input and output variables is constraint by Equation 3.1 and 3.2, where S is the stride of the sliding window and the zero padding size is included in Tix and Tiy.

$$
\begin{aligned}
Tix &= (Tox - 1)\mathrm{S} + \mathrm{Nkx}, \\
Tiy &= (Toy - 1)\mathrm{S} + \mathrm{Nky}.
\end{aligned}
\tag{3.1}
$$

$$
\begin{aligned}
Pix &= Pox, \\
Piy &= Poy.
\end{aligned}
\tag{3.2}
$$

The dimensions or variables (N*, T*, P*) determine the configurations of the three levels of memory hierarchy from the external memory to on-chip buffers to registers and PEs.

Loop Unrolling

As illustrated in Figure 3.1, 3.2, 3.3, and 3.4, unrolling different convolution loops leads to different parallelization of computations, which affects the optimal PE architecture with respect to data reuse opportunities and memory access patterns.

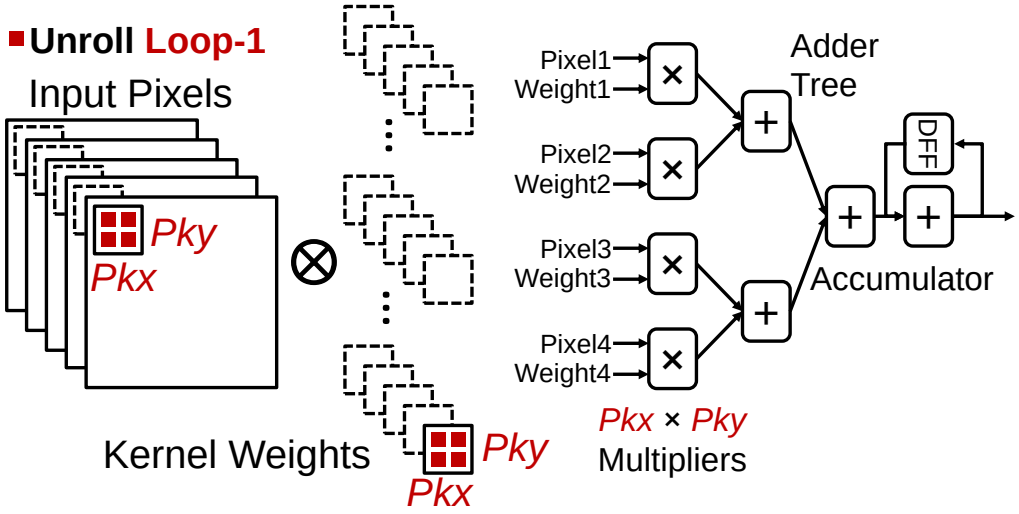

Figure 3.1: Unroll Loop-1 and its corresponding computing architecture.

Loop-1 unrolling (Figure 3.1): in this case, the inner product of $Pkx \times Pky$ pixels (or activations) and weights from different (x, y) locations in the same feature

and kernel map are computed every cycle. This inner product requires an adder tree with fan-in of $Pkx \times Pky$ to sum the $Pkx \times Pky$ parallel multiplication results, and an accumulator to add the adder tree output with the previous partial sum.

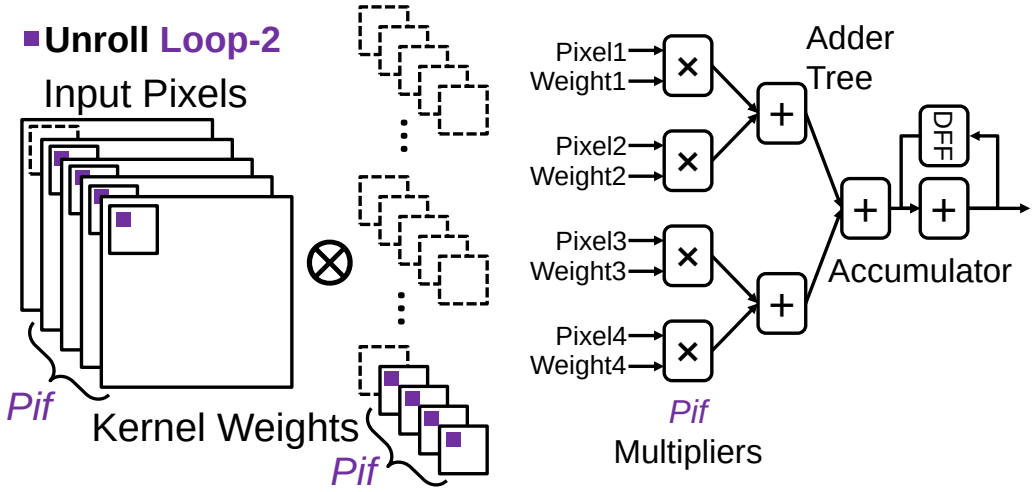

Figure 3.2: Unroll Loop-2 and its corresponding computing architecture.

Loop-2 unrolling (Figure 3.2): in every cycle, Pif number of pixels/weights from Pif different feature/kernel maps at the same (x, y) location are required to compute the inner product. The inner product operation results in the same computing structure as in unrolling Loop-1, but with a different adder tree fan-in of Pif.

Loop-3 unrolling (Figure 3.3): in every cycle, $Pix \times Piy$ number of pixels from different (x, y) locations in the same feature map are multiplied with the identical weight. Hence, this weight can be reused $Pix \times Piy$ times. Since the $Pix \times Piy$ parallel multiplication contributes to independent $Pix \times Piy$ output pixels, $Pix \times Piy$ accumulators are used to serially accumulate the multiplier outputs and no adder tree is needed.

Loop-4 unrolling (Figure 3.4): in every cycle, one pixel is multiplied by Pof

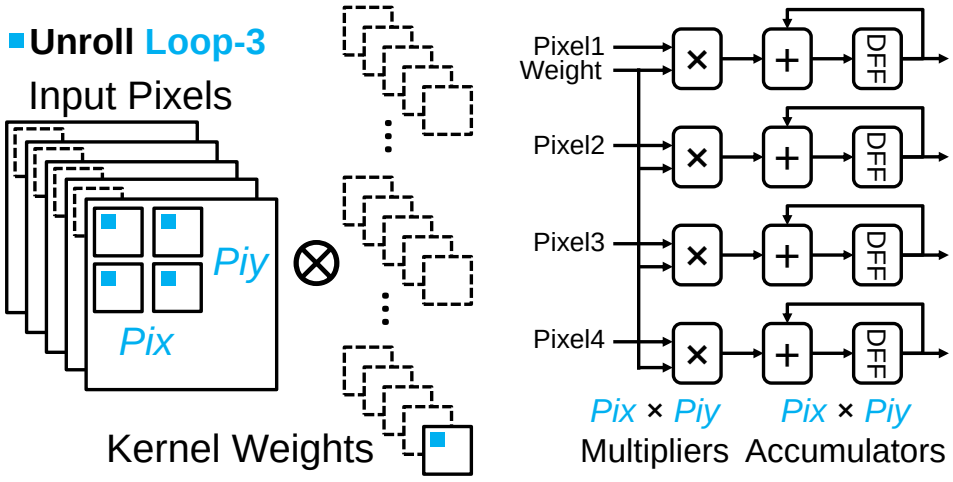

Figure 3.3: Unroll Loop-3 and its corresponding computing architecture.

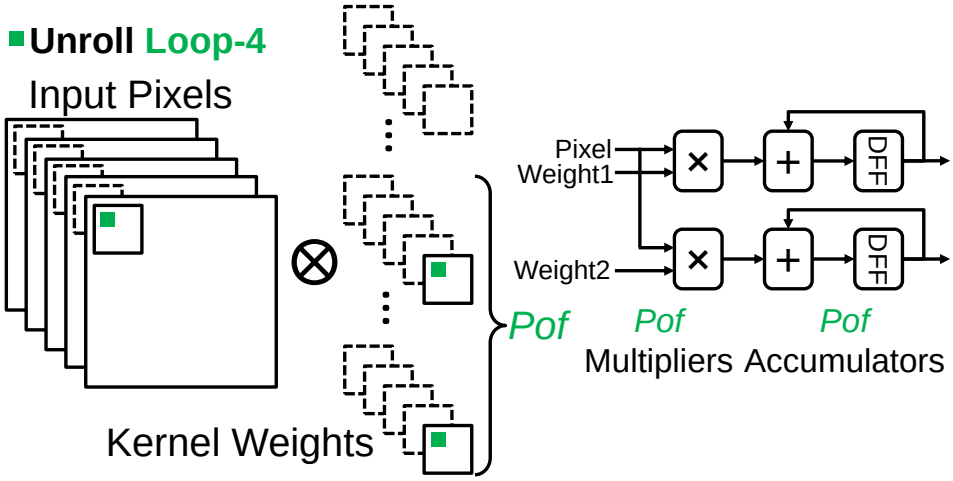

Figure 3.4: Unroll Loop-4 and its corresponding computing architecture.

weights at the same (x, y) location but from Pof different kernel maps, and this pixel is reused Pof times. The computing structure is identical to unrolling Loop-3 using Pof multipliers and accumulators without an adder tree.

The unrolling variable values of the four convolution loops collectively determine the total number of parallel MAC operations as well as the number of required multipliers (Pm):

$$Pm = Pkx \times Pky \times Pif \times Pix \times Piy \times Pof. \tag{3.3}$$

Loop Tiling

On-chip memory of FPGAs is not always large enough to store the entire data of deep CNN algorithms. Therefore, it is reasonable to use denser external DRAMs to store the weights and the intermediate pixel results of all layers.

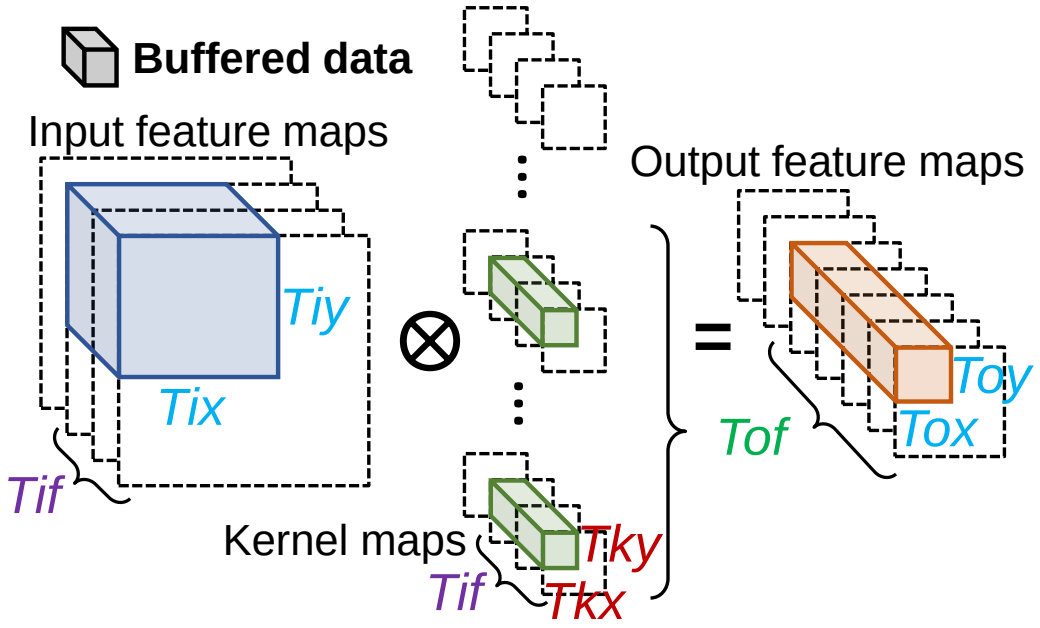

Figure 3.5: Loop tiling determines the size of data stored in on-chip buffers.

Loop tiling is used to divide the entire data into multiple blocks, which can be accommodated in the on-chip buffers, as illustrated in Figure 3.5. With proper assignments of the loop tiling size, the locality of data can be increased to reduce the number of DRAM accesses, which incurs long latency and high-power consumption. The loop tiling sets the lower bound on the required on-chip buffer size. The required size of input pixel buffer is $Tix \times Tiy \times Tif \times (pixel\_datawidth)$. The size of weight

buffer is $Tkx \times Tky \times Tif \times Tof \times (weight\_datawidth)$. The size of output pixel buffer is $Tox \times Toy \times Tof \times (pixel\_datawidth)$.

Loop Interchange

Loop interchange determines the order of the sequential computation of the four convolution loops. There are two kinds of loop interchange, namely intra-tile and inter-tile loop orders. Intra-tile loop order determines the pattern of data movements from on-chip buffer to PEs. Inter-tile loop order determines the data movement from external memory to on-chip buffer.

3.2 Analysis on Design Objectives of CNN Accelerator

In this section, we provide a quantitative analysis of the impact of loop design variables ($P*$ and $T*$) on the following design objectives that our CNN accelerator aims to minimize:

1. *Computing latency* depends strongly on the loop unrolling factors $P*$, but can also be affected by inefficient utilization of PEs and external memory transactions.

2. The requirement of *partial sum storage* is mainly determined by the computation order of loops. The earlier the final pixel output can be obtained, the fewer the number of partial sums will need to be stored.

3. To reduce *the number of on-chip buffer accesses*, the pixels and weights fetched from the on-chip buffer need to be reused as much as possible, which is largely determined by loop unrolling strategy.

4. *The number of external memory accesses* primarily relies on the size of on-chip buffers, which is determined by the loop tiling variables $T*$.

3.2.1 Computing Latency

The number of multiplication operations per layer (Nm) is

$$\text{Nm} = \text{Nif} \times \text{Nkx} \times \text{Nky} \times \text{Nof} \times \text{Nox} \times \text{Noy}. \tag{3.4}$$

Ideally, the number of computing cycles per layer should be Nm/Pm, where Pm is the number of multipliers. However, for different loop unrolling and tiling sizes, the multipliers cannot necessarily be fully utilized for every convolution dimension.

The number of actual computing cycles per layer is

$$\#cycles = \#inter\text{-}tile\_cycles \times \#intra\text{-}tile\_cycles, \tag{3.5}$$

where

$$\begin{aligned} &\#inter\text{-}tile\_cycles = \\ &\lceil \text{Nif}/Tif \rceil \lceil \text{Nkx}/Tkx \rceil \lceil \text{Nky}/Tky \rceil \lceil \text{Nof}/Tof \rceil \lceil \text{Nox}/Tox \rceil \lceil \text{Noy}/Toy \rceil, \end{aligned} \tag{3.6}$$

$$\begin{aligned} &\#intra\text{-}tile\_cycles = \\ &\lceil Tif/Pif \rceil \lceil Tkx/Pkx \rceil \lceil Tky/Pky \rceil \lceil Tof/Pof \rceil \lceil Tox/Pox \rceil \lceil Toy/Poy \rceil. \end{aligned} \tag{3.7}$$

Here we assume that the multipliers receive input data continuously without idle cycles. If the ratio of N* to T* or T* to P* is not an integer, the multipliers or the external memory transactions are not fully utilized. In addition to considering computing latency, memory transfer delay must also be considered for the overall system latency.

3.2.2 Partial Sum Storage

A partial sum (psum) is the intermediate result of the inner product operation that needs to be accumulated over several cycles to obtain one final output data.

Therefore, partial sums need to be stored in memory for the next few cycles and sometimes have to be moved between PEs. An efficient acceleration strategy has to minimize the number of partial sums and process them locally as soon as possible to reduce data movements.

The flow chart to calculate the number of partial sums stored in memory (#psum) is shown in Figure 3.6. To obtain one final output pixel, we need to finish Loop-1 and Loop-2. Therefore, if both Loop-1 and Loop-2 are fully unrolled, the final output pixel can be obtained right after the inner product operations with minimal #psum. If the loop tile size can cover all pixels and weights in Loop-1 (Tkx = Nkx & Tky = Nky) and Loop-2 (Tif = Nif), then the partial sums can be consumed within this tile as described in (9.2) – (9.5) inside Figure 3.6. In this case, the number of partial sums, determined by P^* or T^*, is small and can be stored in local registers ((9.2) inside Figure 3.6) or in on-chip buffers ((9.3) inside Figure 3.6). If the loop tile cannot include all data for Loop-1 and Loop-2, partial sums from one tile need to be stored in on-chip or off-chip memory until it is consumed by another tile as in (9.6) – (9.9) inside Figure 3.6. In this case, the partial sums need to be stored in on-chip buffers ((9.6) inside Figure 3.6) or even in external memory ((9.7) inside Figure 3.6). The loop computing order also affects the number of partial sums, and the earlier Loop-1 and Loop-2 are computed, the fewer is the number of partial sums. The requirement to store partial sums in different levels of memory hierarchy significantly worsens data movements and associated energy cost Chen *et al.* (2016), since partial sums involve both read and write memory operations and typically require higher precision than pixels and weights.

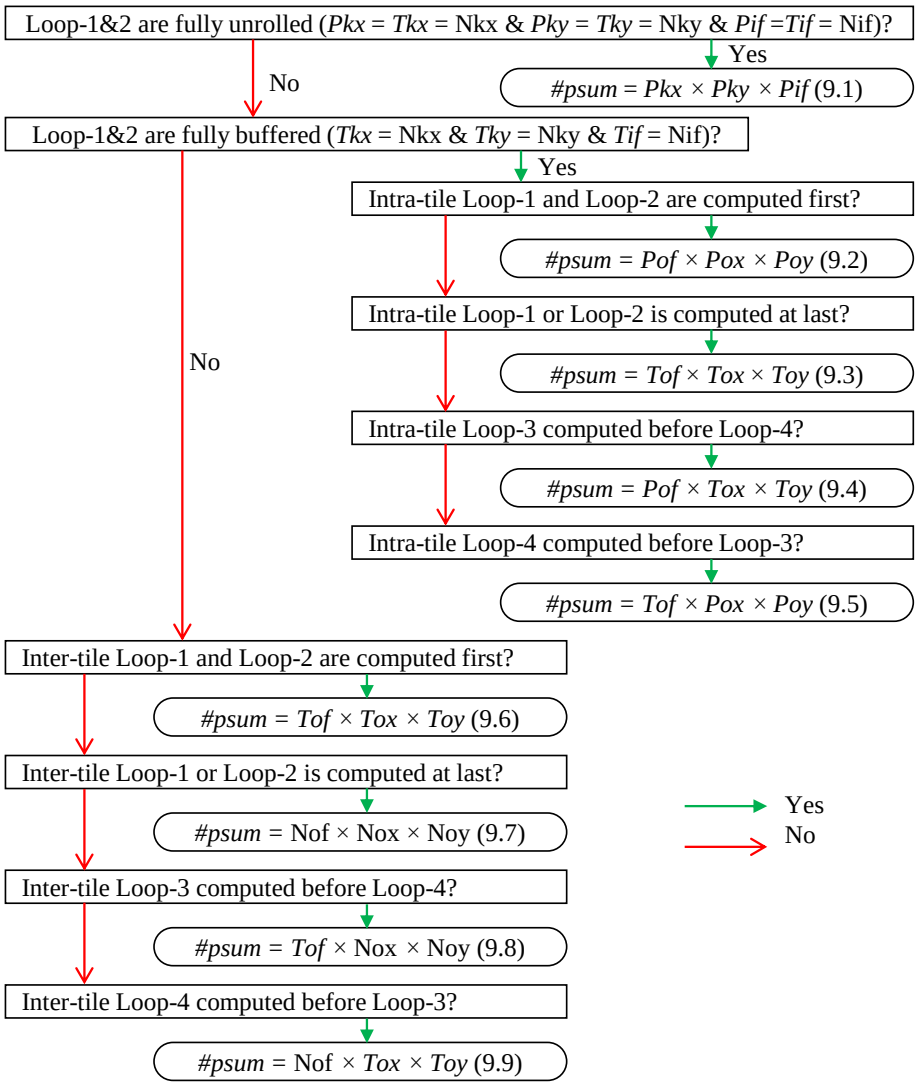

Figure 3.6: Design space exploration of the total number of partial sums that need to be stored in memory.

3.2.3 Data Reuse

Reusing pixels and weights reduces the number of read operations of on-chip buffers. There are mainly two types of data reuse: spatial reuse and temporal reuse. *Spatial reuse* means that, after reading data from on-chip buffers, a single pixel or weight is used for multiple parallel multipliers within one clock cycle. On the other

hand, *temporal reuse* means that a single pixel or weight is used for multiple consecutive cycles.

Having Pm parallel multiplications per cycle requires Pm pixels and Pm weights to be fed into the multipliers. The number of distinct weights required per cycle is:

$$Pwt = Pof \times Pif \times Pkx \times Pky \tag{3.8}$$

If Loop-1 is not unrolled ($Pkx = 1, Pky = 1$), the number of distinct pixels required per cycle (Ppx) is:

$$Ppt = Pif \times Pix \times Piy \tag{3.9}$$

Otherwise, Ppx is:

$$Ppx = Pif \times ((Pix - 1)\mathrm{S} + Pkx) \times ((Piy - 1)\mathrm{S} + Pky) \tag{3.10}$$

Note that 'distinct' only means that the pixels/weights are from different feature/kernel map locations and their values may be the same. The number of times a weight is spatially reused in one cycle is:

$$Reuse\_wt = Pm/Pwt = Pix \times Piy \tag{3.11}$$

where the spatial reuse of weights is realized by unrolling Loop-3 ($Pix > 1 or Piy > 1$). The number of times of a pixel is spatially reused in one cycle ($Reuse\_px$) is:

$$Reuse\_px = Pm/Ppx \tag{3.12}$$

If Loop-1 is not unrolled, $Reuse\_px$ is:

$$Reuse\_px = Pof \tag{3.13}$$

otherwise, $Reuse_px$ is:

$$Reuse\_px = \frac{Pof \times Pkx \times Pky \times Pix \times Piy}{((Pix - 1)\mathrm{S} + Pkx) \times ((Piy - 1)\mathrm{S} + Pky)} \tag{3.14}$$

The spatial reuse of pixels is realized by either unrolling Loop-4 ($Pof > 1$) or unrolling both Loop-1 and Loop-3 together. Only unrolling Loop-1 ($Pix = 1, Piy = 1$) or only unrolling Loop-3 ($Pkx = 1, Pky = 1$) hampers reusing pixels, and $Reuse\_px = Pof$.

If intra-tile Loop-3 is computed first, the weights can be reused for $Tox \times Toy/(Pox \times Poy)$ consecutive cycles. If intra-tile Loop-4 is computed first, the pixels can be reused for Tof/Pof consecutive cycles.

3.2.4 Access of On-chip Buffer

With the data reuse, the number of on-chip buffer accesses can be significantly reduced. Without any data reuse, the total read operations from on-chip buffers for both pixels and weights are Nm, as every multiplication needs one pixel and one weight. With data reuse, the total number of read operations from on-chip buffers for weights becomes:

$$\#read\_wt = \mathrm{Nm}/Reuse\_wt \tag{3.15}$$

and the total number of read operations of buffers for pixels is:

$$\#read\_px = \mathrm{Nm}/Reuse\_px \tag{3.16}$$

If the final output pixels cannot be obtained within one tile, their partial sums are stored in on-chip buffers. The number of write and read operations to/from on-chip buffers for partial sums per cycle is $2 \times Pof \times Pox \times Poy$, where all partial sums generated by Loop-1 (Pkx, Pky) and Loop-2 (Pif) are already summed together right after multiplications. The total number of write and read operations to/from buffers for partial sums is:

$$\#wr\_rd\_psum = \#\_cycles \times (2 \times Pof \times Pox \times Poy) \tag{3.17}$$

The number of times output pixels are written to on-chip buffers (i.e. $\#write\_px$) is identical to the total number of output pixels in the given CNN model. Finally,

the total number of on-chip buffer accesses is:

$$\#buffer\_access = \#read\_px + \#read\_wt + \#wr\_rd\_psum + \#write\_px \quad (3.18)$$

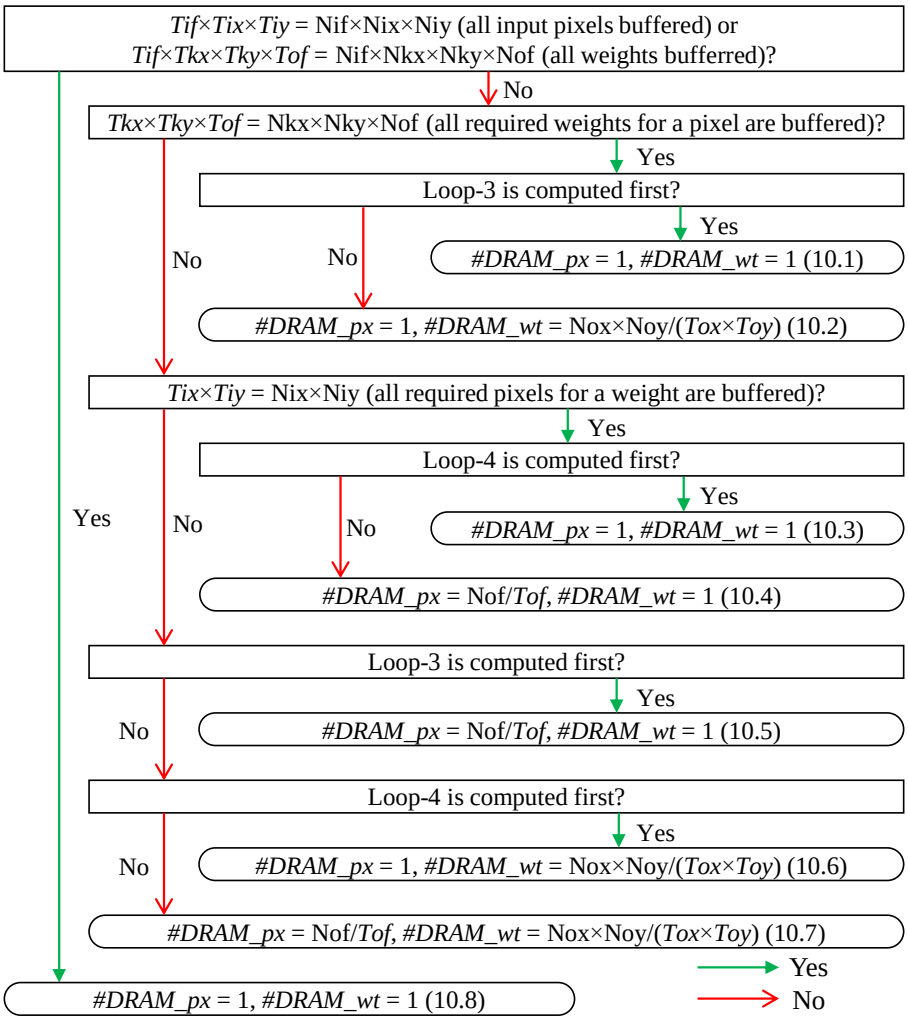

Figure 3.7: Design space exploration of the number of external memory accesses.

3.2.5 Access of External Memory

In our analysis, both the weights and intermediate results of pixels are assumed to be stored in external memory (DRAM), which is a necessity when mapping large-scale CNNs on moderate FPGAs. The costs of DRAM accesses are higher latency and

energy than on-chip Block RAM (BRAM) memory accesses Chen *et al.* (2016)Han *et al.* (2016a), and therefore it is important to reduce the number of external memory accesses to improve the overall performance and energy efficiency. The minimum number of DRAM accesses is achieved by having sufficiently large on-chip buffers and proper loop computing orders, such that every pixel and weight needs to be transferred from DRAM only once. Otherwise, the same pixel or weight has to be read multiple times from DRAM to be consumed for multiple tiles.

The flow chart to estimate the number of DRAM accesses is shown in Figure 3.7, where $\#DRAM\_px$ and $\#DRAM\_wt$ denote the number of DRAM access of one input pixel and one weight, respectively. After fetched out of DRAM, all data should be exhaustedly utilized before being kicked out of the buffer. Therefore, if the tile size or the on-chip buffer can fully cover either all input pixels or all weights of one layer, the minimum DRAM access can be achieved as (10.8) inside Figure 3.7. By computing Loop-3 first, weights stored in buffer are reused and $\#DRAM\_wt$ is reduced as in (10.1) and (10.5) inside Figure 3.7. Similarly, by computing Loop-4 first, pixels can be reused to reduce $\#DRAM\_px$ as in (10.3) and (10.6) inside Figure 3.7. However, computing Loop-3 or Loop-4 first may postpone the computation of Loop-1 or Loop-2, which would lead to a large number of partial sums.

The DRAM access of output pixels is not considered in the analysis because it is constant as every output pixel is written to DRAM only once. As Nkx $>$ S or Nky $>$ S, there are overlaps of pixels on the boundary of two tiles, and these pixels may be read twice by the two tiles. Since the number of the additional read is negligible, we do not include them in the analysis.

3.3 Loop Optimization in Related Works

In this section, the acceleration schemes of the state-of-the-art hardware CNN accelerators are compared. The loop unrolling strategy of current designs can be categorized into the four types:

(A) Unroll Loop-1, Loop-2, Loop-4 Suda *et al.* (2016) Guo *et al.* (2018) Li *et al.* (2016) Motamedi *et al.* (2016)

(B) Unroll Loop-2, Loop-4 Zhang *et al.* (2015) Ma *et al.* (2016)

(C) Unroll Loop-1, Loop-3 Chen *et al.* (2017) Chen *et al.* (2016) Du *et al.* (2017)

(D) Unroll Loop-3, Loop-4 Ma *et al.* (2017b) Rahman *et al.* (2016)

By unrolling Loop-1, Loop-2 and Loop-4 in type-(A), parallelism is employed in kernel maps, input and output feature maps. However, kernel size (Nkx $\times$ Nky) is normally very small ($\leqslant 11 \times 11$) so that it cannot provide sufficient parallelism and other loops need to be further unrolled. A more challenging problem is that kernel size may vary considerably across different convolution layers in a given CNN model (e.g., AlexNet Krizhevsky *et al.* (2012), ResNet He *et al.* (2016a)), which may cause workload imbalance and inefficient utilization of the PEs Du *et al.* (2017). To address this, PEs need to be configured differently for layers with different kernel sizes Zhang *et al.* (2016b), which increases control complexity. In type-(C), every row in the kernel window is fully unrolled (Pkx = Nkx) and Loop-3 is also partially unrolled. By this means, pixels can be reused by the overlapping caused by Loop-1 and Loop-3 as in 3.14, and weight reuse can also be realized by unrolling Loop-3 as in 3.11. However, Loop-4 is not unrolled and further pixel reuse cannot be achieved. The PE efficiency issue caused by unrolling Loop-1 also affects type-(C) Du *et al.* (2017). In type-(A)

and type-(B), Loop-3 is not unrolled, which implies that weights cannot be reused. Type-(B) only unrolls Loop-2 and Loop-4, but Nif $\times$ Nof of the first convolution layer is usually small ($\leqslant 3 \times 96$) and cannot provide sufficient parallelism, which results in low utilization and throughput. If the first layer is computation bounded or the DRAM delay is not overlapped with the computation, the throughput degradation will affect the overall performance, especially for shallow CNNs, e.g., AlexNet and NiN. In type-(D), both Loop-3 and Loop-4 are unrolled so that both pixels and weights can be reused. In addition, Nox $\times$ Noy $\times$ Nof ($\geqslant 7 \times 7 \times 64$) is very large across all the convolution layers in AlexNet, VGG and ResNet so that high level of parallelism can be achieved even for largest FPGA available with $\approx$ 3,600 DSP slices. By this means, a uniform configuration and structure of PEs can be applied for all the convolution layers.

Loop tiling has been used in prior hardware CNN accelerators to fit the large-scale CNN models into limited on-chip buffers. However, only a few prior works Guo *et al.* (2018) Rahman *et al.* (2016) have shown their tiling configurations that determine the on-chip buffer size, but the trade-off between the loop tiling size and the number of external memory accesses is not explored. The tiling size in Rahman *et al.* (2016) does not cover Loop-1 and Loop-2, e.g., $Tkx = Tky = Tif = 1$, which could significantly increase the number and movements of partial sums.

The impact of loop interchange has not been rigorously studied in prior works, but it can greatly impact the number of partial sums as well as the resulting data movements and memory access.

3.4 Proposd Acceleration Scheme

Based on the design objectives and analysis in Section 3.2, the optimization process of our proposed acceleration scheme is presented in this section, which includes

appropriate selection of the convolution loop design variables.

3.4.1 Minimizing Computing Latency

We set variables P^* to be the common factors of T^* for all the convolution layers to fully utilize PEs, and T^* to be the common factors of N* to make full use of external memory transactions. For CNN models with only small common factors, it is recommended to set $\lceil$ N*/T^* $\rceil$ − N*/T^* and $\lceil$ $T^*/P^*\rceil$ − T^*/P^* as small as possible to minimize the inefficiency caused by the difference in sizes of CNN models.

3.4.2 Minimizing Partial Sum Storage

To reduce the number and movements of partial sums, both Loop-1 and Loop-2 should be computed as early as possible or unrolled as much as possible. To avoid the drawback of unrolling Loop-1 as discussed in Section 3.3 and maximize the data reuse as discussed in Section 3.2.3, we decide to unroll Loop-3 ($Pox > 1$ or $Poy > 1$) and Loop-4 ($Pof > 1$). By this means, we cannot attain the minimum partial sum storage as (9.1) inside Figure 3.6.

Constrained by $1 \leqslant P^* \leqslant T^* \leqslant$ N*, the second least number of partial sum storage is achieved by (9.2) among (9.2) – (9.9) inside Figure 3.6. To satisfy the condition for (9.2), we serially compute Loop-1 and Loop-2 first and ensure the required data of Loop-1 and Loop-2 are buffered, i.e., $Tkx =$ Nkx, $Tky =$ Nky and $Tif =$ Nif. Therefore, we only need to store $Pof \times Pox \times Poy$ number of partial sums, which can be retained in local registers with minimum data movements.

3.4.3 Minimizing Access of On-chip Buffer

The number of on-chip buffer accesses is minimized by unrolling Loop-3 to reuse weights as shown in Equation 3.11 and unrolling Loop-4 to reuse pixels as shown

in Equation 3.13. As our partial sums are kept on local registers, they do not add overhead to the buffer access and storage.

3.4.4 Minimizing Access of External Memory

As we first compute Loop-1 and Loop-2 to reduce partial sums, we cannot achieve the minimum number of DRAM access described in (10.1) and (10.3) inside Figure 3.7, where neither the pixels nor the weights are fully buffered for one convolution layer. Therefore, we can only attain the minimum DRAM access by assigning sufficient buffer size for either all pixels or all weights of each layer as in (10.8) inside Figure 3.7.

Then, the optimization of minimizing the on-chip buffer size while having minimum DRAM access is formulated as below:

$$\begin{aligned} &\textbf{minimize} \quad bits\_BUF\_px\_wt \\ &\textbf{subject to} \quad \#Tile\_px_L = 1 \quad \text{or} \quad \#Tile\_wt_L = 1 \\ &\textbf{with} \quad \forall L \in [1, \#CONVs] \end{aligned} \tag{3.19}$$

where $\#Tile\_px_L$ and $\#Tile\_wt_L$ denote the number of tiling blocks for input pixels and weights of layer L, respectively, and $\#CONVs$ is the number of convolution layers.

$bits\_BUF\_px\_wt$ is the sum of pixel buffer size ($bits\_BUF\_px$) and weight buffer size ($bits\_BUF\_wt$), which are given by,

$$bits\_BUF\_px\_wt = bits\_BUF\_px + bits\_BUF\_wt. \tag{3.20}$$

Both pixel and weight buffers need to be large enough to cover the data in one tiling block for all the convolution layers. This is expressed as:

$$\begin{aligned} bits\_BUF\_px = \text{MAX}(words\_px_L) \times pixel\_datawidth \\ \text{with} L \in [1, \#CONVs] \end{aligned} \tag{3.21}$$

$$bits\_BUF\_wt = \text{MAX}(words\_wt_L) \times weight\_datawidth \quad \text{with} L \in [1, \#CONVs] \tag{3.22}$$

where $words\_px_L$ and $words\_wt_L$ denote the number of pixels and weights of one tiling block in layer L, respectively. These are expressed in terms of loop tiling variables as follows,

$$words\_px_L = Tix_L \times Tiy_L \times Tif_L + Tox_L \times Toy_L \times Tof_L \tag{3.23}$$

$$words\_wt_L = Tof_L \times Tif_L \times Tkx_L \times Tky_L \tag{3.24}$$

where $words\_px_L$ is comprised of both input and output pixels. The number of tiles in 3.19 is also determined by T^* variables,

$$\#Tile\_px_L = \lceil \text{Nif}_\text{L}/Tif_L \rceil \times \lceil \text{Nox}_\text{L}/Tox_L \rceil \times \lceil \text{Noy}_\text{L}/Toy_L \rceil \tag{3.25}$$

$$\#Tile\_wt_L = \lceil \text{Nkx}_\text{L}/Tkx_L \rceil \times \lceil \text{Nky}_\text{L}/Tky_L \rceil \times \lceil \text{Nif}_\text{L}/Tif_L \rceil \times \lceil \text{Nof}_\text{L}/Tof_L \rceil \tag{3.26}$$

By solving 3.19, we can find an optimal configuration of T* variables that result in minimum DRAM access and on-chip buffer size. However, since we have already set Tkx = Nkx, Tky = Nky, Tif = Nif as in Section 3.4.2, we can only achieve a sub-optimal solution by tuning Tox, Toy and Tof, resulting in larger buffer size requirement. If the available on-chip memory is sufficient, we set Tox = Nox so that an entire row can be buffered to benefit the DMA transactions with continuous data.

Finally, we have to solve 3.19 by searching Toy and Tof, because it has a non-linear objective function and constraints with integer variables. Since Toy and Tof in VGG-16 consist of $2 \times \#CONVs = 26$ variables and each variable can have about 4 candidate values constrained by T^*/P^* = integer and N^*/T^* = integer, the total number of Toy and Tof configurations is about $426 = 4.5 \times 10^{15}$, which becomes an enormous solution space. In ResNet-50/ResNet-152, the $\#CONVs$ are increased to be 53 and 155, respectively, which makes the solution space even larger to be about

$4^{106} = 6.6 \times 10^{63}$ and $4^{310} = 4.4 \times 10^{186}$, respectively. Therefore, it is impossible to enumerate all the candidate solutions. In Ma *et al.* (2017b), we randomly sampled the configurations of VGG-16, and it took about 10 hours to obtain a relatively good solution. However, in ResNets, this method becomes infeasible due to the dramatically increased solution space.

In this work, a new method is proposed to empirically find a satisfactory solution for a given on-chip memory capacity that takes advantage of the property of CNNs. CNNs normally have large pixel data volume and small weight sizes in the beginning few layers. As we proceed into deeper layers, the pixel sizes become smaller with extracted features, and the weight sizes become larger with more channels. This trend is illustrated in Figure 3.8, where the bars denote data sizes in each convolution layer. To benefit from the data distribution property in different layers, we only need to make pixel buffers fully cover the last few layers and weight buffers fully cover the beginning few layers. Then, the middle layers with both relatively large pixel and weight sizes become the constraints of the buffer sizes, and we only need to take care of these bounding layers, which significantly shrinks the solution space. The dashed lines in Figure 3.8 are the minimal buffer sizes we found while guaranteeing minimum DRAM accesses, and the bounding layers are pointed out by arrows. If this buffer size still cannot be fit into the FPGA on-chip memory, then we need to either change the tiling strategy or decrease the buffer sizes at the cost of more DRAM accesses as discussed in Ma *et al.* (2017b).

3.4.5 Optimized Loop Design Variables

According to the aforementioned optimization process, we propose a convolution acceleration scheme for a high-performance and low-communication CNN accelerator, which is visualized in Figure 3.9.

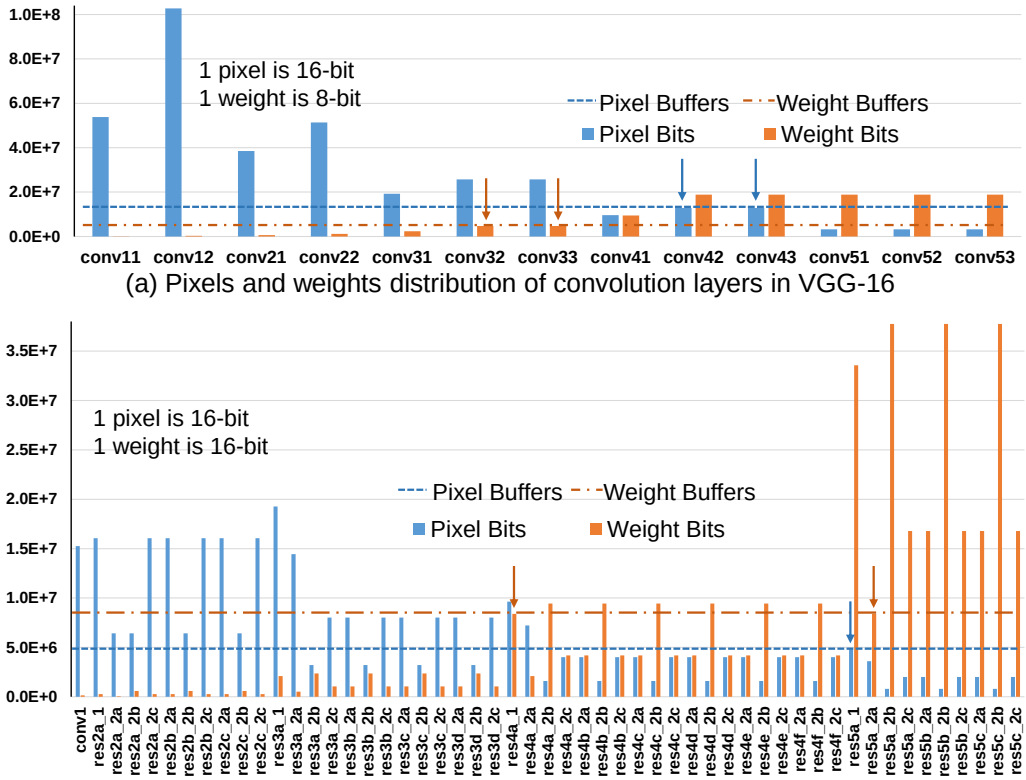

Figure 3.8: To guarantee minimum DRAM accesses, either all pixels (blue bars) are covered by pixel buffers (blue dashed lines) or all weights are covered by weight buffers in one layer. Then, we try to lower the total buffer sizes/lines.

Loop Unrolling

For all the convolution layers, Loop-1 and Loop-2 are not unrolled, which means $Pkx = 1$, $Pky = 1$ and $Pif = 1$. For VGG-16, we set $Pox = 14$, $Poy = 14$ and $Pof = 16$, which enables Pm = 3,136 parallel multiplications. For ResNets, we set $Pox = 7$, $Poy = 7$ and $Pof = 32$, which needs Pm = 1,568 parallel multiplications. Since the minimum feature map dimensions (NOX and NOY) of ResNets are 7, Pox and Poy are reduced to be 7. The more complex structure and new type of layers force ResNets to use less multipliers than VGG-16. By setting $P*$ to be constant across all the convolution layers, a uniform structure and mapping of PEs can be realized

to reduce the architecture complexity.

Loop Tiling

For loop tiling, we set Tkx = Nkx, Tky = Nky, Tif = Nif as in Section 3.4.2 and shown in Figure 3.9 so that data used in Loop-1 and Loop-2 are all buffered and Tox = Nox to benefit DMA transfer. Details of Toy and Tof are described in Section 3.4.4.

Loop Interchange

For loop interchange, we first serially compute Loop-1 and then Loop-2 as described in Section 3.4.2. Finally, we compute Loop-3 and Loop-4, where the exact computation order of these two loops does not have a pronounced impact on the cost, based on our P^* and T^* choices.

3.5 Proposd CNN Accelerator

To implement the optimized convolution acceleration scheme in Section 3.4.5, a data router is proposed with high flexibility for different convolution sliding settings, e.g. strides and zero paddings, using variant data buses. A corresponding hardware PE architecture is also designed that minimizes on/off-chip memory accesses and data movements.

3.5.1 Data Bus from Buffer to PE (BUF2PE)

In Ma *et al.* (2017b), a register array architecture is designed to rearrange and direct the pixel stream from buffers into PEs. This method takes advantage of convolution stride being 1 in VGG-16 so that pixels can be reused by the adjacent register array in the next computing cycles. However, if stride is 2 or more, which frequently

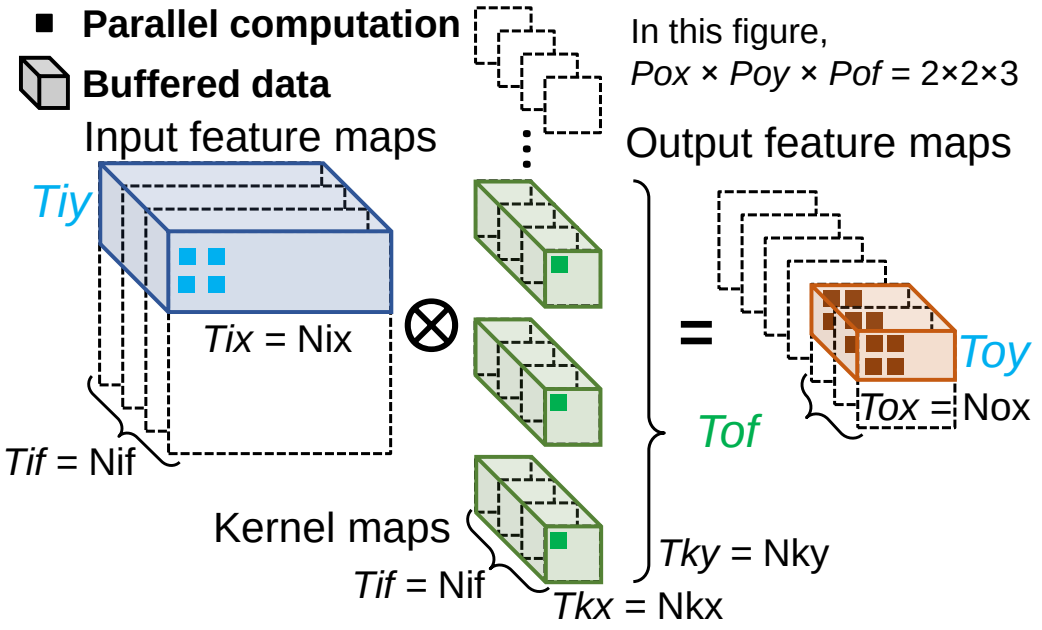

Figure 3.9: The optimized loop unrolling and tiling strategy. The parallelism is within one feature map ($Pox \times Poy$) and across multiple kernels (Pof). The tiling variables Tiy, Toy and Tof can be tuned that decide the buffer sizes.

occurs in CNN algorithms Krizhevsky *et al.* (2012)He *et al.* (2016a), pixels need to wait for Nkx × (Stride − 1) cycles to be reused by the neighboring register array. This makes the control logic and wire routing among registers much more complicated. Therefore, we propose a BUF2PE data bus in Figure 3.10 to implement the dataflow using FIFO to temporally store pixels to be reused by the adjacent register array. This method is similar to line buffer design in Bosi *et al.* (1999), where FIFOs are used to align pixels from multiple feature rows to a kernel window so that parallelism can be employed within a kernel window, i.e. unrolling Loop-1, whereas this work unrolls Loop-3 to parallel compute within one feature map. By this means, the wire routing within and across register arrays is simplified, and the data router can follow the same pattern for convolution with different strides and zero paddings.

The detailed design of BUF2PE data bus is illustrated in Figure 3.10. Pixels from

input buffers are loaded into the corresponding registers as shown by the blue dashed box to the blue solid box. Then, the pixels are sent to PEs or MAC units and are also sent to FIFOs during cycles 0 to 5, waiting to be reused by the adjacent register array. Register arrays except the rightmost one start reading input pixels from FIFOs at cycle 3, as shown by the purple pixels in Figure 3.10. Meanwhile, the new pixels are fed into the rightmost register array from buffers. In this work, the offset caused by west zero padding is handled by shifting the connection between buffers and register arrays, whereas Ma *et al.* (2017b) has to change the storage pattern within one address of input buffer by a padding offset that increases the complexity of transferring data from DRAM to buffers.

The coarse-grained dataflow is shown in Figure 3.11 at feature map row level for stride = 1 and stride = 2. The data flow in Figure 3.11(a) is the same as Figure 3.10, where more clock cycles of operation is shown after cycle 8. In Figure 3.11(b), the dataflow with stride = 2 and zero padding = 3 is shown, which follows the same pattern as the case with stride = 1. The buffer storage pattern is adjusted according to different stride and padding settings. Three rows of zeros are added to the buffer due to the north zero padding of 3. With stride = 2, every two rows of pixels are continuously distributed across Poy buffer banks. These adjustments are handled by the buffer write enable and address signals during the reception of pixels from DRAM. Since the data movement within a register array or a feature map row is different for different settings of stride and zero padding, various BUF2PE data buses are needed for each dataflow, and the set of data buses are called data router. If these settings are identical, one BUF2PE bus can handle different kernel sizes (Nkx $\times$ Nky) without penalty of idle cycles as we serially compute Loop-1. Therefore, the BUF2PE bus in Figure 3.11(b) can be applied for conv1 in ResNet with kernel size = 7 $\times$ 7, stride = 2 and zero padding = 3. For other sliding settings in ResNet, e.g. stride = 2 and

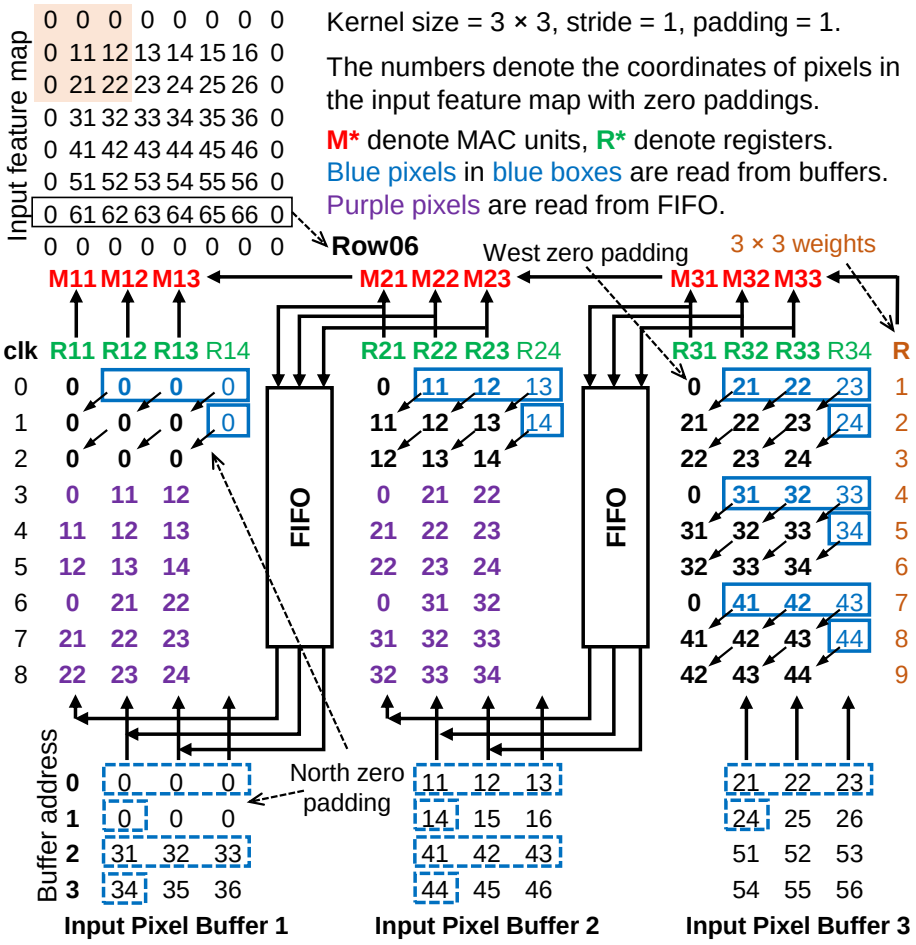

Figure 3.10: The BUF2PE data bus directs the convolution pixel dataflow from input buffers to PEs (i.e. MAC units), where $Pox = 3$ and $Poy = 3$.

zero padding = 0, the corresponding variants of BUF2PE buses are designed to direct the dataflow. The global control logic controls the switch among different BUF2PE buses inside the data router.

After Nkx × Nky cycles, we complete one kernel window sliding (Loop-1) and move to the next input feature map with the same dataflow until the last one as shown in Figure 3.11. After Nkx × Nky × Nif cycles, both Loop-1 and Loop-2 are completed and we obtain $Pox \times Poy \times Pof$ final output pixels.

In summary, the proposed dataflow is scalable to Nkx × Nky by changing the

control logic, and it can handle various sliding settings using variant BUF2PE data buses inside the data router, whereas the MAC units are reused and kept busy.

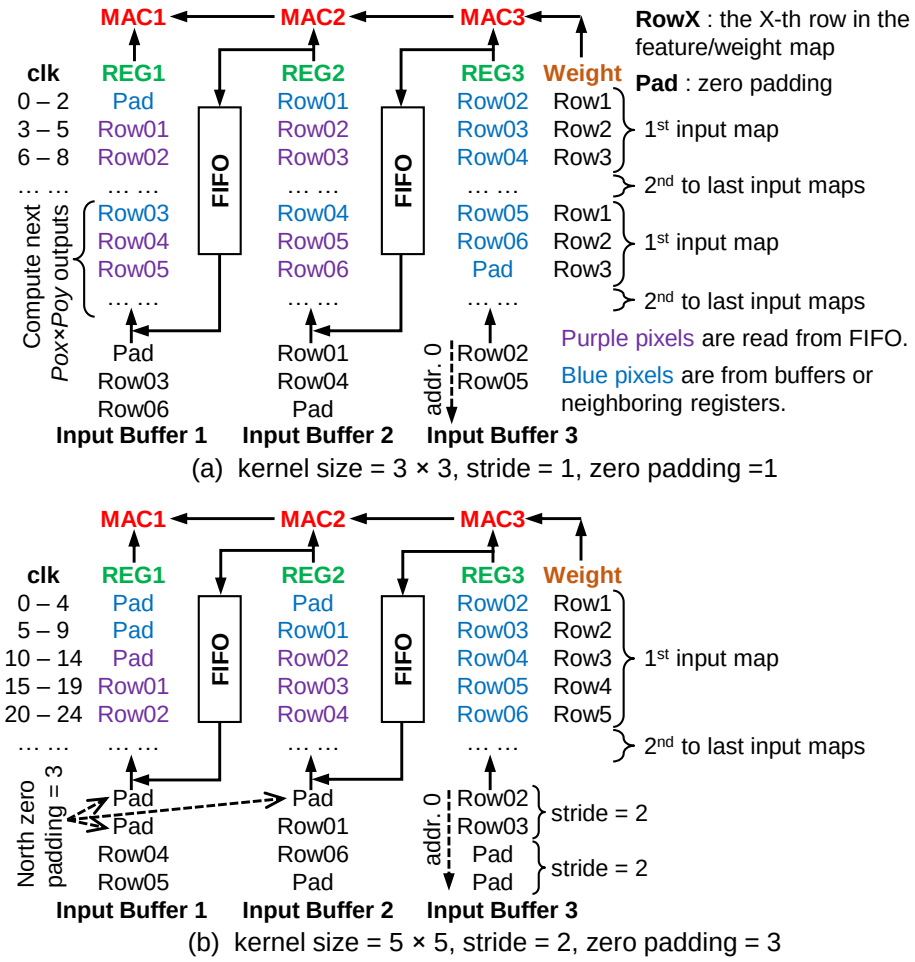

Figure 3.11: The coarse-grained designs of BUF2PE data buses for (a) strides = 1 and zero padding = 1 and (b) stride = 2 and zero padding = 3.

3.5.2 Convolution PE Architecture

The PE architecture of convolution layers shown in Figure 3.12 is designed according to the proposed acceleration strategy and dataflow. It is comprised of $Pox \times Poy \times Pof$ PEs, and every PE in our architecture is an independent MAC unit con-

sisting of one multiplier followed by an accumulator. As Loop-1 and Loop-2 are not unrolled, no adder tree is needed to sum the multiplier outputs. The partial sum is consumed inside each MAC unit until the final results are obtained, such that the data movements of partial sums are minimized. Pixels read from input pixel buffers are shared by Pof MAC units and sliding overlapped pixels are also reused by the data router. Weights read from weight buffers are shared by $Pox \times Poy$ MAC units. The proposed architecture is implemented with parameterized Verilog codes and is highly scalable to different CNN models in FPGAs or even ASICs by modifying design variables such as Pox, Poy and Pof. After the completion of Loop-1 and Loop-2, the partial sums need to be added with biases as in Figure 2.3 to obtain the final output pixels. Therefore, every Nkx $\times$ Nky $\times$ Nif cycles, MAC units output the partial sums into the adders to add with biases. Since $Poy <$ Nkx $\times$ Nky $\times$ Nif for all the layers, we serialize the $Pox \times Poy \times Pof$ MAC outputs into Poy cycles. Then, we only need $Pox \times Pof$ adders to add the biases in parallel. The data width of one output buffer can also be reduced to be Pox and we store the pixels of one output feature map in one buffer bank, which could need totally Pof output buffers. If Pof is large, e.g. $Pof = 32$ in ResNet, it would require many output buffers with shallow depth, resulting in low utilization of on-chip BRAMs (e.g. M20K memory block). In addition, batch normalization (Bnorm) layers in ResNet still need $Pox \times Pof$ adders and multipliers that are expensive. We further serialize the $Pox \times Pof$ parallel outputs to be $Pox\times$ #OUTBUF using multiplexers with neighboring output feature maps stacked in one output buffer, as illustrated in Figure 3.12. In ResNet, we set #OUTBUF = 16 to ensure $Poy \times Pof$ / (#OUTBUF) $<$ Nkx $\times$ Nky $\times$ Nif or the number of serial output cycles is smaller than the MAC unit output interval cycles. By this means, the parallelism of adders and multipliers for bias and Bnorm is significantly reduced, as well as the output buffer bandwidth and the used M20K BRAMs.

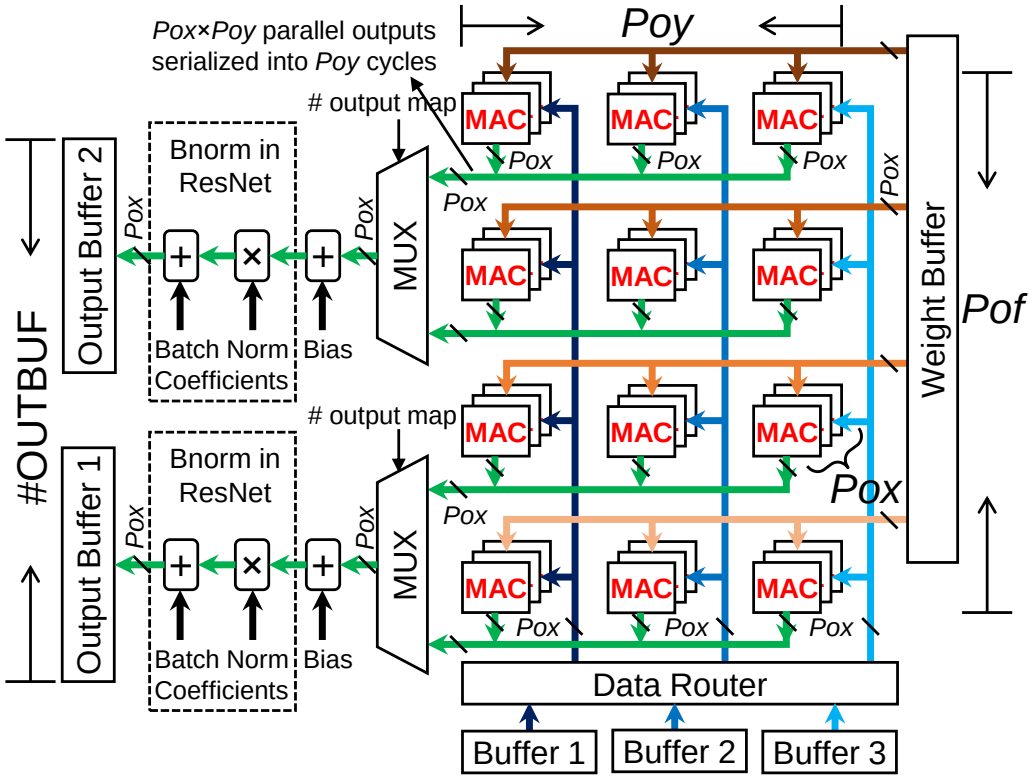

Figure 3.12: Convolution acceleration architecture with $Pox \times Poy \times Pof$ MAC units.

3.5.3 Pooling Layers

Pooling is commonly used to reduce the feature map dimension by replacing pixels within a kernel window (e.g., 2×2, 3×3) by their maximum or average value. The output pixels from previous convolution layers are stored row-by-row in the output pixel buffers. As pooling operation only need pixels, after one tile of convolution is finished, we directly compute pooling with pixels read from output pixel buffers to eliminate the access of external memory. The unrolling factors of all the pooling layers are the same. Since the width of output pixel buffer is Pox, we can enable Pox $\times$ #OUTBUF parallel pooling operations, which is large enough considering that pooling layers involve much less operations compared to convolution layers. Register arrays are used to reshape the pooling input pixels and ensure continuous feeding

of pixels into pooling PEs without idle cycles. The PEs are either comparators for max-pooling or accumulators followed by constant coefficient multipliers for average-pooling. The outputs of pooling are written back to the output pixel buffers and then transferred to the external memory.

3.5.4 Fully-connected Layers

The inner-product layer or fully-connected (FC) layer is a special form of the convolution layer with Nkx = Nky = Nox = Noy = 1, or there are no Loop-1 and Loop-3. Therefore, we only unroll Loop-4 and reuse the same MAC unit array used in convolution layers for all the FC layers. Contrary to convolution layers, FC layers have large amount of weights but small amount of operations, which makes the throughput of FC layers primarily bounded by the off-chip communication speed. Due to this, dual FC weight buffers are used to overlap the inner-product computation with off-chip communication for VGG implementation. In ResNet, the size of FC weights (= 2.0M) is significantly reduced compared to that of VGG (= 123.6M), and we reuse the convolution weight buffers for FC weights and start the FC computations after the weights are read from DRAM. FC layer output pixels are directly stored in on-chip buffers as their size is small (< 20 KB).

3.6 Experimental Results

3.6.1 System Setup

The proposed hardware CNN inference accelerator is demonstrated by implementing NiN Lin *et al.* (2013), VGG-16 Simonyan and Zisserman (2014) and ResNet-50/ResNet-152 He *et al.* (2016a) CNN models on two Intel FPGAs. In NiN, VGG-16 and ResNet-50/ResNet-152, there are 12/13/53/155 convolution layers, 3/5/1/1 max-

pooling, 1/0/1/1 average-pooling, 0/3/1/1 FC and 0/0/16/50 element-wise (Eltwise) layers, respectively. Some convolution layers are followed by batch normalization (Bnorm) and ReLU layers. The two Intel FPGAs, e.g. Stratix V GXA7 / Arria 10 GX 1150, consist of 234.7K/427.2K adaptive logic modules (ALM), 256/1,518 DSP blocks and 2,560/2,713 M20K BRAM blocks, respectively. The underlying FPGA boards for Stratix V and Arria 10 are Terasic DE5-Net and Nallatech 385A, respectively, and both are equipped with two banks of 4GB DDR3 DRAMs.

The overall CNN acceleration system on the FPGA chip shown in Figure 3.13 is coded in parametrized Verilog scripts and configured by the proposed CNN compiler in Ma *et al.* (2017a) for different CNN and FPGA pairs. If a layer does not exist in the CNN model, the corresponding computing module is not synthesized and the dataflow just bypasses this module, for example, VGG-16 does not have Eltwise layer and this layer is not compiled. With two DRAM banks, both kernel and feature maps are separated into these two banks to enable full off-chip communication. Two Modular Scatter-Gather DMA (mSGDMA) engines provided by Intel are used to simultaneously read and write from/to these two DRAM banks. Data scatter and gather in Ma *et al.* (2017a) are used to distribute the data stream from DMA into multiple input buffers and collect data from multiple output buffers into one DMA stream, respectively. After the input images and weights are loaded into DRAMs, the CNN inference acceleration process starts. When the computation of one loop tile completes, the output pixels are transferred to DRAM, and then the weights and pixels for the next loop tile are loaded from DRAM to on-chip buffers. The controller governs the iterations of the four convolution loops and the layer-by-layer sequential computation. The buffer read and write addresses are also generated by the controller.

The fixed-point data representation is used, and both pixels and weights are 16-

bit. The decimal points are dynamically adjusted according to the ranges of pixel values in different layers to fully utilize the existing data width Guo *et al.* (2018). By this means, the top-1 and top-5 ImageNet classification accuracy degradation is within 2% compared with software floating point implementation in Zhang *et al.* (2016b) Suda *et al.* (2016) Aydonat *et al.* (2017) Guo *et al.* (2018) Ma *et al.* (2016).

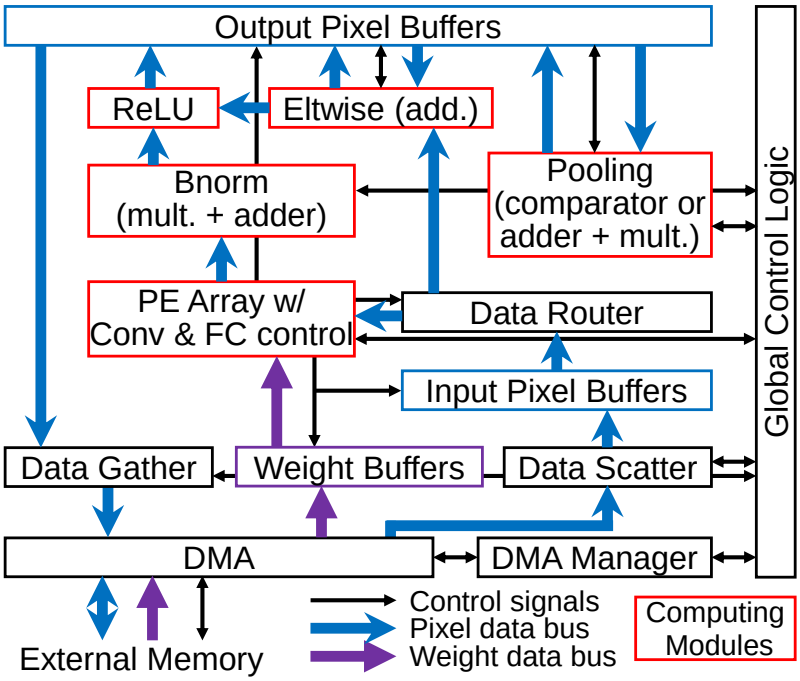

Figure 3.13: Overall FPGA-based CNN hardware acceleration system.

3.6.2 Analysis of Experimental Results

The performance and specifications of our proposed CNN accelerators are summarized Table 3.2. In Stratix V and Arria 10, one DSP block can be configured as either two independent 18-bit × 18-bit multipliers or one multiplier followed by an accumulator, i.e. one MAC. Since one multiplier consumes much more logic than one adder, we use the DSP as two independent multipliers and implement the accumulator inside the MAC unit by ALMs. Since Arria 10 has 1.8× more ALMs and 5.9× more DSP

blocks than the Stratix V we use, larger loop unrolling variables ($Pox \times Poy \times Pof$) can be achieved in Arria 10 to obtain $> 2\times$ throughput enhancement than Stratix V.

Compared with Ma *et al.* (2017b), the unrolling variables, i.e. $Pox \times Poy \times Pof$, of VGG-16 are set to be $7 \times 7 \times 64$ on Arria 10 instead of $14 \times 14 \times 16$, where the number of MAC units (= 3,136) are the same and both sets of $P*$ variables are the common factors of the feature/kernel map sizes resulting in the same computation cycles. The data router in Figure 3.10 and the data buses after MAC units in Figure 3.12 are only related with Pox and Poy, whereas the data buses related with Pof from weight buffers to MAC units in Figure 3.12 are relatively simple. To reduce the data bus width and required logic, we choose smaller $Pox \times Poy$ in this work as 7×7 with a larger Pof as 64. Since the greatest common factors of feature/kernel maps, e.g. Nox $\times$ Noy $\times$ Nof, of all convolution layers in ResNets are $7 \times 7 \times 64$, we still set $Pox \times Poy \times Pof$ to be $7 \times 7 \times 64$. Since ResNets have more complex structure and more types of layers, e.g. Eltwise and Bnorm, they consume more logic elements than NiN and VGG-16 on Arria 10 and cannot achieve the same parallel degree as NiN and VGG-16 on Stratix V. Since the two FPGAs have close capacity of on-chip BRAMs, the loop tiling variables ($T*$) of the same CNN is set to be the same for both FPGAs, which leads to similar BRAM consumption.

The breakdown of the processing time per image of each CNN is shown in Fig. 17 with batch size = 1. The MAC computation time of convolution layers, e.g. "Conv MAC", dominates the total latency by over 50%. "Conv DRAM" includes DRAM transaction delay of convolution weights and input/output pixels. The FC latency includes the inner-product computation delay and the DRAM transfer delay of FC weights. "Others" include the delay of average pooling, Eltwise and pipeline stages.

The logic utilization in ALMs of each module is shown in Figure 3.15. Most multipliers in MAC units are implemented by DSPs, and logic elements are mainly

Table 3.2: Our Implementation of Different CNNs on Different FPGAs

CNN	NiN	VGG-16	ResNet-50	ResNet-152
# Operations (GOP)	2.20	30.95	7.74	22.62
# of Parameters	7.59 M	138.3 M	25.5 M	60.4 M
Precision (fixed)	16 bit	16 bit	16 bit	16 bit
FPGA / Tech.	Intel Stratix V GXA7 / 28 nm			
Clock	150 MHz	150 MHz	150 MHz	150 MHz
$Pox \times Poy \times Pof$	$7 \times 7 \times 32$	$7 \times 7 \times 32$	$7 \times 7 \times 24$	$7 \times 7 \times 24$
# of MAC Units	1,568	1,568	1,176	1,176
DSP Blocks	256 (100%)	256 (100%)	256 (100%)	256 (100%)
Logic (ALMs)	228K (97%)	218K (93%)	176K (75%)	185K (78%)
BRAM (M20K)	1,512 (59%)	2,210 (86%)	1,950 (76%)	2,385 (93%)
Delay/Image (ms)	7.9	88.8	31.8	81.8
Overall Throughput (GOPS)	278.2	348.8	243.3	276.6
FPGA / Tech.	Intel Arria 10 GX 1150 / 20 nm			
Clock	200 MHz	200 MHz	200 MHz	200 MHz
$Pox \times Poy \times Pof$	$7 \times 7 \times 64$	$7 \times 7 \times 64$	$7 \times 7 \times 64$	$7 \times 7 \times 64$
# of MAC Units	3,136	3,136	3,136	3,136
DSP Blocks	1,518 (100%)	1,518 (100%)	1,518 (100%)	1,518 (100%)
Logic (ALMs)	161K (38%)	138K (32%)	221K (52%)	235K (55%)
BRAM (M20K)	1,528 (56%)	2,232 (82%)	1,931 (71%)	2,365 (87%)
Delay/Image (ms)	3.8	43.2	12.7	32.0
Overall Throughput (GOPS)	584.8	715.9	611.4	707.2

used to implement accumulators in MAC units. With the same parallel computation degree, the MAC units of the four CNNs use about the same amount of ALMs. As VGG-16 is highly uniform with only one convolution sliding setting, e.g. stride = 1 and padding = 1, only one BUF2PE bus is needed, which leads to less logic and BRAM consumption of data router compared to NiN and ResNets. Convolution and FC layers share the MAC units but have their own control logic to govern the

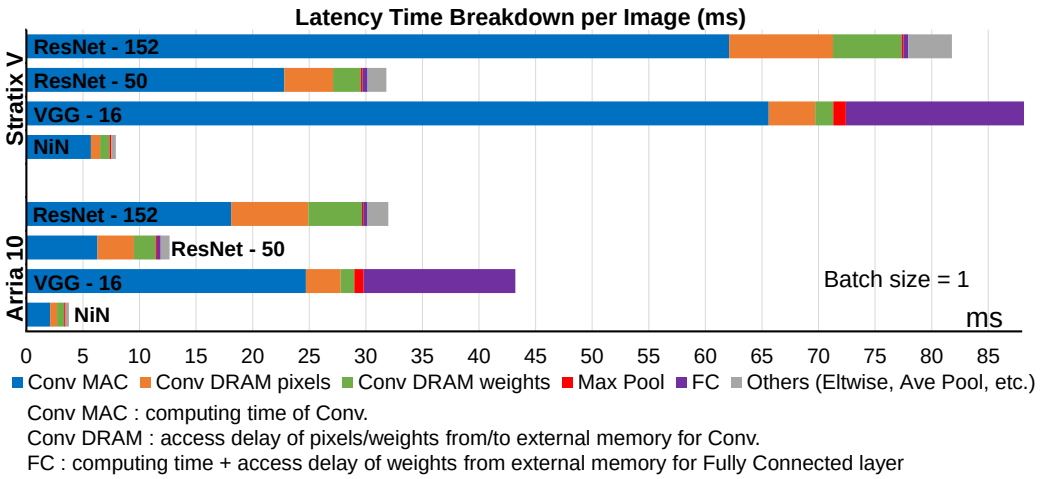

Figure 3.14: Latency breakdown per image of ResNet-50/152 and VGG-16.

sequential operations. Eltwise layers use adders to element-wise add pixels from two branches of layers. "Others" include the system interconnections, global control logic, bias adders, and configuration registers.

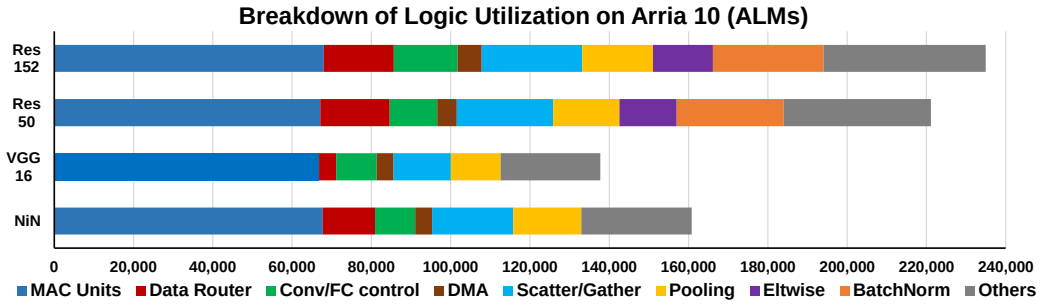

Figure 3.15: Logic utilization breakdown of ResNet-50/152 and VGG-16.

The breakdown of the on-chip memory usage is shown in Figure 3.16. ResNet-152 uses more BRAMs than ResNet-50, because more Bnorm coefficients are saved in BRAMs and the number of instructions for DMA manager is increased due to the additional layers. FIFOs in data router are implemented by BRAMs. "FC" in Figure 3.16 only includes the buffer to store intermediate FC pixels. FC and convolution layers share the weight buffers.

Table 3.3: Comparison with Previous CNN FPGA Implementations

	Rahman *et al.* (2016)	Li *et al.* (2016)	Aydonat *et al.* (2017)	Guo *et al.* (2018)	Suda *et al.* (2016)	Zhang *et al.* (2016b)	Wei *et al.* (2017)	Guan *et al.* (2017)	Guan *et al.* (2017)
CNN	AlexNet	AlexNet	AlexNet	VGG-16	VGG-16	VGG-16	VGG-16	VGG-19	ResNet-152
FPGA	Virtex-7 VC707	Virtex-7 VC709	Arria 10 GX 1150	Zynq XC7Z045	Stratix V GSD8	Virtex-7 VX690t	Arria 10 GT 1150	Stratix V GSMD5	Stratix V GSMD5
Clock (MHz)	160	156	303	150	120	150	231.85	150	150
# Oper. (GOP)	1.33	1.46	1.46	30.76	30.95	30.95	30.95	39.26	22.62
# Parameters	2.33 M	60.95 M	60.95 M	50.18 M	138.3 M	138.3 M	138.3M	143.7M	60.4M
Precision	32bit fixed	16bit fixed	FP16bit	16bit fixed	8-16bit fixed	16bit fixed	8-16bit fixed	16bit fixed	16bit fixed
DSP[a]	2,688 (96%)	2,144 (60%)	1,476 (97%)	780 (87%)	-	-	1,500 (99%)	1,044 (66%)	1,044 (66%)
Logic[b]	45K (9.2%)	274K (63%)	246K (58%)	183K (84%)	-	-	313K (73%)	45.7K (27%)	45.7K (27%)
BRAM[c]	543 (53%)	956 (65%)	2,487 (92%)	486 (89%)	-	-	1668 (61%)	959 (48%)	959 (48%)
Delay/Image (ms)	-	8×2.56[d]	-	224.6	262.9	151.8	26.85	-	-
Throughput (GOPS)	147.82	565.94	1.38 TFLOPS	136.97	117.8	203.9	1,171.3	364.36	226.47

[a] Xilinx FPGAs in DSP slices and Intel FPGAs in DSP blocks.

[b] Xilinx FPGAs in LUTs and Intel FPGAs in ALMs.

[c] Xilinx FPGAs in BRAMs (36Kb) and Intel FPGAs in M20Ks (20Kb).

[d] The reported delay of one pipeline stage is 2.56 ms and the number of pipeline stages equals 8.

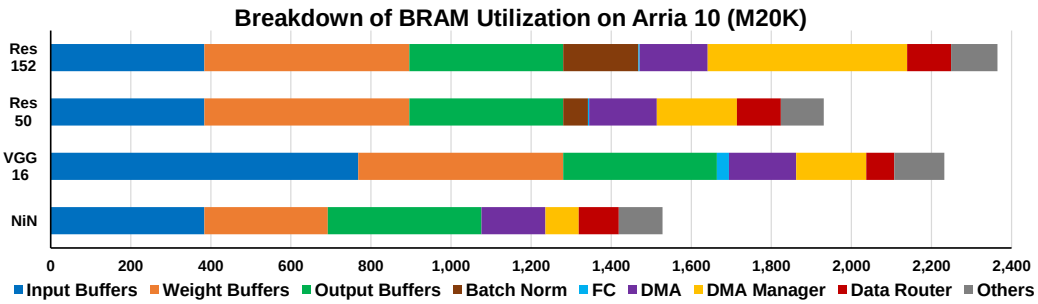

Figure 3.16: On-chip BRAM breakdown of ResNet-50/152 and VGG-16.

3.6.3 Comparison with Prior Works

The reported results from recent CNN FPGA accelerators are listed in Table 3.3. Rahman *et al.* (2016) only implements convolution layers in AlexNet and uses the similar strategy as us to unroll Loop-3 and Loop-4, which can also achieve high DSP utilization. However, their loop tiling strategy is only along Loop-3 and Loop-4, which significantly postpones the acquisition of the final pixels resulting in more memory accesses and data movements of partial sums. In Zhang *et al.* (2016b) and Li *et al.* (2016), the layer-by-layer computation is pipelined using different part of one or multiple FPGAs resources to improve hardware utilization and thus throughput. However, with the highly increasing number of convolution layers He *et al.* (2016a), it becomes very difficult to map different layers onto different resources and balance the computation among all the pipeline stages. In addition, pipelining can increase the throughput but not necessarily the latency. Batch computing with multiple input images is applied in Chen *et al.* (2016), Zhang *et al.* (2016b), Aydonat *et al.* (2017), and Li *et al.* (2016). The biggest advantage of this technique is to share the weights transferred from off-chip DRAM among multiple images and thus increase the throughput at the cost of increased latency per image and external memory storage of multiple images. Benefit from batch computing and using 2,144 DSP slices, which en-

ables high parallelism degree, Li *et al.* (2016) also achieves high throughput of 565.94 GOPS for AlexNet. In Aydonat *et al.* (2017), an OpenCL-based CNN accelerator is implemented on Arria10 FPGA, where the Intel FPGA SDK for OpenCL provides a pre-generated platform that ensures timing closure at higher frequency than our RTL design. The Winograd transform is applied for convolution layers that reduces multiplication operations by 2× or improves the throughput by 2× using the same number of DSPs. The 16-bit floating-point data format is used with shared exponent, which allows directly using fixed-point 18-bit × 18-bit multipliers for floating-point operations. Wei *et al.* (2017) proposed an OpenCL-based systolic array architecture to implement convolution on Arria 10, which reduces the global PE interconnect fanout to achieve high frequency and resource utilization. The VGG-16 throughput of Wei *et al.* (2017) is higher than ours mainly due to 1) higher frequency, 2) lower precision of weights, and 3) dual buffer scheme to hide DRAM latency. Guan *et al.* (2017) proposed an RTL-HLS hybrid framework to automatically generate FPGA hardware and implements convolution and FC as matrix multiplication. Although the Stratix-V GSMD5 (with 1590 DSP blocks) used in Guan *et al.* (2017) has 6.2× more DSP blocks than our Stratix-V GXA7, our accelerator on Stratix V can realize 1.2× higher throughput for ResNet-152 by higher hardware (DSP and logic) utilization through the proposed loop optimization technique and exploiting logic elements to implement multipliers as well as DSPs.

With the optimized CNN acceleration scheme and low-communication dataflow, the proposed CNN accelerator uses uniform unrolling factors for all the convolution layers and fully utilizes the DSPs. The proposed methodology is also demonstrated by implementing ResNet, which exhibit a highly irregular and complex structure.

3.7 Summary

In this chapter, we present an in-depth analysis of convolution loop acceleration strategy by numerically characterizing the loop optimization techniques. The relationship between accelerator objectives and design variables are quantitatively investigated, and we provide design guidelines for an efficient acceleration strategy. A corresponding new dataflow and architecture is proposed to minimize data communication and enhance throughput. Our CNN accelerator implements end-to-end NiN, VGG-16 and ResNet-50/ResNet-152 CNN models on Stratix V and Arria 10 FPGA, achieving the overall throughput of 348 GOPS and 715 GOPS, respectively.

Chapter 4

AUTOMATIC COMPILATION OF DIVERSE CNNS ONTO FPGA

4.1 Overview of Proposed CNN RTL Compiler

The dimensions and connections of CNN layers and pre-trained kernel weights are obtained from Caffe Jia *et al.* (2014), and provided as inputs to the CNN compiler. The various dimensional parameters of the CNN algorithm and the accelerator design variables, e.g. loop unrolling and tiling sizes as shown in Figure 4.1 (described in detail in Section 4.2), can be tuned by the user to balance the performance and required hardware resources. Then, a layer-by-layer execution schedule (see Figure 4.2(a) and Figure 4.2(b)) is generated from the CNN graph representation. The execution schedule is translated into the global control logic on the FPGA, and it also determines the order of the reads and writes of certain kernel weights or pixels from different layers that are stored in external memory. The associated read and write addresses are generated and sorted to control the transactions between external and on-chip memories.

The RTL module library consists of manually coded Verilog templates describing the computations and dataflow of various types of layers. The templates are built on the optimized CNN acceleration strategy described in Ma *et al.* (2018a). That strategy is designed to minimize the memory access and data movements while maximizing the resource utilization. The Verilog parameters that determine the size of PEs and buffers are configured based on the design variables. The parameters for runtime control are initialized by compiler and stored in configuration registers. The intra-tile execution flow of layers, as shown in Figure 4.2(c), is predefined in the templates

and can be customized by the compiler to enable execution of certain layers during run time. The top-level accelerator system template, shown in Figure 4.3, integrates these modules with the reconfigurable dataflow, where only the required computing modules are compiled for a given CNN model, bypassing the unused modules.

4.2 Acceleration of Convolution Loops

4.2.1 Convolution Loop Optimization and Design Variables

Convolution involves three-dimensional multiply and accumulate operations (MAC) of input feature maps and kernel weights as illustrated in Figure 4.1, where the parameters (N*) prefixed with capital N denote the algorithm-defined dimensions of feature and kernel maps of one Conv layer. Since convolution dominates the CNN operations, the acceleration strategy of convolution loops dramatically impacts the parallel computation efficiency and memory access requirements. Therefore, we employ the loop optimization techniques in Ma *et al.* (2018a) to customize the convolution computation and communication patterns. Loop unrolling design variables (P*) determine the degree of parallelism of certain convolution loops, and thus the required size and architecture of PEs. Loop tiling increases the data locality by dividing the entire data of one layer into multiple tiles, which can be fit into the on-chip buffers. The loop tiling design variables (T*) determine the required minimum sizes of the on-chip buffers, and affect the required external memory accesses.

4.2.2 Convolution Acceleration Strategy

The design of the module templtes at the RTL is based on the CNN acceleration strategy described in Ma *et al.* (2018a). It achieves a uniform mapping of PEs and reduces the accelerator architecture complexity. Figure 4.1 shows the dimensions

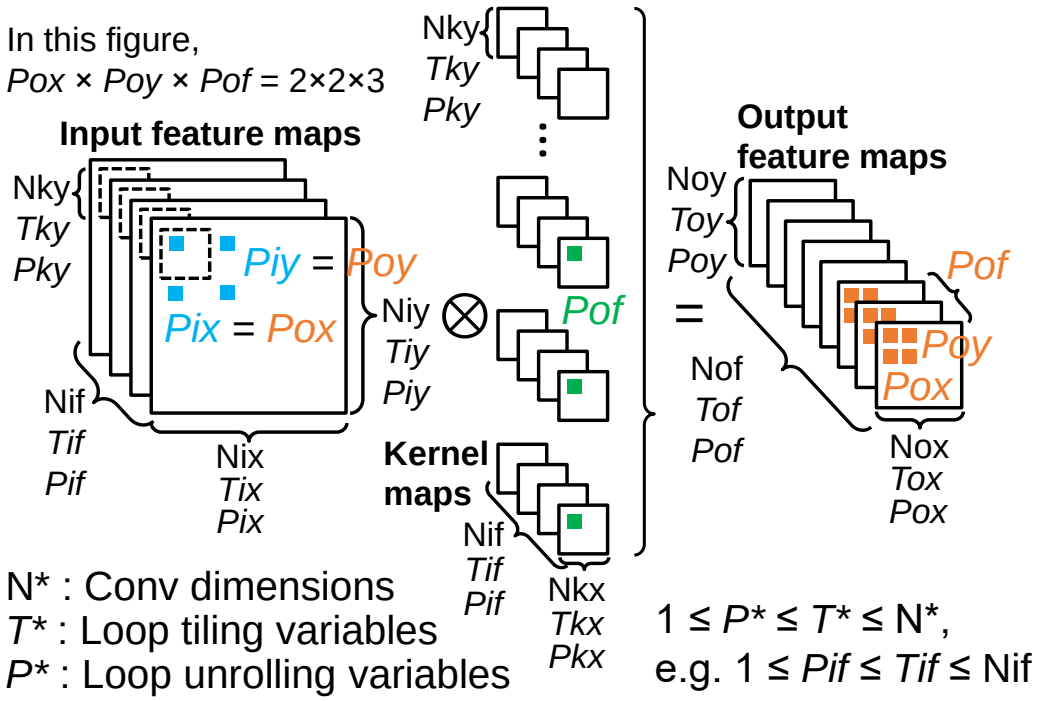

Figure 4.1: Convolution loop dimensions (N*) and accelerator design variables of loop unrolling (P*) and loop tiliing (T*). Type: i: input; o: output; k: kernel; f: feature.

of the inputs (input feature maps and kernel maps) and the output feature maps. The loop unrolling or the parallel computations are only employed within one input feature map and across multiple kernel maps. The computations shown in Figure 4.1 are as follows.

1. $Pix = Pox > 1$ and $Piy = Poy > 1$: in every cycle, $Pix \times Piy$ number of pixels from different (x, y) locations in the same input feature map are multiplied with one identical weight;

2. $Pof > 1$: in every cycle, one input pixel is multiplied by Pof weights from Pof different kernel maps, which contributes to Pof pixels in Pof output feature maps.

The total number of parallel operations is $Pox \times Poy \times Pof$ with $Pkx = Pky = Pif = 1$. By this means, each PE contributes to one independent output pixel and no adder tree is needed to total the partial sums of different PEs Ma *et al.* (2018a). Therefore, a PE is a MAC unit consisting of one multiplier followed by an accumulator in this work. Both pixels and weights are reused by multiple MAC units and high degree of parallelism can be supported with large Nox × Noy × Nof. The data required to compute one final output pixel are fully buffered to minimize the partial sum storage, i.e. $Tkx = \text{Nkx}, Tky = \text{Nky}, Tif = \text{Nif}$. We also set $Tox = \text{Nox}$ so that an entire row is buffered to improve the DRAM transactions with data from continuous addresses. Furthermore, the required buffer sizes can be changed by tuning Toy and Tof. Following the above optimized settings, different $P*$ and $T*$ design variables can be adjusted by the user to explore the best trade-off between performance and hardware resource usage, e.g. DSP blocks and block RAMs (BRAMs), for the target FPGA platform.

4.3 End-to-end CNN Accelerator

4.3.1 Layer-by-layer Execution Schedule

In conventional CNN algorithms, different layers are connected in sequence, which allows for a straightforward layer-by-layer serial computation. The recent CNN algorithms (e.g. ResNet He *et al.* (2016a)) are DAGs, with combinations of serial and parallel branches. A reconfigurable layer-by-layer execution schedule is designed to handle the different combinations of stacked layers and the DAG as shown in Figure 4.2. Therefore, the present mapping of a DAG onto an FPGA still results in a serial computation of the layers.

There are many types of layers in a CNN algorithm, and the number and order

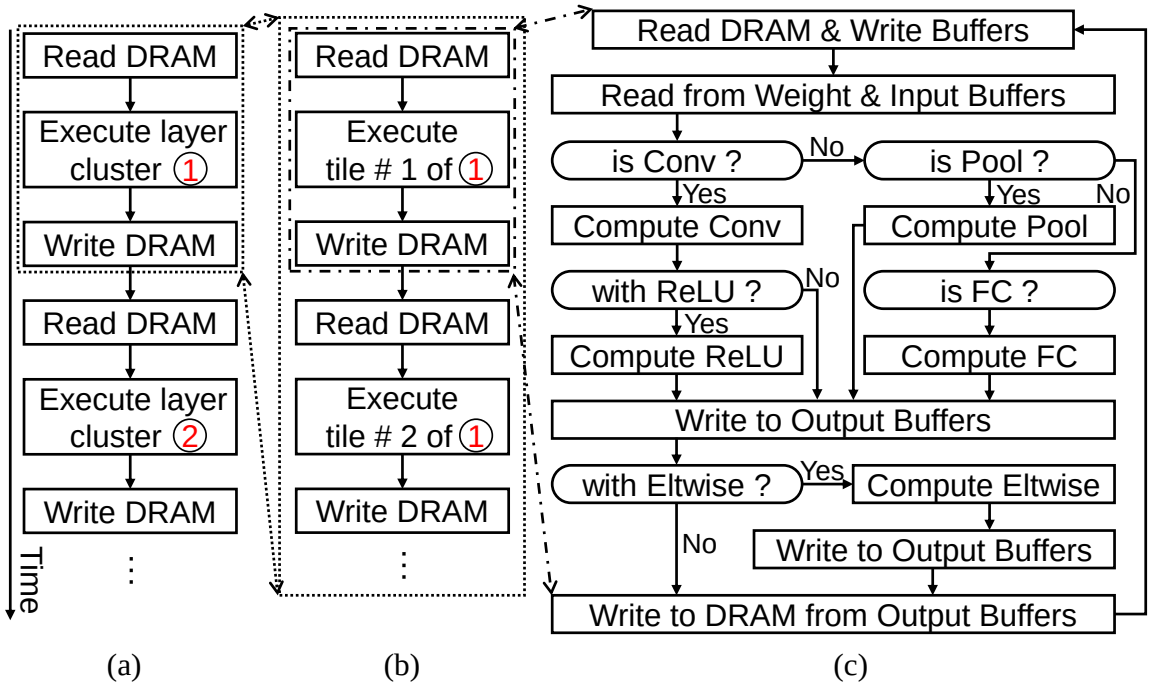

Figure 4.2: The execution schedule is designed to handle different CNN topology: (a) layer-by-layer execution (b) inter-tile execution inside one layer (c) intra-tile process inside one tile.

of these stacked layers could be quite different. A CNN layer that reads the DRAM for its input is referred to as a *key-layer*. Therefore, Conv, Pool and FC are assigned as key-layers so that the computation or design variable settings between these layers are relatively independent, while all other layers are *affiliated-layers* to the key-layers. The DRAM access of an affiliated-layer can be eliminated, however its computing pattern, e.g. unrolling and tiling variables, must depend on the key-layer configuration, which hampers its design flexibility. A layer *cluster* is a subgraph of the DAG that consists of a key-layer and zero or more affiliated-layers. The example DAG shown in Figure 1.1 has six clusters, numbered ① through ⑥. The Conv1(①), Pooling(②) and FC(⑥) layers in Figure 1.1 are individual key-layers (i.e. clusters with only a

key-layer) whereas cluster ⑤ has one key-layer (Conv4) and three affiliated-layers (Batchnorm, Eltwise and ReLu). The layer-by-layer serial computation is essentially the serial execution of clusters as illustrated in Figure 1.1 and Figure 4.2(a). The order of computation of the clusters is set before compilation, and the only rule is to ensure that all the predecessors of any key-layer is executed prior to that key-layer.

When tiling of loops is performed, each cluster is divided into multiple tiles to fit into the on-chip buffers. This is illustrated in Figure 4.2(a)(b). As clusters may contain different kinds of layers, (e.g. layer cluster ④ in Figure 1.1 does not have BatchNorm and Eltwise), a general intra-tile execution schedule is designed as shown in Figure 4.2(c) to control whether or not a layer is executed for a specific cluster during runtime. The select signals, e.g. "is Conv?" in Figure 4.2(c), are stored in the configuration registers and initialized based on the input CNN topology during compilation. If a layer does not exist in the given CNN, the select signal becomes constant to be "No". This schedule is also flexible as it allows introduction of new types of layers by the simple addition of new select signals.

Three levels of control logic, namely global, inter-tile, and local control logic, are required to govern the layer-by-layer, inter-tile and intra-tile sequential execution (Figure 4.2). The parameters of each layer, e.g. kernel sizes, feature map dimensions, unrolling and tiling variables, and iteration numbers, are stored in configuration registers. The global control logic keeps track of the number of executed clusters, and loads the current layer's parameters from the configuration registers into the local control logic registers. Each type of layer module has its own local control logic to perform the iterations within the layer. By this means, we can just use one set of control logic for layers with varying dimensions by initializing configuration registers for different layers during compilation.

4.3.2 Top-level Acceleration System and Dataflow

The overall CNN acceleration system and dataflow is shown in Figure 4.3, where different types of layers are modularized to establish the RTL module library. During compilation, if a certain type of layer does not exist in the given CNN model, its corresponding module will not be compiled or synthesized to save the hardware resources, and the dataflow just bypasses this module. During runtime, whether or not a layer is executed is controlled by the global control logic by asserting "start" signal to the module following the execution schedule. After receiving a "done" signal from the current layer, global control logic iterates to the next layer.

The reconfigurable computing modules, as shown by the red boxes in Figure 4.3, are manually coded as maximally parameterized Verilog scripts. Each type of module template is designed to be reused by any layer of the same type, in any CNN. The varying layer sizes and loop design variables are handled by initializing the configuration registers based on the layer property. This RTL module library is designed to be easily extended with new layers for more CNN algorithms and the existing modules can also be further optimized for performance and efficiency. The detailed design of the computing modules is discussed in Section 4.5.

The direct memory access (DMA) engine is used to transfer data between external and on-chip memories. The data scatter module is designed to distribute a data stream from one DMA write port to multiple input buffers, and the data gather module is designed to collect data from multiple output buffers into one DMA read port. The detailed memory system design is presented in Section 4.4.

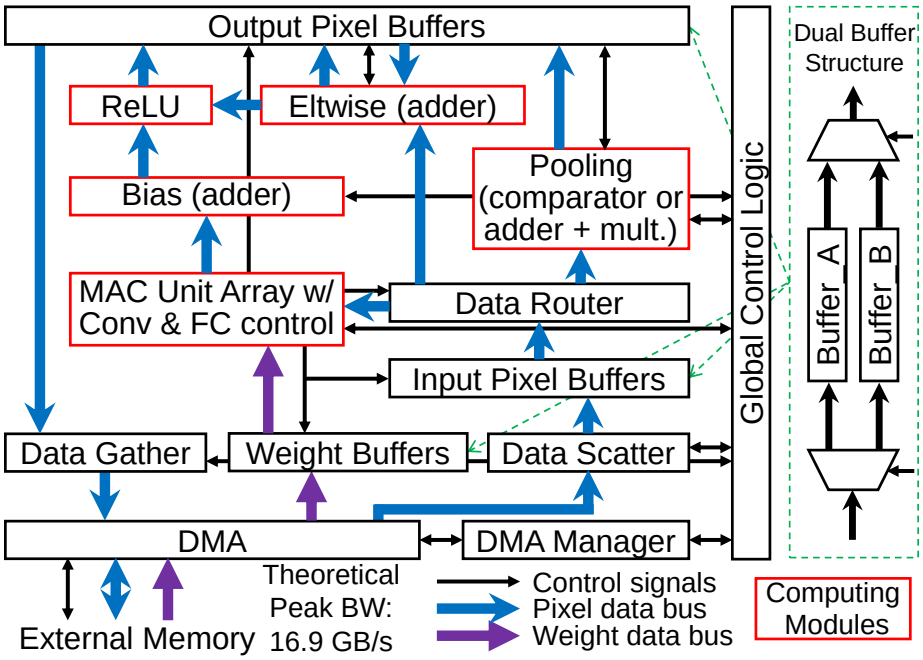

Figure 4.3: Reconfigurable top-level CNN acceleration system, where the dataflow is from external memory to input buffers and then into computing modules, the results are stored in output buffers and finally sent back to external memory.

4.4 External and On-chip Memory System

4.4.1 Storage Pattern in DRAM

Due to the limited capacity of on-chip BRAMs, both kernel weights and intermediate pixel results are stored in external memory, i.e. the DRAM, and the on-chip BRAMs are used as buffers between DRAM and PEs. The proposed storage pattern of kernel weights and intermediate pixel results in the DRAM are illustrated in Figure 4.4(a). The pre-trained kernel weights and the input images are loaded into DRAM before the acceleration. All the intermediate output pixels are organized in the form from row-by-row, map-by-map to layer-by-layer in continuous DRAM

addresses.

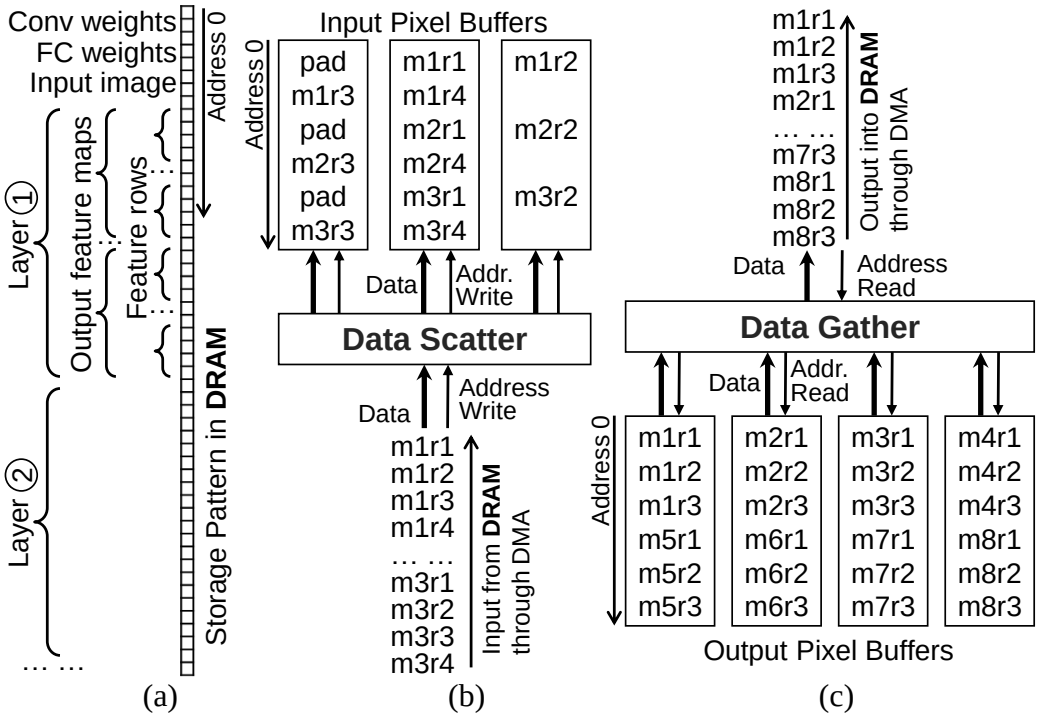

Figure 4.4: (a) Storage pattern in DRAM. (b) Data scatter. (c) Data gather, where mXrY denotes the Y-th row in the X-th feature map.

4.4.2 DMA Manager

The DMA engine is used to communicate data between DRAM and on-chip BRAMs. A custom DMA manager module is designed to control the DMA operation using preload descriptors. The descriptor sets the source and destination addresses and the transaction bytes. Given the CNN parameters, loop design variables and the order of computation of the layers, the descriptors are generated by the compiler and stored in the on-chip BRAM. As weights are loaded into the DRAM before acceleration, we have the freedom the reorganize the weight storage pattern during compi-

lation to enable the continuous DRAM read operations. Therefore, a tile needs only one descriptor to read the weights. As we compute multiple output feature maps in parallel, Pof weights from Pof kernel maps are grouped together and continuously stored in DRAM. The weight groups are stored in the order along Nkx, Nky, Nif and Nof dimensions. To read/write the pixels from/to the DRAM, one descriptor is responsible to transfer a portion of one input/output feature map, e.g. $Tix \times Tiy$ continuous pixels. If one entire feature map is buffered, e.g. $Tix = \text{Nix}, Tiy = \text{Niy}$, one descriptor can read/write multiple feature maps because these pixels across different maps are also continuously stored.

4.4.3 Data Scatter and Gather

The accelerator has two memory mapped slave ports to receive/send data from/to one DMA, respectively. The data stream from the DRAM is in continuous form and a data scatter is designed to distribute and rearrange data to multiple input pixel buffers as illustrated in Figure 4.4(b), where mXrY denotes the Y-th row in the X-th feature map. With different length of feature map rows, one mXrY may occupy different number of addresses. The data scatter module counts the number of received pixels based on the received DMA write signal and generates the write addresses and write enable signal for the buffers. Similarly, the data gather module in Figure 4.4(c) is designed to collect data from multiple output pixel buffers into continuous form to benefit DMA transactions.

4.4.4 Dual Buffer Structure

The dual buffer structure (or ping-pong buffer structure) Zhang *et al.* (2015) is employed to overlap the PE computation with external memory communication to decrease the overall latency. By this means, while the DMA is writing/reading one

buffer, the PE array can read/write the other buffer simultaneously, as illustrated in Figure 4.5. With one DRAM bank, the DMA is designed not to read and write DRAM at the same time to avoid potential conflict, and the DMA only sequentially writes input/weight buffers and reads output buffers at different times.

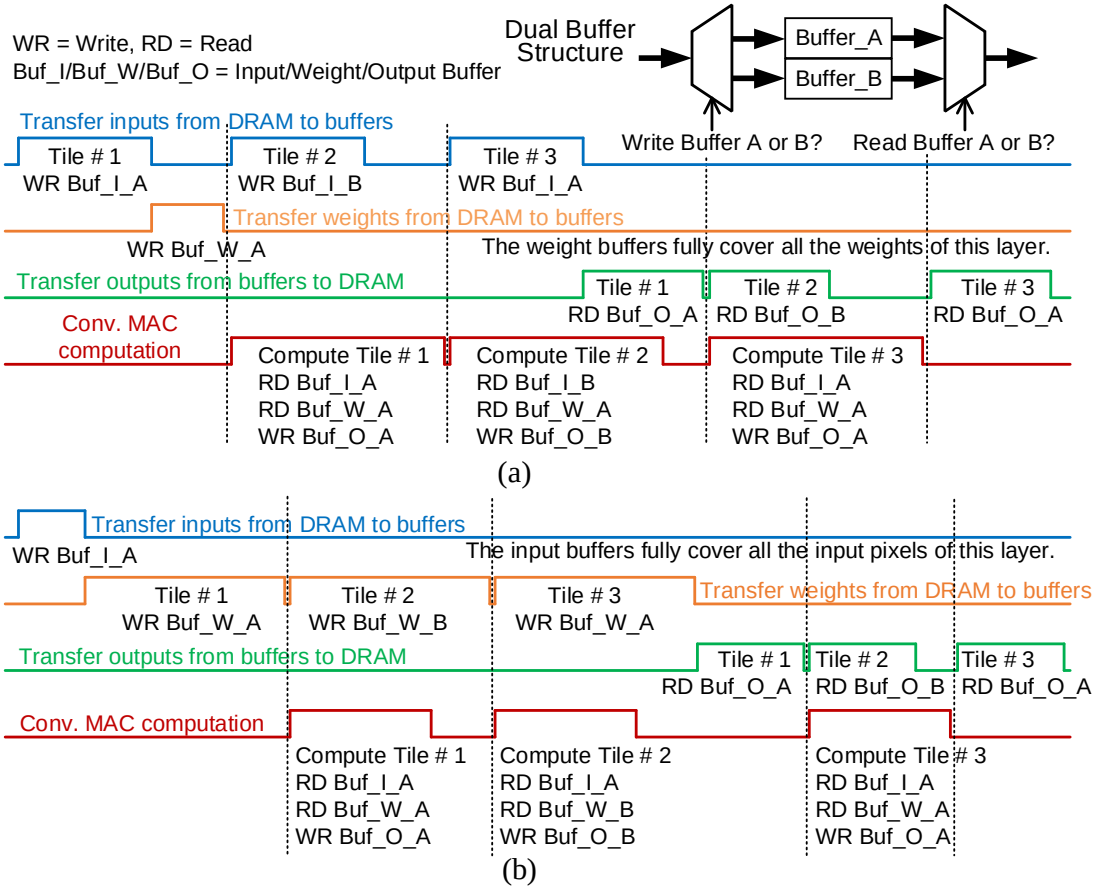

Figure 4.5: The dual buffer structure and its pipeline schedule is used to overlap computation with memory communication to improve the throughput. (a) All the weights of this layer are fully buffered and the weights only need to be read once from DRAM. (b) All the pixels of this layer are fully buffered and the pixels only need to be read once from DRAM.

Figure 4.5(a) illustrates the pipeline schedule when the weight buffers fully cover

all the weights of this Conv layer, where we only need to read the weights from DRAM once and different tiles can reuse the weights without reloading them from the DRAM again. Similarly, the input buffers fully cover all the input pixels of this Conv layer in Figure 4.5(b), and different tiles can reuse the pixels. If the buffers cannot fully cover either all pixels or all weights of one layer, the same pixels or weights need to be read multiple times from the DRAM Ma *et al.* (2018a).

Before the computation of Tile #1, we need to load both input pixels and weights of Tile #1 into the buffers. While computing Tile #1, we can start to load the inputs (Figure 4.5(a)) or weights (Figure 4.5(b)) of Tile #2 into the other buffer and write outputs to the output buffer. In Figure 4.5(a), the computation time of Tile #1 is longer than the delay of loading input buffer, so Tile #2 can only start after the completion of Tile #1 computation, which means its overall delay is bounded by the computation delay. On the other hand, in Figure 4.5(b), the memory delay is longer than the computation time of Tile #1, so Tile #2 can only start after the memory transaction is finished, which makes its delay bounded by the memory communication delay. Since the DMA can only start reading the output buffer after the computation of this tile is fully completed, the outputs of Tile #1 are transferred to DRAM during/after the computation of Tile #2, while the outputs of Tile #2 are written into the other output buffer. To simplify the control logic, the pipeline of computation and memory transaction is currently only within each layer. By this means, the write of input/weight buffers of the first tile and the read of output buffer of the last tile are not overlapped with computation, which limits the efficiency of dual buffer structure to further improve the throughput.

4.4.5 Computation Bounded vs. Memory Bounded

The roofline model is introduced in Zhang *et al.* (2015) to analyze the performance bottleneck of the CNN accelerator, which is mainly affected by the available computation resources (DSP or MAC units) and the external memory (DRAM) bandwidth. When overlapping computations with external memory transactions, if the computation delay exceeds the memory delay, the design is said to be *computation bounded*, with the bound referred to as the *computation roof throughput*. Otherwise, it is said to be *memory bounded*, with the bound referred to as the *memory roof throughput*. The computation roof throughput (DSP_roof) is defined as:

$$\begin{aligned} DSP\_roof(GOPS) &= \frac{\#operations(GOP)}{DSP\_delay(s)}, \\ DSP\_delay(s) &= \frac{\#operations}{2 \times \#MACs} \times clock\_period(s). \end{aligned} \tag{4.1}$$

where $\#operations$ is the number of operations and $\#MACs$ is the number of MAC units. One MAC unit computes two operations (one multiplication and one addition) at one clock cycle. Therefore, the DSP_roof is determined by the number of MAC units and the operating clock frequency. The memory roof throughput ($DRAM\_roof$) is defined as:

$$\begin{aligned} DRAM\_roof(GOPS) &= \frac{\#operations(GOP)}{DRAM\_delay(s)}, \\ DRAM\_delay(s) &= \frac{\#data(GB)}{DRAM\_BW(GB/s)}. \end{aligned} \tag{4.2}$$

where $DRAM\_BW$ is the external memory bandwidth, and $\#data$ is the data size of memory accesses including both reading inputs/weights from DRAM and writing outputs to DRAM. The *roof throughputs* (DSP_roof and $DRAM\_roof$) are shown in Figure 4.6 for each Conv layer of different CNN algorithms. The DSP_roof of Arria 10/Stratix 10 are computed with different number of MAC units at 240/300 MHz, respectively. The $DRAM\_roof$ is directly proportional to computation to

communication ratio (CTC) Zhang *et al.* (2015) by memory bandwidth, e.g. 12 GB/s in Figure 4.6. If DSP_roof is lower than $DRAM\_roof$, the design is computation bounded, otherwise it is memory bounded. Obviously, the attainable throughputs are lower than both roof throughputs. With relatively large intermediate feature map dimensions and kernel sizes, VGG-16 has a larger CTC ratio or memory roof throughput than NiN, GoogLeNet and ResNet, which makes its implementation easier to be computation bounded as shown in Figure 4.6. By this means, the increase of hardware resources, e.g. from Arria 10 to Stratix 10, is expected to benefit the throughput improvements of VGG-16 more than the other three algorithms, which will be demonstrated in Section 4.6. The DSP_roof with 6,272 MAC units on Stratix 10 are already larger than $DRAM\_roof$ of most layers in NiN, GoogLeNet and ResNet as in Figure 4.6 that makes the design memory bounded, which means the increase of the number of MAC units to be 8,192 will only bring insignificant performance enhancement. Limited by the utilization of computation resources and the efficiency of external memory accesses, the real throughput of one layer may not be able to achieve the roof throughput of this layer.

4.5 Reconfigurable CNN Computing Modules

4.5.1 Convolution Modules (Conv)

Based on our convolution acceleration strategy, the module template of Conv layer is designed as in Figure 4.7, which follows the computing architecture in Ma *et al.* (2018a). There are $Pox \times Poy \times Pof$ independent PEs in Conv module, and each PE is a MAC unit consisting of one multiplier followed by an accumulator. With judiciously chosen loop unrolling scheme, both pixels and weights are reused by multiple MAC units to reduce buffer read operations. The partial sums are consumed inside each

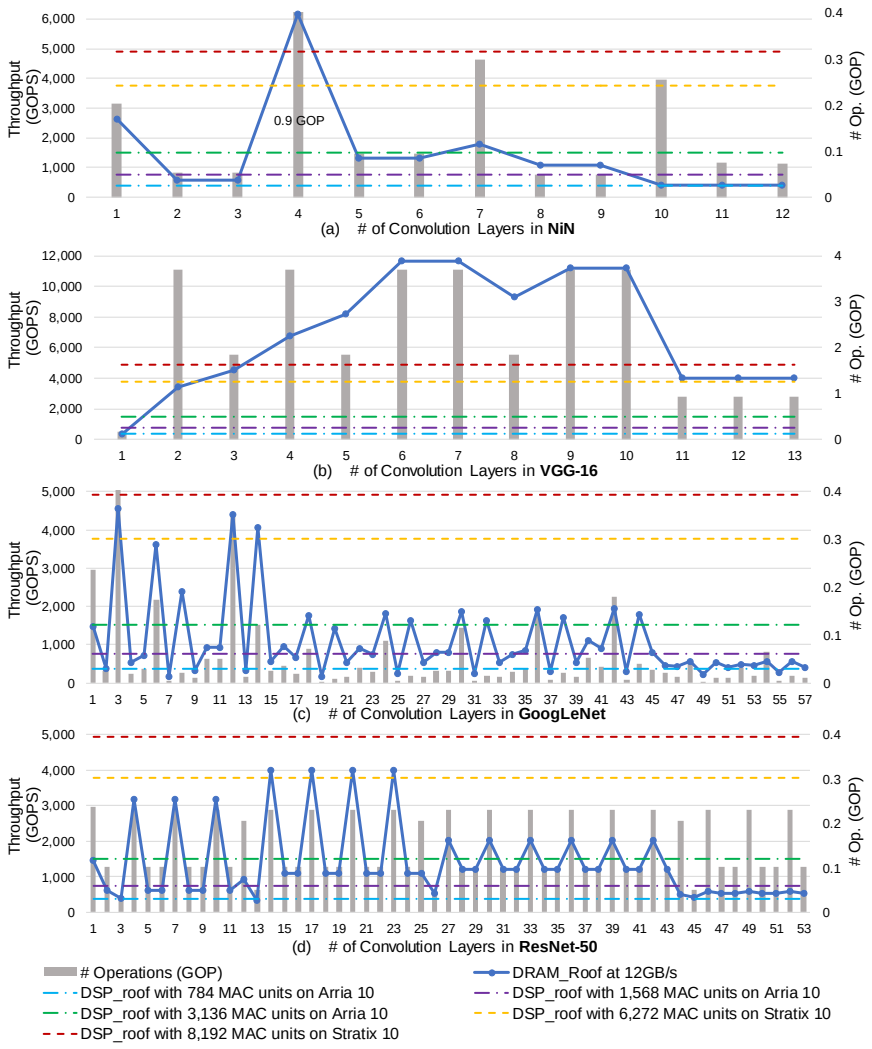

Figure 4.6: The *roof throughputs* are limited by computation resources and memory bandwidth at different layers of diverse CNN algorithms.

MAC unit so that the movements of partial sums are minimized.

The *local control logic* inside the Conv module receives the start flag signal from global control logic and controls the sequential computation of the four convolution loops. It is composed of multiple counters, which iterates from 0 to the dimensions of the feature and kernel maps, the number of input and output feature maps, respec-

tively. These parameters are read from configuration registers by the global control logic during runtime. By this means, for different Conv layers, the compiler only needs to generate associated parameters for each layer and maintain the same logic implementation. The combination of the counter values in local control logic generates the buffer read and write addresses. Instead of assigning individual Conv module for each Conv layer as in Ma *et al.* (2018b), the computing module in this work is reused by all the layers of the same type, thanks to the uniform mapping of PEs and shared local control logic.

The *data router* inside the Conv module is used to reshape the data form and continuously feed input pixels from buffers into MAC units. It is comprised of multiple data buses to handle the dataflow of different configurations of sliding strides and zero paddings for different Conv layers. The control logic governs the switch among different data buses for the corresponding layer. The data router can easily handle different kernel sizes without penalty of idle clock cycles and additional logic resources, which is realized by sequentially sliding the kernel window ($Pkx = Pky = 1$). The compiler only needs to change the iteration boundary of the counters inside the control logic for the corresponding kernel size.

There are $Pox \times Poy \times Pof$ parallel outputs from the MAC units, and they are serialized into Poy consecutive clock cycles to reduce the required number of bias adders and the data width of output buffers. The $Pox \times Pof$ outputs are further serialized to be $Pox\times$ #OUTBUF using multiplexers with output feature maps stacked in the output buffer as shown in Figure 4.7.

4.5.2 Pooling Modules (Pool)

Pooling layer (Pool) is commonly employed to reduce the dimensionality of feature maps by replacing pixels within a pooling sliding window by their maximum or

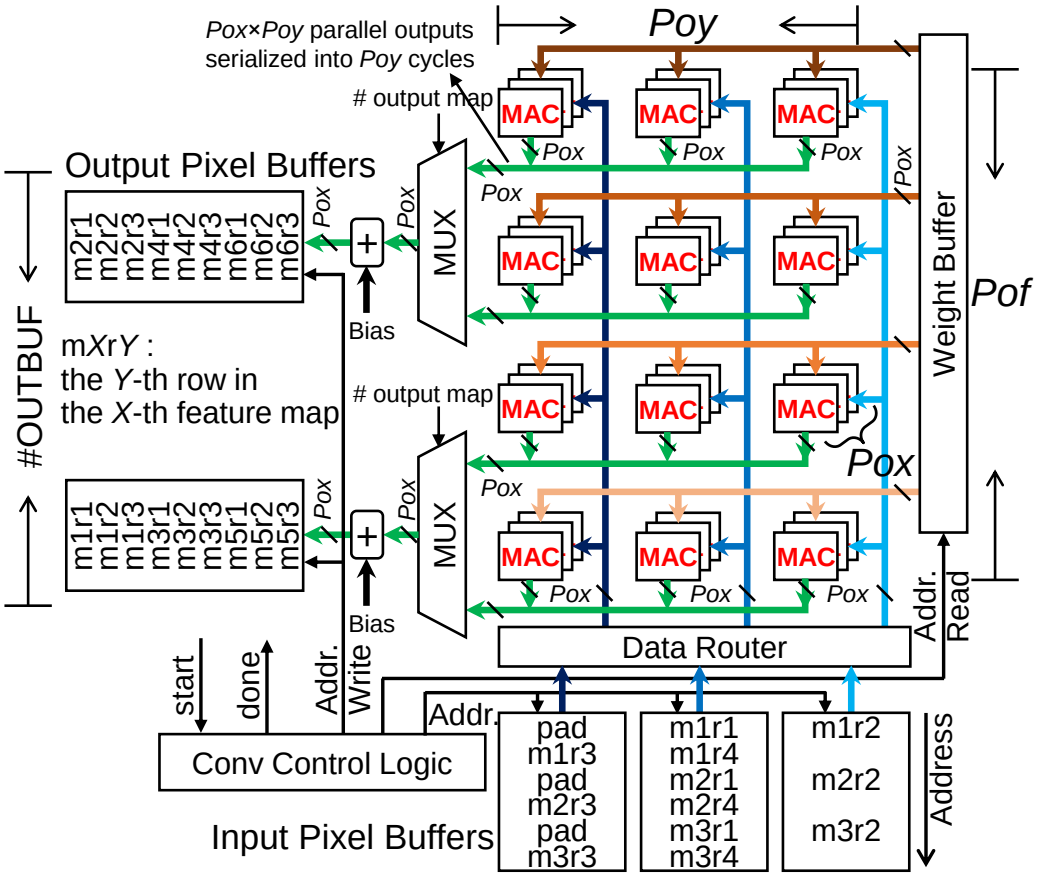

Figure 4.7: Convolution computing module (Conv) including buffers, where one MAC is comprised of one multiplier followed by an accumulator.

average value. Pool only needs pixels from its previous layer, so it can be treated as an affiliated layer to eliminate DRAM accesses as in Ma *et al.* (2018b) Ma *et al.* (2018a). However, the loop design variables of Pool must depend on its key layer and this dependency can worsen the design flexibility. If the key layer has $Toy <$ Noy, the pixels of one sliding window may be separated into two tiles, which demands the storage of pixels from the last tile and causes imbalance of pooling operations across tiles. Therefore, we treat Pool as a key layer to enable independent design configurations at the cost of DRAM access delay. Considering the small number of

Pool layers in CNNs, the overhead in the total latency is insignificant. Since average-pooling (Ave-Pool) is normally at the end, where Noy is small with Toy = Noy, it is not affected by the tiling problem. To that end, we still implement Ave-Pool as an affiliated layer by reading input data directly from output buffers of its previous layer.

The Max-Pool module is shown in Figure 4.8, which consists of local control logic, register arrays and PEs. The difference from the Ave-Pool module is that the input data are from output buffers. The counters inside the local control logic control the sliding within one feature map and across different feature maps, and generate the buffer read and write addresses. The Pool PEs ("POOL" component in Figure 4.8) are either comparators for Max-Pool or accumulators followed by constant coefficient multipliers for Ave-Pool. Pixels from one feature map are stored in one input buffer and processed by one row of PEs as illustrated in Figure 4.8. The different data storage pattern in the input buffers from that of Conv layer is handled by the data scatter module. The column size of PE array is constrained by the input buffer output width and the row size equals to the number of used input buffers, which can be adjusted before compilation. The data router in Pool is employed to ensure continuous feeding of pixels into PEs without idle cycles.

4.5.3 Batch Normalization and Scale (Bnorm)

Batch normalization followed by scale has been commonly used in recent CNN models He *et al.* (2016a) Szegedy *et al.* (2017), enabling fast training convergence. Their operations are depicted in (4.3) and (4.4):

$$y = \frac{x - bn0}{\sqrt{bn1}}, \tag{4.3}$$

$$z = sc0 \times y + sc1. \tag{4.4}$$

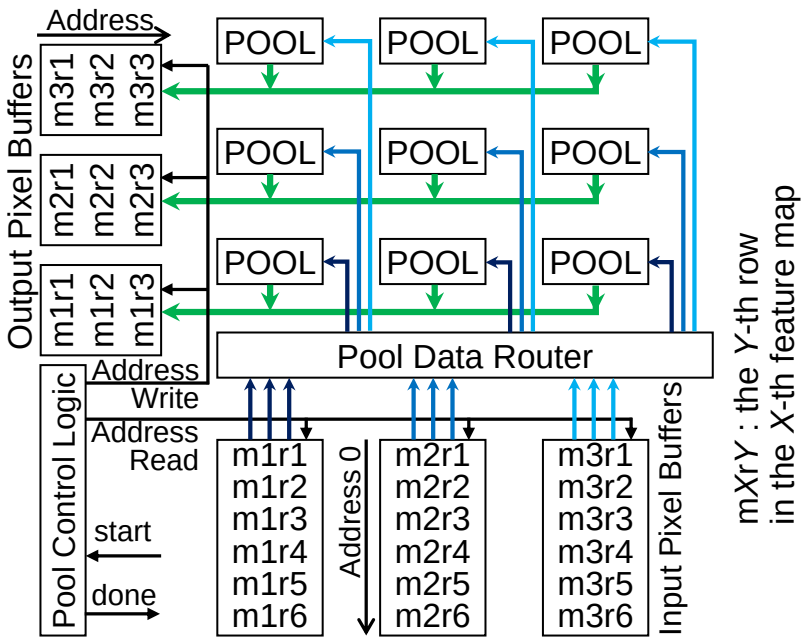

Figure 4.8: Max-pooling computing module (Max-Pool) including buffers.

During the inference process, $bn0$, $bn1$, $sc0$ and $sc1$ are all constants for each output feature map along Nof. Therefore, we can combine batch normalization with scale (Bnorm) to be a single equation:

$$z = A \times x + B. \tag{4.5}$$

where $A = sc0/\sqrt{bn1}$, and $B = sc1 - sc0 \times bn0/\sqrt{bn1}$. However, (4.5) still requires multipliers and adders that are expensive. To further save the computation resources, we continue to merge Bnorm with its preceding Conv layer. The convolution operation can be briefly expressed as (4.6):

$$x(no) = \sum_{ni=1}^{\mathrm{Nif}\times\mathrm{Nky}\times\mathrm{Nkx}} p(ni) \times w(ni, no) + bias(no), \quad no \in [1, \mathrm{Nof}]. \tag{4.6}$$

where $p(ni)$ is the input pixel and $w(ni, no)$ is the kernel weight, and the Conv output, e.g. $x(no)$, is the input to Bnorm in (4.5). After applying (4.6) to (4.5), we have:

$$z(no) = \sum_{ni=1}^{\text{Nif}\times\text{Nky}\times\text{Nkx}} p(ni) \times A(no) \times w(ni, no) \\ + A(no) \times bias(no) + B(no), no \in [1, \text{Nof}]. \tag{4.7}$$

By this mean, the Conv layer merged with Bnorm has new weights as $A(no)\times w(ni, no)$ and new biases as $A(no) \times bias(no) + B(no)$, with $no \in [1, \text{Nof}]$. Then, we can get rid of the Bnorm computations during inference, and the new weights and biases of Conv are pre-computed off-line to replace the original data. Therefore, there is no Bnorm module in Figure 4.3.

4.5.4 Element Wise (Eltwise)

The Eltwise layer performs element-wise addition to connect two branches of layers in ResNet CNNs as shown in Figure 1.1. As discussed in Section 4.3, we serially compute the two branches. Eltwise is treated as an affiliated layer to the key Conv layer in one branch and the other branch is computed first.

Eltwise is performed after its previous layer in the same branch has stored all the results into the output buffers. Then, the pixels from the other branch are read from DRAM and written into the input pixel buffers. Subsequently, the pixels from the two branches are element-wise added by the adders and finally stored back into the output pixel buffers, as illustrated in Figure 4.9. The output buffers are implemented as dual-port RAMs so that the adder results can be written back to the output buffers at their addends' original locations without using additional buffers. A few pipeline stages are introduced in the adders to avoid the conflict of writing and reading at the same output buffer address.

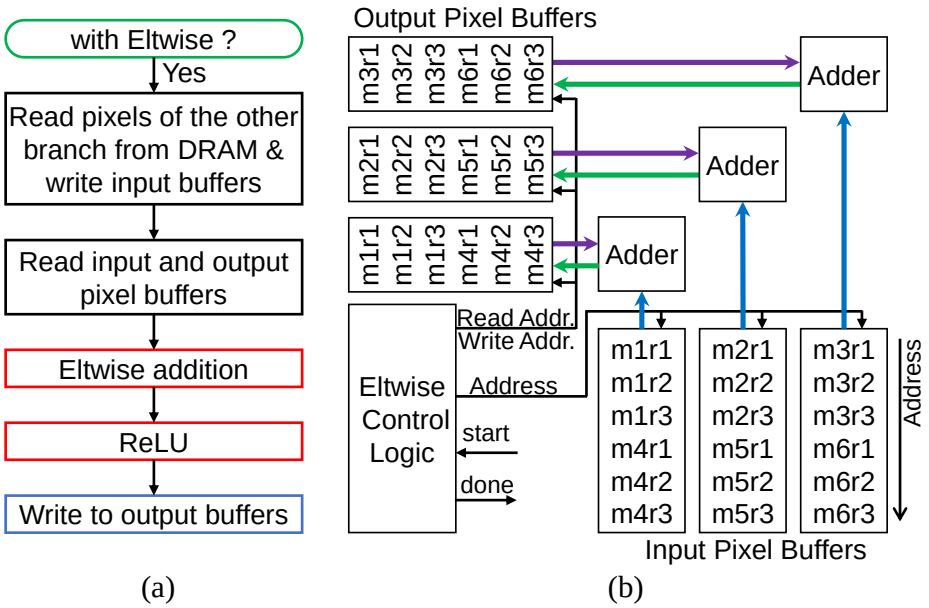

Figure 4.9: (a) Eltwise exectuion schedule (b) Eltwise module architecture.

4.5.5 Concat Layer

The Concat layer is used to concatenate the outputs of multiple layers together as shown in Figure 1.1. In this work, we assume the concatenation is only along multiple channels and all the input layers must have the same feature map sizes (Nix × Niy), which is the case for GoogLeNet Szegedy *et al.* (2015) and Inception Szegedy *et al.* (2017). If the inputs of one layer is from Concat, the compiler generates DMA descriptors that control DMA to read multiple layers of the Concat from different DRAM addresses as the inputs. Since there is no computation in Concat, it does not add overhead to the hardware.

4.5.6 Fully-connected (FC)

The FC layer can be treated as a special form of Conv with kernel size as Nkx × Nky = 1 × 1 and feature map size as Nox × Noy = 1 × 1. As the kernel weights are not

shared by pixels of one feature map, FC layer normally has a large volume of weights but with only a few operations, which makes FC layers memory intensive. Therefore, FC layers reuse the weight buffers with Conv layers, and the dual buffer technique is still used to overlap the memory delay with computation, which improves the FC latency especially for VGG implementation with heavy FC layers. The parallel computation of FC matrix-vector multiplication is only employed across different output feature maps such that only one pixel is multiplied with multiple weights simultaneously, and the MAC units in Conv are reused for FC layers.

4.6 Experimental Results

4.6.1 Experimental Setup

The proposed CNN compilation methodology is demonstrated by accelerating the inference process of both conventional CNNs, e.g. NiN and VGG, and complex DAG form CNNs, e.g. GoogLeNet and ResNet, on two Intel FPGAs. The two Intel FPGAs, e.g. Arria 10 GX 1150 / Stratix 10 GX 2800 FPGA, consist of 427K/933K adaptive logic modules (ALM), 3,036/11,520 fixed-point 18-bit $\times$ 18-bit DSP blocks, and 2,713/11,721 M20K BRAM blocks, where each M20K BRAM exhibits 20 Kbit storage. The underlying FPGA boards for Arria 10 and Stratix 10 are Nallatech 385A and Stratix 10 GX FPGA Development Kit, respectively, and both are equipped with DDR3 DRAM with theoretical peak memory bandwidth of 16.9 GB/s. The compiled Verilog scripts are synthesized by Quartus Prime. The fixed point data representation is employed by the compiler with dynamic quantization, which dynamically adjusts the decimal point according to the ranges of data values in different layers to fully utilize the existing data width Guo *et al.* (2018) Ma *et al.* (2018a). The data precision can be tuned to trade classification accuracy for hardware utilization and throughput.

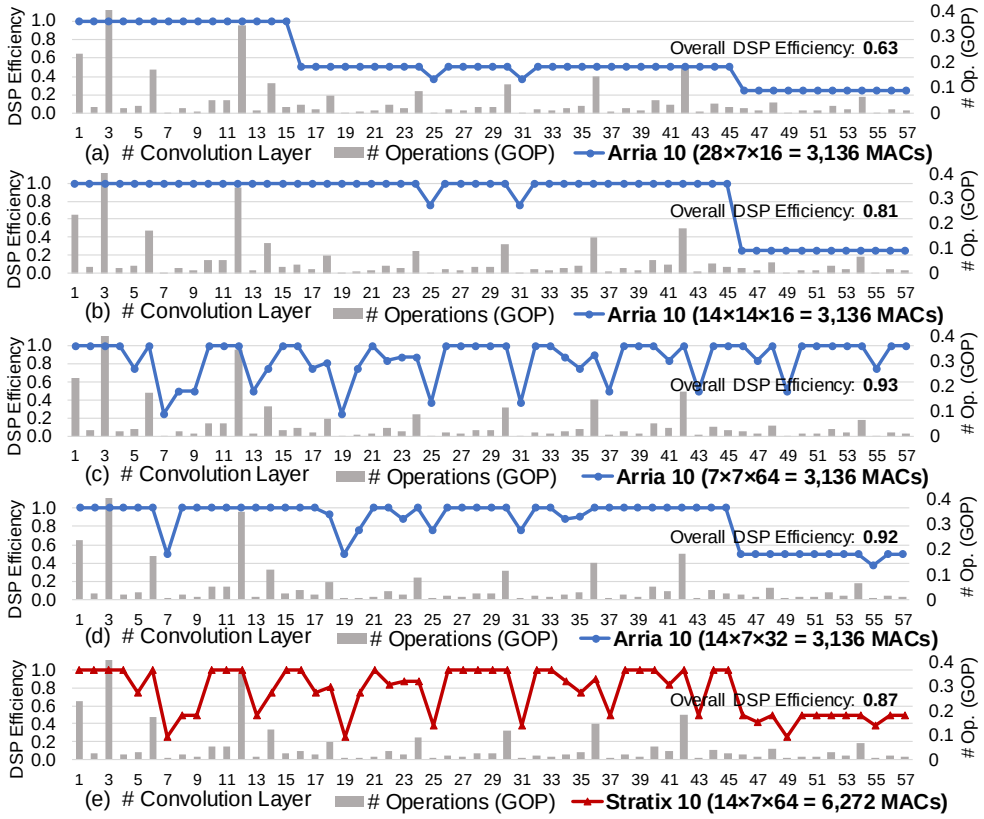

Figure 4.10: The DSP efficiency of different convolution layers in GoogLeNet is shown to measure the degree of matching between loop dimensions and loop unrolling ($Pox \times Poy \times Pof$), where (a)(b)(c)(d) have the same size of loop unrolling (= 3,136) but with different shapes, and (e) has larger loop unrolling size with 6,272 MAC units.

4.6.2 Parallel Computation Efficiency

Considering that the DSP blocks in Arria 10 and Stratix 10 can implement 3,036 and 11,520 fixed-point multipliers, respectively, the maximum number of MAC units on the two FPGAs can be around 3,000 and 11,000, respectively. To achieve better performance with higher parallelism, we attempt to maximize the usage of DSP blocks for the MAC operations. Based on the optimized acceleration strategy, the parallel

or unrolled loop computations are within one feature map ($Pox \times Poy$) and across multiple output channels (Pof). Since the feature map sizes (Nox × Noy) and the number of output channels (Nof) vary significantly across different layers in different CNN algorithms, the loop unrolling degree and shape may not perfectly match the feature map size and dimension, which causes inefficient utilization of DSP blocks or MAC units. Therefore, the DSP efficiency Wei *et al.* (2017) is defined to measure how well the parallel computation scheme matches the convolution loop dimension:

$$DSP\_efficiency = \frac{\#\quad effective\quad ops.}{\#\quad actual\quad performed\quad ops.}. \tag{4.8}$$

The DSP efficiency of different convolution layers is shown in Figure 6.4 using GoogLeNet as an example. Although Figure 6.4 (a)(b)(c)(d) have the same number of parallel MAC units ($Pox \times Poy \times Pof$ = 3,136) on Arria 10, their loop unrolling shape is different, which results in significant difference of the overall DSP efficiency from 0.63 to 0.93. The first several layers of GoogLeNet have large feature map sizes, e.g. 114×114 and 57×57, so that the loop unrolling sizes, e.g. 28×7 and 14×14, can be easily fit into the feature maps. The layers at the end has small feature map sizes, e.g. 14×14 and 7×7, which leads to DSP efficiency degradation except for Figure 6.4(c) with small $Pox \times Poy = 7 \times 7$. However, GoogLeNet still has layers with small number of output channels, e.g. 16 and 32, in the middle, which hurts the DSP efficiency of Figure 6.4(c) with large $Pof = 64$. Finally, Figure 6.4(c) and Figure 6.4(d) show similar overall DSP efficiency, and they are better than the other unrolling scenarios. Stratix 10 in Figure 6.4(e) has larger parallel degrees ($= 14 \times 7 \times 64$) than Arria 10, which makes it more difficult to exactly match the loop dimensions of all the layers and results in lower DSP efficiency.

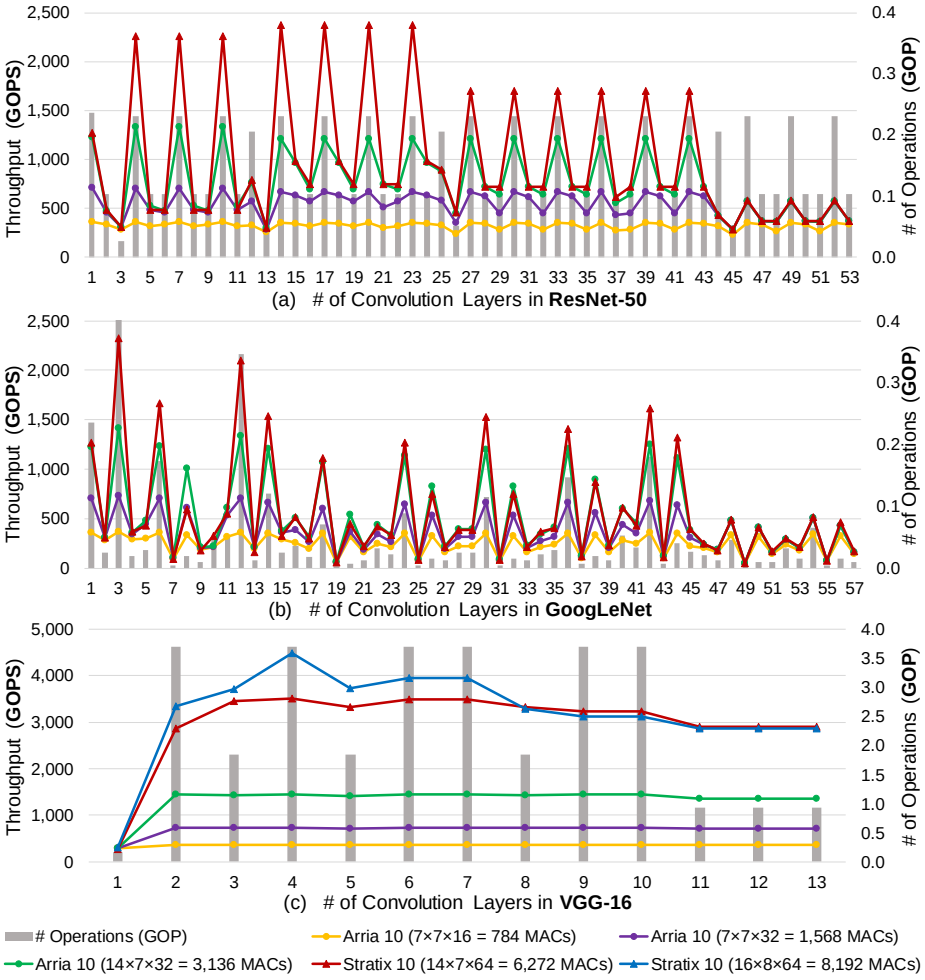

Figure 4.11: The throughput of each convolution layer in ResNet-50, GoogLeNet and VGG-16 with different number of MAC units on Arria 10 (240MHz) and Stratix 10 (300MHz).

4.6.3 Performance Analysis

The throughput of the CNN accelerator is collectively determined by the employed computation resources and memory bandwidth as discussed in Section 4.4.5, as well as the DSP efficiency, the number of external memory accesses, and the overlapping of computation and memory transactions. The throughput of each convolution layer in

ResNet-50, GoogLeNet, and VGG-16 is shown in Figure 6.5 with different number of MAC units on Arria 10 (running at 240 MHz) and Stratix 10 (running at 300 MHz). If the memory bandwidth is unlimited, the shape of the throughput curve should well match their corresponding DSP efficiency curve. However, with limited memory bandwidth, layers with small number of operations or small CTC ratios tend to be memory bounded, e.g. Conv #1 in VGG-16, Conv #7 in GoogLeNet and Conv #3 in ResNet-50 in Figure 6.5. With the increased number of MAC units, the design is more likely to be memory bounded with the same memory bandwidth, which limits further improvement of throughput by using more MAC units. As expected in Section 4.4.5, layers in VGG-16 have large CTC on Arria 10 and Stratix 10, whose throughputs can be significantly improved with the increase of MAC units. On the contrary, a lot of layers in ResNet and GoogLeNet are memory bounded, especially for Stratix 10, which limits the additional improvement of throughput on Stratix 10.

As mentioned in Section 4.6.2, even if the number of MAC units is the same, the different loop unrolling shapes may considerably impact the DSP efficiency, which will further affect the performance. The effect of different loop unrolling shapes on the throughput of different CNNs is shown in Figure 4.12 on Arria 10 with 3,136 MAC units. Although the loop unrolling of $14 \times 14 \times 16$ has worse DSP efficiency than $7 \times 7 \times 64$ for GoogLeNet in Figure 6.4 resulting in longer computation latency, the throughput of $14 \times 14 \times 16$ is higher than that of $7 \times 7 \times 64$ in Figure 4.12, which means $14 \times 14 \times 16$ of GoogLeNet allows better overlapping of computation and memory communication that overcompensates its longer computation time. The loop unrolling configuration of $14 \times 7 \times 32$ shows supreme throughput than other configurations for all the CNNs in Figure 4.12, thus we take it as our optimal choice for the Arria 10 implementation. The normalized convolution throughputs (Conv GOPS / DSP) are shown in Fig. 4.13(b) to measure the performance provided by

a single DPS block or MAC unit, which tend to decrease with more MAC units as the throughputs are saturated due to the lower DSP efficiency and limited memory bandwidth. VGG-16 exhibits higher normalized throughputs than other algorithms due to the higher CTC ratio to benefit more from the increase of DSP blocks.

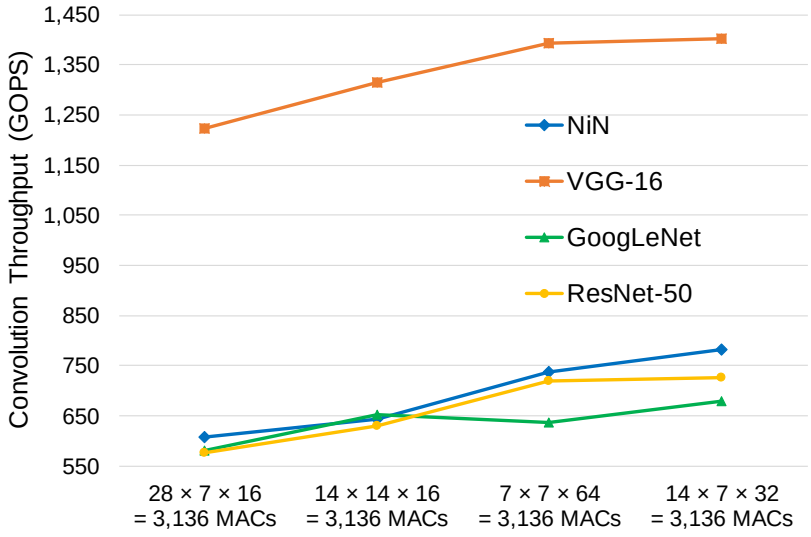

Figure 4.12: The convolution throughput of different CNNs on Arria 10 with the same number of MAC units is affected by the shape of loop unrolling ($Pox \times Poy \times Pof$).

The convolution throughputs of different CNNs on Arria 10 and Stratix 10 with different number of MAC units are shown in Figure 4.13. As mentioned before, most layers in VGG-16 have large CTC ratios that makes them more likely to be computation bounded, thus the throughput improvement of VGG-16 can benefit more from the increase of MAC units than the other three CNNs. The implementations of NiN, GoogLeNet and ResNet with 6,272 MAC units on Stratix 10 are already memory bounded so that more MAC units can only result in negligible throughput improvement, meanwhile more hardware resources are needed. If we target at smaller FPGA devices with less computation resources, e.g. DSP and logic, the compiler is

scalable to decrease the number of MAC units by assigning smaller loop unrolling size to reduce the resource requirements at the cost of lower performance.

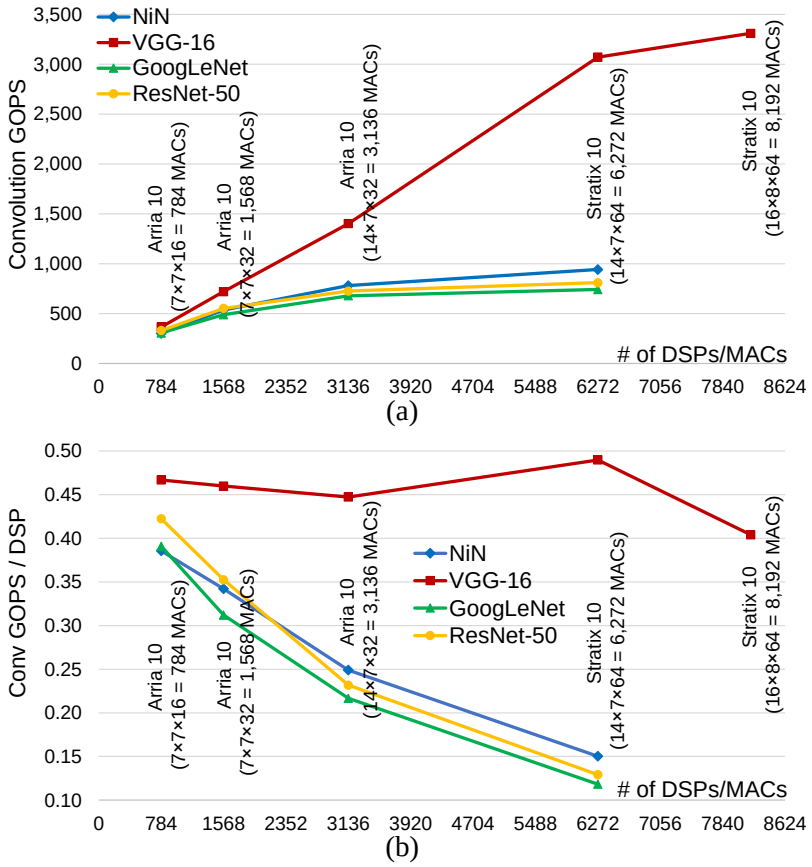

Figure 4.13: The compiler is scalable to change the number of MAC units ($Pox \times Poy \times Pof$) to trade the throughput for resource usage, e.g. DSP blocks. The increasing of throughputs with more MAC units are saturating due to lower DSP efficiency and limited memory bandwidth. (b) With the increased number of DSPs, the convolution throughputs normalized to one DSP (Conv GOPS / DSP) tend to decrease due to the saturation of throughputs.

Table 4.1: CNN Accelerators on Arria 10 and Stratix 10 FPGAs (Batch Size = 1)

CNN	NiN	VGG-16	GoogLeNet	ResNet-50	ResNet-152
# Oper. (GOP)	2.2	30.95	3.18	7.74	22.62
# of Parameters	7.59 M	138.3 M	6.07 M	25.5 M	60.4 M
Precision (fixed)	16 bit	8/16 bit	16 bit	16 bit	16 bit
FPGA / Tech.	Intel Arria 10 GX 1150 / 20 nm				
Clock	240 MHz	240 MHz	240 MHz	240 MHz	240 MHz
$Pox \times Poy \times Pof$	$14 \times 7 \times 32$	$14 \times 7 \times 32$	$14 \times 7 \times 32$	$14 \times 7 \times 32$	$14 \times 7 \times 32$
DSP Blocks	3,036 (100%)	3,036 (100%)	3,036 (100%)	3,036 (100%)	3,036 (100%)
Logic (ALMs)	256K (60%)	208K (49%)	277K (65%)	286K (67%)	335K (78%)
BRAM (M20K)	1,605 (59%)	2,319 (85%)	1,849 (68%)	2,356 (87%)	2,692 (99%)
Delay/Image (ms)	3.01	31.97	6.05	12.87	32.37
Overall GOPS	732.36	968.03	524.98	599.61	697.09
GOPS/DSP	0.24	0.32	0.17	0.20	0.23
FPGA / Tech.	Intel Stratix 10 GX 2800 / 14 nm				
Clock	300 MHz	300 MHz	300 MHz	300 MHz	300 MHz
$Pox \times Poy \times Pof$	$14 \times 7 \times 64$	$16 \times 8 \times 64$	$14 \times 7 \times 64$	$14 \times 7 \times 64$	$14 \times 7 \times 64$
DSP Blocks	6,304 (55%)	8,216 (71%)	6,304 (55%)	6,304 (55%)	6,304 (55%)
Logic (ALMs)	487K (52%)	469K (50%)	528K (57%)	559K (60%)	623K (67%)
BRAM (M20K)	1,915 (16%)	2,421 (21%)	1,949 (17%)	3,014 (26%)	3,350 (29%)
Delay/Image (ms)	2.56	19.29	5.70	11.85	28.59
Overall GOPS	858.66	1604.57	557.08	651.49	789.44
GOPS/DSP	0.14	0.20	0.09	0.10	0.13

4.6.4 Results of the CNN Inference Accelerator

The specifications and performance of the proposed compiler configured CNN FPGA accelerators are compared in Table 4.1. As discussed before, although Stratix 10 provides $> 3.3\times$ higher computation capability than Arria 10, the overall throughput improvements of Stratix 10 over Arria 10 are from $1.06\times$ to $1.66\times$ due to the lower DSP efficiency and limited external memory bandwidth, which considerably reduce the normalized throughputs (GOPS/DSP) of Stratix 10. Suffered from heavy FC layers, which are memory bounded, the overall throughput of VGG-16 on Arria 10/Stratix 10 (968/1,604 GOPS) is much lower than the convolution throughput (1,402/3,309 GOPS), respectively. The latency improvements brought by dual buffer structure is shown in Figure 4.14. Since the computation and the memory transaction cannot be perfectly fully overlapped as mentioned in Section 4.4.4, the actual total latency is larger than the theoretical minimum latency, which equals to the larger one of computation delay and DRAM delay. As the Stratix 10 FPGA board has only one DRAM bank, we also keep using one DRAM bank for the Arria 10 implementation for comparison purposes in this work. Therefore, the throughput of ResNet on Arria 10 is lower than that in Ma *et al.* (2017a) using two DRAM banks, even though the dual buffer structure is used in this work. If the Arria 10 implementations in Ma *et al.* (2017a) also use one DRAM bank, the throughputs of ResNet-50 and ResNet-152 could be decreased to 440 GOPS and 530 GOPS, which are $1.36\times$ and $1.32\times$ worse than this work, respectively. Although the external memory bandwidth in this work is only half of that in Ma *et al.* (2017a), the throughputs of NiN and VGG-16 are still $1.25\times$ and $1.34\times$ higher than Ma *et al.* (2017a), respectively, mainly due to the dual buffer structure, higher frequency and lower precision of weights in VGG-16. Despite of smaller loop tiling sizes used in this work, the on-chip memory usage of M20K on

Arria 10 is still higher than Ma *et al.* (2017a) due to the dual buffer structure, which directly doubles the M20K consumption of buffers.

Aimed at deep CNNs, our compiler stores all the weights and intermediate pixel results in DRAM by default. Considering current trends towards compressed CNNs with dramatically reduced data bit-width and small CNNs for simpler applications, it would be possible to fit the entire CNN model into FPGA on-chip BRAM. The potential modification of our compiler is to connect the DMA engine with a large enough BRAM instead of DRAM serving as the global memory, while retaining the computing architecture the same. With decreased data size and precision, more MAC units, higher frequency and less memory access delay could be possible to obtain higher throughput.

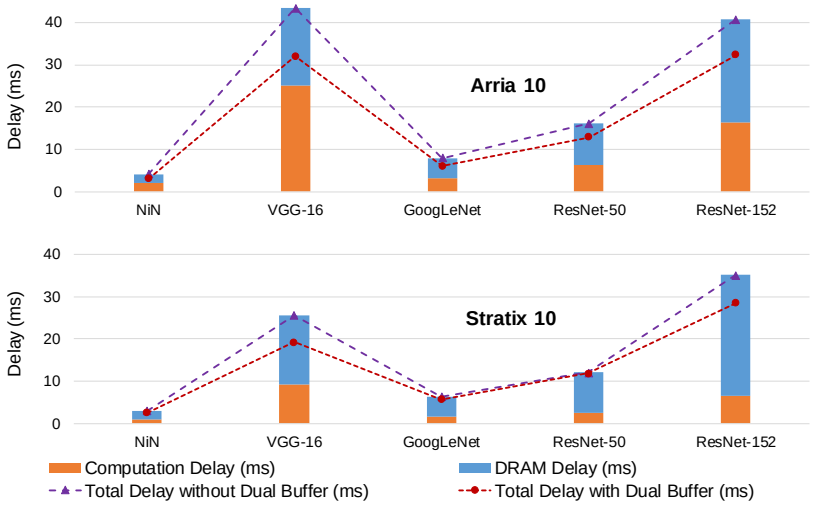

Figure 4.14: The dual buffer structure is used to overlap compuation delay with DRAM dealy to reduce the overall total delay.

4.6.5 Comparision with Prior Works

GPUs have been widely used to accelerate the training and inference tasks of CNN algorithms, which are realized by thousands of parallel processing cores, high operating clock frequencies at GHz level, and large memory bandwith of hundreds of GB/s. However, the power consumption of high performance GPUs is too high (>150W) for power constrained applications Guo *et al.* (2018)Zhang *et al.* (2016a). Furthermore, GPUs are best suited to process large batches of images together to fully utilize all the resources and realize high throughput (>1TFLOPS) at the cost of longer latency per image, which does not benefit the latency-critical applications, e.g. autonomous drive, that require real-time recognition results. On the other hand, numerous hardware accelerators based on application specific integrated circuits (ASICs) are recently developed achieving impressively high energy efficiency, e.g. up to 10 TOPS/W with 4-bit precision in Moons *et al.* (2017) or even higher for fixed custom accelerators with binary precision Bankman *et al.* (2018). However, CNNs with binary precision often incur accuracy loss, the ASIC based accelerators are too specific to efficiently handle various CNN algorithms, and the long development time of ASIC makes it difficult to catch up with the rapid evolution of CNN algorithms.

Benefited from the high reconfigurability and the freedom to customize the architecture, FPGAs have gained increasing popularity and there have been several works on automatically generating FPGA accelerators for CNN algorithms Ma *et al.* (2018b) Wang *et al.* (2016) Guo *et al.* (2018) Sharma *et al.* (2016) Zhang *et al.* (2016a) Venieris and Bouganis (2016) Guan *et al.* (2016) Wei *et al.* (2017) Zeng *et al.* (2018). The performance and hardware utilization of these related state-of-the-art works are listed in Table 4.2. Compared with previous works, our RTL compiler exhibits higher flexibility by handling not only conventional CNNs but also highly

Table 4.2: Related Works on Automated FPGA Accelerators

	Wang *et al.* (2016)	Guo *et al.* (2018)	Zhang *et al.* (2016a)	Zhang *et al.* (2016a)	Guan *et al.* (2016)	Guan *et al.* (2016)	Wei *et al.* (2017)	Zeng *et al.* (2018)
FPGA	Zynq 7045	Zynq XC7Z045	Ultrascale KU060	Virtex7 690t	Stratix V GSMD5	Stratix V GSMD5	Arria 10 GT 1150	Stratix V GXA7
Tech.	28 nm	28nm	20 nm	28 nm	28 nm	28 nm	20 nm	28 nm
Clock	100 MHz	150 MHz	200 MHz	150 MHz	150 MHz	150 MHz	232 MHz	200 MHz
CNN	NiN	VGG-16	VGG-16	VGG-16	VGG-16	ResNet-152	VGG-16	VGG-16
Precision	fixed	16 bit	16 bit	16 bit	16 bit	16 bit	8-16 bit	16 bit
DSP[a]	-	780 (87%)	1,058 (38%)	2,833 (78%)	1,036 (65%)	-	1,500 (99%)	256 (100%)
Logic[b]	-	183K (84%)	100K (31%)	300K (81%)	42.3K (25%)	-	313K (73%)	107K (46%)
RAM[c]	-	486 (89%)	-	-	-	-	1,668 (61%)	1,377 (73%)
Memory BW	12.8 GB/s	12.8 GB/s	12.8 GB/s	14.9 GB/s	-	-	19 GB/s	5 GB/s
Delay[d]	52 ms	-	-	-	-	-	26.85 ms	-
GOPS	-	137	266	354	364.36	226.47	1,171.3	669.1
GOPS/DSP	-	0.18	0.25	0.13	0.18	0.11	0.39	1.31

[a] Xilinx FPGAs in DSP slices and Intel FPGAs in DSP blocks.

[b] Xilinx FPGAs in LUTs and Intel FPGAs in ALMs.

[c] Xilinx FPGAs in BRAMs (36Kb) and Intel FPGAs in M20Ks (20Kb).

[d] Latency per image with batch size = 1. References without latency reported may use batch size > 1 to enhance throughput.

complex and irregular CNNs, e.g. GoogLeNet and ResNet, through reconfigurable execution schedule, on two different scale FPGAs, e.g. Arria 10 and Stratix 10. Our compiled CNN accelerators also significantly outperform prior works in terms of performance, which is achieved by hardware level optimization to accelerate convolution loops as in Ma *et al.* (2018a) with high hardware utilization and low data communication.

In Wang *et al.* (2016), AlexNet and NiN are implemented to evaluate their FPGA accelerator generators. Our NiN implementation on Arria 10 (20 nm and 3,036 DSP blocks) obtains $\sim 17.3\times$ speedup compared to [14], which needs over 50 ms runtime on Xilinx Zynq-7045 (28 nm and 900 DSP slices). Guo *et al.* (2018) presents a programmable and flexible CNN accelerator architecture, where the fixed 3×3 convolver used to parallel compute a kernel window could significantly degrade the DSP efficiency and throughput for irregular CNNs with varying kernel sizes, e.g. GoogLeNet and ResNet. Zhang *et al.* (2016a) proposes a HW/SW co-designed CNN FPGA accelerator based on high level synthesis (HLS). Our VGG-16 implementation on Arria 10 provides $2.7\times$ and $3.6\times$ overall throughput enhancement compared to Zhang *et al.* (2016a) using Virtex7 690t (28 nm and 3,600 DSP slices) and Ultrascale KU060 (20 nm and 2,760 DSP slices) FPGAs, respectively. Guan *et al.* (2016) proposes FP-DNN framework to automatically generate FPGA hardware to accelerate DNN with RTL-HLS hybrid templates. Although the Stratix V GSMD5 (28 nm and 3,180 DSP blocks) used in Guan *et al.* (2016) has more DSP blocks than our Arria 10, our accelerator on Arria 10 can achieve $3.1\times$ higher throughput for ResNet-152 by higher frequency and DSP utilization through the loop optimization technique Ma *et al.* (2018a). Wei *et al.* (2017) proposes an OpenCL-based automation flow to generate CNN design from high level C code to FPGA using systolic array architecture, which reduces the global PE interconnect fanout to achieve high frequency and resource

utilization. The VGG-16 implementation on Arria 10 in Wei *et al.* (2017) has 1.19× better latency than ours, probably because they have more efficient pipeline of dual buffering and can achieve higher memory bandwidth, e.g. 19 GB/s, which is especially important for memory bounded FC layers that comprise 28% of our Arria 10 VGG-16 total latency. However, Wei *et al.* (2017) only evaluated two conventional CNNs, e.g. AlexNet and VGG-16, which have relatively regular data shape and network structure. The framework proposed in Zeng *et al.* (2018) automatically generates CNN accelerators on a CPU+FPGA heterogeneous computing platform, i.e. Intel HARP, where only the convolution layers are performed on FPGA except the first convolution layer in AlexNet. By reducing the convolution operation complexity by about 3× in frequency domain through algorithm optimization, Zeng *et al.* (2018) can achieve high normalized throughputs, e.g. 1.31 GOPS/DSP, with a small number of DSPs, e.g. 512. In Venieris and Bouganis (2016), the fpgaConvNet framework for mapping a CNN onto a Zynq-7000 FPGA platform is designed based on HLS and evaluated on several relatively small CNN models, e.g. Convolutional Face Finder (CFF), LeNet-5 and MPCNN. The automatic CNN accelerator generation framework proposed in Sharma *et al.* (2016) is designed based on proposed instruction set architecture and accelerator template for both Intel and Xilinx FPGAs. The absolute performance numbers are not reported in Sharma *et al.* (2016) so that direct comparison cannot be made. Since our compiler generated accelerator is coded in Verilog, it is not difficult to implement on FPGAs from other vendors by changing FPGA board specified components, e.g. external memory controller.

4.7 Summary

In this chapter, a library-based RTL compiler is proposed to automatically generate customized FPGA accelerator for the inference task of a given CNN algorithm,

which enables high-level mapping of CNN from software to FPGA and keeps the benefit of low-level hardware optimization. An RTL library is developed to modularize the commonly used layers in CNNs with hand coded Verilog templates. These building block modules are built on the optimized acceleration strategy and configured by the hardware design variables to be scalable for different FPGAs. The topology of the given CNN is transformed into a DAG to configure the proposed execution schedule that controls runtime layer-by-layer serial processing. The flexibility of the proposed CNN compilation methodology is demonstrated on two Intel FPGAs, e.g. Arria 10 and Stratix 10, with different computing resources to implement both traditional CNNs, e.g. NiN and VGG-16, and complex CNNs, e.g. GoogLeNet and ResNets. Our compiled CNN accelerators on Stratix 10 exhibit superior performance compared to prior automation-based works by $> 1.4\times$ for various well-known CNNs algorithms.

Chapter 5

PERFORMANCE MODELING FOR CNN INFERENCE ON FPGA

5.1 Introduction

In this chapter, a high-level performance model is proposed to estimate the FPGA-based CNN inference accelerator throughput, on-chip buffer size and the number of external and on-chip memory accesses, which enables the efficient exploration of the design space to identify the performance bottleneck and obtain the optimal design configurations. The performance model is validated for a specific design strategy across a variety of CNN algorithms comparing with the on-board test results on two different FPGAs. The techniques that may further enhance the performance of our current design by improving the efficiency of DRAM transactions and PE utilization are also evaluated throughput the performance model.

The starting point of this work is a general system-level model of a CNN accelerator shown in Figure 2.5, which includes the external memory, on-chip buffers, and PEs. The hardware architectural parameters, e.g. buffer sizes, are determined by the design variables that control the loop unrolling and tiling. Combining the design constraints and the choices of the acceleration strategy, a more fine-grained performance model is built to achieve better prediction for a specific design implementation, e.g. the design strategy in Ma *et al.* (2018a). By this means, the proposed performance model makes it possible to identify the performance bottleneck and design limitations in the early development phase by exploring the design space through unrolling and tiling variables.

5.2 Coarse-grained Performance Model

In this section, a coarse-grained performance model of a general CNN accelerator that is independent of a specific acceleration strategy, is presented. Then, more detailed design choices and constraints (e.g. unrolling and tiling variable settings, memory storage pattern, and computation dataflow) are introduced to create a more precise and fine-grained model in the following sections. Table 5.1 lists the mainly used abbreviations and units in this chapter, which indicate the meaning of the variables discussed afterwards.

Table 5.1: List of Abbreviations and Units

Abbreviation	Description	Abbreviation	Description
Px	Pixel	Rd	Read
Wt	Weight	Wr	Write
Buf	Buffer	$InBuf$	Input Buffer
$WtBuf$	Weight Buffer	$OutBuf$	Output Buffer
BW	Bandwidth	$1T$	One Tile
Unit	Description	Unit	Description
bit / byte	Data Size	*word*	RAM Depth
ms	Delay Time	MHz	Frequency

5.2.1 Computation Latency

The number of multiplication operations per layer is Nm = Nif×Nkx×Nky×Nof×Nox×Noy. The number of PEs that determines the degree of parallel computations by unrolling is $Pm = Pif \times Pkx \times Pky \times Pof \times Pox \times Poy$. A similar reasoning is applied to determine the number of clock cycles for one buffered tile ($1T$) of convolution. This

is denoted by $\#cycles\_1T$, and is expressed as follows,

$$\#cycles\_1T = \left\lceil \frac{Tif}{Pif} \right\rceil \left\lceil \frac{Tkx}{Pkx} \right\rceil \left\lceil \frac{Tky}{Pky} \right\rceil \left\lceil \frac{Tof}{Pof} \right\rceil \left\lceil \frac{Tox}{Pox} \right\rceil \left\lceil \frac{Toy}{Poy} \right\rceil. \tag{5.1}$$

The number of tiles for one convolution layer is

$$\#tiles = \left\lceil \frac{\mathrm{Nif}}{Tif} \right\rceil \left\lceil \frac{\mathrm{Nkx}}{Tkx} \right\rceil \left\lceil \frac{\mathrm{Nky}}{Tky} \right\rceil \left\lceil \frac{\mathrm{Nof}}{Tof} \right\rceil \left\lceil \frac{\mathrm{Nox}}{Tox} \right\rceil \left\lceil \frac{\mathrm{Noy}}{Toy} \right\rceil. \tag{5.2}$$

The total number of computation clock cycles of one convolution (CV) layer is

$$\#cycles\_1CV = \#tiles \times \#cycles\_1T. \tag{5.3}$$

5.2.2 On-chip Buffer Size

Determined by the tiling variables, the input buffer ($InBuf$) size (bit) requirement to store one tile of input pixels is

$$bit\_InBuf = Tix \times Tiy \times Tif \times bit\_Px, \tag{5.4}$$

where bit_Px is the bit width of one pixel (Px). Similarly, the size (bit) requirement of weight buffer ($WtBuf$) to store one tile of weights is

$$bit\_WtBuf = Tkx \cdot Tky \cdot Tif \cdot Tof \cdot bit\_Wt, \tag{5.5}$$

where bit_Wt is the bit width of one weight (Wt). The output buffer ($OutBuf$) size (bit) requirement to store one tile of output pixels is

$$bit\_OutBuf = Tox \times Toy \times Tof \times bit\_Px. \tag{5.6}$$

The theoretical sizes of the input, weight and output buffers are the maximum possible values of bit_InBuf, bit_WtBuf and bit_OutBuf of all the convolution layers, respectively. In an actual implementation, the sizes of the buffers used may be larger than these values due to inefficient storage pattern and extra garbage data.

5.2.3 DRAM Access and Latency

In theory, the size of one tile of data read from or written to the external DRAM should be the same as the size of buffered data. Therefore, the size (bytes) of input pixels (Px) read (Rd) from DRAM for one convolution tile is $byte\_RdPx = bit\_InBuf/8$. The size (bytes) of one tile of weights (Wt) read from the DRAM is $byte\_RdWt = bit\_WtBuf/8$. The size (bytes) of one tile of output pixels written (Wr) to the DRAM is $byte\_WrPx = bit\_OutBuf/8$. The latency (milliseconds or ms) of DRAM transactions of one tile ($1T$) of data is determined by the size of DRAM access and the memory bandwidth. This is given by

$$ms\_DRAM\_1T = \frac{byte\_DRAM\_1T}{BW\_Memory \times 10^6}, \tag{5.7}$$

where BW_Memory is the external memory bandwidth (GByte/s), and $byte\_DRAM\_1T$ is the size of DRAM access of one tile, which can be $byte\_RdPx$, $byte\_RdWt$, or $byte\_WrPx$.

5.2.4 On-chip Buffer Access

The size (bits) of on-chip buffer access (bit_Buf_Access) is computed by multiplying the number of access clock cycles ($\#cycles\_Access$) with the total bit width of the corresponding buffers ($width\_Buf$).

$$bit\_Buf\_Access = \#cycles\_Access \times width\_Buf. \tag{5.8}$$

During computation, it is assumed that data are continuously read from input and weight buffers and the results are written into the output buffers every clock cycle. Then, to estimate the buffer access during computation, $\#cycles\_Access$ equals the number of computation cycles, and $width\_Buf$ can be the total bit width of input/weight/output buffers. The size (bits) of buffer access by DMA that writes

into input and weight buffers and reads from output buffers is the same as the size of external memory access. The data stored in the input or weight buffers may be read multiple times during computation, hence the size of data read from buffers may be larger than the size of data written into buffers from DRAM. Since each result is written into output buffers only once, the size of write and read operations of output buffers are the same.

5.3 Modeling of DRAM Access

In this section, more accurate models of the DRAM access are constructed by including the design constraints and the variables of loop acceleration described in Section 3.4.5.

5.3.1 Data Size of Convolution DRAM Access

The direct memory access (DMA) engine shown in Figure 2.5 is used to transfer data to and from off-chip DRAM. To achieve the maximum bandwidth, the data width of both the DMA (bit_DMA) and the DRAM controller (bit_DRAM) are set to be 512 bits.

Pox represents the number of pixels that are computed in parallel in each output feature map. For the feature map transfer, the number of groups of Pox pixels associated with one DMA address is then given by $\#PoxGroup = \lfloor bit\_DMA/(Pox \times bit\_Px) \rfloor$, where bit_Px is the bit width per pixel. The effective or actual DMA bandwidth (as a fraction of the maximum) is then given by

$$eff\_DMA\_Px = \frac{\#PoxGroup \times Pox \times bit\_Px}{bit\_DMA}. \tag{5.9}$$

For example, if $Pox = 7$, $bit\_DMA = 512$ and $bit\_Px = 16$, then there are $\#PoxGroup = 4$ groups of Pox pixels in one DMA address, and $4 \times 7 \times 16 = 448$ bits are

the effective number of bits out of the DMA bit width of 512 bits, resulting in $eff\_DMA\_Px = 0.875$.

The intermediate pixel results stored in DRAM are arranged row-by-row, map-by-map, and layer-by-layer. One convolution tile needs $Tix \times Tiy \times Tif$ input pixels. Then, the size (bytes) of the input pixels read (Rd) from the DRAM for one tile is

$$byte\_RdPx = \frac{Tix \times Tiy \times Tif \times bit\_Px}{eff\_DMA\_Px \times 8}. \tag{5.10}$$

Note that if $eff\_DMA\_Px < 1$, it implies more bytes are read than necessary, due to the alignment of data storage. Similarly, the size (bytes) of output pixels written (Wr) to DRAM for one convolution tile is

$$byte\_WrPx = \frac{Tox \times Toy \times Tof \times bit\_Px}{eff\_DMA\_Px \times 8}. \tag{5.11}$$

For convolution weights, the ratio of effective DRAM bandwidth to the maximum of reading weights from DRAM is

$$eff\_DMA\_Wt = \frac{\lfloor bit\_DMA/bit\_Wt \rfloor \times bit\_Wt}{bit\_DMA}. \tag{5.12}$$

The size (bytes) of input weights read from DRAM for one convolution tile is

$$byte\_RdWt = \frac{Tkx \cdot Tky \cdot Tif \cdot Tof \cdot bit\_Wt}{eff\_DMA\_Wt \times 8}. \tag{5.13}$$

5.3.2 DRAM Access Delay of One Tile (1T)

The data width of the DRAM controller interface to the FPGA is assumed to be bit_DRAM, running at frequency of MHz_DRAM. This means the theoretical maximum DRAM bandwidth (BW_DRAM in GB/s) is $(bit\_DRAM/8) \times (MHz\_DRAM/10^3)$, which is normally very difficult to sustain due to the non-contiguous DRAM access. For example, if $bit\_DRAM = 512$ bits, with MHz_DRAM

= 266 MHz, then $BW\_DRAM = (512/8) \times (266/10^3) = 17.0$ GB/s as the maximum DRAM bandwidth.

In the CNN acceleration system described in Ma *et al.* (2018a), DMA is operated at the same clock frequency as the CNN accelerator (i.e. $MHz\_Accelerator$) with read/write data-width (bit_DMA) of 512 bits. An asynchronous FIFO can be inserted between DMA and the DRAM controller to synchronize data across the two clock domains. Then, the DMA bandwidth (BW_DMA) is ($bit\_DMA/8$) $\times$ ($MHz\_Accelerator/10^3$). By this means, the bandwidth of the external memory is bounded by the effective bandwidth of both the DRAM controller and the DMA as $BW\_Memory = \min(BW\_DRAM, BW\_DMA)$, which is used in Equation (5.7) to calculate the DRAM latency.

The more accurate and specific DRAM access sizes of one tile ($byte\_DRAM\_1T$) are discussed in this section, including $byte\_RdPx$, $byte\_WrPx$, and $byte\_RdWt$. Then, we can use Equation (5.7) to compute their corresponding DRAM access delay (ms_DRAM_1T), e.g. ms_RdPx, ms_WrPx, and ms_RdWt, respectively.

5.3.3 DRAM Access of Other Layers

The DRAM access and performance of other layers, e.g. max-pooling, fully-connected (FC) and Eltwise, are also investigated and included in our performance model. Since the analysis process of theses layers are similar to the convolution layer, for simplicity, their detailed formulas used in the performance model are not presented.

The pixels of max-pooling layers are also transferred to and from the DRAM with loop tiling performed, depending on the adopted design choices Ma *et al.* (2018a)Ma *et al.* (2017a). For max-pooling, the calculation of the DRAM transfer sizes of input and output pixels are similar to $byte\_RdPx$ in Equation (5.10) and $byte\_WrPx$ in

Equation (5.11), respectively.

The weights of fully-connected (FC) layers are stored in DRAM in the same way as convolution, and reuse the same weight buffers. Since the intermediate results of FC layers are small (< 20KB), they are always kept in the on-chip RAMs.

The Eltwise layer performs element-wise summation of the output pixels of two convolution layers. We identify one convolution layer as the *key layer* Ma *et al.* (2017a), so that Eltwise is executed directly after its key layer, and this key layer is executed after the other convolution layer. Eltwise layer can directly read the outputs of its key layer, which are stored in the output buffers, without accessing DRAM. However, Eltwise layer also needs to read the outputs of the other convolution layer from DRAM as the output buffers were already refreshed. Therefore, the size of pixels read from DRAM for one Eltwise tile equals to $byte\_WrPx$ of its key convolution layer.

5.4 Modeling of Latency

5.4.1 Computation Delay (ms) of One Convolution Tile

Setting $Pif = Pkx = Pky = 1$, $Tif = \text{Nif}$, $Tkx = \text{Nky}$, $Tkx = \text{Nky}$, and $Tox = \text{Nox}$ as described in Section 3.4.5, Equation (5.1) can be written as

$$\#cycles\_1T = \text{Nif} \cdot \text{Nkx} \cdot \text{Nky} \cdot \left\lceil \frac{Tof}{Pof} \right\rceil \cdot \left\lceil \frac{\text{Nox}}{Pox} \right\rceil \cdot \left\lceil \frac{Toy}{Poy} \right\rceil. \tag{5.14}$$

Then, the computation delay (ms) of one convolution tile is

$$ms\_Compute = \frac{\#cycles\_1T}{MHz\_Accelerator \times 10^3}, \tag{5.15}$$

where $MHz\_Accelerator$ is the clock frequency of the accelerator in MHz. The number of tiles of one convolution layer ($\#tiles$) is $\lceil \text{Nof}/Tof \rceil \lceil \text{Noy}/Toy \rceil$ based on Equation (5.2) with $\text{Nif} = Tif$, $\text{Nkx} = Tkx$, $\text{Nky} = Tky$, and $\text{Nox} = Tox$ as described in Section 3.4.5.

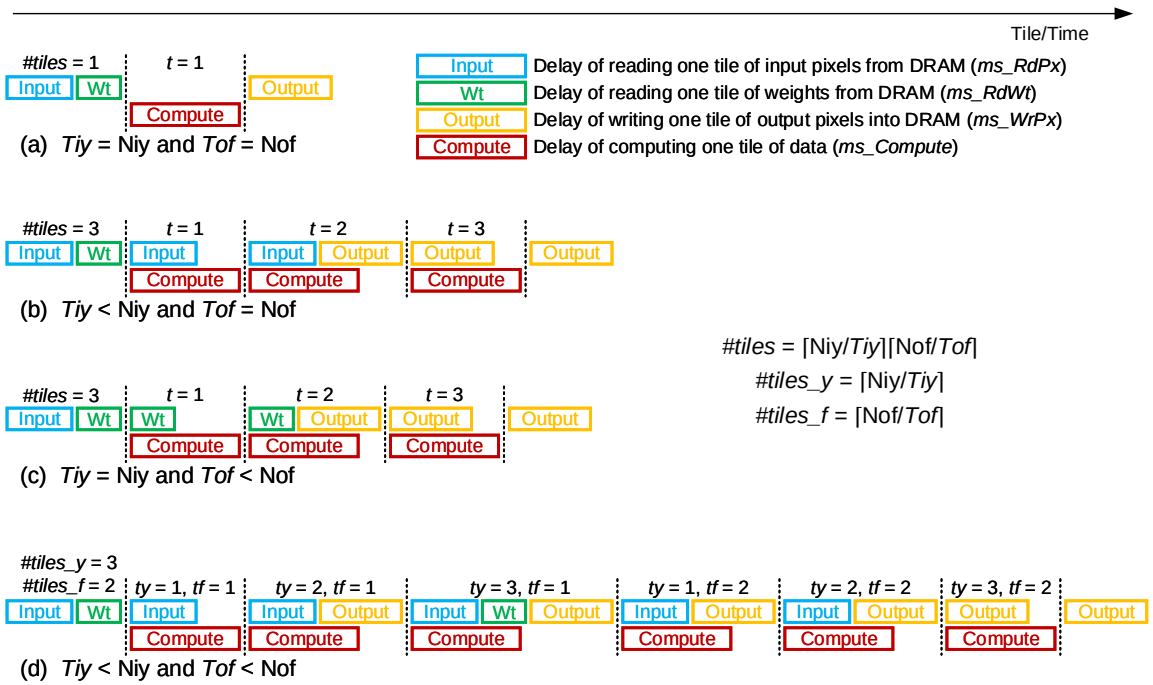

Figure 5.1: The tile-by-tile delay of one convolution layer, and the DRAM access delay is overlapped with the computation delay due to dual buffering technique. (a) Both inputs and weights fully buffered, (b) only weights fully buffered, (c) only inputs fully buffered, (d) neither inputs nor weights fully buffered.

5.4.2 Overall Delay (ms) of One Convolution Layer

With dual buffering technique, the DRAM access delay is overlapped with the computation delay to improve the performance Zhang *et al.* (2015)Wei *et al.* (2017). The overall tile-by-tile delay of one convolution layer is illustrated in Figure 5.1. Since the dual buffering pipeline is only within one layer with the current design choice, after the start of one layer and before the computation of the first tile, both the input pixels and weights (Wt) of one tile are first read from DRAM. This is shown as "Input+Wt" at the beginning of one layer in Figure 5.1. Similarly, after the completion of the last tile's computation, its output pixels are transferred back into

DRAM, which is shown as "Output" at the end in Figure 5.1. Therefore, for each convolution layer, the delay of transferring inputs of the first tile and outputs of the last tile cannot be overlapped with the computation, and this delay is denoted as

$$ms\_Mem = ms\_RdPx + ms\_RdWt + ms\_WrPx. \tag{5.16}$$

If the convolution layer has only one tile that is Tiy = Niy and Tof = Nof, there is no overlapping of memory transfer and computation as shown in Figure 5.1(a), and the delay of this tile (e.g. $t = 1$ in Figure 5.1(a)) is only determined by the computation delay as in Algorithm 1 (line 2).

If the convolution layer has multiple tiles and all its weights are fully buffered, i.e. $Tiy <$ Niy and Tof = Nof, then the weights only need to be read from DRAM once and can be reused by different tiles as illustrated in Figure 5.1(b). The procedure to estimate the delay of this convolution layer is summarized in Algorithm 1 (line 3 to line 12). The computation of the first tile (e.g. $t = 1$ in Figure 5.1(b)) is overlapped with fetching the input pixels of the next tile, and there is no DMA transfer of output pixels of the previous layer, thus the delay of this tile is determined by Algorithm 1 (line 6). The computation of the last tile (e.g. $t = 3$ in Figure 5.1(b)) is overlapped with transferring the output pixels of its previous tile, and its delay is calculated by Algorithm 1 (line 8). For the other tiles (e.g. $t = 2$ in Figure 5.1(b)), the communication with DRAM includes both reading input pixels and writing output pixels, and the delay of one tile is expressed by Algorithm 1 (line 10). The overall delay of this convolution layer is the sum of all the tiles as well as the DRAM access delay before the first tile and after the last tile, i.e. ms_Mem.

If the convolution layer has multiple tiles and all its pixels are fully buffered, i.e. Tiy = Niy and $Tof <$ Nof, then the pixels only need to be read from DRAM once and can be reused by different tiles as illustrated in Figure 5.1(c). Similarly, the procedure

Algorithm 1: Delay estimation of one convolution layer (ms_1CV), where $C = ms\_Compute$, $I = ms\_RdPx$, $W = ms\_RdWt$, and $O = ms\_WrPx$.

input : C, I, W, O, $\#tiles$, $\#tiles\_y$, $\#tiles\_f$
output: ms_1CV

if $Tiy =$ Niy and $Tof =$ Nof **then**
  $T[1] = C$
else if $Tiy <$ Niy and $Tof =$ Nof **then**
  for $t = 1$ **to** $\#tiles$ **do**
    if $t = 1$ **then**
      $T[t] = \max(C, I)$
    else if $t = \#tiles$ **then**
      $T[t] = \max(C, O)$
    else
      $T[t] = \max(C, I + O)$
    end
  end
else if $Tiy =$ Niy and $Tof <$ Nof **then**
  for $t = 1$ **to** $\#tiles$ **do**
    if $t = 1$ **then**
      $T[t] = \max(C, W)$
    else if $t = \#tiles$ **then**
      $T[t] = \max(C, O)$
    else
      $T[t] = \max(C, W + O)$
    end
  end
else
  for $tf = 1$ **to** $\#tiles\_f$ **do**
    for $ty = 1$ **to** $\#tiles\_y$ **do**
      $t = ty + (tf - 1) \times \#tiles\_y$;
      if $ty = 1$ and $tf = 1$ **then**
        $T[t] = \max(C, I)$
      else if $t = \#tiles$ **then**
        $T[t] = \max(C, O)$
      else if $ty = \#tiles\_y$ **then**
        $T[t] = \max(C, I + W + O)$
      else
        $T[t] = \max(C, I + O)$
      end
    end
  end
end
$ms\_1CV = \sum_{t=1}^{\#tiles} T[t] + ms\_Mem$

to estimate the delay of this convolution layer is summarized in Algorithm 1 (line 13 to line 22).

If neither the weights nor the pixels of the convolution layer can be fully buffered, i.e. $Tiy <$ Niy and $Tof <$ Nof, its pipeline schedule is shown in Figure 5.1(d) and the associated delay is estimated in Algorithm 1 (line 23 to line 37). In this case, either the pixels or the weights need to be re-fetched multiple times from the DRAM. In our current design, the input pixels are re-fetched and the weights only need to be read once. If the DRAM access requirement of input pixels is more than weights, we can also re-fetch weights instead and only read input pixels once by changing the DMA instructions and associated control logic. Before the computation, the first tile of weights are loaded and reused by the following consecutive $\#tiles\_y = \lceil \text{Niy}/Tiy \rceil$ tiles of pixels to perform convolution. Then, the next tile of weights are loaded and reused by the following $\#tiles\_y$ tiles of pixels. This process iterates by $\#tiles\_f = \lceil \text{Nof}/Tof \rceil$ times to complete the computation with all the $\#tiles\_f$ tiles of weights. By this means, the pixels are re-fetched by $\#tiles\_f$ times. A normal tile needs to read input pixels of the next tile from DRAM and write output pixels of the previous tile into DRAM, where the required weights are already loaded during the previous tile and reused. Therefore, the delay of a normal tile is estimated as in Algorithm 1 (line 34). As the first tile does not have a previous tile, there is no transfer of output pixels back to DRAM as in Algorithm 1 (line 28). For the last tile, there is no need to read input pixels for the next tile as in Algorithm 1 (line 30). When $\#tiles\_y$ tiles of weights are finished (e.g. $ty = 3$ and $tf = 1$ in Figure 5.1(d)), the new tile of weights are loaded from DRAM, and the DRAM access also includes the transfer of pixels as in Algorithm 1 (line 32).

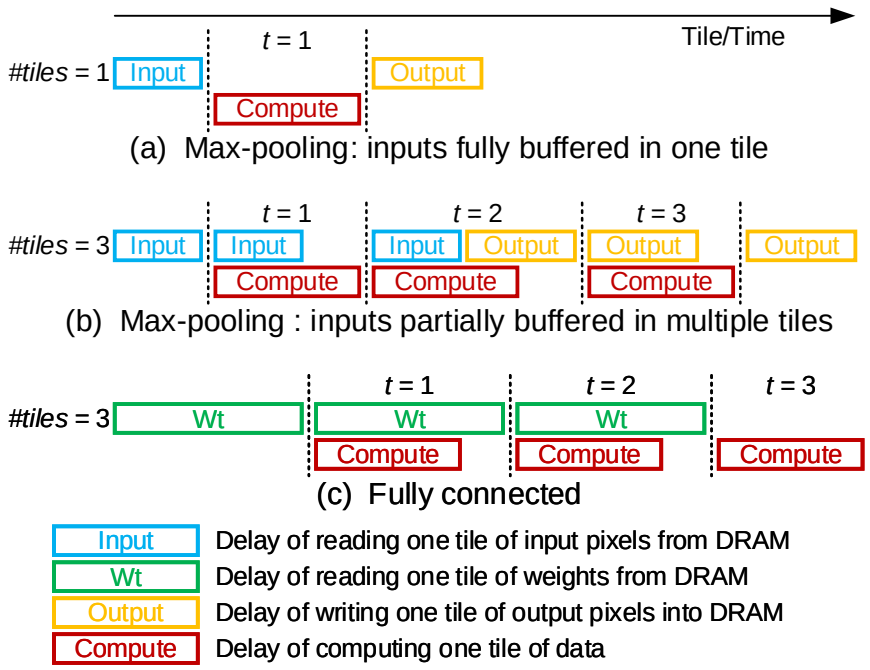

Figure 5.2: The tile-by-tile delay of one pooling/fully-connected layer, and the DRAM access delay is overlapped with the computation delay due to the dual buffering technique.

5.4.3 Delay Estimation of Other Layers

With dual buffering technique employed, the overall tile-by-tile process of one max-pooling layer is illustrated in Figure 5.2(a)(b), which is similar to the convolution layer except that pooling does not need weights. If the pooling layer has only one tile, which means the inputs of one pooling layer can be fully buffered, there is no overlapping between memory transfer and computation as shown in Figure 5.2(a). Figure 5.2(b) illustrates the dual buffering pipeline of one pooling layer with multiple tiles. Similar to Algorithm 1, we can compute the overall latency of max-pooling layers according to the tile-by-tile execution schedule, with the delay of max-pooling computation and DRAM access calculated similar to the convolution layer.

Figure 5.2(c) shows the pipeline schedule of FC layer, where weights are fetched

before the corresponding computation and no outputs are transferred back to DRAM. The storage format of FC weights in the weight buffer allows us to read Pof weights simultaneously every clock cycle to parallel compute Pof outputs. Then, the computation cycles of one FC tile equal to the depth of buffered FC weights. The overall delay of FC is bounded and determined by the computation delay or the DRAM access delay of weights.

The delay of Eltwise layer is comprised of the DRAM access delay of pixels from the second convolution layer and the computation time of element-wise addition. The same size of pixels of the two convolution layers are separately stored in the input and output buffers Ma *et al.* (2017a). Every clock cycle, pixels are continuously read from the input and output buffers and computed in parallel. The number of computation cycles of one Eltwise tile equals to the depth of its key convolution layer's results in the output buffers. Then, the overall delay of one Eltwise layer is the product of the delay of one Eltwise tile and the number of tiles of its key convolution layer.

5.5 Size Requirement of On-chip Memory

With the specific data storage pattern of buffers, we can more precisely calculate the required on-chip buffer sizes than the rough estimation in Section 5.2.2.

5.5.1 Size and Storage of Input Buffers

Figure 5.3 illustrates the proposed storage pattern of convolution input pixels, which benefits the dataflow of $Pox \times Poy$ pixels from buffers into MAC units Ma *et al.* (2018a). The width of one input buffer is determined by Pox to feed data for parallel computation of Pox pixels in one feature map row. The number of input buffers is determined by Poy to feed data for parallel computation of Poy multiple output rows. In Figure 5.3, $c(x)$ denotes one input pixel in the x-th column of a

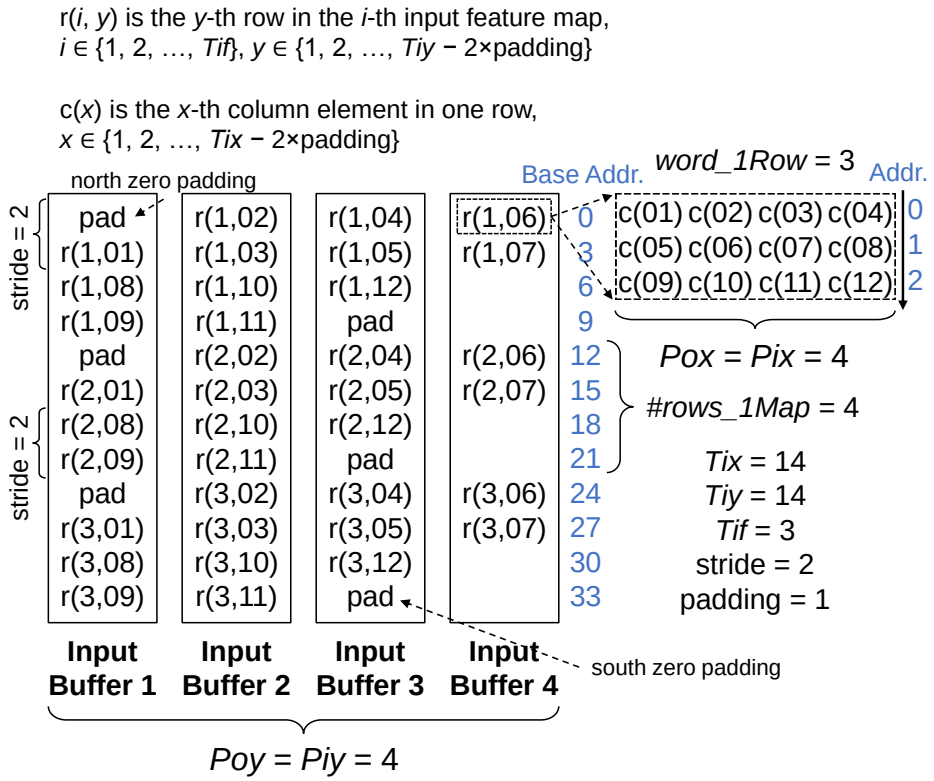

Figure 5.3: The convolution data storage pattern in the input pixel buffers.

certain row, where $x \in \{1, 2, \ldots, Tix - 2 \times padding\}$ and Tix includes both the east and west zero padding. The east and west zero paddings are not stored in buffers and instead they are masked out by control logic before loading into the MAC units. The number of addresses or words occupied by one row is

$$word\_1Row = \lceil (Tix - 2 \times padding)/Pox \rceil. \tag{5.17}$$

In Figure 5.3, $r(i, y)$ is the y-th row of the i-th input feature map, where $i \in \{1, 2, \ldots, Tif\}$ and $y \in \{1, 2, \ldots, Tiy\}$. The Tiy rows of one input feature map including north and south zero paddings if they exist are distributed across the Poy number of input buffers. With stride $= 2$ as in Figure 5.3, two adjacent rows are continuously stored in the same buffer according to the dataflow requirement. Then,

the number of rows of one feature map, i.e. $r(i, y)$, in one buffer is

$$\#rows\_1Map = \lceil \lceil Tiy/stride \rceil / Poy \rceil \times stride. \tag{5.18}$$

The storage location of the subsequent input feature maps are aligned with the first feature map to simplify the address generation logic, which causes some overhead due to the noncontinuous storage pattern as shown by the blank spaces in the buffers in Figure 5.3. By this means, the depth or words requirement of one input buffer ($InBuf$) storing Tif input feature maps for one convolution layer is expressed as

$$word\_InBuf = word\_1Row \cdot \#rows\_1Map \cdot Tif. \tag{5.19}$$

The data width of one input buffer is $Pox \times bit\_Px$ and the number of input buffers is $Poy \times Dual$ with $Dual = 2$, where $Dual$ represents doubling of the number of buffers due to the dual buffer structure. Therefore, in every clock cycle, $Pox \times Poy$ pixels can be fed into the MAC units. The input buffer size requirement of one convolution layer is

$$bit\_InBuf = Dual \times Poy \times Pox \times bit\_Px \times word\_InBuf. \tag{5.20}$$

The final input buffer size is the maximum bit_InBuf of all the convolution layers. The actual input buffer size in Equation (5.20) is larger than the rough estimation in Equation (5.4) due to the mismatch of tile and buffer dimensions caused by the specific storage pattern.

5.5.2 Size and Storage of Weight Buffers

The storage pattern of weight buffer is illustrated in Figure 5.4. The $k(x, y)$ in Figure 5.4 denotes one weight inside the Nkx × Nky kernel window, where $x \in \{1, 2, \ldots, Tkx\}$ and $y \in \{1, 2, \ldots, Tky\}$. In the chosen design, we always have $Tkx =$

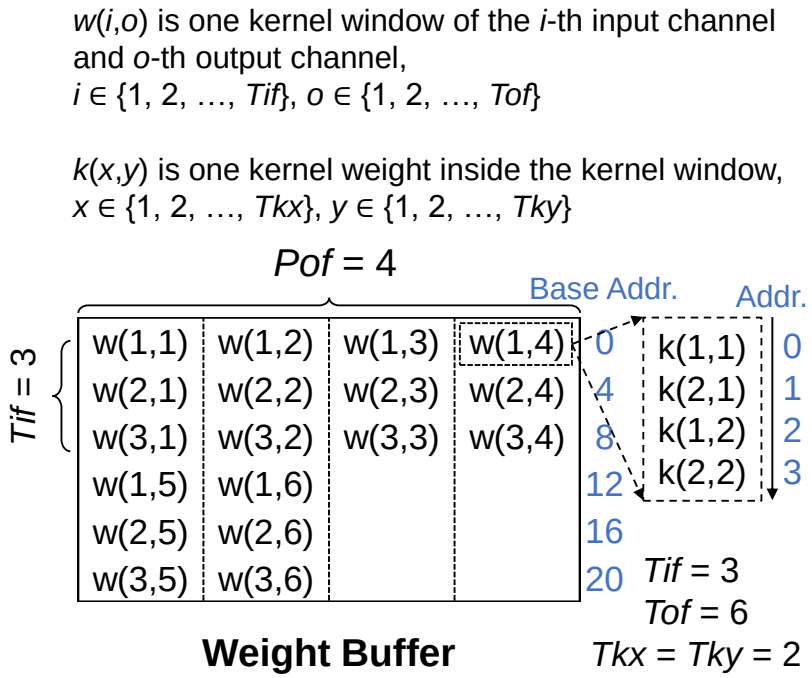

Figure 5.4: The convolution data storage pattern in the weight buffer.

Nkx and $Tky =$ Nky, so that one kernel window is fully buffered. These $Tkx \times Tky$ weights, i.e. $k(x, y)$, are stored in continuous addresses as we serially compute one kernel window, e.g. $Pkx = Pky = 1$. In Figure 5.4, $w(i, o)$ denotes one kernel window of the i-th input channel and o-th output channel, which is comprised of $Tkx \times Tky$ weights. Weights from different input channels (Tif) are stacked in different addresses as we serially compute each input channel. To compute Pof output channels in parallel, the weights of Pof output channels are stored at the same address of the weight buffer. Therefore, the bit width of the weight buffer is $Pof \times bit\_Wt$. The words or depth of the weight buffer $(WtBuf)$ is

$$word\_WtBuf = Tkx \times Tky \times Tif \times \lceil Tof/Pof \rceil. \tag{5.21}$$

With dual buffering, the number of weight buffers is two. The weight buffer size requirement of one convolution layer is

$$bit\_WtBuf = Dual \cdot Pof \cdot bit\_Wt \cdot word\_WtBuf. \tag{5.22}$$

If Tof/Pof is not an integer, some blank spaces in the weight buffer are wasted as in Figure 5.4. The final weight buffer size is the maximum bit_WtBuf of all the convolution layers.

5.5.3 Size and Storage of Output Buffers

After every $Nkx \times Nky \times Nif$ clock cycles, there are $Pox \times Poy \times Pof$ outputs from MAC units. To reduce the bit width of data bus and the bandwidth requirement of output buffers as in Figure 5.5, the parallel outputs are serialized into $Poy \times \lceil Pof/\#OutBuf \rceil$ clock cycles, where $\#OutBuf$ is the number of output buffers excluding the dual buffer structure with $\#OutBuf \leqslant Pof$. By this means, the data width of one output buffer is $Pox \times bit\_Px$, as shown in Figure 5.5, to store the parallel Pox outputs from the same feature map.

The output buffer storage pattern is illustrated in Figure 5.5, where $c(x)$ is the x-th column element in one row with $x \in \{1, 2, \ldots, Tox\}$ and $r(o, y)$ is the y-th row in the o-th output feature map with $o \in \{1, 2, \ldots, Tof\}$ and $y \in \{1, 2, \ldots, Toy\}$. The outputs of the same feature map are continuously stored in the same buffer in a row-major order. One row ($r(o, y)$) is comprised of Tox elements ($c(x)$) continuously stored in $\lceil Tox/Pox \rceil$ addresses, and we set $Tox =$ Nox so that one entire row is processed while maintaining the row-major order. One feature map has Toy number of rows stored in one buffer and it occupies $Toy \times \lceil Tox/Pox \rceil$ addresses. One output buffer stores $\lceil Tof/\#OutBuf \rceil$ number of feature maps. Then, the number of words

or the depth of one output buffer ($OutBuf$) for one convolution layer is

$$word\_OutBuf = \lceil Tof/\#OutBuf \rceil \times Toy \times \lceil Tox/Pox \rceil. \tag{5.23}$$

The output buffer size requirement of one convolution layer is

$$bit\_OutBuf = (Dual \times \#OutBuf) \times (Pox \times bit\_Px) \times word\_OutBuf. \tag{5.24}$$

If $Tof/\#OutBuf$ is not an integer, the blank spaces in the output buffers as in Figure 5.5 are wasted.

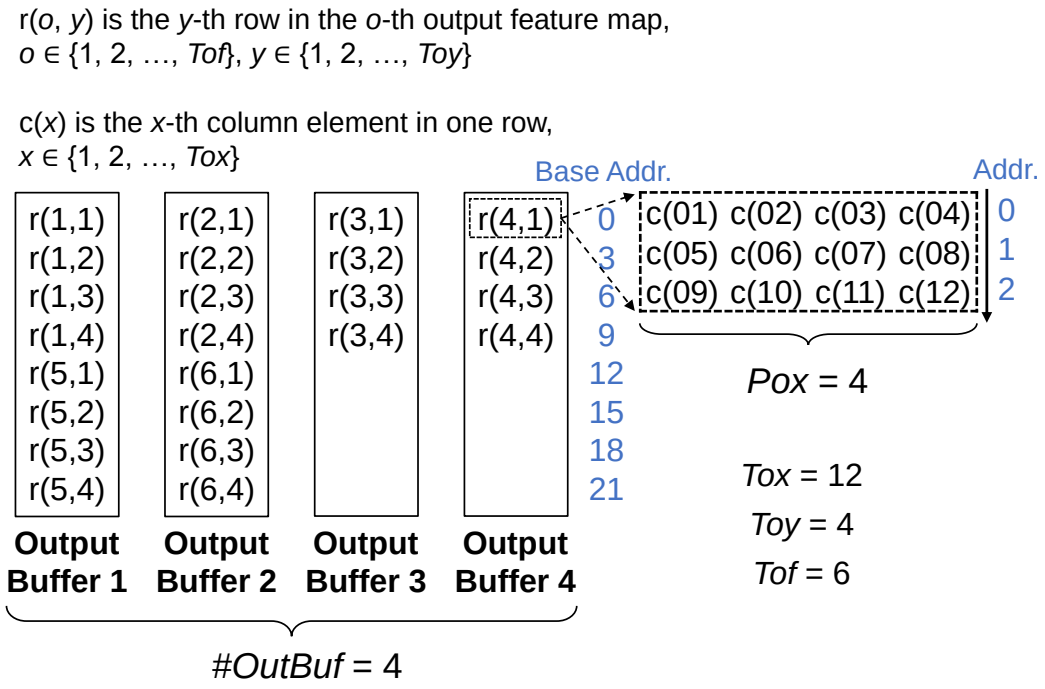

Figure 5.5: The convolution data storage pattern in the output pixel buffers.

5.5.4 Size and Storage of Pooling Buffers

The max pooling layers share the input and output buffers with convolution layers. Due to the different dataflow requirement, the max-pooling input storage pattern in the input buffers is different from convolution inputs, but it is the same as the output

storage pattern of convolution outputs in Figure 5.5. In addtion, the output buffer storage pattern of max-pooling layers is also the same as the convolution outputs in Figure 5.5. The pixels from the same feature map are stored in the same buffer, and different feature maps are distributed across different buffers. Therefore, the input and output buffer depth of one tile of max pooling is similar to Equation (5.23). The buffer size requirement of pooling layers is ensured to be smaller than that of the convolution layers by using smaller pooling tiling variables so that there is no overflow of pooling data.

5.6 Modeling of On-chip Buffer Access

The energy cost of accessing data in the buffers dominates the on-chip memory energy consumption Chen *et al.* (2016) Han *et al.* (2016b), so it is essential to reduce the size of buffer accesses for energy-efficient design. To reduce the buffer access size, data should be reused as much as possible either by multiple PEs or by different execution tiles, which will be discussed in this section.

5.6.1 Read Input and Weight Buffers of Convolution

Based on Equation (5.8) to estimate the buffer access, we need to compute $\#cycles\_Access$ first. In this case, $\#cycles\_Access$ is the MAC computation clock cycles of one tile, which is $\#cycles\_1T$ in Equation (5.14). Then, the computation clock cycles of all the convolution layers are

$$\#cycles\_C = \sum_{L=1}^{\#CONVs} \#cycles\_1T[L] \times \#tiles[L], \tag{5.25}$$

where $\#CONVs$ is the number of convolution layers and $\#tiles$ is the number of tiles. The size (bit) of data read (Rd) from input buffers ($InBuf$) for convolution layers is computed by multiplying the read clock cycles with the total input buffer

data width as

$$bit\_RdInBuf = \#cycles\_C \cdot (Pox \cdot Poy \cdot bit\_Px), \tag{5.26}$$

where every $Pox \times Poy$ pixels are reused by Pof MAC units and the number of input buffer accesses is reduced by Pof times. Similarly, the size (bit) of data read (Rd) from weight buffers ($WtBuf$) for all the convolution layers is

$$bit\_RdWtBuf = \#cycles\_C \times (Pof \times bit\_Wt), \tag{5.27}$$

where every Pof weights are reused by $Pox \times Poy$ MAC units and the number of weight buffer accesses is reduced by $Pox \times Poy$ times.

5.6.2 Write Input and Weight Buffers of Convolution

Before computation, the input data are written into the input and weight buffers from DMA. As discussed in Section 5.4.2, not every tile needs to read both pixels and weights from DRAM, because some pixels or weights of one tile can be reused by the following adjacent tiles. The number of tiles of one convolution layer that write new weights (Wt) to the weight buffer is

$$\#tiles\_Wt = \lceil \mathrm{Nof}/Tof \rceil. \tag{5.28}$$

The number of tiles of one convolution layer that write new input pixels (In) to the input buffers is

$$\#tiles\_In = \begin{cases} \lceil \frac{\mathrm{Noy}}{Toy} \rceil \lceil \frac{\mathrm{Nof}}{Tof} \rceil, & \text{if } Toy < \mathrm{Noy} \text{ and } Tof < \mathrm{Nof} \\ \lceil \frac{\mathrm{Noy}}{Toy} \rceil, & \text{otherwise} \end{cases} \tag{5.29}$$

When neither weights nor pixels are fully buffered, i.e. $Toy < \mathrm{Noy}$ and $Tof < \mathrm{Nof}$, the same pixels are re-loaded $\lceil \mathrm{Nof}/Tof \rceil$ times into input buffers as shown in

Figure 5.1(d). Similar to Equation (5.20), the size (bit) of one tile ($1T$) of pixels written into the input buffers is

$$bit\_WrIn\_1T = word\_InBuf \cdot Poy \cdot Pox \cdot bit\_Px. \tag{5.30}$$

The size (bit) of data loaded into the input buffers of all the convolution layers is

$$bit\_WrInBuf = \sum_{L=1}^{\#CONVs} bit\_WrIn\_1T[L] \times \#tiles\_In[L]. \tag{5.31}$$

Similarly, the size (bit) of one tile of weights written into the weight buffers is

$$bit\_WrWt\_1T = word\_WtBuf \times Pof \times bit\_Wt, \tag{5.32}$$

and the size (bit) of data written into the weight buffers of all the convolution layers is

$$bit\_WrWtBuf = \sum_{L=1}^{\#CONVs} bit\_WrWt\_1T[L] \times \#tiles\_Wt[L]. \tag{5.33}$$

5.6.3 Data Access of Output Buffers of Convolution

The number of clock cycles to write outputs into output buffers during one tile is the same as $word\_OutBuf$, where one word of data is written into one output buffer in one cycle. Since every tile of one layer has outputs to be saved, the clock cycles of writing outputs to output buffers is $word\_OutBuf \times \#tiles$. Then, the total cycles to load outputs into output buffers ($OutBuf$) are summed up across all the convolution layers as

$$\#cycles\_WrOutBuf = \sum_{L=1}^{\#CONVs} word\_OutBuf[L] \times \#tiles[L]. \tag{5.34}$$

The size (bit) of results written into the output buffers is

$$bit\_WrOutBuf = \#cycles\_WrOutBuf \times \#OutBuf \times Pox \times bit\_Px. \tag{5.35}$$

Since each output is written into and read from the output buffers only once, the size (bit) of data read from output buffers ($bit\_RdOutBuf$) by DMA equals to $bit\_WrOutBuf$.

5.6.4 Data Access of Buffers of Other Layers

During the max-pooling computation, input pixels are read from input buffers every clock cycle similar to the convolution layer. Then, the size (bit) of data read from input buffers for max-pooling layers is computed by multiplying the read clock cycles with the total input buffer data width similar to Equation (5.26). The size of one tile of pooling inputs written into the input buffers is the same as the input buffer size requirement of one pooling tile similar to Equation (5.31). The data access of output buffer of pooling layers is computed similar to that of the convolution layer, e.g. $bit\_RdOutBuf$ and $bit\_WrOutBuf$, which is determined by the tiling size of max-pooling outputs.

The buffer access of FC layer is mainly from weights. Each FC weight is loaded into the weight buffer from DRAM only once, and every FC weight is read from the weight buffer only once during the computation. Then, the sizes of read and write of the weight buffer of FC layers are the same, which are determined by the FC weight size and the buffer storage utilization.

5.7 Experiments and Analysis

In this section, the proposed performance model is used to explore the design space by tuning the key design variables, e.g. unrolling and tiling sizes, DRAM bandwidth and accelerator frequency, to identify the performance bottleneck and obtain the optimal design configurations.

5.7.1 Design Space Exploration of Tilling Variables

The loop tiling strategy determines how many data of each layer are buffered, which affects the buffer capacity requirement, the number of DRAM accesses, and the accelerator performance. Although we have fixed Tkx = Nkx, Tky = Nky, Tif = Nif and Tox = Nox, the remaining two tiling variables Toy and Tof still give us a huge design space as mentioned in Ma *et al.* (2018a). For example, VGG-16 has 13 convolution layers, and there are $13 \times 2 = 26$ tiling variables and each variable can have 4 or more candidate values determined by Noy/Poy or Nof/Pof, then the total number of Toy and Tof choices is roughly $4^{26} = 4.5 \times 10^{15}$, which results in an enormous solution space that cannot be enumerated. Therefore, we randomly sample 30,000 tiling configurations for different CNN algorithms to explore their impact on the memory access and performance as in Figure 5.6, Figure 5.7 and Figure 5.8, where we set loop unrolling variables as $Pox \times Poy \times Pof = 7 \times 7 \times 32$.

The relationship between tiling variables and the number of DRAM accesses is investigated in Figure 5.6 with 16-bit data. The total convolution DRAM access size is computed by

$$\begin{aligned} byte\_DRAM = \sum_{L=1}^{\#CONVs} \big(& byte\_RdPx \cdot \#tiles\_In + \\ & byte\_RdWt \cdot \#tiles\_Wt + byte\_WrPx \cdot \#tiles \big), \end{aligned} \tag{5.36}$$

where the right-hand side variables are computed by Equation (5.10) (5.11) (5.13) (5.28) (5.29). The DRAM accesses of other layers are also included in Figure 5.6. One circle in Figure 5.6 represents one design point of the tiling variables Toy and Tof. Since the buffer size is determined by the layer with the maximum tiling size, there could be multiple different tiling configurations in other layers leading to the same buffer size. The buffer size in Figure 5.6 includes input/weight/output buffers, which equals to $max(bit\_InBuf) + max(bit\_WtBuf) + max(bit\_OutBuf)$ from Equation

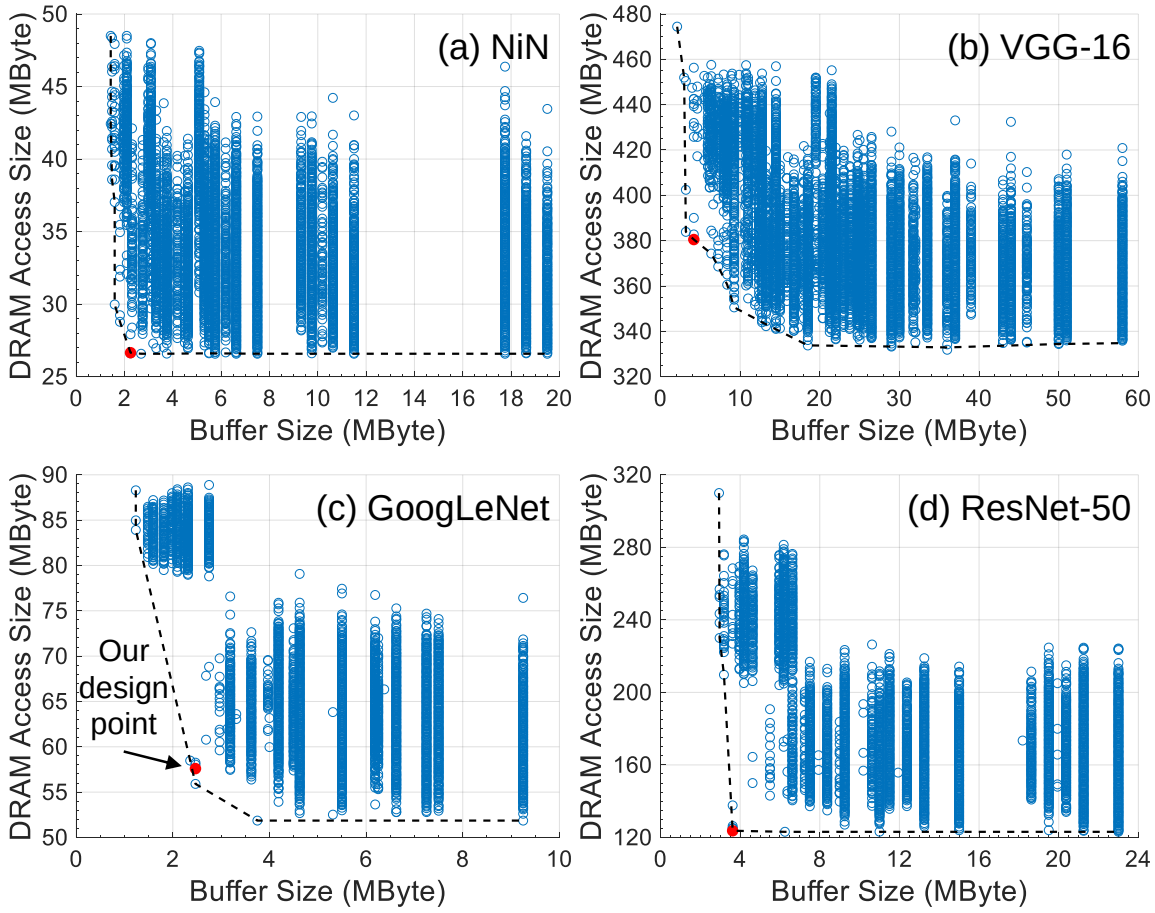

Figure 5.6: The tiling variables (Toy and Tof) are swept to explore the relationship between the size of DRAM accesses and the total input/weight/output buffer size requirement, where $Pox \times Poy \times Pof = 7 \times 7 \times 32$ with 16-bit data.

(5.20) (5.22) (5.24). With the increase of tiling and buffer sizes, the number of DRAM accesses is decreasing as shown by the dashed line in Figure 5.6. After the buffer size is increased to be large enough, we can achieve the minimum DRAM accesses. The red dot in Figure 5.6 is our optimal design choice of Toy and Tof that balances the buffer size requirement and the number of DRAM accesses.

Figure 5.7 shows the relationship between tiling sizes and the convolution throughputs, where the accelerator operating frequency is 240 MHz and the DRAM band-

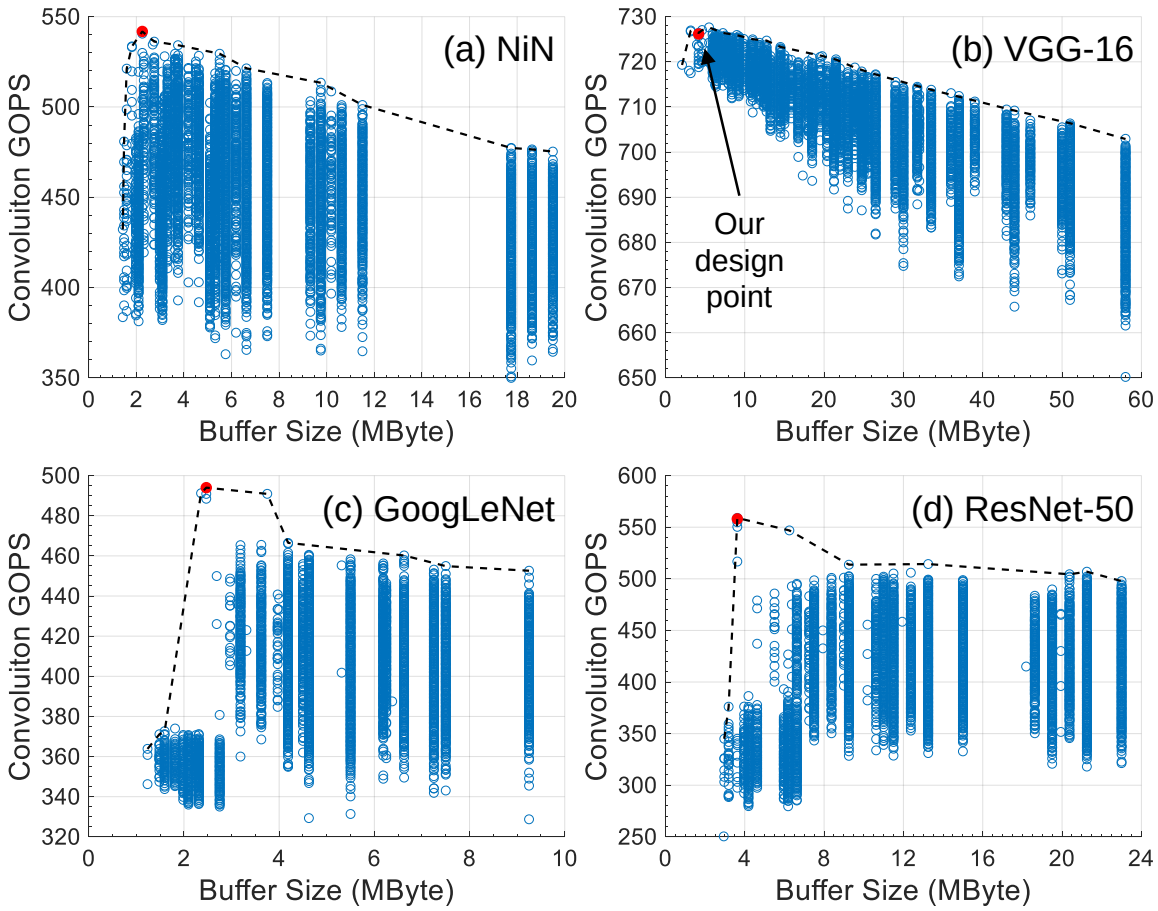

Figure 5.7: The tiling variables (Toy and Tof) are swept to explore the relationship between the convolution throughputs and the total input/weight/output buffer size requirement, where $Pox \times Poy \times Pof = 7 \times 7 \times 32$, $MHz\_Accelerator = 240$, $BW\_DRAM = 14.4$ GB/s.

width is 14.4 GB/s. The throughput is computed by $\#operations/delay$, where $\#operations = 2$Nm including both multiply and addition, and $delay$ is the sum of ms_1CV over all the convolution layers. If the tiling or buffer size is too small, the number of DRAM access and the associated latency is significantly increased, which degrades the throughput. If the tiling size is too large or there is only one tile in one layer, the DRAM access latency cannot be well overlapped with the computation

delay as mentioned in Section 5.4.2, which results in lower throughput. This trend is shown by the dashed line in Figure 5.7. The dashed lines of GoogLeNet and ResNet-50 are not as smooth as those of NiN and VGG-16. It is mainly because GoogLeNet and ResNet-50 have more layers resulting in much larger design space, which makes it more difficult to cover all the design choices through random sampling. The red dots in Figure 5.7 are our design choices of Toy and Tof, which are the same in Figure 5.6, to achieve the best throughputs.

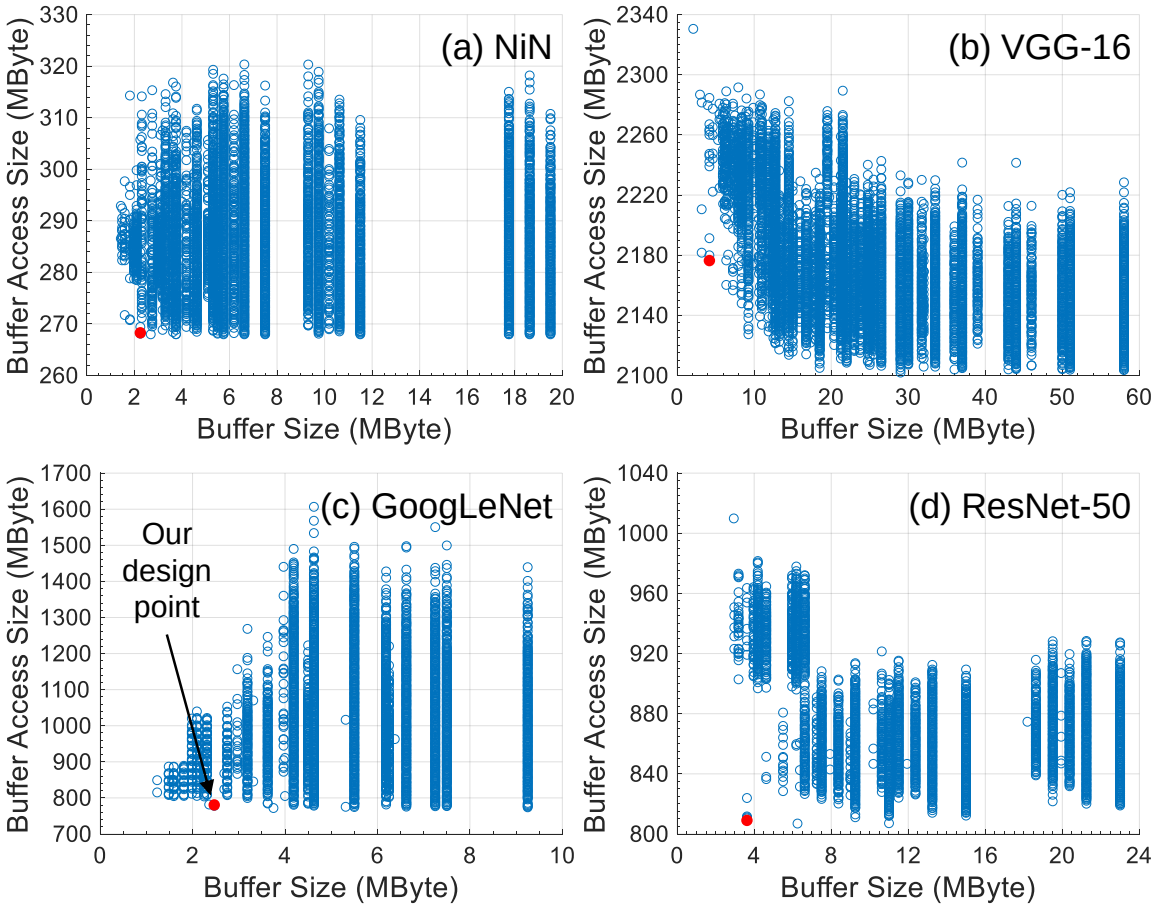

Figure 5.8: The tiling variables (Toy and Tof) are swept to explore the relationship between the size of on-chip buffer accesses and the size requirement of buffers, where $Pox \times Poy \times Pof = 7 \times 7 \times 32$.

Figure 5.8 shows the relationship between tiling sizes and the number of on-chip buffer accesses for different CNN algorithms, which include both read and write operations of input/weight/output buffers of all the layers in a given CNN algorithm. Based on our acceleration strategy Ma *et al.* (2018a), the partial sums are accumulated inside the MAC units, which do not involve buffer access. The estimation of the number of on-chip buffer accesses is discussed in Section 5.6. Our design choices of Toy and Tof shown by red dots in Figure 5.8 can achieve close to the optimal number of buffer accesses while having best throughputs and low level of DRAM accesses.

5.7.2 Design Space Exploration for Performance

As convolution dominates the CNN operations Krizhevsky *et al.* (2012) Lin *et al.* (2013) Simonyan and Zisserman (2014) He *et al.* (2016a), we focus on the design space exploration of convolution throughputs. The convolution throughput is affected by several factors, namely the accelerator operating frequency, external memory bandwidth and the loop unrolling variables, These are explored in Figure 5.9 using GoogLeNet as an example. With a small number of MAC units and high DRAM bandwidth (BW_DRAM) as shown in Figure 5.9(a), the accelerator throughput is mainly bounded by computation, and thus the throughput is almost linearly increasing with the frequency when $BW_DRAM > 12.8$GB/s. If the DRAM bandwidth is too low, e.g. 3.2 GB/s, the design is more likely to be memory bounded and the throughput stops increasing with the frequency. With more MAC units and higher frequency, the throughputs are tend to increase, as shown in Figure 5.9, until the design touches the memory roof which is illustrated in Figure 5.10.

The memory roof throughput Zhang *et al.* (2015) in Figure 5.10 is the maximum

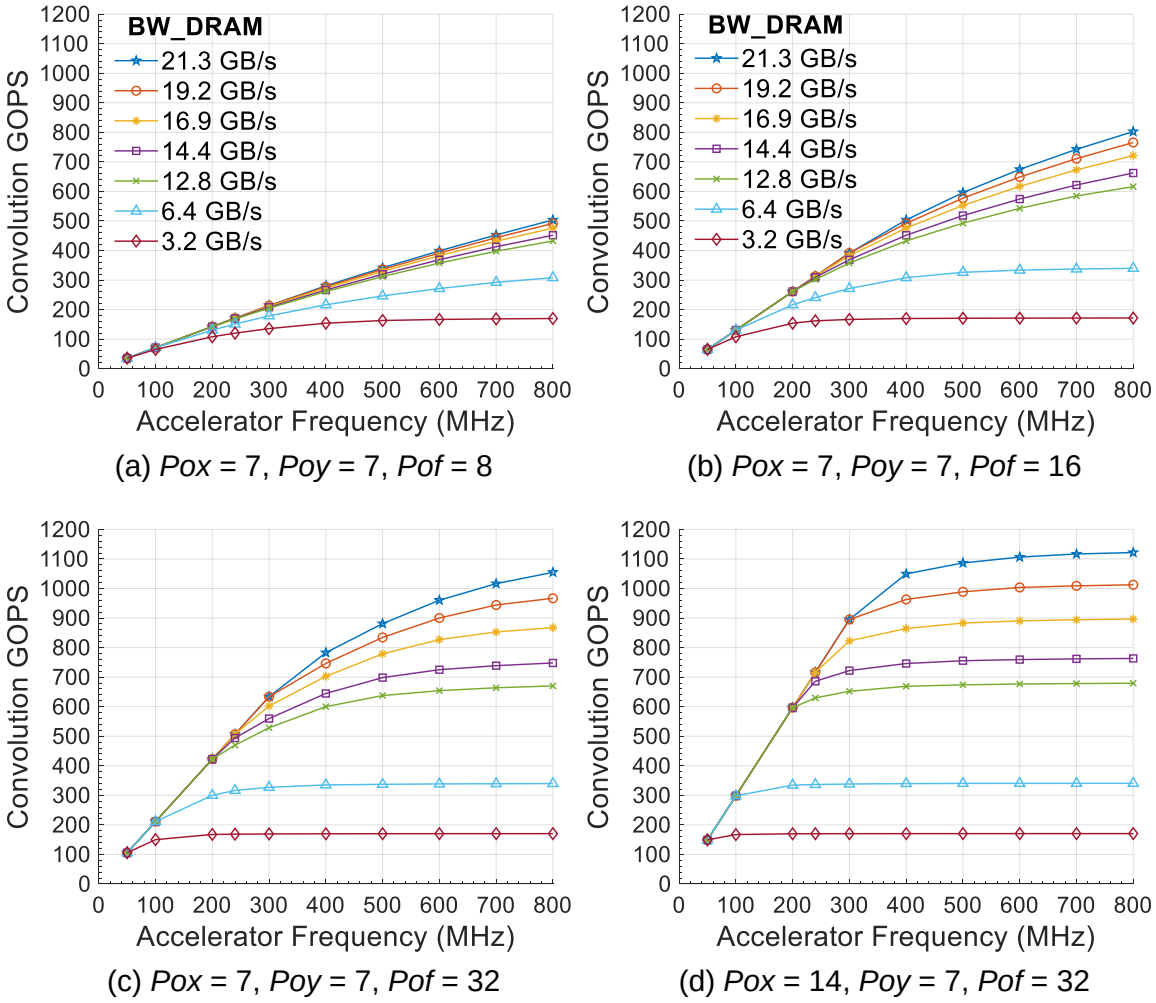

Figure 5.9: The convolution throughput is affected by the accelerator operating frequency, DRAM bandwidth, and the number of MAC units. GoogLeNet is shown as an example here.

achievable throughput under a certain memory bandwidth, which is defined as,

$$\begin{aligned} DRAM\_roof(GOPS) &= \frac{\#operations(GOP)}{DRAM\_delay(s)} \\ &= \frac{\#operations(GOP)}{\#data(GByte)} BW\_Memory(GB/s), \end{aligned} \tag{5.37}$$

where $\#data$ is the data size of DRAM accesses. Since the computation-to-communication ratio (CTC), i.e. $\#operations/\#data$, is a constant under a certain tiling setting,

$DRAM\_roof$ is directly proportional to BW_Memory. With the same setting of BW_Memory for GoogLeNet and VGG-16, the shape of the curves in Figure 5.10(a) and (b) are similar. Since VGG-16 has a higher CTC, its memory roof throughput is much higher than GoogLeNet in Figure 5.10. As discussed in Section 5.3.2, the memory bandwidth (BW_Memory) is bounded by both the DRAM controller (BW_DRAM) and the DMA (BW_DMA). At low frequency, BW_Memory is limited by BW_DMA, and $DRAM\_roof$ is linearly increasing with the increase of frequency as in Figure 5.10. After BW_DMA is larger than BW_DRAM, BW_Memory is limited by BW_DRAM instead, and $DRAM\_roof$ stops growing with the frequency. The saturated throughputs in Figure 5.9 are lower than $DRAM\_roof$ in Figure 5.10, which is mainly because there are redundant DRAM transfers and the computation delay is not fully overlapped with the DRAM latency.

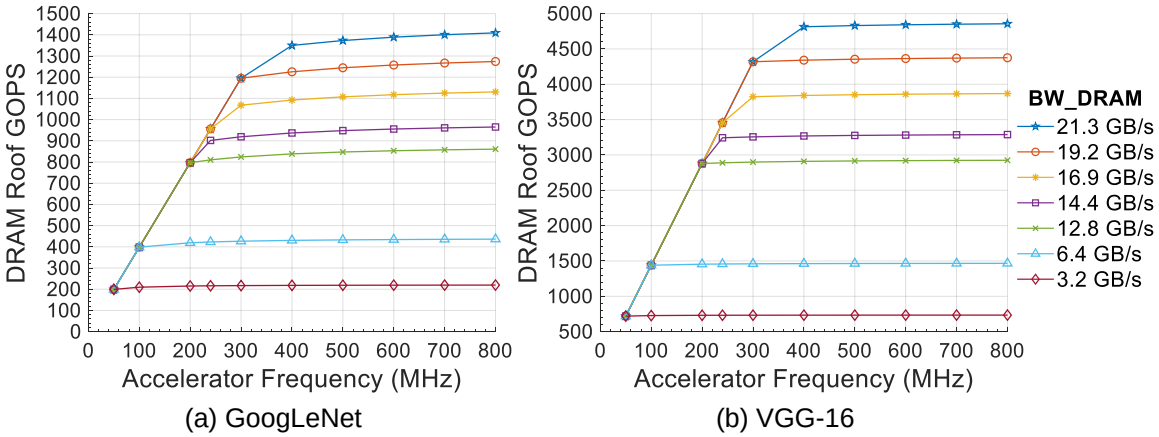

Figure 5.10: The external memory roof throughput ($DRAM\_roof$) is the maximum achievable throughput under a certain memory bandwidth.

5.7.3 Performance Model Validation

Figure 5.11 shows the comparison of throughput and latency between the performance model and the on-board test results on Arria 10 and Stratix 10 with different

number of MAC units, where both pixels and weights are 16-bit fixed point data. The differences between the estimation and on-board results are about 5%, which are mainly due to the DRAM transfer latency mismatch, minor layers (e.g. average pooling), and some pipeline stages in the real implementation. The compilation of our FPGA design using Quartus Pro 17.1 on 16-core Intel Xeon CPU E5-2650 v3 normally takes six to eight hours, while the performance model running on laptop Intel Core i7-7500U CPU using MATLAB takes about 1 to 5 seconds per design.

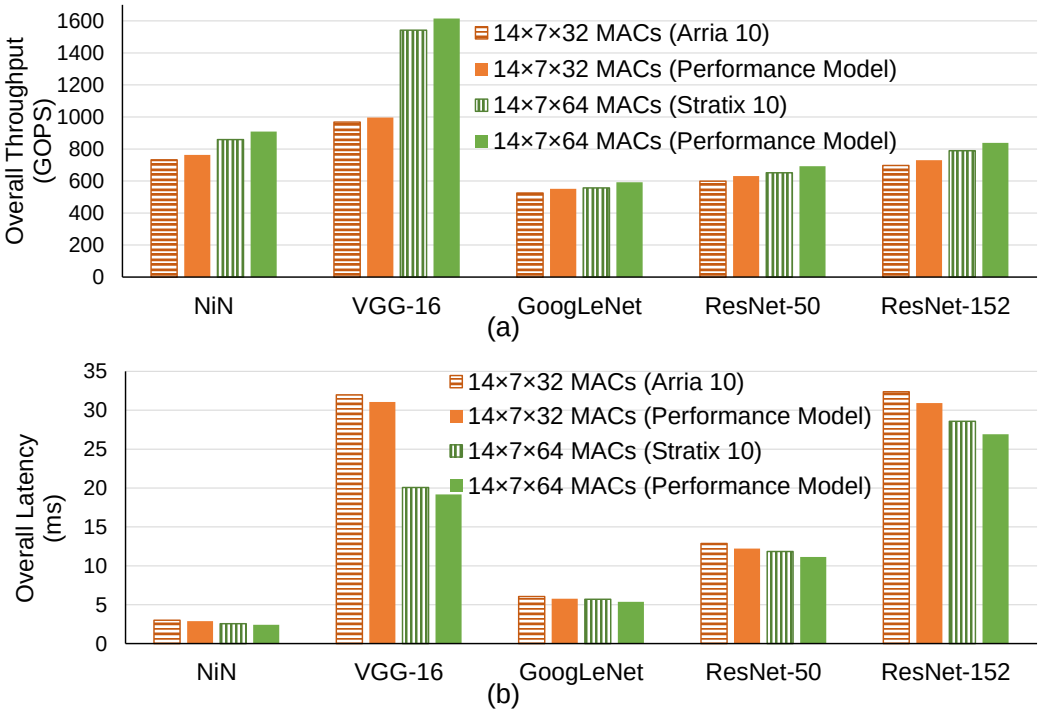

Figure 5.11: The performance model results are compared with on-board test results of Arria 10 and Stratix 10 on overall (a) throughput and (b) latency.

5.8 Further Improvement Opportunities

In this section, we use the proposed performance model to evaluate the opportunities that may further enhance the performance of the accelerator by improving the

efficiency of DRAM transactions and DSP utilization.

5.8.1 Improving DRAM Bandwidth Utilization

To maximize the bandwidth of DRAM access with $bit\_DRAM = 512$, we need to set the DMA datawidth (bit_DMA) to be 512 bits, which means one DMA transfer address can accommodate $\lfloor bit\_DMA/bit\_Px \rfloor$ number of pixels. To simplify the control logic of data bus from DMA to input buffers, different feature map rows are aligned in different addresses in our current design. By this means, if the number of pixels in one row is smaller than $\lfloor bit\_DMA/bit\_Px \rfloor$, the successive row directly starts from the next address instead of continuously using the same address resulting in the waste of DMA datawidth. For example, with $bit\_Px = 16$, one address can accommodate $512/16 = 32$ pixels, if the width of the feature map is Nix = 14, then the actual number of pixels of one row read from DRAM in Equation (5.10) is $Tix = 32$, where $32 - 14 = 20$ data are redundant. Some CNN models, e.g. GoogLeNet and ResNet, have a lot of convolution layers with small Nix, e.g. 7 or 14, then their throughputs are significantly affected by the inefficient utilization of DMA datawidth.

To improve the DRAM bandwidth utilization, one method is to store multiple rows in one DMA address, which involves the modifications of control logic and extra data paths from DMA to input buffers. The other method is to keep the data aligned, but narrow the bit width of the data bus between DMA and input buffers. To attain the same data transfer rate, higher frequency is needed, and asynchronous FIFO may be used. In the performance model, we reduce bit_DMA to be 256 and 128 and increase their corresponding frequency of the data bus to predict the potential throughput improvements. With $bit\_Px = 16$ in our experiments as shown in Figure 5.12, setting bit_DMA to be 256 or 128 has the same effect as supporting two or four rows in

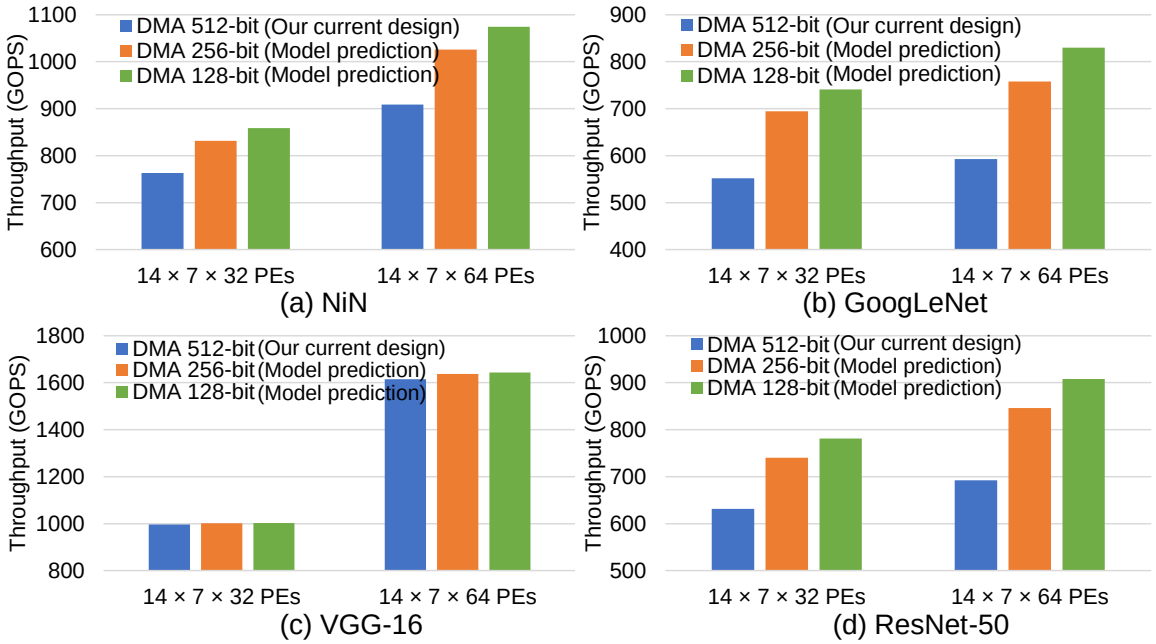

Figure 5.12: Performance model predicts that the throughput will be improved by increasing the DRAM bandwidth utilization, which is achieved by decreasing the DMA bit width to reduce the redundant DRAM accesses.

one address with $bit\_DMA = 512$, respectively. In Figure 5.12, our current design (DMA 512-bit) serves as the baseline with data aligned. Figure 5.12 shows that NiN, GoogLeNet and ResNet can benefit a lot from decreasing the DMA bit width or improving the DRAM bandwidth utilization, mainly because they have many layers with small Nix and the layers with small Nix are memory bounded. On the contrary, VGG-16 cannot benefit from higher DRAM bandwidth utilization as the design is still computation bounded. Based on the prediction of the performance model, it is compelling to improve our design for higher DRAM bandwidth utilization.

5.8.2 Merging the First Layers

In GoogLeNet and ResNet, there are multiple parallel branches of layers, and the first layer of each branch reads input pixels from the same precedent layer. If these

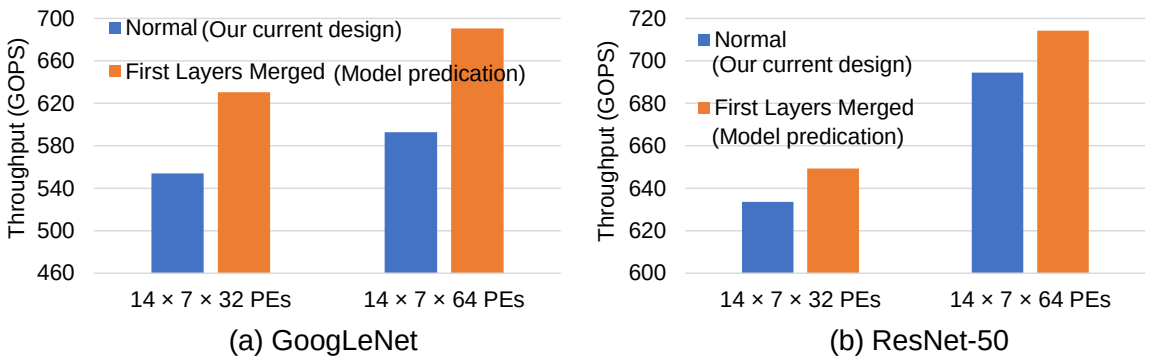

Figure 5.13: Performance model predicts that the throughput will be improved by merging the first layers of different parallel branches, which read from the same precedent layer, to eliminate the repeated DRAM access, where "Normal" denotes our current design as baseline.

convolution layers also have the same kernel size and stride, they can be merged into one layer along the output feature map dimension (Nof). By this means, the input pixels can be shared by the first layers of different branches and only need to be read from DRAM once, as proposed in Lin *et al.* (2018). We change the corresponding settings of our performance model, e.g. $byte\_RdPx$ in Equation (5.10), to estimate the effect of eliminating the repeated DRAM accesses of the precedent layer as shown in Figure 5.13. Since GoogLeNet and ResNet are already memory-bounded in our current design, reducing the DRAM access can considerably improve the throughputs. The required modifications of our current design to merge the first layers involve changing the control logic and the descriptors of DMA transactions, and there is no significant overhead of additional hardware resources.

5.8.3 Improving PE Efficiency

Due to the highly varying dimensions of different convolution layers in a given CNN algorithm, it is a challenge task to efficiently distribute workloads across PEs,

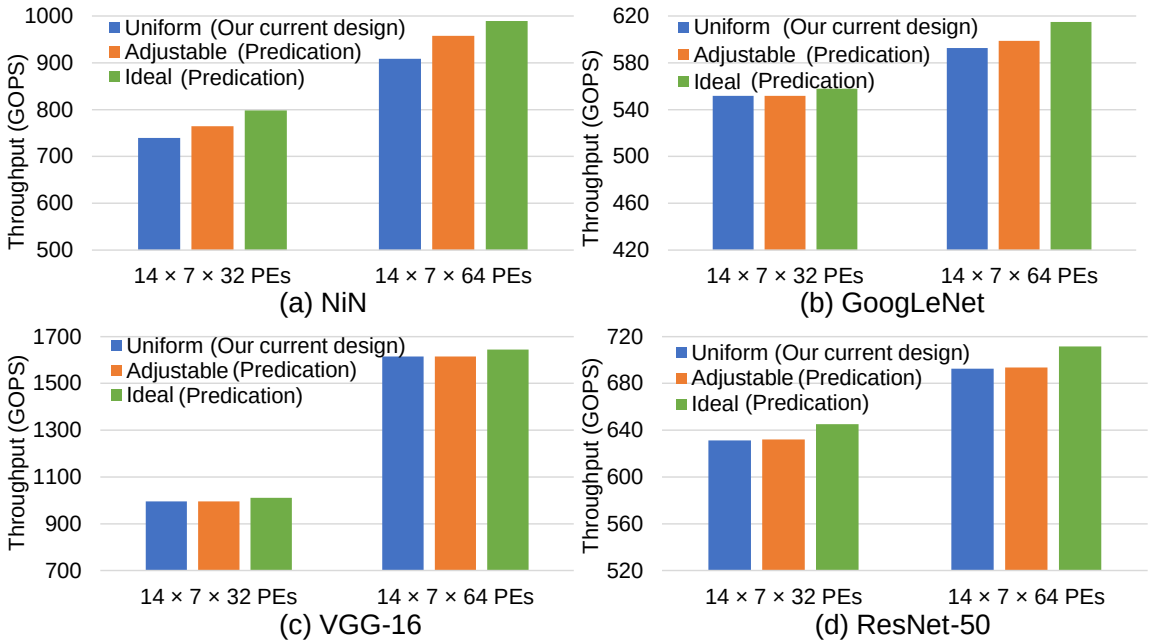

Figure 5.14: Uniform: our current design as baseline with uniform PE mapping; Adjustable: dynamically adjust the unrolling variables for different layers to improve PE utilization; Ideal: force PE utilization to be 100%.

or we need to make the loop dimensions (N$*$) divisible by their corresponding unrolling variables ($P*$). In Song *et al.* (2016) Putic *et al.* (2018), adaptive parallelism scheme is proposed to dynamically adjust the mapping of operations on different PEs, or the unrolling variables can be changed for each layer to maximize the PE utilization. This requires the ability to dynamically redirect the data flow from buffers to PEs, which may need complex control logic, incur penalty of additional resources, and aggravate the burden on timing closure.

Instead of using uniform PE mapping and unrolling variables in the current design, we adjust the unrolling variables ($Pox \times Poy \times Pof$) for different layers to achieve better PE utilization in the performance model as shown by "Adjustable" in Figure 5.14. We also force the PE utilization to be 100% by removing the ceiling functions in Equation (5.1), which is denoted by "Ideal" in Figure 5.14. However, the

throughput improvements from adjustable unrolling strategy are very limited ($< 10\%$) for our design, mainly because 1) the Nox $\times$ Noy $\times$ Nof dimensions of most layers have already been able to provide large enough parallelism for our uniform unrolling strategy, and 2) most of our layers are memory-bounded and the reduction of computation latency has little effect on the throughput. Considering the large amount of necessary design efforts for adjustable PE mapping and low expected improvements, we surmise it is not a primary task in our future work to adopt this technique.

5.9 Summary

In this chapter, a high-level performance model is proposed to estimate the key specifications, e.g. throughput, of FPGA accelerators for CNN inference, which enables the design space exploration to identify performance bottleneck in the early development phase. The design strategy and resource costs are formulated using the design variables of loop unrolling and tiling. The proposed performance model is validated for a specific acceleration strategy across a variety of CNN algorithms comparing with on-board test results on two different FPGAs.

Chapter 6

ALGORITHM-HARDWARE CO-DESIGN OF DEEP LEARNING

6.1 Algorithm Customization for FPGA

The recently achieved substantial improvements in speed and accuracy of CNN for image recognition are now being demonstrated in object detection algorithms. The Single Shot Detector (SSD) Liu *et al.* (2016) algorithm uses VGG-16 CNN as the base feature extractor to predict the bounding boxes and classification probability, and then uses additional convolution layers at the end to predict objects from multi-scale feature maps. With its simplified architecture, the SSD algorithm demonstrates faster performance with higher accuracy, compared to Faster RCNN Ren *et al.* (2015) and YOLO Redmon *et al.* (2016). However, it is still very difficult to directly implement SSD on hardware accelerator to achieve real-time detection with high energy efficiency, due to (1) the large volume of data and operations, (2) the use of complex nonlinear functions, and (3) the highly varying layer sizes and configurations. Therefore, in this chapter, we propose to customize the deep learning based detection algorithm, e.g. SSD, to benefit its hardware implementation with low data precision at the cost of marginal accuracy degradation.

Unlike software (CPU-GPU) implementations, direct hardware implementation normally favors performing massive numbers of linear computations in parallel, and with a uniform dataflow, as this maximizes the utilization of the hardware resources and reduces the complexity of the control logic. Therefore, it is necessary to tailor the original software implementation of SSD object detection algorithm to benefit the hardware implementation, while maintaining sufficient accuracy. The modification

methods and their corresponding accuracies are shown and summarized in Figure 6.1 and Table 6.1.

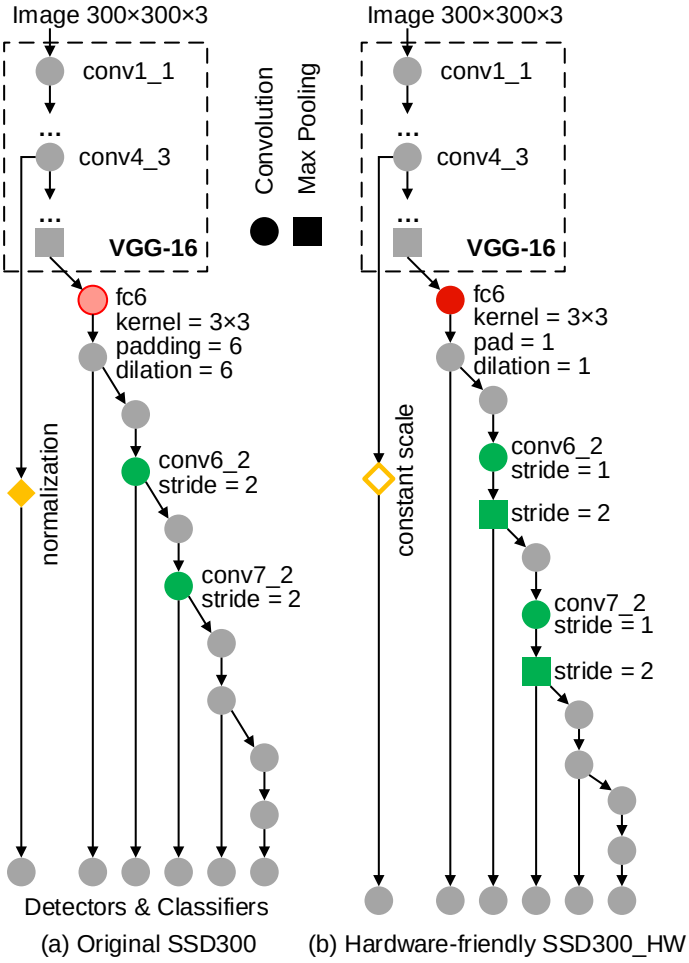

Figure 6.1: Customization of SSD300 to be hardware-friendly SSD300_HW by (1) replacing dilated convolution, (2) using constant scale instead of normalization and (3) using uniform convolution stride.

6.1.1 Dilated Convolution

To speed up the training and inference time in the original SSD algorithm Liu *et al.* (2016), the fully connected layers, e.g. fc6 and fc7, are converted to convolution

Table 6.1: Experiments of SSD customization for hardware inference with mAP tested on VOC07+12 test database Everingham *et al.* (2012)

Model	Dilated Conv. (fc6)	Norm	Constant Scale	Different Conv. Strides	mAP
SSD300	√	√	-	√	77.30%
SSD300_1	×	√	-	√	77.34%
SSD300_2	√	×	0.01	√	77.81%
SSD300_3	√	×	0.015	√	77.88%
SSD300_4	√	×	0.02	√	77.19%
SSD300_5	√	√	-	×	77.41%
SSD300_HW	×	×	0.015	×	77.10%

layers. In addition, the fc6 layer is implemented as dilated convolution to expand the receptive field without loss of resolution or coverage Yu and Koltun (2015). However, the change of the computation pattern in convolution makes the dataflow into the PEs significantly different from the original convolution, which requires new data bus and control logic in hardware.

One solution is to implement the dilated convolution as original convolution, filling the intervals inside the kernel window with zeros. The cost of this increases redundant computation. In SSD, the configuration of fc6 is kernel size = 3, dilation = 6, and zero-pad = 6. These can be implemented as a normal convolution with kernel size = $3 + 2 \times (6 - 1) = 13$, dilation = 1 and zero pad = 6. By this means, the number of fc6 operations is dramatically increased from 3.4 GOP to 64 GOP. This is even larger than the total number of operations in the original SSD algorithm, i.e. 62 GOP, and is obviously unacceptable.

Another solution is to change the dilated convolution into a normal convolution

directly and make the convolution configurations uniform with other layers. Therefore, we set fc6 to be kernel size = 3, dilation = 1, and zero-pad = 1. This makes the output feature map size have the same number of operations. After retraining the SSD model, the mAP of SSD300 with the modified fc6, e.g. SSD300_1, is 77.34% as shown in Table 6.1. This is even slightly better than the original one as 77.30%. By this means, we can keep using the existing data bus and control logic to implement fc6 without any performance penalty.

6.1.2 Normalization

Since conv4_3 in SSD has a different feature scale compared to the other layers, Liu *et al.* (2016) applies the L2 normalization combined with scale at each location in the feature map and learn the scale during back propagation. The normalization operation of conv4_3_norm in SSD is expressed as:

$$\begin{gathered} out(x,y,m) = \frac{scale(m) \times input(x,y,m)}{\sqrt{\sum_{m=1}^{M} input(x,y,m)^2}}, \\ x \in [1,X], y \in [1,Y], m \in [1,M], \end{gathered} \tag{6.1}$$

where X and Y are the feature map width and height, respectively, and M is the number of feature map channels. Computing Equation 6.1 requires sum of squares, square root and division operations, which are complex in hardware and require large number of logic resources. Instead of directly implementing hardware for these computations, we can alternatively approximate this nonlinear function by using lookup tables to store limited points of the function, which also requires significant amount of on-chip memory and logic. Since conv4_3_norm is only used to scale the feature values to be the same level as other layers, we directly scale all the conv4_3 features with a constant number during training and use the same scale value for inference. As shown in Table 6.1, we have tried several scale values, e.g. 0.01 for SSD300_2,

0.015 for SSD300_3, and 0.02 for SSD_4, and find that 0.015 scale results in the best mAP of 77.88% (SSD300_3), which is even better than the original 77.30%. By this means, we can directly scale all the features of conv4_3 by a constant number, which significantly simplifies the control logic and reduces the required hardware computing resources.

6.1.3 Convolution with Different Sliding Strides

Different sliding strides and zero padding in convolutions lead to different dataflow of input features into the PEs. This requires different databus and control logic to govern the dataflow and ensure that the proper input data are continuously fed into PEs without idle clock cycles. Therefore, the hardware design favors regular and uniform convolution structures, e.g. VGG-16, to reduce the design efforts and complexity as well as the required hardware resources. In the original SSD, conv6_2 and conv7_2 use stride of 2 to scale down the output feature map size for multi-scale detection and all other convolution layers have stride of 1, which is not favored by hardware design. Therefore, we change the stride of conv6_2 and conv7_2 to be 1 and add a subsequent max pooling layer with stride of 2 to downsample the feature map. The additional max pooling layers reuse the existing hardware module for the previous pooling layers, which does not add overhead to the hardware resources. This modification adds about 0.64 GOP operations ($\approx$ 1.0% of the total SSD operations) and does not affect the overall performance noticeably. The accuracy of this modification is shown in Table 6.1 to be 77.41% as SSD300_5.

6.1.4 Hardware-friendly SSD300_HW

After collectively applying all the aforementioned modifications of (1) removing dilated convolution, (2) using constant scale instead of normalization and (3) employ-

ing uniform convolution stride, we obtain the final hardware-friendly SSD300_HW as shown in Table 6.1 with mAP of 77.10%, which is slighlty lower than the original SSD300 by 0.20%.

6.2 FPGA Inference with Limited Precision

Although 32-bit floating point precision may be required for the training phase, such a high precision is not necessary for inference, and thus most of the hardware inference works to date use fixed-point data precision without significant loss of accuracy Suda *et al.* (2016) Qiu *et al.* (2016) Wei *et al.* (2017) Ma *et al.* (2017b) **?** Shin *et al.* (2017).

Using data with low precision reduces considerably the requirement of on-chip memory capacity and external memory bandwidth. It also improves the hardware efficiency and performance by allowing the use of fixed-point arithmetic operations, which demands significantly fewer FPGA computing resources, e.g. logic and DSP, compared to floating-point operations.

6.2.1 Fixed-point Data Representation

Quantization is one of the most commonly used method to convert floating-point represented real numbers into fixed-point format with lower precision. The bit width of a signed fixed-point number (bit_total) is comprised of one sign bit (bit_sign), integer bits (bit_int) and fractional bits (bit_fra) as shown by Equation 6.2:

$$bit\_total = bit\_sign + bit\_int + bit\_fra. \tag{6.2}$$

In conventional fixed-point hardware implementation, the decimal point is fixed, and defines the portion between the integer and fractional bits of all the numbers. The

integer bit of all the numbers ($\boldsymbol{x}$) is determined as:

$$bit\_int = \lceil \log_2 \max(|\boldsymbol{x}|) \rceil . \tag{6.3}$$

If bit_int is larger than $bit\_total - 1$, it causes overflow error due to the large scale of the numbers. If bit_int is smaller than $1 - bit\_total$, there is underflow problem due to the small scale of the numbers, which may lead to significant precision loss. The fixed-point integer number $\boldsymbol{X}$ can be obtained by rounding to the nearest integer as Equation 6.4:

$$\text{Rounding} : \boldsymbol{X_R} = \left\lfloor \boldsymbol{x} \times 2^{bit\_fra} \right\rceil , \tag{6.4}$$

or truncated to the largest previous integer as Equation 6.5, which is easier to implement in hardware by right shifting or discarding the least significant bits (LSB):

$$\text{Truncation} : \boldsymbol{X_T} = \left\lfloor \boldsymbol{x} \times 2^{bit\_fra} \right\rfloor . \tag{6.5}$$

6.2.2 Dynamic Quantization

Due to the large range and variance in the data in a given CNN algorithm, the conventional fixed-point representation has to increase bit_total to solve the issue of overflow and underflow resulting in higher usage of hardware resources, e.g. memory and logic.

To overcome this problem, we employ the *dynamic quantization* method in Qiu *et al.* (2016) Ma *et al.* (2017b) Shin *et al.* (2017) to use fixed-point representation within one layer and vary the decimal point across different layers. This exploits the characteristic that the range of data in one layer is much smaller than the range across all the layers as shown by Figure 6.2. By this means, all the weights or all the features of one layer share the same exponent, e.g. bit_fra, and have at most $bit\_total - 1$ bits of significand, whereas in a floating-point representation each number has its own exponent and fixed bits of significand. The constraint on the bit_int is relaxed

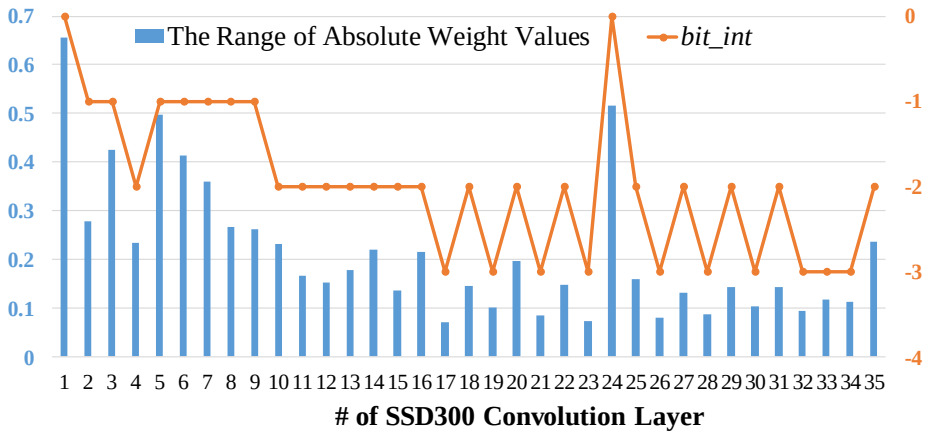

Figure 6.2: The range of absolute values of each convolution layer's kernel weights in SSD300 and their corresponding *bit_int*.

to be any integer number, which allows for a wide range of values. For example, if the maximum absolute value of the weights in one layer is 5678_{10} and $bit\_total = 8$, then $bit\_int = 13$ according to Equation 6.3 and $bit\_fra = 8 - 1 - 13 = -6$. For one weight in this layer, e.g. $x = 2345.625_{10} = 100100101001.101_2$, its corresponding fixed point number after truncation is $X_T = 36_{10} = 100100_2$ by Equation 6.5, where we have 6 bit significand with the rest LSB discarded.

6.2.3 Dynamic Quantization on Hardware

In Intel Arria 10 and Stratix 10 FPGAs, there are limited number of DSP blocks to implement multipliers for convolution operations. One DSP block can support either one single-precision floating-point multiplier or two 18-bit $\times$ 18-bit fixed-point multipliers. Based on this, fixed-point arithmetic can potentially achieve at least twice the throughput compared to floating-point arithmetic by more efficiently utilizing the available DSP resources Aydonat *et al.* (2017). Moreover, lower precision also benefits the memory transactions to reduce the memory access delay and energy cost.

The design of the MAC units to compute convolution and fully-connected layers

Example	*bit_total*	*bit_sign*	*bit_int*	*bit_frac*	*Real number*	*Fixed-point integer*
Input Pixel	16	1	13	2	6789.625	27158
Weight	8	1	-2	9	0.203125	104
Output Pixel	16	1	12	3	1379.142	11033

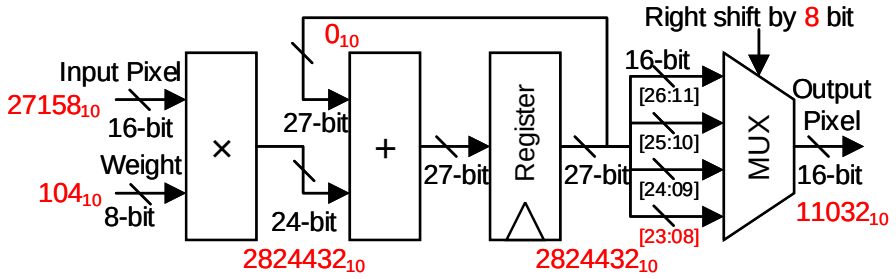

Figure 6.3: The design of one MAC unit with dynamic quantization for convolution and FC operations, where the multiplier is implemented by DSP and the adder is implemented by logic.

are shown in Figure 6.3 with an example to illustrate dynamic quantization. The inputs, weights and outputs are assumed to have bit_total = 16, 8, and 16 and bit_fra = 2, 9, and 3, respectively, as listed in the table inside Figure 6.3, where the multiplier has 24 (=16+8) bit of outputs and the adder has 27 bit of outputs with 3 redundant bit for accumulation. Since the data range of weights and features in one layer could be quite different, we set independent exponents or bit_fra for weights and features. The different bit_fra of inputs and outputs is caused by the different feature value ranges between different layers, or the decimal point is floated across different layers. In order to fit the 27 bit MAC output into the same number of bits as the 16 bit input, the 27 bit output must be truncated or right shifted. The number of bit to be right shifted (bit_right) is determined by the bit_fra of input, weight and output:

$$bit\_right = \\ bit\_fra_{input} + bit\_fra_{weight} - bit\_fra_{output}. \tag{6.6}$$

In the example inside Figure 6.3, the MAC output 2,824,432 needs to be right shifted by 8 ($= 2 + 9 - 3$) bits or discarding the 8 LSB to be 11032, which is different from 11033 in the table because of the error caused by truncation and limited precision. Since different layers may have different bit_right, a multiplexer is needed at the end to choose different truncated outputs with different bit_right, which is the only hardware overhead caused by dynamic quantization compared to the static fixed point design. For the inference phase, the weights are pre-trained so that we can calculate bit_fra and bit_int of each layer off-line before execution as shown in Figure 6.2. Then, all the weights are dynamically quantized by rounding to be fixed point integer numbers as in Equation 6.4 and stored in external DRAM to be used by the hardware CNN accelerator. The ranges of feature values are obtained from testing the overall dataset, and then bit_fra and bit_int of each layer are calculated. By this means, the bit_right of each layer is calculated by Equation 6.6 to control the multiplexer inside the MAC unit.

The detection accuracies of floating-point arithmetic, dynamic quantization and conventional fixed point arithmetic on VOC07+12 test dataset are compared in Table 6.2 for original SSD300 and hardware friendly SSD300_HW. 16-bit precision with dynamic quantization can provide the same level of accuracy compared with single-precision floating-point arithmetic for both original and modified SSD algorithms. For conventional fixed-point arithmetic, bit_int has to be large enough to cover the wide range of data of the entire SSD algorithm leading to fewer bit_fra and lower precision. Compared with weights, features are more sensitive to precision and require more bit width. Since 8-bit weights do not reduce the accuracy significantly and can save a considerable amount of logic and memory usage, we decide to use 8-bit weights and 16-bit features with dynamic quantization.

Table 6.2: The accuracies of original SSD300 and hardware-friendly SSD300_HW with different inference precisions are compared on VOC07+12 test set, and the highlighted precision is chosen for FPGA implementation.

Model	Weight Precision	Pixel Precision	Dynamic Quantization	mAP
SSD300	FP-32	FP-32	-	77.30%
SSD300	16	16	√	77.29%
SSD300	8	16	√	77.06%
SSD300	16	8	√	59.36%
SSD300	8	8	√	58.82%
SSD300	16	16	×	75.21%
SSD300	8	16	×	74.68%
SSD300_HW	FP-32	FP-32	-	77.10%
SSD300_HW	16	16	√	77.11%
SSD300_HW	**8**	**16**	√	**76.94%**
SSD300_HW	6	16	√	35.12%
SSD300_HW	16	8	√	53.60%
SSD300_HW	8	8	√	53.23%
SSD300_HW	16	16	×	74.85%
SSD300_HW	8	16	×	74.10%

6.3 Experiments

6.3.1 Experimental Setup

CPU and GPU: The baseline CPU used in the experiment is Intel Core i7-5930K with 6 cores, and the GPU is NVIDIA GeForce GTX 1080 Ti. Their detailed specifications are listed in Table 6.4. The software deep learning framework we used is Caffe Jia *et al.* (2014).

FPGA: The two Intel FPGAs used in the experiment are Arria 10 GX 1150 and

Stratix 10 GX 2800. The main FPGA computation resources are DSP blocks and adaptive logic modules (ALM). The main memory resource on FPGA chip is the block random-access memory (BRAM) in terms of M20K with each M20K having 20 Kbit capacity. There are 1,518/5,760 DSP blocks, 427K/933K ALMs, and 2,713/ 11,721 M20K BRAMs on the used Arria 10 and Stratix 10, respectively. The underlying FPGA boards for Arria 10 and Stratix 10 are Nallatech 385A and Stratix 10 FPGA Development Kit, respectively, and both are equipped with DDR3 DRAM with peak memory bandwidth of 16.9 GB/s.

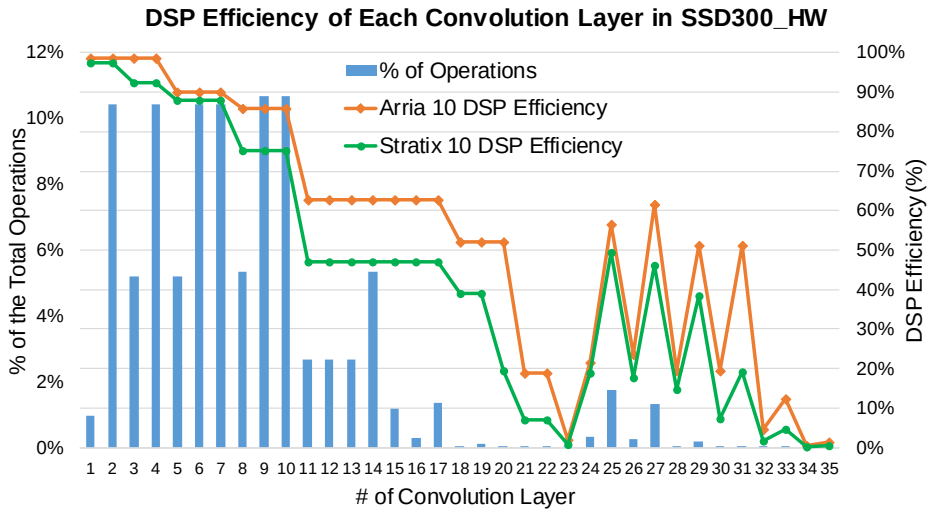

Figure 6.4: The DSP efficiency of each convolution layer in SSD300_HW is used to measure the match degree between parallel computation scheme and the feature maps.

6.3.2 Discussion of Results

Parallel Computation Efficiency

To achieve better performance with higher parallelism, we attempt to maximize the usage of DSP blocks for the MAC operations. Each DSP supports two fixed-point

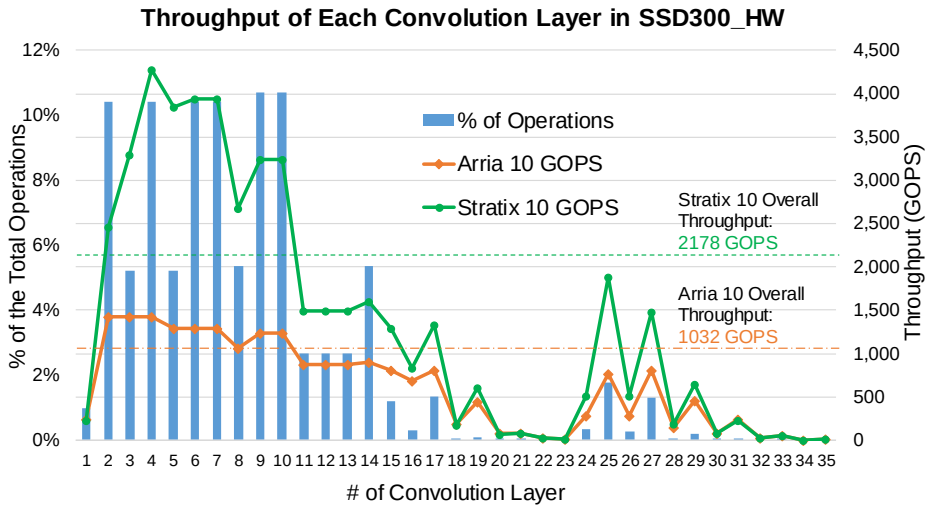

Figure 6.5: The throughput of each convolution layer in SSD300_HW is constrained by the DSP efficiency and memory bandwidth.

multipliers in two MAC units. Constrained by the number of available DSP blocks, we set the number of MAC units on Arria 10 and Stratix 10 to be 3,072 ($= 8 \times 6 \times 64$) and 8,192 ($= 16 \times 8 \times 64$), respectively. This means 8×6 or 16×8 features within the same output feature map are processed in parallel and such 64 output feature maps are simultaneously computed. Since the feature map sizes and output channel numbers vary significantly across different layers in SSD, the parallel degree and shape may not perfectly match the feature map size and dimension, which causes inefficient utilization of DSP blocks or MAC units. Therefore, the DSP efficiency Wei *et al.* (2017) is defined to measure how well the parallel computation scheme matches the feature maps:

$$DSP\_eff. = \frac{\#\quad effective\quad ops.}{\#\quad actual\quad performed\quad ops.}. \tag{6.7}$$

The DSP efficiency of each convolution layer is shown in Figure 6.4. The first several layers in SSD300 have large feature map sizes, e.g. 300×300 and 150×150, so that the parallel dimension can easily fit into the feature maps. The layers at the end for

multi-scale detection have much smaller feature maps, e.g. 10×10 and 5×5, which leads to considerable degradation of DSP efficiency. Fortunately, the first several layers account for most of the total operations as shown in Figure 6.4, which makes the overall DSP efficiency still high as 81.8% on Arria 10 and 71.5% on Stratix 10. Stratix 10 has larger parallel degrees than Arria 10, which makes it more difficult to match all the feature maps and results in lower DSP efficiency.

Table 6.3: Comparison of SSD300_HW with baseline SSD300_3 on Arria 10 and Stratix 10 FPGAs

FPGA	Arria 10 GX 1150		Stratix 10 GX 2800	
Model	SSD 300_3	SSD 300_HW	SSD 300_3	SSD 300_HW
Precision	8-16 bit	8-16 bit	8-16 bit	8-16 bit
mAP	77.45%	76.94%	77.45%	76.94%
Clock (MHz)	200	240	240	300
# MAC units	3,072	3,072	8,192	8,192
DSP Block	1,518	1,518	4,370	4,363
Logic (ALM)	220K	175K	618K	532K
BRAM (M20K)	2,586	2,581	3,862	3,844
Latency (ms)	72.2	61.4	35.2	29.1
GOPS	876	1,032	1,798	2,178

Throughput

The throughput of each convolution layer in SSD300_HW, which is determined by the number of MAC units, DSP efficiency, buffer sizes, and external memory bandwidth, is shown in Figure 6.5. If there is unlimited memory bandwidth, the shape of the

throughput curve in Figure 6.5 should match the DSP efficiency curve in Figure 6.4. With limited memory bandwidth, the memory access delay may be larger than the computation delay in some layers, or these layers are memory-bounded. For example, the first convolution layer (conv1_1) is memory bounded for both Arria 10 and Stratix 10. Although Stratix 10 can compute the MAC operations faster, it can only achieve the same throughput as Arria 10, because both of them are memory bounded with the same memory bandwidth. With higher computation speed and the same memory bandwidth, Stratix 10 encounters memory-bounded situations more often than Arria 10, which poses limitations on the throughput improvements of Stratix 10. With 8,192 MAC units operated at 300 MHz, the theoretical maximum throughput of Stratix 10 is 4,915 GOPS, which is 3.3× larger than the Arria 10 maximum throughput of 1,474 GOPS. However, Stratix 10 achieves 2.1× enhancement of throughput over Arria 10 due to the limited memory bandwidth and lower DSP efficiency.

SSD300_HW vs. Baseline SSD300_3

To evaluate the effect of tailoring SSD300 to achieve an efficient hardware implementation , e.g. SSD300_HW, we also implement SSD300_3 as in Table 6.1, where dilated convolution (fc6) and different convolution strides are unchanged. The detailed comparison results are listed in Table 6.3, including resource utilization and throughput. Due to the special dataflow of dilated convolution, dedicated control logic and data path router are designed in SSD300_3, which need extra design time and efforts. To support convolution layers with strides of two, additional data buses are used to feed proper data into the PEs. Therefore, SSD300_3 implementations on Arria 10 and Stratix 10 consume about 26% and 16% more logic elements (ALMs) than SSD300_HW, respectively, as in Table 6.3. Even worse, the additional data buses tighten the critical path and decrease the operating frequency leading to 1.17× and

1.21× throughput reduction compared to SSD300_HW, on Arria 10 and Stratix 10, respectively. The complex nonlinear function involved in the normalization of conv4_3 is expected to require considerably more design efforts and hardware resources that may result in even lower performance. Hence we did not continue to implement normalization for the baseline design. The example detection results of SSD300_HW are shown in Figure 6.6.

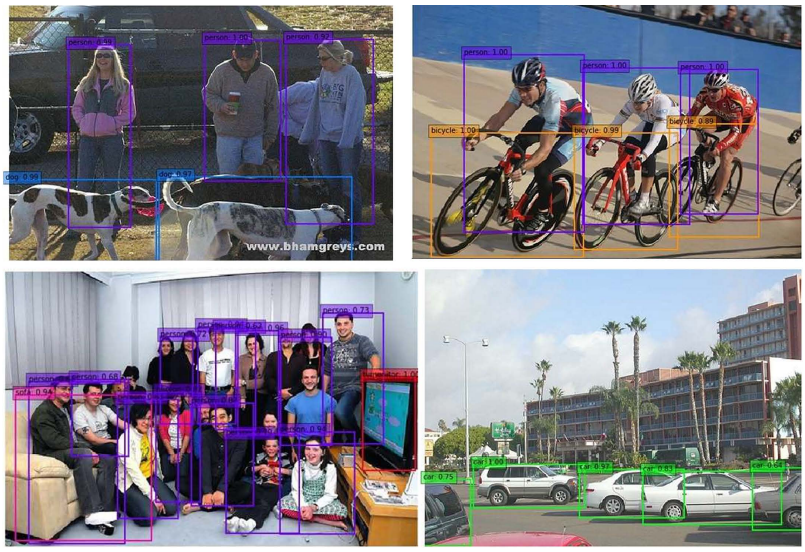

Figure 6.6: Example detection results of SSD300_HW.

FPGA vs. CPU, GPU

In Table 6.4, we compare our FPGA-based inference engine with CPU and GPU platforms, for SSD300 implementation. Many latency-critical inference applications, e.g. autonomous drive and surveillance, require the completion of detection at the speed of incoming data stream. Although the high batch size can improve the throughput by sharing the memory transfer delay, it worsens the latency between one input image and its detection result. Therefore, we set the batch size to be 1 for all the platforms to achieve the minimum latency per image. The results of CPU and GPU are based

Table 6.4: SSD300 Inference Performance and Efficiency Comparison on Different Platforms with Batch Size = 1

Platform	Intel Core i7-5930K CPU	NVIDIA GeForce GTX 1080 Ti GPU	Intel Arria 10 GX 1150 FPGA	Intel Stratix 10 GX 2800 FPGA
Technology	22 nm	16 nm	20 nm	14 nm
Clock Frequency	3.50 GHz	1.48 GHz	240 MHz	300 MHz
Max. Memory BW	68 GB/s	484 GB/s	16.9 GB/s	16.9 GB/s
Precision	FP-32 bit	FP-32 bit	fixed 8-16 bit	fixed 8-16 bit
mAP of SSD300	77.30%	77.30%	76.94%	76.94%
Latency/Image (ms)	3,272.2	32.58	61.45	29.11
Overall Throughput	19.5 GFLOPS	1,956 GFLOPS	1,032 GOPS	2,178 GOPS
Power (W)	140	250	40	100
Energy/Image (J)	458	8.1	2.4	2.9
Efficiency (GOP/J)	0.14	7.82	25.8	21.8

Note that we employed the SSD300 algorithm with data augmentation, which shows 77.3% mAP but GPU performance was not reported in Liu *et al.* (2016). For SSD300 without data augmentation, 46 fps was reported for Titan X GPU, but mAP was degraded to 74.3%.

on the original SSD300 algorithm using single-precision floating-point numbers, and the FPGA results are based on the hardware-friendly SSD300_HW as in Table 6.2, which uses 8-bit weights and 16-bit features with dynamic quantization to achieve the same accuracy level as software. Aided by the customized hardware architecture specific for CNN inference acceleration, Arria 10 achieves 53× higher performance than CPU and Stratix 10 obtains 1.12× better throughput than GPU, even if FPGAs suffer

from lower clock frequency and much less memory bandwidth. Due to the difficulty of directly measuring the power of CPU, GPU and FPGA, the listed power numbers are from their datasheet specifications for only rough estimation. Based on this, Arria 10 and Stratix 10 FPGAs can achieve 3.3× and 2.8× better energy-efficiency compared to GPU with 6.3× and 2.5× less power consumption, respectively.

6.4 Summary

In this chapter, we presents an efficient hardware implementation of the SSD300 object detection algorithm, tailored for an FPGA. The proposed design, SSD300_HW, achieves this through three basic innovations. These are: 1) replacing the dilated convolution with a normal convolution, 2) using a constant scale instead of normalization, and 3) using a uniform convolution sliding stride. Fixed-point arithmetic is employed to reduce the computation resource usage, which significantly enhances the FPGA inference performance, and the dynamic quantization is used to remain the detection accuracy of floating-point representation. The proposed FPGA-based inference engines achieve 1.03 TOPS and 2.18 TOPS throughput for SSD300_HW on Intel Arria 10 and Stratix 10 FPGA, respectively, and they also consume 6.3× and 2.5× less power and obtain 3.3× and 2.8× better energy efficiency, respectively, compared to a high-end GPU.

Chapter 7

CONCLUSION

Many reported successes of deep learning algorithms for computer vision tasks have motivated the development of hardware implementations of CNNs. In particular, there has been increased interest in FPGAs as a platform to accelerate the post-training inference computations of CNNs. To achieve high performance and low energy cost, a CNN accelerator must 1) fully utilize the limited computing resources to maximize the parallelism, 2) exploit data locality by saving only the required data in on-chip buffers to minimize the cost of DRAM accesses, and 3) manage the data storage patterns in buffers to increase the data reuse.

In this dissertation, a complete framework is proposed to compile the software deep CNN algorithms and automatically map the inference processes onto the high-performance FPGA accelerator, where an efficient dataflow and hardware architecture are designed based on the convolution loop optimization and the design space is explored through the proposed performance model.

The convolution loop optimization strategy is quantitatively analyzed in Chapter 3 aimed at efficient accelerator dataflow and high performance hardware architecture. Chapter 4 presents the RTL compiler that enables fast and automatic mapping of various deep CNN algorithms from software deep learning frameworks, e.g. Caffe, onto FPGA hardware. A high-level performance model is proposed in Chapter 5 to estimate the throughput and resource utilization of the CNN inference accelerators allowing design space exploration at early design stage. Chapter 6 performs software-hardware co-design to customize the SSD object detection algorithm to benefit its hardware implementation with low data precision.

REFERENCES

Aydonat, U., S. O'Connell, D. Capalija, A. C. Ling and G. R. Chiu, "An OpenCL^{TM}deep learning accelerator on Arria 10", in "Proc. of ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)", (Feb., 2017). 1, 2.4, 3.6.1, 3.3, 3.6.3, 6.2.3

Bacon, D. F., S. L. Graham and O. J. Sharp, "Compiler transformations for high-performance computing", ACM Computing Surveys (CSUR) **26**, 4, 345–420, URL `http://doi.acm.org/10.1145/197405.197406` (1994). 1.1.1, 3.1.1

Bankman, D., L. Yang, B. Moons, M. Verhelst and B. Murmann, "An always-on 3.8μJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28nm CMOS", in "IEEE Int. Solid-State Circuits Conf. (ISSCC)", pp. 222–224 (Feb., 2018). 4.6.5

Bosi, B., G. Bois and Y. Savaria, "Reconfigurable pipelined 2-D convolvers for fast digital signal processing", IEEE Trans. on Very Large Scale Integration (VLSI) Systems (TVLSI) pp. 299–308 (1999). 3.5.1

Chen, Y., J. S. Emer and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks", in "ACM/IEEE Int. Sym. on Computer Architecture (ISCA)", pp. 367–379 (Jun., 2016). 1.1.1, 3.2.2, 3.2.5, (C), 3.6.3, 5.6

Chen, Y., 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 (JSSC) pp. 127–138 (2017). (C)

Du, L., Y. Du, Y. Li and M. F. Chang, "A reconfigurable streaming deep convolutional neural network accelerator for Internet of Things", IEEE Trans. Circuits Syst. I, Reg. Papers (TCAS-I) (2017). (C), 3.3

Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results", (2012). (document), 6.1

Gokhale, V., J. Jin, A. Dundar, B. Martini and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks", in "IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) Workshops", pp. 696–701 (2014). 1

Guan, Y., H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: an automated framework for mapping deep neural networks onto fpgas with RTL-HLS hybrid templates", in "IEEE Int. Sym. on Field-Programmable Custom Computing Machines (FCCM)", (May, 2016). 4.6.5, 4.2

Guan, Y., H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates", in "IEEE Int. Sym. on Field-Programmable Custom Computing Machines (FCCM)", (May, 2017). 2.4, 3.3, 3.6.3

Guo, K., L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang and H. Yang, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA", IEEE Trans. on CAD of Integrated Circuits and Systems **37**, 1, 35–47 (2018). 1.1.1, (A), 3.3, 3.6.1, 3.3, 4.6.1, 4.6.5, 4.6.5, 4.2

Han, S., 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 "ACM/IEEE Int. Sym. on Computer Architecture (ISCA)", pp. 243–254 (Jun., 2016a). 3.2.5

Han, S., H. Mao and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding", URL `http://arxiv.org/abs/1510.00149` (2016b). 5.6

He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", in "IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)", (Jun., 2016a). 1.1.1, 1.1.2, 2.1, 2.2, 3.3, 3.5.1, 3.6.1, 3.6.3, 4.3.1, 4.5.3, 5.7.2

He, K., X. Zhang, S. Ren and J. Sun, "Identity mappings in deep residual networks", in "Computer Vision (ECCV)", (Oct., 2016b). 1.1.2

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", arXiv preprint arXiv:1408.5093 (2014). 1.1.2, 4.1, 6.3.1

Krizhevsky, A., I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in "Conf. on Neural Information Processing Systems (NIPS)", (2012). 1, 1.1.1, 1.1.2, 2.1, 2.1, 2.2, 3.3, 3.5.1, 5.7.2

Li, H., X. Fan, L. Jiao, W. Cao, X. Zhou and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks", in "Int. Conf. on Field Programmable Logic and Applications (FPL)", pp. 1–9 (Aug., 2016). (A), 3.3, 3.6.3

Lin, M., Q. Chen and S. Yan, "Network In Network", CoRR URL `http://arxiv.org/abs/1312.4400` (2013). 1.1.2, 2.2, 3.6.1, 5.7.2

Lin, X., S. Yin, F. Tu, L. Liu, X. Li and S. Wei, "LCP: A layer clusters paralleling mapping method for accelerating Inception and Residual networks on FPGA", in "Proc. of Design Automation Conference (DAC)", (Jun., 2018). 5.8.2

Liu, W., D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu and A. C. Berg, "SSD: single shot multibox detector", in "Computer Vision (ECCV)", (Oct., 2016). 1.1.4, 6.1, 6.1.1, 6.1.2, 6.4

Ma, Y., Y. Cao, S. Vrudhula and J. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA", IEEE Trans. on Very Large Scale Integration (VLSI) Systems (2018a). 1.1.3, 4.1, 4.2.1, 4.2.2, 4.2.2, 4.4.4, 4.5.1, 4.5.2, 4.6.1, 4.6.5, 5.1, 5.3.2, 5.3.3, 5.5.1, 5.7.1, 5.7.1

Ma, Y., Y. Cao, S. Vrudhula and J. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks", in "Proc. of ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)", (Feb., 2017b). 2.4, (D), 3.4.4, 3.5.1, 3.6.2, 6.2, 6.2.2

Ma, Y., Y. Cao, S. Vrudhula and J. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks", in "Int. Conf. on Field Programmable Logic and Applications (FPL)", (Sep., 2017a). 1, 2.4, 3.6.1, 4.6.4, 5.3.3, 5.4.3

Ma, Y., N. Suda, Y. Cao, J. Seo and S. B. K. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA", in "Int. Conf. on Field Programmable Logic and Applications (FPL)", pp. 1–8 (Aug., 2016). (B), 3.6.1

Ma, Y., N. Suda, Y. Cao, S. Vrudhula and J. Seo, "ALAMO : FPGA acceleration of deep learning algorithms with a modularized RTL compiler", Integration, the VLSI Journal (2018b). 2.4, 4.5.1, 4.5.2, 4.6.5

Moons, B., R. Uytterhoeven, W. Dehaene and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI", in "IEEE Int. Solid-State Circuits Conf. (ISSCC)", pp. 246–247 (Feb., 2017). 4.6.5

Motamedi, M., P. Gysel, V. Akella and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks", in "Asia and South Pacific Design Automation Conference (ASP-DAC)", pp. 575–580 (Jan., 2016). 1.1.1, (A)

Putic, M., S. Venkataramani, S. Eldridge, A. Buyuktosunoglu, P. Bose and M. Stan, "Dyhard-DNN: even more DNN acceleration with dynamic hardware reconfiguration", in "Proc. of Design Automation Conference (DAC)", (Jun., 2018). 5.8.3

Qiu, J., J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network", in "Proc. of ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)", (Feb., 2016). 2.4, 6.2, 6.2.2

Rahman, A., J. Lee and K. Choi, "Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array", in "Design, Automation Test in Europe Conf. Exhibition (DATE)", pp. 1393–1398 (Mar., 2016). (D), 3.3, 3.3, 3.6.3

Redmon, J., S. K. Divvala, R. B. Girshick and A. Farhadi, "You only look once: Unified, real-time object detection", in "2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016", pp. 779–788 (2016). 6.1

Ren, S., K. He, R. B. Girshick and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks", in "Conf. on Neural Information Processing Systems (NIPS)", (Dec., 2015). 6.1

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", International Journal of Computer Vision (IJCV) (2015). 2.2

Sharma, H., J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra and H. Esmaeilzadeh, "From high-level deep neural models to fpgas", in "IEEE/ACM Int. Sym. on Microarchitecture (MICRO)", (Oct., 2016). 4.6.5

Shin, D., J. Lee, J. Lee and H. Yoo, "14.2 DNPU: an 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks", in "2017 IEEE International Solid-State Circuits Conference, ISSCC 2017, San Francisco, CA, USA, February 5-9, 2017", pp. 240–241 (2017). 1, 6.2, 6.2.2

Simonyan, K. and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", URL `http://arxiv.org/abs/1409.1556` (2014). 1.1.1, 1.1.2, 2.2, 3.6.1, 5.7.2

Song, L., Y. Wang, Y. Han, X. Zhao, B. Liu and X. Li, "C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization", in "Proc. of Design Automation Conference (DAC)", (Jun., 2016). 5.8.3

Suda, N., V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks", in "Proc. of ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)", (2016). 1.1.1, 1.1.2, 2.4, (A), 3.6.1, 3.3, 6.2

Sutskever, I., O. Vinyals and Q. V. Le, "Sequence to sequence learning with neural networks", in "Conf. on Neural Information Processing Systems (NIPS)", pp. 3104–3112 (Dec, 2014). 1

Szegedy, C., S. Ioffe, V. Vanhoucke and A. A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning", in "Proc. of Conf. on Artificial Intelligence (AAAI)", (Feb., 2017). 1.1.2, 2.1, 2.1, 4.5.3, 4.5.5

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going deeper with convolutions", in "IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)", (Jun., 2015). 1.1.2, 2.1, 2.2, 4.5.5

Venieris, S. I. and C. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs", in "IEEE Int. Sym. on Field-Programmable Custom Computing Machines (FCCM)", (May, 2016). 1.1.2, 2.4, 4.6.5

Wang, Y., J. Xu, Y. Han, H. Li and X. Li, "DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family", in "Proc. of Design Automation Conference (DAC)", (Jun., 2016). 4.6.5, 4.2

Wei, X., C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs", in "Proc. of Design Automation Conference (DAC)", (2017). 1, 2.4, 3.3, 3.6.3, 4.6.2, 4.6.5, 4.2, 5.4.2, 6.2, 6.3.2

Yu, F. and V. Koltun, "Multi-scale context aggregation by dilated convolutions", CoRR URL `http://arxiv.org/abs/1511.07122` (2015). 6.1.1

Zeng, H., R. Chen, C. Zhang and V. K. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs", in "Proc. of ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)", (Feb., 2018). 2.4, 4.6.5, 4.2

Zhang, C., Z. Fang, P. Zhou, P. Pan and J. Cong, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks", in "Proc. of Int. Conf. on Computer-Aided Design (ICCAD)", (Nov., 2016a). 1.1.3, 2.4, 4.6.5, 4.6.5, 4.2

Zhang, C., P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks", in "Proc. of ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)", (2015). 1, 1.1.1, 1.1.2, 1.1.3, 2.4, 3.1.1, (B), 4.4.4, 4.4.5, 4.4.5, 5.4.2, 5.7.2

Zhang, C., D. Wu, J. Sun, G. Sun, G. Luo and J. Cong, "Energy-efficient CNN implementation on a deeply pipelined (FPGA) cluster", in "Int. Symp. on Low Power Electronics and Design (ISLPED)", pp. 326–331 (Aug., 2016b). 1.1.1, 3.3, 3.6.1, 3.3, 3.6.3