UNIVERSITY OF CALIFORNIA

Los Angeles

FPGA Overlay Processor for Deep Neural Networks

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical and Computer Engineering

by

Yunxuan Yu

2020

ABSTRACT OF THE DISSERTATION

FPGA Overlay Processor for Deep Neural Networks

by

Yunxuan Yu
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2020
Professor Lei He, Chair

The rapid advancement of Artificial intelligence (AI) is making our everyday life easier with smart assistants, automatic medical analyzer, bank plagiarism checkers and traffic predictions, etc. Deep learning algorithms, especially deep convolutional neuron networks (DCNNs), achieve top performance on AI tasks, but suffers from dense computational requirements, which calls for hardware acceleration. In this thesis we propose several architectures including compilation flow for general DCNN acceleration using FPGA platform.

Starting from late 2015 we began to design customized accelerators for popular DCNNs such as VGG and YOLOv2. We reformulate the convolution computation by flattening it to large-scale matrix multiplication between feature maps and convolution kernels, which can be computed as inner product. With this formulation, the accelerators across all layers can be unified to enhance resource sharing, and maximize utilization of computing resources. We also quantized the network into 8bit with negligible accuracy loss to reduce memory footprint and computation resources. Different parallelism optimization strategies are explored for different networks. The VGG16 accelerator achieved 1.15x throughput under 1.5x lower frequency compared with state-of-the art designs. The YOLOv2 accelerator was commercialized and employed for real-time subway X-ray auto-hazard detection.

Based on the experience we gained through customized accelerator designing, we designed a RTL compiler as an end-to-end solution to automatically generate RTL design for given CNN network and FPGA platform, which greatly reduced the human effort in developing a specific network accelerator. The compiler applies analytical performance models to optimize parameters for modules based on a handwritten template library, such that the overall throughput is maximized. Several levels of parallelism for convolution are explored, including inter feature-map, intra-kernel-set, input/output channel, etc. We also optimize architectures for block RAM and input buffers to speed up data flow. We tested our compiler on several well-known CNNs including AlexNet and VGGNet for different FPGA platforms. The resulting AlexNet is 113.69 GOPS on Xilinx VCU095 and 177.44 GOPS on VC707, and VGGNet is 226 GOPS on VCU095 under 100MHZ. These are 1.3×, 2.1× and 1.2× better than the best reported FPGA accelerators at that time, respectively.

However, network-specific accelerator requires regeneration of logic and physical implementation whenever network is updated. Moreover, it certainly cannot handle cascaded network applications that are widely employed in complex real-world scenarios. Therefore, we propose a domain-specific FPGA overlay processor, named OPU to accelerate a wide range of CNN networks without re-configuration of FPGA for switch or update of CNN networks. We define our domain-specific instruction architecture with optimized granularity to maintain high efficiency while gaining extra progammability. We also built hardware micro-architectures on FPGA to verify ISA efficiency, and a compiler flow for parsing, optimization ans instructuin generation. Experiments show that OPU can achieve an average of 91% run-time MAC efficiency (RME) among various popular networks. Moreover, for VGG and YOLO networks, OPU outperforms automatically compiled network-specific accelerators in the literature. In addition, OPU shows 5.35× better power efficiency compared with Titan Xp. For a case using cascaded CNN networks, OPU is 2.9× faster compared with edge computing GPU Jetson Tx2 with a similar amount of computing resources. Our OPU platform was employed in an automatic curbside parking charging system in real-world.

Using OPU as base design, we extend different versions of OPU to handle the newly emerged DCNN architectures. We have *Light-OPU* for light-weight DCNNs acceleration, where we modified the OPU architecture to fit the memory bounded light-weight operations. Our instruction architecture considers the sharing of major computation engine between LW operations and conventional convolution operations. This improves the run-time resource efficiency and overall power efficiency. Our experiments on seven major LW-CNNs show that *Light-OPU* achieves 5.5× better latency and 3.0× higher power efficiency on average compared with edge GPU NVIDIA Jetson TX2. Moreover, we also have Uni-OPU for the efficient uniform hardware acceleration of different types of transposed convolutional (TCONV) networks as well as conventional convolution (CONV) networks. Extra stage in compiler would transform the computation of Zero-inserting based TCONV (Zero-TCONV), nearest-neighbor resizing based TCONV (NN-TCONV) and CONV layers into the same pattern. The compiler conducts the following optimizations: (1) Eliminating up to 98.4% of operations in TCONV by making use of the fixed pattern of TCONV upsampling; (2) Decomposing and reformulating TCONV and CONV processes into streaming paralleled vector multiplication with uniform address generation scheme and data flow pattern. *Uni-OPU* can reach throughput up to 2.35 TOPS for TCONV layer. We evaluate *Uni-OPU* on a benchmark set composed of six TCONV networks from different application fields. Extensive experimental results indicate that *Uni-OPU* is able to gain 1.45× to 3.68× superior power efficiency compared with state-of-the-art Zero-TCONV accelerators. High acceleration performance is also achieved on NN-TCONV networks, whose acceleration have not been explored before. In summary, we observe 15.04× and 12.43× higher power efficiency on Zero-TCONV and NN-TCONV networks compared with Titan Xp GPU on average. To the best of our knowledge, we are the first in-depth study to completely unify the computation process of both Zero-TCONV, NN-TCONV and CONV layers.

In summary, we have been working on FPGA acceleration for deep learning vision algorithms. Several hand-coded customized accelerator as well as an auto-compiler that generates RTL code for customized accelerator have been developed. An initial tool-chain for an FPGA based overlay processor was also finished, which can compile DCNN network configuration file from popular deep learning platforms and map to processor for acceleration.

The dissertation of Yunxuan Yu is approved.

Abeer Alwan

Yong Chen

Puneet Gupta

Lei He, Committee Chair

University of California, Los Angeles

2020

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

VITA

2010–2014	B.S., Microelectronics, Department of Electrical Engineering and Computer Science, Peking University, Beijing, China.
2014–now	MS/PhD program, Department of Electrical and Computer Engineering, University of California, Los Angeles, California, USA
2015 - 2019	Teaching Assistant/Research Assistant, ECE, 205A (3 quarters)/113DA (3 quarters)/113DB (3 quarters)/209 (one quarter)
2018	Internship, Synposys FPGA Eumlation Group, San Jose
2020	Internship, Amazon Annapurna lab, San Jose
2020	Internship, Synposys Design Compiler Group, San Jose

PUBLICATIONS

Yu, Y., Wu, C., Zhao, T., Wang, K. and He, L., 2019. OPU: An fpga-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(1), pp.35-47.

Yu, Y., Zhao, T., Wang, K. and He, L., 2020, February. Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks. In The *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 122-132)*

Yu, Y., Zhao, T., Wang, M., Wang, K. and He, L., 2020. Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*

Yu, Y., Wu, C., etc. "Overview of a FPGA-Based Overlay Processor." *2019 China Semiconductor Technology International Conference (CSTIC)*. IEEE, 2019.

Yu, Y., and He, L. "FPGA Power Estimation Using Automatic Feature Selection." *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016.

CHAPTER 1

Introduction

1.1 Background

1.1.1 The rising of deep learning

Neural Network based deep learning has obtained tremendous advancement in the past few years, which brought Artificial Intelligence (AI) to our everyday life. The smart assistants in our home and cell phone, automatic medical analyzer, bank plagiarism checkers, traffic predictions and shopping recommendations help make our life easier.

The power of deep learning was recognized in 2012 ImageNet [RDS^{+}15] contest, when AlexNet [KSH12] beat the traditional machine learning methods with handcrafted features and won. From then we have seen the emergence of various more accurate and powerful networks. In 2013 ZFNet [ZF14] reduce the top-5 error rate from 15.3% to 11.2%; In 2014 Inception V1 [SLJ^{+}15] and VGGNet [SZ14a] were presented, not only did these two networks reduced the top-5 error rate to 6% to 7%, they also became the most popular backbones for numerous future networks for both classification, detection and segmentation. Another DCNN structure that has big influence on later DCNN architecture design is the residual layer concept used by ResNet [HZRS16b] in 2016. At this stage the accuracy of deep learning for classification has already achieved "superhuman performance", which means the machine running DCNN can be more accurate then an actual human being in this classification task specifically.

1.1.2 DCNN for computer vision tasks

DCNN has great performance on various traditional computer vision tasks. **Object Classification** requires recognizing previous learned object classes in a 2D image or 3D poses in the scene. The previous mentioned AlexNet, VggNet, ResNet, etc. handles classification problems very well. **Object Detection** is another important area in computer vision, which requires detect certain types of the object in give image and draw bounding box as well as give out classification probability. The development of object detection DCNN has been through several stages, famous networks include Fast-RCNN [Gir15], Mask-RCNN [HGDG17], YOLO V1-V4 [RDGF16][RF17][RF18][BWL20] have become very popular. Furthermore, Unet [RFB15], FCN [XTW$^+$18] performed well on **segmentation** tasks. FSR-CNN [DLT16] outperforms traditional algorithm on **Image Super Resolution**. On **Image synthesis** area GANs [ZXL$^+$17][TCAT17][KCK$^+$17][WZZH17] have opened various possibilities including style transfer, 3D Object generation, Photo Blending, Face Frontal View Generation, etc.

The outstanding performance of DCNN on computation vision tasks has render DCNN more and more important as a research topic.

1.1.3 The curse of computation complexity

The great performance of DCNN comes with a price, it requires far more computation compared with traditional method. The 2D convolution operation is widely applied in DCNN (will be introduced in detail in 2.2). However, a single layer of convolution with $224 \times 224 \times 16$ input image size, $224 \times 224 \times 64$ output image size and a 3×3 kernel size would require $462M$ multiplication operations. And a general DCNN would have tens to hundreds convolution layers. VggNet needs $30G$ operations in total to process one image, which may take a low-end CPU several seconds to execute.

High computation complexity of DCNN constraints its application scenario and calls for hardware acceleration solutions.

1.2 Motivation

1.2.1 FPGA Acceleration of DCNNs

High computation of DCNN motivates accelerations by computing clusters [DCM+12], GPUs [CWV+14], Domain Specific ASICs [LCL+15][LCL+15][JYP+17] and FPGAs [ORK+15]. The comparison between different platforms are listed in Table 1.1. It can be seen that FPGA has relatively good properties on verious aspects can obvious drawbacks. Moreover, FPGA has the advantage of lower Non-recurrent engineering as it does not require the long and expensive tape out process. An update of the design can be accomplished by simple reconfiguration. This can be an advantage in DCNN acceleration area as the development of new DCNN architecture is at a high speed with new operators emerging every year. ASIC can fall into the dilemma of trying to be general to accommodate more future operators and lose it's domain specific advantage.

Therefore, we designed and implemented a customized FPGA accelerator for VGGnet (in 2016), which is the start-of-the art classification network at that time. Details can be find in Chapter 2.

Table 1.1: DCNN Acceleration platform comparison

	GPU	ASIC	FPGA
Flexibility	high	low	medium
Power Efficiency	low	high	medium
Unit Price	high	low	medium
Technology iteration speed	fast	slow	fast

1.2.2 An Auto-code generator

Customized FPGA accelerator has been widely explored since 2016 [ZLS$^+$15a, QWY$^+$16, SCD$^+$16, SPM$^+$16, MCVS17a, ZFZ$^+$16, WYZ$^+$17], demonstrating FPGA as a promising acceleration platform for DCNN. However, design and implement a high-performance customized FPGA accelerator for a certain network can be time-consuming as it normally involves parallel architecture exploration, memory bandwidth optimization, area and timing tuning, as well as software-hardware interface development.

Therefore, an auto-code generator that can produce RTL code based on FPGA board constraints (memory, DSP, LUT, DDR bandwidth, etc.) and Network configuration (kernel size, layer types, input/output channel, etc.) can be a useful end-to-end solution for FPGA accelerator users. Here we proposed automatic DCNN accelerator RTL generator based on a hand-coded architecture template. Detailes can be found in Chapter 3.

1.2.3 Deep Learning for Edge

The huge computation requirement of deep learning constrains it to data center and servers. Meanwhile, the ever-increasing number of Internet of Things (IoT) devices are creating enormous amount of data at edge. Sending these data back to data center would cause huge network burden, and slow the processing speed as well. For example, parking surveillance cameras at the light poles are capturing high resolution (1920 x1080) pictures at 24 frames/s, which generates 712MB data per one second for a five camera device. If we can process these data right at the source, we can reduce them to less than 1KB of result before sending back to the server. Edge computing for AI is in high demand.

However, current platforms for edge deep learning acceleration have limitations. GPU based platform is the most popular one, but it suffers from high single unit price and lack of domain specific architecture, which greatly reduces its acceleration effect on deep learning algorithm with uncommon architectures, such as light-weight CNNs and Transpose CNNs.

ASIC based platform requires very high NRE cost for circuit design and fabrication. Applications with low volume can hardly meet the price bar. Moreover, ASIC designs have potential adaptation issues considering the fast evolvement of deep learning algorithms. As a re-configurable platform, FPGA can be used to implement the domain specific acceleration architecture for deep learning, but existing FPGA designs are mostly limited to customized accelerator for specific network, which lacks generality and cannot be widely adopted. So we developed an FPGA-based overlay processor (OPU) for the acceleration of various deep learning algorithms on the edge. We studied and evaluated all the computation patterns of popular CNN families and designed an instruction set architecture (ISA) that is flexible yet still maintaining domain specific optimization advantages. We then implemented the corresponding tool chain for the ISA, including algorithm parser, instruction compiler and the data flow driven hardware micro-architecture. We than evaluated our architecture with selected set of CNNs that cover the applications of object classification, object detection, human pose detection, semantic segmentation, image super-resolution and automatic image generation. We design and fabricate our own mother board(Fig. 1(a)) to achieve only half unit price compared with the most popular existing edge deep learning acceleration platform, while achieving 6.14x better performance and consumes 3.67x less energy [YWZ$^+$19, YZWH20, YZW$^+$20]. Details of OPU and its extension can be found in Chapter 456.

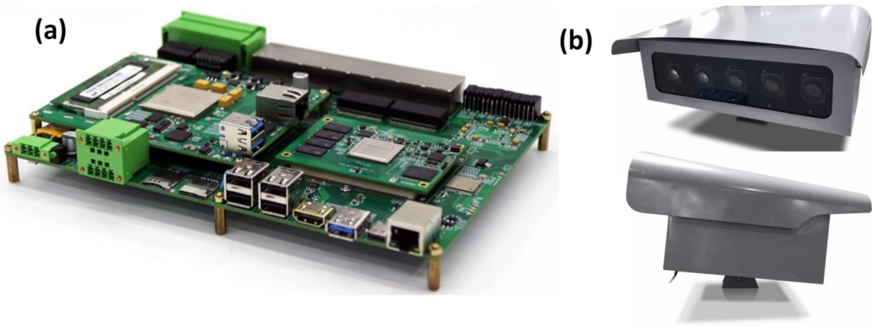


Figure 1.1: (a). OPU platform; (b). OPU platform installed inside box, connected wit 5 camera lines for smart city application

1.3 Organization of the Dissertation

The research presented in this dissertation mainly focuses on architecture design for deep learning acceleration on FPGA. The remaining parts of this dissertation are organized as follows:

- **Chapter 2: Customized FPGA Accelerator for CNN using uniform inner product engine.**
  A customized FPGA Accelerator is designed for the acceleration of VGGnet, which is a deep convolutional Neural Network (CNN) for image classification. We reformulate the convolution computation by flattening it to large-scale matrix multiplication between feature maps and convolution kernels, which can be computed as inner product (IP). With this formulation, the accelerators across all layers can be unified to enhance resource sharing,and maximize utilization of computing resources.

- **Chapter 3: Auto-compiler for customized FPGA accelerator**
  We present a RTL compiler to automatically generate Verilog code for given CNN network and for given FPGA platform. The compiler is based on a parameterized template, and applies analytical performance models to optimize parameters for modules in the template such that the overall throughput is maximized.

- **Chapter 4: *OPU*: An FPGA Overlay Processor for Convolutional Neural Networks**
  we propose a domain-specific FPGA overlay processor, named OPU to accelerate a wide range of CNN networks without re-configuration of FPGA for switch or update of CNN networks. We define OPU instructions including instructions for data block handling. We also optimize the granularity of instructions, microarchitectures on FPGA, and compiler to maximize parallelism.

- **Chapter 5: *Light-OPU*: An FPGA-based Overlay Processorfor Lightweight**

Convolutional Neural Networks

We extend our OPU overlay architecture of OPU to accelerate the newly emerged Light-weight CNNs. This Software-hardware co-designed Light-OPU reformulates and decomposes lightweight operations for efficient acceleration.

- **Chapter 6: *Uni-OPU*: An FPGA based Uniform Accelerator for Convolutional and Transposed Convolutional Networks** We extend our OPU overlay architecture of OPU for the efficient uniform hardware acceleration of different types of transposed convolutional (TCONV) networks and conventional convolution (CONV) networks. Specifically, a software compiler is provided to transform the computation of various TCONV and CONV layers, *i.e.*, Zero-inserting based TCONV (Zero-TCONV), nearest-neighbor resizing based TCONV (NN-TCONV) and CONV layers into the same pattern.

- **Chapter 7: Summary**

  The summary and future works are discussed.

- **Appendix** ISA specification for OPU.

CHAPTER 2

Customized FPGA Accelerator for CNN using Uniform Inner Product Engine

2.1 Introduction

In the early stage, accelerators have computation engines customized for each layers. Several FPGA based deep CNNs are proposed in 2009 - 2016 [FPHL09, CMB$^+$10, CSJC10, PSM$^+$13, ZLS$^+$15a]. Among them, [FPHL09] uses FPGA as a vectorial arithmetic unit, and implements CNN mainly on a 32bit soft processor for flexibility. Work in [ORK$^+$15, CMB$^+$10, CSJC10] design specific accelerator for each layer, while accelerators cannot be shared between layers[1]. The key to a high performance FPGA design is to make full use of on-chip resources, including both computing resources, (e.g., DSP slides), and enormous on-chip memory bandwidth. Using specific accelerators for each layer actually creates a dilemma between maximizing computational capability and memory bandwidth. To keep all the accelerators busy (optimize computational capability), accelerators have to be designed as a pipeline, which requires intermediate data for each accelerator. Due to the data size, they are typically stored in off-chip memory. Off-chip memory bandwidth becomes the bottleneck when multiple accelerators access them simultaneously.

This dilemma motivates the proposed FPGA architecture which speeds up CNN using uniform accelerators. Benefiting from uniform accelerators, we can process one layer at

[1]Work in [ZLS$^+$15a] designs uniform accelerator for convolutional layers, but the accelerator cannot be used for the fully-connected layer.

a time, virtually using all the computing resource. Moreover, the data required for one layer can be easily buffered in on-chip memory with massive bandwidth. This optimizes the memory access. Naturally, convolutions with different kernel size or stride cannot be implemented by the same accelerator. To tackle this problem, we reformulate the convolution to two separate steps, 1) convolution flattening (CF) to flatten the feature map and kernel from high-dimensional matrices to big 2-D matrices, 2) inner product (IP) between each rows and columns of the flattened feature map and kernel matrices. As the CF accelerators involve only light-weight memory manipulation, they consumes less resource compared with IP accelerators.

Our design philosophy is to share the computing resource across layers by formulate the convolution operations as CF operations and IP operations. While the CF operations at different layers might be different, the IP operations are unified at each layer, hence the expensive IP accelerators can be shared across all layers (even including the the fully connected (FC) layers).

Moreover, we compress the precision of the network parameters without impairing accuracy, therefore we can fit more accelerators on the same hardware with almost no accuracy loss. We use 8-bit fixed-point numbers instead of the single-precision floating-point number to represent the feature map and kernel weights in the network. On a large validation set, we verified that the CNN top-1 inference accuracy only drops 2.3%, and top-5 accuracy even increases 1% with more compact fixed-point numbers. On the other hand, we are able to implement about 6 times more accelerators on the same hardware when we optimize the algorithm w.r.t. fixed-point arithmetic unit.

Experimental results indicate that the proposed design achieves 245.04 GOPS at 125 MHZ. This is the fastest design compared with the state-of-the-art. Moreover, our design flow is semi-automated. The CF and IP accelerators with given parameters can be automatically synthesized, which make it easier to adopt the proposed architecture to new CNNs and new FPGA devices.

The remaining of this chapter is organized as follows: Section 2.2 introduces the background of CNN. Section 2.3 discusses the data quantization and its impact on accuracy and resource consumption. Section 2.4 presents the reformulation of convolution and the design of uniform accelerators. Section 2.5 describes the proposed architecture implementation details, evaluation and comparison with other related work. Section 2.6 concludes this chapter.

2.2 Basic CNN Architecture

2.2.1 Preliminary of Convolutional Neural Network

The area of Artificial Neural Networks (ANN) was originally inspired by the goal of modeling biological neural systems and has achieved good results in classification [RDS+15]. Regular ANN has an input vector and several hidden layers, where each layer has a set of neurons fully connected to the neurons in previous layer, resulting a large parameters matrix.

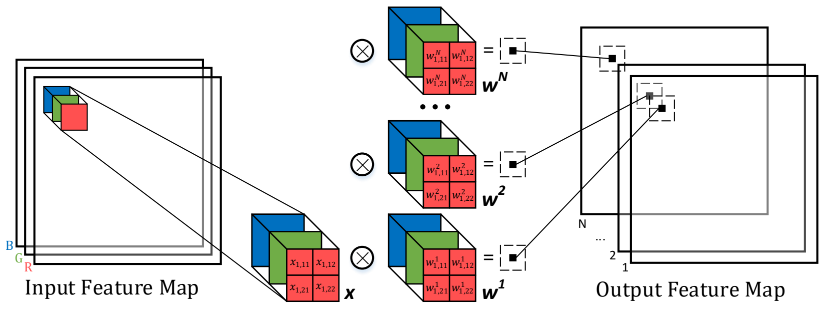


Figure 2.1: Convolutional Layer

Convolutional neural network (CNN) is proposed to reduce the number of parameters in regular ANN by featuring local connectivity and parameter sharing. As illustrated in Fig. 2.1, first, each neuron at convolutional layer is only connected to a local region of the input instead of the entire input feature map. Second, all neurons in a single depth slice are using the same weight vector. Those two features are reasonable for high dimensional data, such as images, where one pixel is only relevant to the pixels nearby.

2.2.2 Network Architecture of a Real-Life CNN

Typical CNN layers include convolutional layer, fully-connected (FC) layer and pooling layer. The convolutional and FC layers count for the majority of computation, while pooling layers only perform very simple calculation, e.g. getting the maximum or average out of a block of feature map. The architecture of a real-life CNN, VGG-16 designed by Visual Geometry Group at university of Oxford [SZ14a], is illustrated in Fig. 2.2. It is a winning top-5 classifier in ILSVRC 2014 [RDS+15], which categorizes the objects from high-resolution images into 1000 different classes.

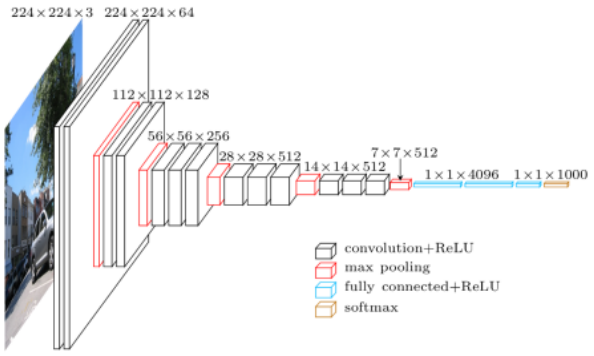


Figure 2.2: An illustration of the architecture of 16-layer VGG network (VGG-16) [SZ14a]

As shown in Fig. 2.2, the VGG-16 network consists of 16 computing layers (13 convolution layer and 3 FC layer), which are grouped in 8 layer sets. Other layers, such as rectifier and normalization layers, are not plotted in Fig. 2.2. Using 224x224x3 image as input, 2 convolutional layers produce 224x224x64 output feature map, which is then compressed to 112x112x64 via a max pooling layer. Those 3 layers form the first convolutional-pooling layer block. After going through 5 such convolutional-pooling layer blocks, the feature map is compressed 7x7x512 at the output of pool5. This feature map is then processed by 3 FC layers to generate an 1x1x1000 output, which can be used to infer the object class in the original image out of 1000 classes .

A detailed breakdown of the network parameters at each layer is presented in Table 2.1. It is obvious that convolutional layers and fully-connected layers count for almost 100% of the computational load, and the convolutional layers are the majority (97.2%) of them.

There are at lease two challenges in designing specific hardware for an accurate, but cumbersome network like VGG-16.

- First, their are more than 138 million network parameters in this network. Using single precision floating point number (32-bit), it could take 4222 Mbit (Mb) to store those network parameters, let alone the feature maps.
- Second, using specific accelerators for each layer actually creates a dilemma between maximizing computational capability and memory bandwidth, which motivates the design of uniform accelerators.

The discussions in Section III and IV address these two problems respectively.

2.3 Data Quantization

In this section, we discuss the data quantization, which improves the throughput of the proposed design at fine-grained data-level.

One major problem for the design of FPGA CNN accelerator is the large memory capacity required due to the enormous amount of kernel weights. While the on-chip memory is typically limited to sub-150 Mb even on some high-end FPGAs (Xilinx Ultrascale series[2]), the kernel weights can easily go up to 4222Mb when using 32bit single-precision floating-point representation. This motivates the development of fixed-point number based CNN, where both the kernel weights and intermediate product feature maps are truncated to less

[2]The particular FPGA used in this chapter (Xilinx Ultrascale XCVU095) has only 60Mb on-chip memory.

Table 2.1: A detail breakdown of the network parameter and computational load in each layer of VGG-16 [SZ14a]

Layer Name		Ker. size / Stride	Output Size	F_I [1]	F_O [1]	# of P. [2] (x10^6)	Gega OPS	Percentage	FPGA Latency[3] (x10^6 cycles)
L1	conv1-1	3x3/1	224x224x64	3	64	0.001	0.173	0.005 %	3.211
	conv1-2	3x3/1	224x224x64	64	64	0.036	3.699	11.626%	3.211
	pool1	2x2/2	112x112x64	64	64				
L2	conv2-1	3x3/1	112x112x128	64	128	0.073	1.850	5.813%	1.606
	conv2-2	3x3/1	112x112x128	128	128	0.147	3.699	11.626%	1.606
	pool2	2x2/2	56x56x128	128	128				
L3	conv3-1	3x3/1	56x56x256	128	256	0.294	1.850	5.813%	0.803
	conv3-2	3x3/1	56x56x256	256	256	0.589	3.699	11.626%	1.606
	conv3-3	3x3/1	56x56x256	256	256	0.589	3.699	11.626%	1.606
	pool3	2x2/2	28x28x256	256	256				
L4	conv4-1	3x3/1	28x28x512	256	512	1.179	1.850	5.813%	0.803
	conv4-2	3x3/1	28x28x512	512	512	2.359	3.699	11.626%	1.606
	conv4-3	3x3/1	28x28x512	512	512	2.359	3.699	11.626%	1.606
	pool4	2x2/2	14x14x512	512	512				
L5	conv5-1	3x3/1	14x14x512	512	512	2.359	0.925	2.907%	0.401
	conv5-2	3x3/1	14x14x512	512	512	2.359	0.925	2.907%	0.401
	conv5-3	3x3/1	14x14x512	512	512	2.359	0.925	2.907%	0.401
	pool5	2x2/2	7x7x512	512	512				
L6	fc6		1x1x4096			102.760	0.822	2.605%	–
L7	fc7		1x1x4096			16.78	0.034	0.108%	–
L8	fc8		1x1x1000			4.10	0.008	0.025%	–
Total						138.357	31.556	100.00%	18.868

[1] F_I/F_O: Number of input/output feature maps.

[2] # of P.: Number of parameters.

[3] FPGA Latency: Latency at each layer based on the implementation presented in this chapter.

precise fixed-point to reduce memory footage and bandwidth requirements. Consider the accuracy degradation that can be caused by data quantization, we need to find the best strategy to use as less bit as possible while maintain reasonable accuracy.

One realistic problem about the fixed-point number is the range of the data. Unlike the floating-point number, fixed-point numbers has very limited range. Without properly scaling, 8-bit fixed-point number can only represent numbers from -128 to 127. The feature map and weight parameters in a CNN could scale up and down at a much larger range. In our implementation we use dynamic fraction length to accommodate the data range of different layers. The process of finding the best fraction length can be described as below:

$$frac = \underset{floc}{\operatorname{argmin}} \sum (float - fix(floc))^2 \tag{2.1}$$

where float is the original single precision representation of kernel weights or the feature map, and fix(floc) is the value after the float is cut into fixed-point based on certain fraction length floc. A summary for different data quantization performance can be find in Table 2.2. It can be seen that when using dynamic fraction length, the accuracy is not impaired much until the precision is down to 8/4 bit.

Table 2.2: Fix Point Strategy Performance Comparison

Word Length		Accuracy (relative)	
Weights	Feature Map	Top-1	Top-5
32bit FL[1]	32bit FL	62.67%	85.14 %
16bit FI[2]	16bit FI	61.38% (-1.3%)	86.14% (+1.0%)
16bit FI	8bit FI	61.38% (-1.3%)	86.14% (+1.0%)
8bit FI	8bit FI	60.41% (-2.3%)	86.14% (+1.0%)
8bit FI	4bit FI	45.54% (-17.1%)	75.24% (-9.9%)
4bit FI	8bit FI	59.40% (-3.3%)	85.15% (-0.0%)

[1]**FI**: Fixed-point number

[2]**FL**: Floating-point number

Table 2.3: Dynamic fraction length strategy

fraction Length	Weights	Bias	FM_{OUT}
Conv1-1	7	5	-2
Conv1-2	9	7	-4
Conv2-1	8	8	-5
Conv2-2	9	7	-6
Conv3-1	9	8	-6
Conv3-2	9	8	-7
Conv3-3	9	7	-8
Conv4-1	10	8	-5
Conv4-2	10	9	-4
Conv4-3	10	8	-4
Conv5-1	10	7	-3
Conv5-2	11	7	-2
Conv5-3	11	3	-1

Finally the dynamic fraction length we chose to use is shown in Table 2.3. Negative fraction length indicates that the data is all in the integer part. For example, -2 for Conv1-1 output means the data is within range [2^9-1, 2^2] for positive number and [-2^9, -2^2] for negative number. Similarly, for fraction length bigger than 8, for instance the 11 for layer Conv5-3 means that the whole layer is in the fraction part, and the value range is within [2^{-4}-1, 2^{-11}] for positive number and [-2^{-4}, -2^{-11}] for negative number.

In order to maintain the precision, for addition between values of different fraction length, we choose to move the value with smaller fraction length left to match the fraction length. Moreover, when doing the precision reduction, we use nearest rounding method for all the values.

Besides saving storage space, the benefits of fix point CNN including reducing bandwidth and decreasing computation resource requirement. For bandwidth, since kernel weights are mainly stored off chip, available IO ports on the chip set limit to the speed of kernel loading.

In our case, to fully utilize the on-chip computation resource, each time two kernels of size 3x3x64 are needed, which require 2x576x32 IO ports if single float precision is employed, but only 832 IO ports are available on chip. So cutting down the length of the kernel weights representation greatly shortens the kernel loading latency.

Moreover, computation unit for fix point arithmetic normally occupies less resources than float arithmetic, with detailed implementation resource estimation in Table **??**. It is worth noting that although it would require the same number of DSP48 slice to realize the Adder for 32bit, 16bit and 8bit, the limited availability of DSP48 slices on-chip prevents using DSP to implement all the computation units. When it comes to the implementation using logic, short length fix point shows its advantage, compared with 32bit precision, 8bit multiplier saves approximately 88.33% resources.

Table 2.4: Computation resource comparison

	resource	32bit FL	16bit FI	8bit FI
Multiplier	DSP48 slice	3	1	1
	LUT6s (Δ)	617	280 (-54.6%)	72 (-88.3%)
Adder	DSP48 slice	2	1	1
	LUT6s (Δ)	379	17 (-95.5%)	9 (-97.6%)

2.4 Uniform Accelerator for Convolutional and Fully-Connected Layer

Apart from compressing the network parameter to fixed-point number with smaller footage, we also optimize the design at algorithm and architecture level. In this section, we reformulate the convolution operation and design uniform accelerators to speed up convolution at different layers.

2.4.1 Reformulation of the Convolution Operation

Naturally, convolutions with different kernel sizes or strides cannot be accelerated by the same accelerators. Yet, as shown in Fig. 2.3, the convolution between a local region of feature map (x) and a kernel (w^i) is essentially inner-product (IP) when they are flattened to vectors xf and wf^i. Note that wf^i can be pre-calculated since the kernels remain unchanged for different inputs, but xf needs to be generated in real-time when input changes. It is acceptable to flatten feature maps at different layer by different accelerators because it only involves memory manipulations, which can be handled in FPGA at low cost. On the

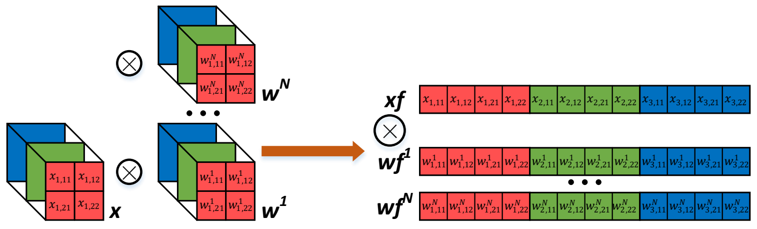


Figure 2.3: Convolution Flattening

other hand, the acceleration of IP is very expensive as it involves all the floating-point operations. At different layers, the lengths of flattened vectors xf and wf^i are typically different. However, the flattened vectors xf and wf^i are typically very long, e.g. for layer 2, the length is 2400 (5x5x96). Given this characteristic, we can break the long vectors into shorter uniform segments, and design uniform accelerators to calculate inner product for each segment.

2.4.2 Design of Convolution-Flattening and Inner-Product Accelerators

According to the reformulation of convolution operation, each convolutional layer has its own light-weight CF accelerators, while all layers share the same uniform IP accelerators, which take care of all the floating-point operations.

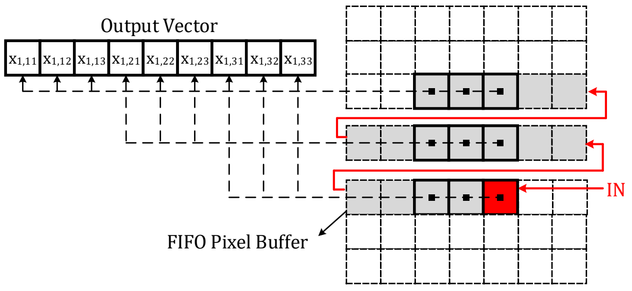

Output Vector
$x_{1,11}$ $x_{1,12}$ $x_{1,13}$ $x_{1,21}$ $x_{1,22}$ $x_{1,23}$ $x_{1,31}$ $x_{1,32}$ $x_{1,33}$
IN
FIFO Pixel Buffer

Figure 2.4: Convolution-Flattening (CF) Accelerator

2.4.2.1 Convolution-Flattening Accelerator

Even though the CF operation only counts for memory manipulation, it is non-trivial to design a CF accelerator. The key of designing an efficient CF accelerator is to reuse the data when the convolutional filter (kernel) slides on the feature map.

As shown in Fig. 2.4, the CF accelerator is designed mainly using a set of shift registers. At each clock cycle, the shift register takes in 1 pixel in the 7x7 feature map, and outputs a 1x9 vector on the left hand side (flattened from the data in the 3x3 window). To generate a vector that is long enough for the inner product, a convolutional layer may consist of multiple CF accelerators.

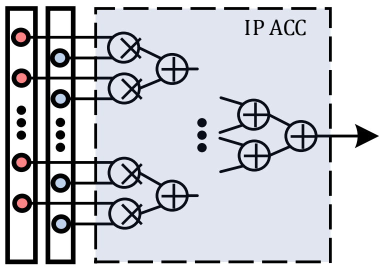


Figure 2.5: Inner-Product (IP) Accelerator

2.4.2.2 Inner-Product Accelerator

The design of IP accelerator is relatively straightforward. As shown in Fig. 2.5, it uses a set of multipliers to calculate the production between each element of two input vectors. The multiplier outputs are summed up to a single result using an adder tree. This IP accelerator is fully streamable, which can process two input vectors at each clock cycle.

The key of achieving a high computational capability is to keep the computing resource (DSPs) always busy. In this design, all convolutional layers shares the same IP accelerators. At each layer, the CF accelerators continuously stream data to the IP accelerator, resulting a high computational capability.

Moreover, we developed tools to automatically generate the CF and IP accelerators with given parameters such as kernel size, stride, IP vector length. Such tools facilitate the adoption of this architecture to other CNNs or FPGA devices.

2.5 Architecture and Performance Evaluation

2.5.1 Overall Architecture

The overall architecture of the proposed FPGA based CNN is illustrated in Fig. 2.6. Clearly, the architecture can be partitioned into 3 parts mainly consisting of logic, DSPs and on-chip memory blocks, respectively. The memory part includes feature map(FM) buffers, kernel buffers and a temp buffer for intermediate results. The logic and DSP parts including CF accelerators for feature map data flattening and IP accelerators for inner product calculating. We are able to implement two CF accelerators and two IP accelerators to maximize parallelism and throughput based on our FPGA resources.

The data flow can be described as following: one of the FM buffers provides feature map data to both CF accelerators, and after CF accelerators the flattened feature map data will be sent to IP accelerator for inner product calculating. On the other hand, one set of kernel buffers will provide two kernel vector to IP accelerators as well. The result after IP accelerators will be sent through two adders for addition between IP accelerators and with former intermediate results if needed. Partial intermediate result will be stored in Temp buffer, and final output feature map will be sent to the other FM buffer. Below we discuss each block one by one.

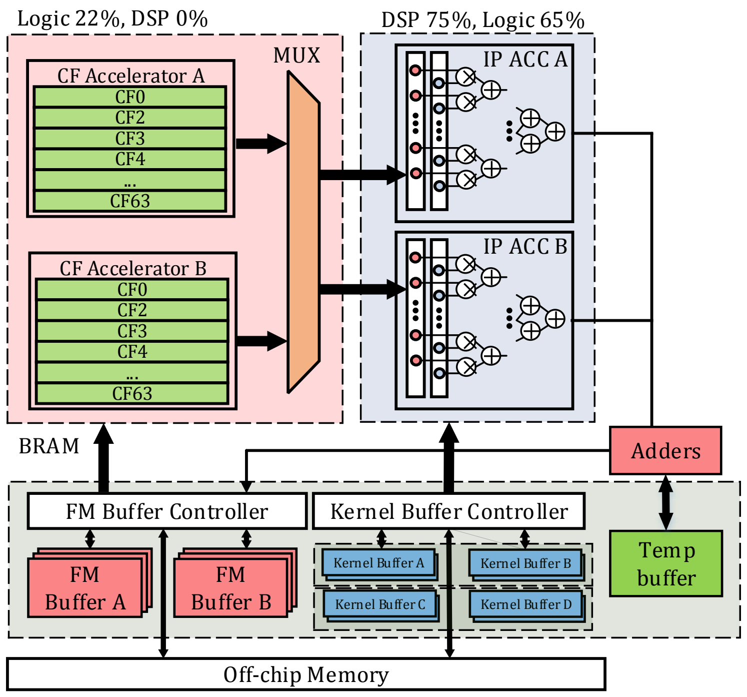


Figure 2.6: Overall architecture

In this architecture, we use two page-flip buffers to store the feature map. While one FM buffer stores the input of the CF accelerators, the other one accepts the output from IP accelerators. Those buffers are implemented as distributed BRAMs on FPGA, allowing multiple(3x3x64x2) 8-bit numbers to be loaded in one cycle. Moreover, once the original image is loaded from off-chip DRAM, the feature map data always stay in the on-chip memory, which saves off-chip memory bandwidth drastically.

We also use two sets of buffers to load kernel data from off-chip memory, each set has two buffers of length 3x3x64x8 bit. While using the kernel data in one buffer set, the controller loads the kernel data of the next computation to the other buffer set. We can achieve very high off-chip memory bandwidth because only block-wise reading operation is involved for kernel loading.

The logic part consists of CF accelerators that flattens the input feature maps. We have two uniform CF accelerators which are responsible for the data flatten for all the layers.

This design minimizes the off-chip memory access by keeping feature map data always on chip, and only load kernel data. It also shares uniform IP accelerators and CF flatten accelerators across layers, which keeps the DSP always busy and maximizes the performance.

2.5.2 Implementation and Accelerator Resource Allocation

The proposed design is synthesized and implemented with Vivado (v2015.4) while targeting to a Xilinx Ultrascale XCVU095 FPGA. The kernel data are extracted from Caffe [JSD+14] and stored in off-chip DRAM. We also use a fixed-point VGG-16 network implemented in Matlab to validate the accuracy of the FPGA-based implementation.

For a given FPGA device, it is straightforward to determine the number and size of the IP accelerators based on available DSPs and CLBs. The CF accelerators and IP accelerator are shared across all the layers, thus allocation of accelerators is a matter of optimizing the usage of both CF accelerator and IP accelerator under limited resources.

In our implementation, the kernel vectors in several layers are of 3x3x64, thus using any single IP accelerator longer than 576 is a waste of resource. We instantiate two IP accelerators, while each of them calculates the IP between two 576x1 vectors. The IP size is chosen as 576 since all of the Layers have kernel lengths which are the multiple of 576.

The input of these IP accelerates are connected to multiple CF accelerator banks via a multiplexer, as illustrated in Fig. 2.6. Hence, we can choose different CF accelerators to accommodate the data at each layer without losing generality. For the VGG-16 network, it happens that the kernel filters at different layers are 3x3. Although the feature map size is different at layers, we can still directly instantiate 2 CF accelerator banks, where each bank consists of 64 CF accelerators. Using these CF accelerators, 576 8-bit numbers can be generated and streamed to each of the IP accelerator at every clock cycle.

Table 2.5: Overall Resource Utilization

	LUT	LUTRAM	FF	BRAM	DSP
used	321603	11915	580160	1230	577
available	537600	76800	1075200	1728	768
Percentage	59.82%	15.51%	53.96%	71.21	75.13%

The overall resource consumption is listed in Table 2.5.

2.5.3 Performance

Table 2.6: Performance Comparison

	ISCA2010[CSJC10]	FPGA2015[ZLS+15a]	FPGA2016 [QWY+16]	FPGA2016[SCD+16]	Our Design
Precision	48bit fixed	32bit floating	8-16bit fixed	16 bit fixed	8bit fixed
FPGA	Virtex5 SX240T	Virtex7 VX485T	Stratix-V GSD8	Kintex-7	Virtex Ultrascale XCVU095
Frequency(MHZ)	120	100	120	150	125
Throughput (GOPS)	16	61.62	136.5	187.80	245.04
Available DSPs	1056	2800	1963	874	768
Utilized DSPs	N/A	2240	727(Alex,no vgg data)	780	576

The proposed design is implemented with clock configured at 125MHz. The throughput of the design is calculated below. At the current stage, all the layers expect the last 3 fully-connected layers in Table 2.1 are implemented on FPGA, which accounts for a total of 30.6 GOP (97.2% of the total computation). These three fully-connected layers can also be easily implemented in our current architecture by sharing the same IP accelerators, at almost no extra area cost.

The proposed design takes a total of 15,656,573 clock cycles to finish one round of process.

Therefore the throughput can be calculated as

$$Throughput = \frac{F_{clk}}{N_{cycle}} \times N_{OP} = 245.04\ GOPS, \tag{2.2}$$

which is substantially higher than the state-of-the-art [CSJC10, ZLS+15a] as presented in Table 2.6.

2.5.4 Automation and Scalability

The proposed architecture involves a semi-automation flow of synthesizing an FPGA design once CNN parameters are given. We have developed the tools that automatically generate the CF and IP accelerators, and pack the CF accelerators to a layer.

It is also convenient to applied the proposed architecture to larger network, or implemented on more advanced FPGA. Since we only store the kernel of the current and the next layer on chip, larger CNN can be easily fit in this architecture, let alone one important trend of FPGA is 3D stacking of large amount of memory to FPGA logic [LM13]. When more advanced FPGAs are available, we can just use more CF accelerators and larger IP accelerators, which are automatically generated by tools.

2.6 Conclusion

CNN achieves very good accuracy at the cost of high computational complexity, which motivates FPGA based acceleration. This chapter proposed an FPGA based CNN using uniform Convention-Flattening(CF) Accelerators and Inner-Product (IP) accelerators shared across multiple convolutional layers. The uniform CF and IP accelerator creates two benefits. First, the throughput is maximized because the uniform accelerators can be fully utilized while processing each layer. Second, it minimizes the off-chip memory access. Instead of accessing the kernel data from all the layers, we only need to process one layer at a time to make use of all the computing resources. This make it easier to buffer the data on chip since we only use the data from adjacent layer. We also developed a semi-automation flow that enable automatic synthesis of the accelerators and other building blocks. Experimental results shown that the proposed design achieves 245.04 GFLOPS at 125 MHZ clock throughput.

CHAPTER 3

Auto-compiler for Customized FPGA Accelerator

3.1 Introduction

Customized FPGA accelerator for deep learning neural network has been extensively studied in recent years. [ZLS$^+$15a] implemented Alexnet [KSH12] on a VC707 board with high throughput using HLS. [QWY$^+$16] proposed a handwritten RTL accelerator for VGGnet[SZ14a]. [SCD$^+$16] used HLS to design accelerators for both Alexnet and VGGnet, etc. These work demonstrated the good ability of FPGA to act as CNN accelerator platform.

However, for customized accelerators optimized for certain DCNNs, while the throughput is optimized for one single network in those designs, the application of the design is limited. The constraint comes from the various configurations for different networks. Despite the distinct layer numbers for each network, the convolution kernel sizes and pool sizes also vary, which makes it hard to directly utilize the accelerator designed for one network to run another.

On the other hand, a general end-to-end solution is needed for generation of accelerators for different CNNs targeted on different FPGA boards. In fact, benefited from the fact that CNNs tend to have the same typical convolution operation, which is exactly where main acceleration is needed, most of the CNNs can share similar parallelism logic and coarse acceleration architecture. As long as the design systems can satisfy different kernel sizes, strides and pooling parameters, a base architecture can be developed. Then sub-modules can be designed with regards to network configuration and platform resources under certain templates.

In this chapter, an automatic compiler is built based on an architecture template, the input of the compiler are network configuration and FPGA specification related information , and the output is the RTL code for the accelerator targeted on specified Xilinx FPGA board.

The template uses a uniform accelerator for different layers in the network, which virtually uses all the computing resources. And the data required for one layer can be easily buffered in on-chip memory with massive bandwidth. This optimizes the memory access. In order to realize the template that is suitable for different kernel sizes and strides, We reformulate the convolution to two separate steps, 1) convolution flattening (CF) to flatten the feature map and kernel from high-dimensional matrices to big 2-D matrices, 2) inner product (IP) between each rows and columns of the flattened feature map and kernel matrices. As the CF accelerators involve only light-weight memory manipulation, they consume less resource compared with IP accelerators. 8-bit fixed-point numbers is used instead of the single-precision floating-point number to represent the feature map and kernel weights in the network, therefore we can fit more accelerators on the same hardware with almost no accuracy loss.

Our main contributions in this work can be summarized as the followings:

- We proposed an architecture template for CNN accelerator. Based on our template design, we develop an analytical model to optimize the module parameters to achieve optimal performance.

- We build an automatic compiler that can generate RTL code for CNN accelerator based on the template for given network configuration and FPGA specification.

- Our design makes use of different levels of parallelism to deal with different computational burden. The flexibility ensure the fully utilization of the resources.

- Our automatically generated accelerators can achieve comparable performance with the single network specified accelerators.

We have experimented with two different networks on more than one platform, the automatically generated accelerators show good performance compared with the state of the art. To be specific, the Alexnet tested on both VC707and XCVU095 exhibit 2.1x and 1.35x times better performance compared with [MGAG16], and VGGnet on XCVU095 performs 1.2 times better than [QWY^{+}16].

The remaining of this chapter is organized as follows: Section 3.2 discusses the overall template architecture. Section 3.4 shows the parameter optimization model. And finally section 3.5 presents the experiment results, evaluation and comparison with other related work. Section 3.6 concludes this chapter.

3.2 Overall Template architecture

The complete architecture is shown in Fig.3.1. The architecture can be partitioned into 4 parts: (1) Convolution acceleration that includes CFA and IPA, which is introduced in previous chapter. (2) Memory where feature map buffer(FM buffer), kernel buffer, bias buffer and temp buffer(used to store intermediate feature map) is implemented. (3) Pooling and output control where the pooling operation is completed and output data is rearranged to fit into FM buffer for next layer's calculation. And (4) top level control where the data flow and connections is set. Below, we discuss the last three parts in detail.

3.2.1 Memory

FM buffers take most of the memory on board. They store the input feature and output feature map for each layer's computation, where the outputs from previous layer serve as the input for next layer. In order to realize the data streaming we use two FM buffers to form the

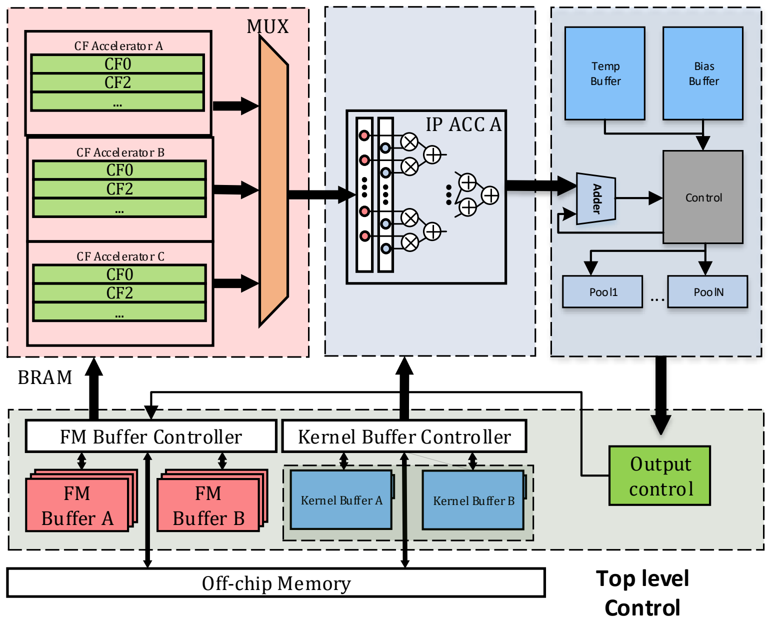


Figure 3.1: An illustration of the complete template architecture[SZ14a]

ping-pong structure, where one acts as the input buffer and the other acts as output buffer at the same time. The kernel buffer is designed the same way, while using the pre-loaded kernel data in one buffer set, the controller loads the kernel data of the next computation to the other buffer set. We can achieve very high off-chip memory bandwidth because only block-wise reading operation is involved for kernel loading.

3.2.2 Pooling and Output Control

The compiler can generate pooling modules with different size and stride according to the network configuration. Since we only store the data after pooling operation, pooling modules consume very small amount of resource. Then the output control module will assign single/multiple outputs into calculated locations of the FM buffer to satisfy the data stream demand of next layer.

3.2.3 top level control

The top level control connects sub-modules and takes care of the data quantization. Since we are using fix point kernel weights and feature map, the addition between two data with different fix-point location needs additional shifting before the operation, at the same time the output data needs to be cut into required data length before store into FM buffer.

All the submodules including the top control are parameterized and can be generated by the compiler given necessary parameters, which will be optimized based on network configuration and resource constraint. The details of the optimization will be illustrated in section 3.5.

3.3 Uniform Convolution Accelerator

Two main requirements need to be satisfied the convolution accelerator template for our automatic compiler:

- Capable to perform convolution for different kernel sizes and strides.
- Able to explore multiple level of parallelism with regard to different computation burden for different layers, as well as different networks. This renders the FPGA computational resources fully utilized over different networks.

As discussed before, We reformulate the convolution operation to fulfill these two purposes.

3.3.1 Reformulation of the Convolution Operation

Details can be found in 2.4.1.

3.3.1.1 Convolution-Flattening Accelerator

The key of designing an efficient CFA is to reuse the data when the convolutional filter (kernel) slides on the feature map, so it is designed mainly using a set of shift registers. Fig. 2.4 illustrates a CFA for convolution with kernel size of 3 and stride 1. At each clock cycle, the shift register takes in 1 pixel from the input feature map, and outputs a 1x9 vector on the left hand side (flattened from the data in the 3x3 window). To generate a vector that is long enough for the inner product, a convolutional layer may consist of multiple CF accelerators.

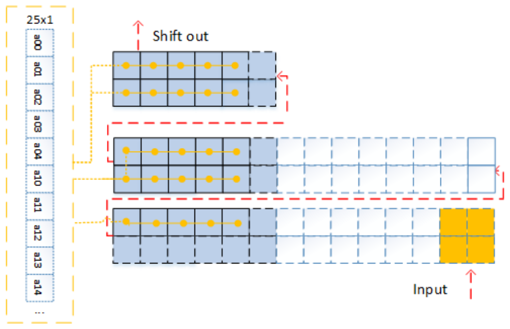


Figure 3.2: Convolution-Flattening (CF) Accelerator

This design can be expand to CFA with other kernel sizes and strides. For example, to deal with kernel size of 5 with stride of 2, the CFA works in the way showed by Fig. 3.2. For each clock cycle, 2x2 pixels from the input feature map is provided and the CFA buffer shifts in the 2x2 square step. At the left side a vector of length 25 will be the output. The input size and shifting distance is decided by the stride. Since the CFA stops only when the complete number of output is generated, in the case where input feature map size is not dividable by input size, extra dummy data is provided at the end of each line to keep the shift going without influencing the output results.

Based on this CFA design, the compiler can generate code for CFA with kernel size K, stride S and with padding or without padding.

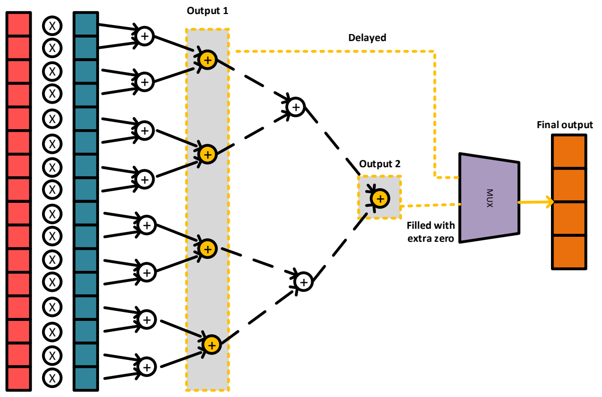


Figure 3.3: Inner-Product (IP) Accelerator

3.3.1.2 Inner-Product Accelerator

The design of IP accelerator is relatively straightforward. It uses a set of multipliers to calculate the production between each element of two input vectors. The multiplier outputs are summed up to a single result using an adder tree. This IP accelerator is fully pipelined, processing two input vectors at each clock cycle.

The IPA is design to achieve the smallest latency for all the layers. But the computation burden can vary a lot between different layers. Taking Alexnet as an example, the first layer has vector length of $3*11*11 = 363$, while the later layers has vector length of 2400, which we use as the IPA length. So during the calculation of first layer, only 15% of the computational units is used. In order to fully utilize all the computational units, for layers with shorter convolution-flattening vectors, multiple outputs is calculated at one round. This means the adder tree will have more than one stage that can act as output. As shown in Fig. 3.3, we can either calculate the one inner product result for length 16 vectors, or four inner product results for length 4 vectors. Then the MUX at the end of IPA chooses which result to use as the final output. Using this structure, in our Alexnet implementation we can calculate 6 outputs at the same time during first layer, which reduce the layer latency by 6 times.

Given total IPA length and sub-IPA lengths(for multiple outputs), the compiler will automatically generate the desired IPA module.

3.4 Module parameter Optimization

In this section, we discuss the parameter optimization. For a given CNN configuration, we have all the network pre-define parameters listed in Table 3.1. Suppose the network has M convolutional layers, then array $ker\_size[M]$ stores the kernel size for each convolution layer.

Then for a given FPGA platform, we can acquire the information about computational and memory resources as listed in Table 3.4.

Table 3.1: Pre-defined network configuration

representation	meaning(of each conv layer)
M	number of convolutional layers
$ker\_size[M]$	kernel size
$stride[M]$	stride length
$layer\_size[M]$	size of input feature map
$layer\_depth[M+1]$	depth of input feature map
$layer\_com[M]$	computation units needed for one complete output
$pool\_size[M]$	pooling size(0 represents no pooling)
$pool\_stride[M]$	pooling size(0 represents no pooling)

Table 3.2: Pre-defined platform resources

representation	meaning
$com\_resource$	computational units available (DSPs, LUT configured as fix-point multiplier)
$BRAM\_resource$	number of 18k/36k BRAM tiles on board
$IO\_resource$	number of IOs
$DRAM\_constraint$	the limit imposed by DRAM speed

The parameters need to be optimized are shown Table 3.3. Among them the key parameter is the IPA length. We consider 3 levels of parallelism while finding the optimal IPA length:

- Intra-kernel level: the convolution process has no data dependency between multiplications within a kernel, so the 9 multiplication in a $3 * 3$ kernel convolution can be flattened and calculated in one multiplier time unit.

- Inter-kernel level: different kernels within a kernel set convolve with different input feature maps, all the multiplication can be finished in parallel.

- Inter-kernel set level: different kernel sets corresponds to different output feature map, so it is also called inter-feature map parallelism. We can calculate the element in the same position for multiple output feature maps concurrently, with no data dependency involved.

The number of computation units each layer needed to generate one complete output calculated in Eq.(3.1) is one of the deciding facts for the IPA length. For large CNNs that we normally face in acceleration tasks, one output would take as many as 2400 computational units to finish, which will occupy most of the computational resources on board. For smaller networks and less computational-intense layer in large networks, inter-kernel set level parallelism will be considered to make full use of the resource.

$$layer\_com[i] = ker\_size[i] \times ker\_size[i] \times layer\_depth[i]^1 \tag{3.1}$$

Then the uniform IPA length can be calculated by solving the optimization problem in Eq.(3.2), where one of the constraint is on board computational resources. As for the other constraints, while using kernel_buffer alleviates the off-chip bandwidth pressure, in later layers where kernel reloading is needed more frequently, DRAM speed still needs be

[1]All the i show up in this equation and the following equations go from 0 to $M - 1$

taken into consideration.

$$\underset{IPA\_len,output\_num}{\text{Minimize}} \quad latency$$
$$= \sum ceil(\frac{layer\_com[i] * layer\_size[i+1] * layer\_size[i+1] * layer\_depth[i+1]}{layer\_IPA\_len[i] * output\_num[i]})$$
$$s.t. IPA\_len \leq \{com\_resource, DRAM\_speed\}$$
$$Resource\ Utilization < BRAM\_resource, LUT\_resource \tag{3.2}$$

Based on the optimal uniform IPA length, the length of input feature map vector that each CFA needs to provide for each layer's calculation can be calculated by Eq.(3.3) and (3.4), which decides the number of CFA we need.

$$layer\_IPA\_len[i] = floor(\frac{IPA\_len}{ker\_size[i]^2}) \times ker\_size[i]^2 \tag{3.3}$$

Table 3.3: optimal parameters

representation	meaning(of each conv layer)
$BRAM\_width$	Width of FM buffer
$BRAM\_depth$	depth of FM buffer
$kernel\_len$	length of the kernel buffer
IPA_len	uniform Inner product accelerator length
$layer\_IPA\_len[M]$	inner product acceleration length
$CFA\_len[M]$	convolutional flattening input length
$output\_num[M]$	number of output per clock cycle

$$CFA\_num[i] = \frac{layer\_IPA\_len[i]}{ker\_size[i]^2} \tag{3.4}$$

CFA does memory manipulation, so it consumes LUT resources mainly, which is not the critical resource constraint on FPGA. But its resources consumption can still be estimated, and it is proportional to the following equation:

$$CFA\_num[i] \times kernel\_size[i] \times layer\_size[i] \tag{3.5}$$

Consequently, the number of CFA decides the bandwidth required by each layer's calculation, thus adding design constraint on the FM BRAM. CFA would take the data stream from FM BRAM and reuse by buffering and rearrangement, which means FM BRAM bandwidth should as least be CFA number times the number of data needed for each CFA. Additionally, the FM BRAM depth each layer needs can be calculated based on the FM BRAM width and feature map size as in Eq.(3.7).

$$BRAM\_width[i] = CFA\_num[i] \times stride[i]^2 \tag{3.6}$$

$$BRAM\_depth[i] = \frac{layer\_size[i]^2 \times layer\_depth[i]}{CFA\_len[i]} \tag{3.7}$$

Then the BRAM architecture can be described by:

$$BRAM\_width[i]_{max} \times BRAM\_depth[i]_{max} \tag{3.8}$$

Since we are taking the maximum for both width and depth. We overestimate the BRAM resources, but we use Eq. (3.8) for two reasons:

- Layers that requires large $BRAM\_depth$ normally has big kernel size, so the $BRAM\_width$ for them are big as well. This BRAM estimation does not exceed the worst situation too much.
- If we decide to only satisfy the maximum of one condition, say BRAM width. When BRAM depth is not deep enough, we will have to rearrange the data to make use of the rest of the width, which greatly complex the input and output control logic.

Kernel load size is known to be the same as IPA_len. Kernel data in the off-chip memory will be pre-arranged so the visiting of DRAM is block-wise and each time a successive chunk of data can be brought in.

3.5 Performance Evaluation

3.5.1 Testing networks and platform

In our experiment we use two large CNNs built on Imagnet dataset [FF10], AlexNet [KSH12] and VGGNet [SZ14a]. The platforms we choose are Virtex-7 XC7VX485T and XCVU095. They represent two types of boards: one with large on-board BRAM, the other with large amount of on-board DSPs. The resource for two boards are listed in Table 3.4.

Table 3.4: resource of platforms

	LUT	LUTRAM	FF	BRAM	DSP
XC7VX485T	303600	130800	607200	1030	2800
XCVU095	537600	76800	1075200	1728	768

Table 3.5: resource utilization

		LUT	LUTRAM	FF	BRAM	DSP
Alexnet	XC7VX485T	53.79%	21.64%	32.29%	52.04%	85.89%
	XCVU095	20.92%	18.71%	15.32%	16.52%	80.60%
VGGnet	XCVU095	62.84%	14.96%	54.34%	76.07%	81.64%

3.5.2 Comparison with existing work

For Alexnet, we implement it on both Virtex-7 XC7VX485T and Ultrascale XCVU095. Frequency of 125MHZ is achieved and throughput is calculated based on the total latency and computation burden. And the throughput is computed by Eq. (3.9).

$$Throughput = \frac{F_{clk}}{N_{cycle}} \times N_{OP} = 177.44\ GOPS \tag{3.9}$$

which is substantially higher than the state-of-the-art [ZLS$^+$15a] [MGAG16] as presented in Table 3.6, where the performance comparison for VGGnet is also listed.

Table 3.6: Performance Comparison

	Alexnet				VGGnet		
	FPGA2015[ZLS+15a]	ASP-DAC2016 [MGAG16]	Our Design 1	Our Design 2	FPGA2016 [QWY+16]	FPGA2016[SCD+16]	Our Design
Precision	32bit floating	32bit floating	8bit fixed	8bit fixed	8-16bit fixed	16 bit fixed	8bit fixed
FPGA	VX485T	VX485T	VX485T	XCVU095	Stratix-V	Kintex-7	XCVU095
Frequency(MHZ)	120	100	100	100	150	100	100
Throughput (GOPS)	61.62	84.2	113.69	177.44	136.5	187.80	226
Available DSPs	2800	2800	2800	768	1963	874	768
Utilized DSPs	2240	NA	2405	619	727	780	627

Our performance for AlexNet on VX485T is 2.1x better than the state-of-art performance [MGAG16], and VGGNet on XCVU095 is 1.2x better than [QWY+16]. Achieving the comparable and even better performance with single network customized accelerator design indicates that FPGA acceleration for CNN can be generalized without losing the performance advantage.

And the higher throughput we are achieving compared to the state-of-art designs can be credited to our fully streaming architecture and the extra parallelism we are exploring in the first few layers that makes full use of the available resources.

The overall resource consumption is listed in Table 3.5. It can be seen that VGGnet takes large amount of BRAM resource compared with Alexnet, thus makes it hard to fit into the XC7VX485T board although the DSP resource is sufficient, which indicates that for CNN application, the BRAM resource can be as important as the computational resource.

3.5.3 Comparison of AlexNet on two different FPGAs

In our experiments, AlexNet accelerators were generated for two different platforms, VX485T and XCVU095. Different performances were achieved due to distinct resource constraints of the two boards. Parameters of the two accelerator are compared in Table 3.7. It can be seen that the parameters are not proportionally scaled when shifting from one FPGA to another. The key difference lies in the IPA_len, which is mainly decided by the computational units available one the FPGA, especially the DSP number. And the IPA_len is closely related to the throughput cause it decides how many rounds it would require for a single output to be finished.

Moreover, larger IPA_len means wider FM_BRAM is needed. It would consume more BRAM tiles to satisfy the bandwidth requirements.

Table 3.7: AlexNet on two different FPGA

	Optimal_IPA_length	Total_CFA_nums	BRAM_width	BRAM_depth
XC7VX485T	2400	365	266	3249
XCVU095	1152	227	128	3249

3.5.4 Comparison of AlexNet and VGGNet on same FPGAs

We also implemented AlexNet and VGGNet on XCVU095. Due to network size difference, VGGNet takes more than 15 times more BRAM than AlexNet. But its total CFA_num is smaller benefited from its rather uniform convolution kernel size.

Although the two network share the same IPA_length, VGGNet's throughput exceeds AlexNet greatly. It is because the high computational requirements for almost all layers of the VGGNet render the IPA fully used all the time, while in AlexNet there are still several layers only using partial of the IPA resources.

Table 3.8: AlexNet and VGGNet on XCVU095

	Optimal_IPA_length	Total_CFA_nums	BRAM_width	BRAM_depth
AlexNet	1152	227	128	3249
VggNet	1152	128	128	50176

3.6 Conclusions

CNN achieves very good accuracy at the cost of high computational complexity, which motivates FPGA based acceleration. While accelerator targeted on single network has achieved very good performance, an automatic compiler for general network accelerator has not been studied in-depth. This chapter developed an auto-compiler that can generate working HDL code for given CNN accelerator on given FPGA platform based on a architecture template. The template has uniform Convention-Flattening(CF) Accelerators and Inner-Product (IP) accelerators shared across multiple convolutional layers. The reformulation of the convolution operation makes it possible to conduct convolution with different sizes and strides using similar structure, and simplify the optimization problem into finding the best IPA length, which is easier to solve. Experimental results shown that, despite the fact the implementation is generated by compiler, the performance is still comparable to network specific accelerators. For example, our automatically generated Alexnet design on VX485T performs 2.1 times better than the state-of-the-art implementation [MGAG16].

CHAPTER 4

OPU: An FPGA Overlay Processor for Convolutional Neural Networks

4.1 Introduction

FPGA acceleration for DCNNs has drawn much attention in recent years [FPHL09, ORK$^+$15, CMB$^+$10, CSJC10, ZLS$^+$15a, QWY$^+$16, SCD$^+$16, SPM$^+$16, MCVS17a, ZFZ$^+$16, WYZ$^+$17]. Well-designed FPGA accelerator for CNN can leverage full capacity of parallelism to achieve low latency and high throughput. Moreover, FPGA's reconfigurability enables fast adoption to new CNN architectures. Additionally, FPGA has higher energy efficiency compared to CPU or GPU. As mentioned in chapter 3, implementing a high-performance FPGA accelerator can be time-consuming as it normally involves parallel architecture exploration, memory bandwidth optimization, area and timing tuning, as well as software-hardware interface development. This leads to the development of automatic compilers for FPGA CNN accelerators[SPM$^+$16, MCVS17a, ZFZ$^+$16, WYZ$^+$17], where hardware description of target accelerators can be generated automatically based on parametric templates, and design space exploration is simplified to parameter optimization with regard to network structure and hardware resource constraints.

However, some disadvantages still exist. First, while RTL code can be generated as the final output of auto-compilers, it still takes logic synthesis, placement and routing to obtain the final bitstream. Second, the resulting design may fail timing requirements. Instead of fixing timing failing paths as in regular FPGA design process, only module parameters

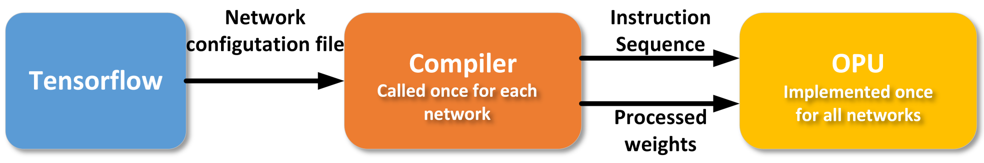

Tensorflow
Network configutation file
Compiler
Called once for each network
Instruction Sequence
Processed weights
OPU
Implemented once for all networks

Figure 4.1: OPU working flow.

or relax timing constraints at the expense of performance degradation can be adjusted in auto-compiler process. Moreover, since complex deep learning tasks usually involve cascaded network flow, this flow may be inefficient, or even impossible to constantly re-burn FPGA for different networks during runtime.

In this work, we propose an FPGA overlay domain-specific processor unit (OPU) applicable to a wide range of CNN networks. OPU accepts network description using CNN frameworks such as Tensorflow [ABC+16]. Each time a new network configuration is given, instead of re-generating a new accelerator, we compile the network into instructions to be executed by OPU. OPU has fine-grained pipe-line, and leverages channel based parallelism. This ensures an average of 91% runtime utilization of computing resources as shown by experiments on nine different networks, including YOLO[RF17], VGG[SZ14b], GoogleNet[SVI+16] and ResNet[HZRS16a]. Moreover, superior power efficiency compared with Titan GPU (both batch = 1 and batch = 64) is observed for all networks in our experiment. In addition, for a test case with cascaded CNN networks, OPU is 2.9× faster compared with edge computing GPU Jetson Tx2 with similar amount of computing resources. Specifically, the proposed overlay processor OPU has following features:

- **CPU/GPU like user friendliness**. As shown in Fig. 5.1, CNN network is compiled into instructions. This is done once for each network. Then instructions are executed by OPU which is implemented on FPGA and is fixed for all networks. The CNN algorithm developer does not need to deal with FPGA implementation.
- **Optimized instruction set**. Our instructions have optimized granularity, which

is smaller than that in [AHB+18] to ensure high flexibility and computational unit efficiency, while a lot larger than those for CPU/GPU to reduce the complexity of compiler. We also define instructions for data block handling to reduce overall memory latency.

- **FPGA based high performance micro-architecture**. These architectures are optimized for computation and data communication and reorganization, and are controlled by parameter registers set directly by instructions.

- **Compiler with comprehensive optimization**. Independent of micro-architecture, operation fusion is performed to merge or concatenate closely related operations to reduce computation time and data communication latency. Data quantization is also conducted to save memory and computation resources. Related to micro-architecture, compiler explores multiple degrees of parallelism to maximize throughput by slicing and carefully mapping the target CNN to overlay architectures.

4.2 Related Work

Deep CNN acceleration by FPGA has been extensively studied, started with developing customized hardware accelerators for specific networks. Farabet et al.[FPHL09] used FPGA as a vector based arithmetic unit, and implemented CNN mainly on a 32-bit soft processor. Authors of [ORK+15], [CMB+10] and [CSJC10] designed specific accelerators for each layer of CNN. [ZLS+15a] implemented Alexnet [KSH12] on a VC707 board using HLS and [QWY+16] hand-coded an RTL accelerator for VGGnet[SZ14a]. Suda et al.[SCD+16] used HLS to design accelerators for both Alexnet and VGGnet. These work demonstrated FPGA's capability as a high performance CNN accelerator platform, but manually designing accelerator for each CNN was inefficient.

More recent work developed automatic compiler to implement CNN accelerators to

FPGA. [SPM$^+$16, MCVS17a] mapped a CNN algorithms to a network of hand-optimized design templates, and gained performance comparable with hand-crafted accelerators. [ZFZ$^+$16] developed a HLS (high level synthesis)-based compiler with bandwidth optimization by memory access reorganization. [WYZ$^+$17] applied an systolic array architecture to achieve higher clock frequency. However, they all generate specific individual accelerators for each CNNs. This has high re-engineering effort when the target CNN changes.

Most recently, [AHB$^+$18] used FPGA overlay to implement CNN accelerators. While instructions are used to decrease the control logic overhead, they still reconfigure the overlay architecture to maximize performance for a specific CNN. Moreover, the granularity of their instructions is larger than that for our OPU. In [AHB$^+$18], one block of about 10 instructions are used for a whole sub-graph defined as a list of chained functions, which is normally a single convolution layer and an optional pooling layer. In contrast, our OPU has the instruction set with smaller granularity (to be discussed in section 5.3) than that in [AHB$^+$18]. Each typical operation in CNN inference is mapped to a specific type of instruction, resulting in high runtime efficiency. Moreover, different CNNs can be compiled and then executed without FPGA reconfiguration.

4.3 Instruction Set Architecture

Instruction set architecture (ISA) is the key to a processor. Our OPU is specific for CNN inference. We identify all the operations during CNN inference and group them into different categories. Each category maps to one type of instruction with adjustable parameters for flexibility. Our instructions are 32-bit long with complicated functions and variant runtimes (up to hundreds of cycles). CNN inference can be executed by OPU without a general processor such as CPU.

We define two types of instructions: Conditional instruction (*C-type*) and Unconditional instruction (*U-type*). *C-type* instruction specifies target operations and sets operation trig-

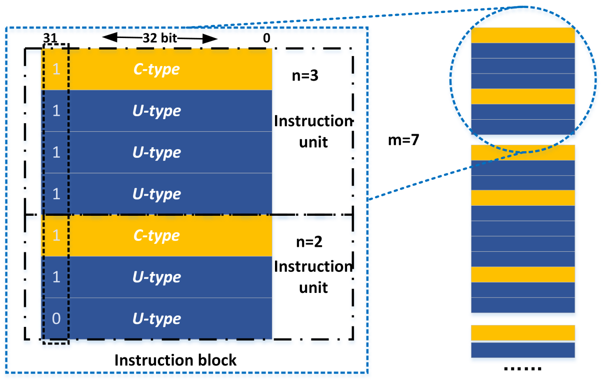


Figure 4.2: The configuration of instruction unit and block.

ger conditions, while *U-type* instruction delivers corresponding operation parameters for its paired *C-type*. As shown in Fig. 4.2, one instruction unit contains one *C-type* instruction with $0 - n$ *U-type* instructions. This instruction block consisting of a number of basic units is fetched together and then is distributed to PE modules. The least significant bit of instruction indicates the end of current instruction block when its value is 0.

4.3.1 Conditional instruction

Conditional- or *C-type* instructions contains operation (OP) code and trigger condition. OP code identifies the target operation while trigger condition defines when operation is ready to execute. Six types of C-instructions are defined below, each operates on a slice of data block:

- *Memory Read* transforms data from external memory to on-board memory. It operates in two modes to accommodate for different data read patterns. Received data will be reorganized and distributed to three destination buffers corresponding to the feature map, kernel weighs and instructions, respectively.
- *Memory Write* sends the block of computational results back to external memory.
- *Data Fetch* performs data read from on-board feature map and kernel buffer, then feeds to computation engine. Its working pattern can be flexibly adjusted by placing

constraint parameters on row and column address counters, read strides, and data reorganize modes.

- *Compute* controls all processing units (PEs). One PE computes the inner product of two 1D vectors of length N (set to 16) in current micro-architecture implementation based on widely used CNNs families' architectures. This sufficiently guarantees the design space exploration for different networks. Results of PEs can be summed up in different modes based on parameter setting.

- *Post Process* includes pooling, activation, data quantization, intermediate result addition and residual operations. Selected combination of before-mentioned operations are executed when *post process* is triggered.

- *Instruction Read* reads a new instruction block from instruction buffer and directs it to target operation modules.

Each instruction introduced above leads one instruction unit. Instead of linking all the operation modules together in a fixed pipeline, we organize our operations in a dynamic pipeline fashion and each module is controlled by an individual instruction unit for more flexibility. For example, after one *memory read* is called for feature map loading, multiple *data fetch* and *compute* may be called to reuse loaded feature map data. Then at certain point during computation, *memory read* can be called again to replace kernel weights data (in the case where kernel size is large). When residual layer is encountered, *memory read* is called one extra time to load feature map data from short cut[HZRS16a] for *post process.* Individual control of each module by instruction greatly simplifies overall hardware control frame, and enhances the architecture applicability to different network configurations.

To realize efficient instruction control, trigger condition is employed, so instruction is not executed immediately upon read. In CNN inference flow, each operation depends on previous operations based on different operating patterns, which have limited variations. We design a trigger condition list for individual operation, then modify trigger condition index

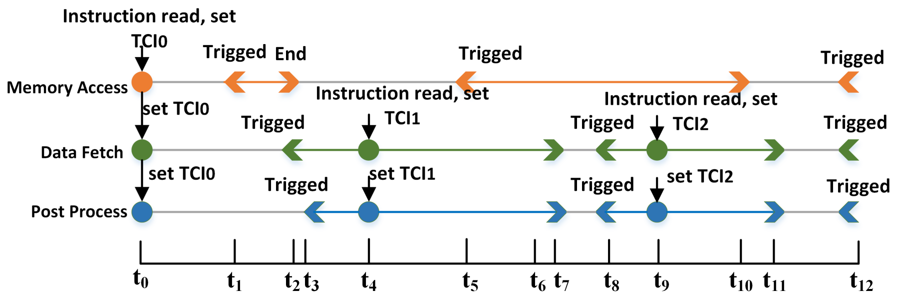


Figure 4.3: Instruction execution order controlled by TCI

(TCI) by instruction at runtime to set module initiation dependency. Moreover, using a dependency based execution strategy relaxes the order enforcement on instruction sequence. Cause memory related operations has the uncertainty in execution time due to extra refresh latency. The resulting system has a simple instruction update scheme.

Several instructions executed at different time points can be put into the same instruction block and read at the same time. As shown in Fig. 5.3, after the first instruction read, initial TCI_0 is set at t_0 for all three operations. At t_1, *memory access* is triggered then executes for $t_1 - t_2$. *Data fetch* is triggered upon finishing memory operation and *post process* is triggered at t_3. Next instruction read that updates TCI_1 for *Data fetch* and *post process* can be performed at any time point between t_3 and t_7. Moreover, we store current TCI to avoid setting the same condition repeatedly when modules operate in one pattern consecutively (at time t_0 and t_5, *memory access* is triggered by the same TCI). This shortens the instruction sequence over 10×.

4.3.2 Unconditional instruction

Unconditional- or *U-type* instruction provides operation related parameters and is generated on an updating demand-based scheme, as parameters are stored to reduce the total length of instruction sequence.

Several *U-type* instructions combined can update the complete parameters list for one *C-type* operation. But in general, when operation pattern switches, only a subset of parameters are changed accordingly. Flexible combinations of *U-type* instructions can update necessary parameters with minimum instruction cost. We group parameters that are closely related to each other and have similar updating rates in one *U-type* instruction. This reduces the possibility of loading futile instruction sections, thus reducing both storage space and communication power.

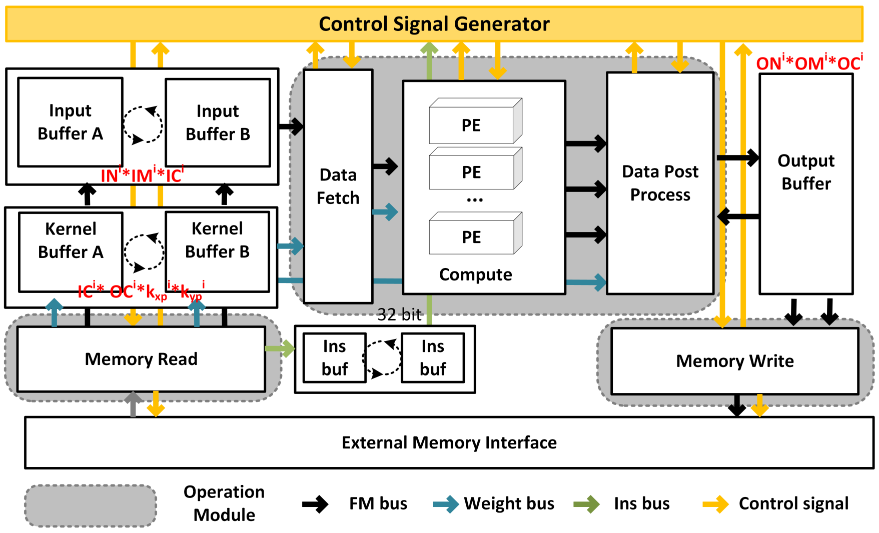


Figure 4.4: Overview of Micro-architecture.

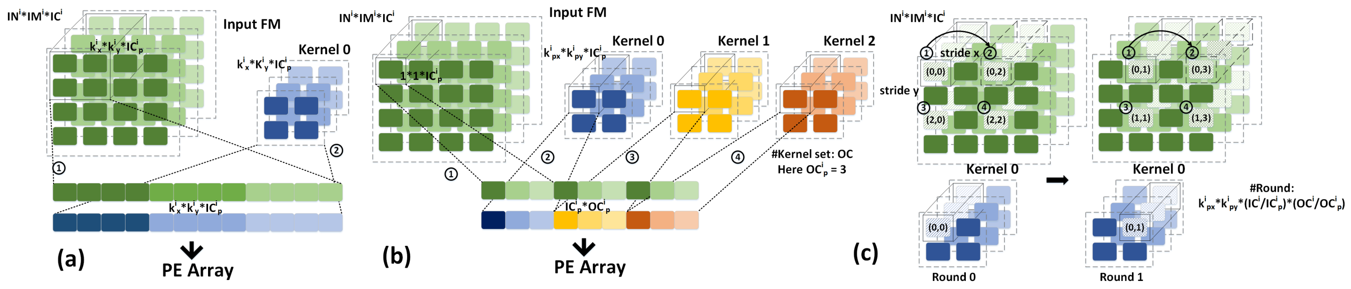


Figure 4.5: (a). Conventional intra-kernel based parallelism. (b). OPU input/output channel based parallelism. (c). Feature map data fetch pattern of OPU.

4.4 Micro-Architecture

Another challenge in OPU design is overlay micro-architecture design. The overlay micro-architecture needs to incur as less control overhead as possible while maintaining easily runtime adjustable and functionality. We design our modules to be parameter customizable, and switch modes at runtime based on parameter registers that directly accepts parameters provided by instructions. The computation engine explores multiple level of parallelisms that generalize well among different kernel sizes. Moreover, CNN operations categorized into the same group are reorganized and combined (see section 5.3.2) so they can be accomplished by the same module to reduce overhead.

As shown in Fig. 5.4, the overlay micro-architecture can be decomposed into six main modules following the instruction architecture definition. Each module can be controlled by instruction to accomplish functionalities defined in section 5.3.2. Besides, four storage buffers (i.e., input feature map buffer, kernel buffer, instruction buffer and output buffer) are placed to cache local data for fast access. With most of the control flow embedded in instruction, overlay only handles the computation of one sub-feature map block. If layer size is larger than the maximum block size allocated, the layer will be sliced into sub-blocks by compiler to fit into the hardware. The optimization of slicing scheme is discussed in section 5.5.

4.4.1 Computation unit

How to utilize the same set of PE structures to accommodate for different layers is dominant in the design of CNN acceleration architecture. Conventional designs tend to explore parallelism within single 2D kernel, which is straightforward but comes with two disadvantages, i.e., complex feature map data management and poor generalization among various kernel sizes. As shown in Fig. 4.5(a), expanding a $k_x * k_y$ kernel sized window of feature map requires multiple data read from row and column directions within single clock cycle (step ①). This poses challenge on limited *Block Ram* bandwidth and generally requires extra complicated data reuse management (like line buffer) to accomplish. Furthermore, data management logic designed for one kernel size cannot be efficiently applied to another one. Similarly, PE architecture optimized for certain kernel size $k_x * k_y$ may not fit other sizes very well. That's why many conventional FPGA designs optimize their architecture on popular $3 * 3$ kernel and perform the best only on networks with pure $3 * 3$ layers.

To address this issue, we explore higher level of parallelism and compute the 2D kernel sequentially. Fig. 4.5(b) explains how it works: At each clock cycle, a slice of input channel of depth IC_p^i with width and height as $1*1$ is read along with corresponding kernel elements. This fits natural data storage pattern and requires much smaller bandwidth. Parallelism is then implemented within input channel slice IC_p^i and output channel slice OC_p^i. Fig. 4.5(c) further shows the computation process. For round 0 cycle 0, input feature map channel slice from position $(0, 0)$ is read. Then we jump stride x ($x = 2$ is used as example here) and read position $(0, 2)$ in next cycle. Read operation continues until all pixels corresponding to kernel position $(0, 0)$ is fetched out and computed. Then we enter round 1 and read starting from position $(0, 1)$ to get all pixels corresponding to kernel position $(0, 1)$. To finish computing this data block of size $IN^i * IM^i * IC^i$ with current slice of kernel with size $k_{px}^i * k_{py}^i * IC_p^i * OC_p^i$ in *kernel buffer*, $k_{px}^i * k_{py}^i * (IC^i/IC_p^i) * (OC^i/OC_p^i)$ rounds are needed.

One implementation of the computation unit is shown in Fig. 4.6. One single Multiplier Unit (MU) computes two 8×8 multiplication with one of the inputs kept the same. This constraint comes from the DSP decomposition implementation. Each PE consists of 16 MU followed by an adder tree structure, thus each PE equals to 32 MACs. For one of our implementations of OPU, we implement 32 PEs within the computation unit, and an adder tree with switch is implemented to sum up results from different group sizes of PEs. The outputs number choices include $[64, 32, 16, 8, 4, 2]$. This allows the computation unit to flexibly fit into the needs of different combinations of input/output channels. For example, current implementation supports 1024 multiplications, which is able to handle 6 $[in_{channel}, out_{channel}]$

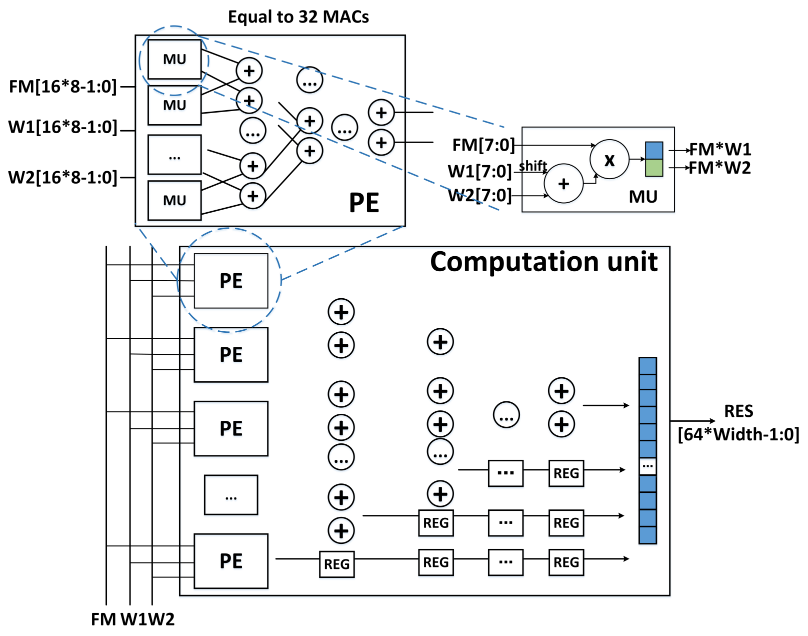


Figure 4.6: Computation Unit

pairs: [512, 2],[256, 4],[128, 8],[64, 16], [32, 32] and [16, 64].

Our computation pattern guarantees the uniform data fetching pattern for any kernel size or stride. This greatly simplifies the data management stage before *compute* operation, and enables higher design frequency with less resource consumption. Moreover, we leverage both the input and output channel level parallelisms. This provides higher flexibility for resource utilization and promises reasonable generalization performance.

4.4.2 Data Fetch and Post-Process

The data fetch module reads feature map and kernel data from on-chip buffer, rearranges the data and then sends to the computation unit. As shown in Fig. 4.7, for input feature map read, *FM ADDR GEN* takes control parameters from instruction and produce feature map buffer read address at each clock cycle. The parameters include [$Xmin$, $Xmax$, $Ymin$, $Ymax$, $Xstride$, $Ystride$, $Xsize$, $Ysize$]. The feature map data read from buffer will be selected and copied by the *FM REARR* to fit the target computation pair of computation

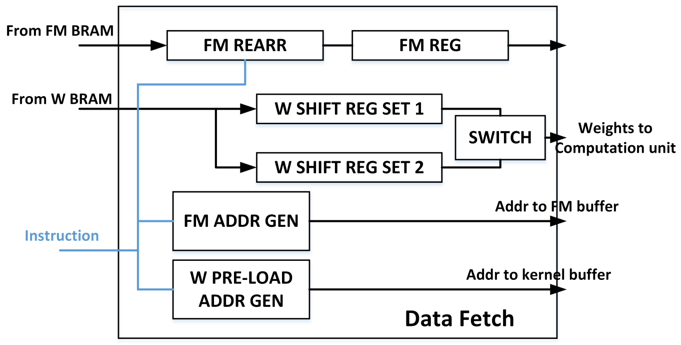


Figure 4.7: Data Fetch module.

unit. For kernel weights, the bandwidth requirement can be as high as 8192 bit/cycle, since all the parallelisms are explored on the kernel side (input channel / output channel). We choose a computation pattern that shares the same set of kernel weights during computation. Accordingly, kernel weights are only pre-loaded once from buffer for each round. We use *W PRE-LOAD ADDR GEN* to generate weights address for buffer, and each weight takes 32 cycles to load. This loading time is overlapped with previous round of data fetch process. A pair of local shift registers using ping-pong structure *[W SHIFT REG SET 1, W SHIFT REG SET 2]* is used to cache the weights.

The data post-process module performs data quantization, partial sum addition, pooling, activation and residual addition. Since pooling, activation and residual are only conducted once for one memory write, they are concatenated with the data memory write module to reduce extra on-chip data movement. As shown in Fig. 4.8, a data concatenation block is placed right after the input port to collect computation output (the output data number can be $[2, 4, 8, 16, 32, 64]$ based on different computation patterns) and formulate a 64-word long array for later process. A set of adders is used for bias addition and partial sum addition. Then the data quantization is achieved by data shift, cut and round modules that work based on parameters provided by instructions. When the complete output is ready, another part of the post process will be called and conduct pooling, activation and residual addition. The detailed corresponding module structure is shown in Fig. 4.8.

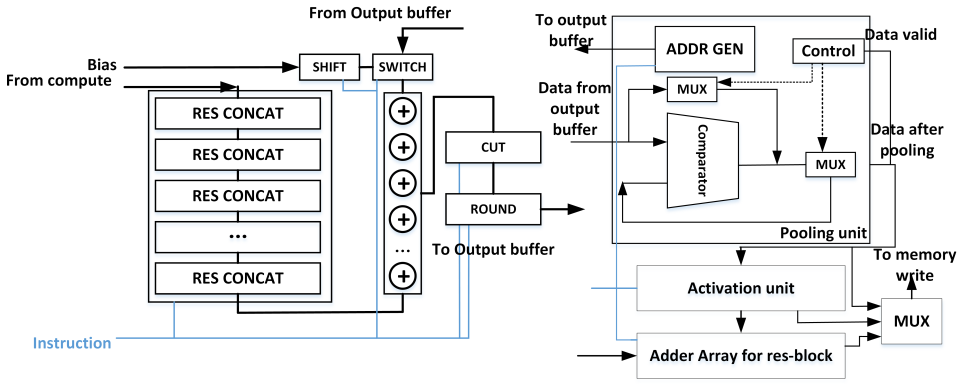


Figure 4.8: Data Post Process module.

4.4.3 Memory management

One crucial issue for CNN acceleration on FPGA is off-chip communication latency. Roof-line model [ZLS$^+$15b] reveals the relationship between bandwidth utilization and computational roof performance. Bandwidth can easily be the bottleneck of performance. Therefore, we utilize a ping-pong structure based caching memory management system to hide off-chip communication latency. While one buffer's data is being fetched, the other buffer can get refilled and updated, which maintains the maximum bandwidth utilization.

Another key point in memory management is data storage format in both local on-board buffer and external memory. For on-board storage format, data from the same channel slice is stored under the same address so that they can be fetched in one clock cycle. The limit of channel slice depth is set to be the width of on-chip buffer to guarantee such memory arrangement. Benefited from computation pattern, enough data bandwidth is provided and no extra memory latency is caused.

Moreover, data storage format in external memory influences data transmission efficiency and complexity of memory access unit. We thereby employ a channel sliced scheme for feature map storage. As shown in Fig. 4.9, data from one channel slice with shape $1 * 1 * IC^i$ is stored adjacently. In this way, data can be streamed on-board in burst mode to fully utilize bandwidth. During the computation process, data chunk loading order is ① $->$ ② $->$

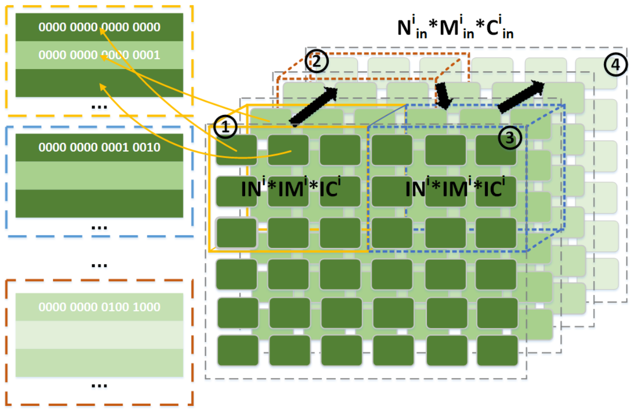


Figure 4.9: Channel based memory storage Management.

③ − > ④. Each chunk of data represents one sub-feature map block.

4.4.4 Irregular operation handling

ResNet [HZRS16a] and GooglNet [SVI$^+$16] exhibit excellent performance with reduced computational requirements compared with VGGNet[SZ14b] and AlexNet[KSH12]. However, they also introduce new non-convolution operations.

Inception module. Channel-wise input concatenation is required for inception module. Thanks to our memory storage pattern, we manipulate output memory addresses of preceding layers to place their outputs adjacently. Therefore, input concatenation can be achieved without any extra computational operation or latency cost. The arranged memory can be loaded in burst mode for efficient off-chip memory transmission.

Residual module. The operation of residual module is matrix element-wise addition, which is not computational intensive but may cause extra stage of off-chip data communication. We embed the addition operation into post process pipeline and cover the additional block loading stage with computation time, so residual operation is executed with negligible latency increment. Fig. 4.10 illustrates the implementation of all types of residual modules. As shown in Fig. 4.10(a), three kinds of residual paths are labeled out. Moreover, an activation module after element-wise addition is optional. We group element-wise addi-

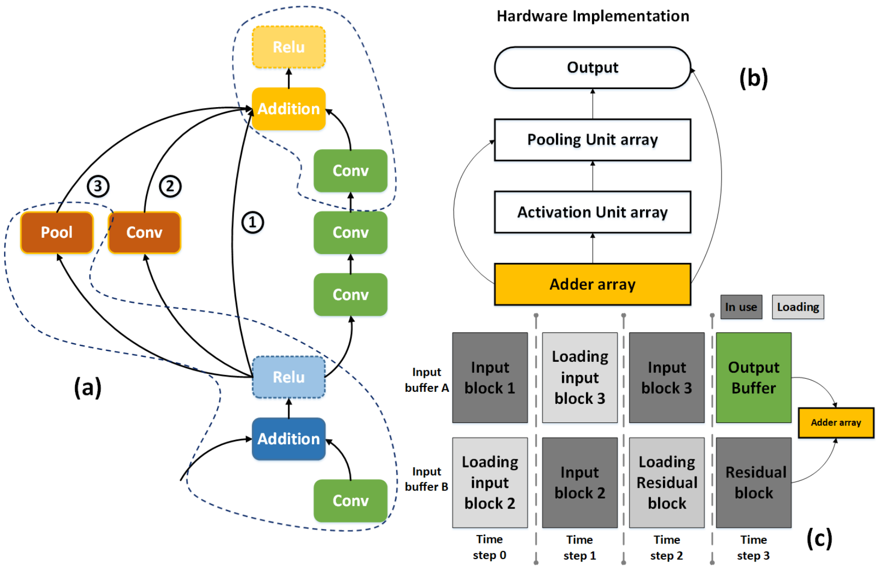


Figure 4.10: (a) Residual module type description; (b) Embedding addition operation into post process data streaming pipeline; (c) Loading time management for residual data.

tion, pooling and activation together and embed to previous convolution layer's operation pipeline. Fig. 4.10(b) shows that an adder array is inserted at the first stage of post-process hardware implementation. All function modules are implemented with bypass logic. With the combination of different bypass and functions, all types of residual layer can be computed. Fig. 4.10(c) shows the ping-pong input buffer at different time steps. At time step 2, current block's computation is at the final stage and uses data from input buffer A. Meanwhile, residual source is being loaded to input buffer B. At time step 3, both matrices for element-wise addition are ready and post process begins with no extra latency incurred.

4.5 Compiler

We develop a compiler to perform operation fusion, network slicing, and throughput optimization on input CNN configuration. There are two stages during the operation of compiler: *Translation* and *Optimization*, as shown in Fig. 4.11. *Translation* extracts necessary information from model definition files and reorganizes them into a uniform intermediate representation (IR) we defined. During this process, operation fusion (introduced in subsection 4.5.1) is conducted to combine closely related operations. Another aspect of *translation* stage is data quantization and arrangement. Fast Data quantization is performed for Kernel Weights/Bias/Feature Maps with negligible accuracy loss. Generated dynamic fixed-point representation system is then merged into IR. Moreover, processed weights get re-arranged based on network slicing and optimization results from the *optimization* stage. *Optimization* stage parses *Translation* generated IR and explores the solution space of network slicing to maximize throughput. For this stage, an optimizing scheme for slicing is developed, which will be introduced in subsection 5.5.2. In the end, the optimization solution is mapped to an instruction sequence.

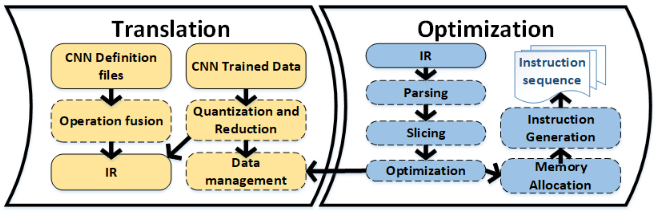


Figure 4.11: Two-step flow of compiler.

4.5.1 Operation fusion

Conventional CNNs contain various types of layer that are connected from top to bottom to form a complete flow. In order to avoid the off-chip memory communication between layers as to reduce computation requirements, operation fusion is employed to merge or concatenate related layer operations. Here we define two types of fusions, *p-fusion* and *r-fusion*. *p-fusion* indicates operation fusion that only contributes to off-chip memory access reduction, and *r-fusion* not only avoids communication latency but reduces the total number of operations and inference time. Among them, *p-fusion* and *r-fusion-I* are hardware micro-architecture independent, while *r-fusion-II* is related to actual implementation scale.

p-fusion. We perform *p-fusion* to concatenate different layers into the same data stream pipeline, which prevents saving intermediate results back to off-chip memory. Convolution and Fully connected layers are major layers. Pooling, Padding, Activation, Residual and Output concatenation layers are treated as affiliated layers, as shown in Fig. 4.12. Originally off-chip memory access is required between each pair of adjacent layers, such as $\{Padding, Convolution\}$, $\{Convolution, ReLU\}$ and $\{ReLU, Pooling\}$. After employing *p-fusion* $1-3$, memory access is reduced to only two times. The fused layers are called a layer group.

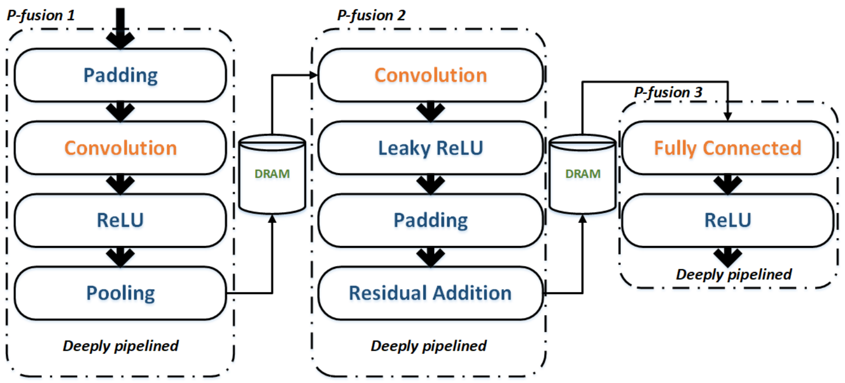


Figure 4.12: *p-fusion* example on a sequence of layers.

r-fusion. Fusion that decreases the amount of hardware arithmetic computations by merging adjacent operations is also conducted. There are mainly two kinds of *r-fusions.*

r-fusion-I is Batch normalization [IS15] elimination. A batch normalization operation can be represented by:

$$y = \gamma \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} + \beta, \tag{4.1}$$

where $\gamma, \mu, \sigma, \beta$ and ϵ are fixed values during inference time. x and y represent input and output matrix of one channel. Eq. (4.1) is essentially a linear function of input matrix, which can be merged to preceding convolutional layer. This merging avoids the separate computation of Batch normalization, thus reducing the inference time. The merged convolutional layer works with modified weights and bias that incorporate Batch normalization coefficients.

r-fusion-II is input sharing. In many cases, layers that share identical inputs can be computed by the same round to fully utilize computational resources and reduce memory communication time. As shown in Fig. 4.13(a),

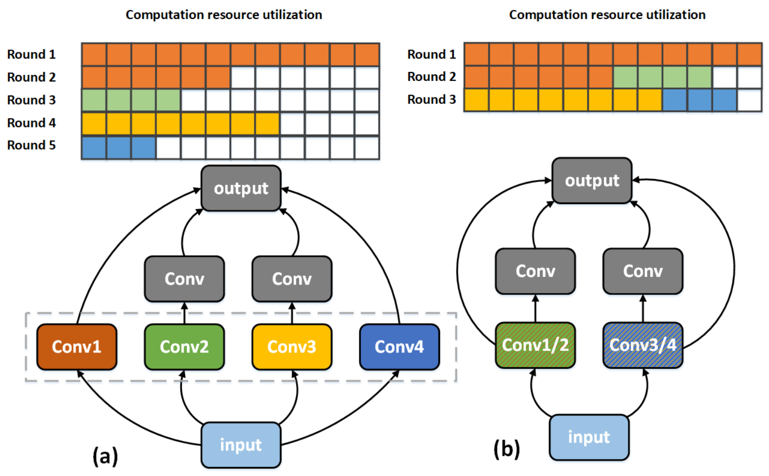


Figure 4.13: (a) Original Inception. (b) Merged Inception.

Conv1-Conv4 share the same input layer, but the numbers of their output channels are either small or cannot be evenly divided by available computing resources. Therefore, it takes 5 rounds to compute the four layers sequentially with relatively low runtime efficiency of PEs. To make use of idle resources, we add an optimization step into compiler that identifies input sharing layers and reassembles them based on their individual resource consumption, as shown in Fig. 4.13(b). Fig. 4.14 shows the improvement of Computation resource runtime efficiency after applying input sharing *r-fusion* to different inception modules. Different groups indicate different scales of computing resource, which increase from left to right. More resources available means higher possibility of resource idling, thus providing more room for input sharing optimization.

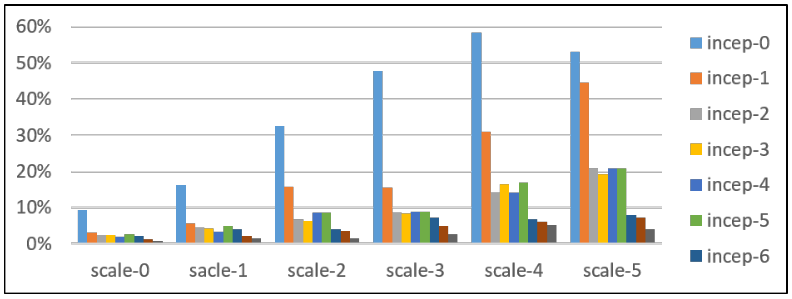


Figure 4.14: Throughput improvements by applying *r-fusion-II* on inception modules.

4.5.2 Data Quantization

It has been proven that CNNs are robust against precision reduction[QWY+16][GSQ+18]. To reduce memory footprint and save computational resources, we use limited precision fixed-point values during computation, data type can be set to fixed-4/8/16 bit based on platform constraint and network accuracy requirements. Taking general network precision redundancy and hardware architecture complexity into consideration, 8 bit is chosen as our data quantization standard for both feature map and kernel weights. We employ a fast yet effective stationary quantization method. Dynamic quantization scheme (similar as [QWY+16][MCVS17b]) has been employed for better accuracy. Each layer's kernel weights and feature maps have their own range for higher precision. The process of finding the best range for each trunk of data is described as follows:

$$\underset{floc}{\operatorname{argmin}} \sum (float - fix(floc))^2, \tag{4.2}$$

where *float* is the original single precision representation of kernel weights or feature maps, and *fix(floc)* is the value after *float* is cut into fixed-point based on certain fraction length *floc*. Table 4.1 shows the quantization accuracy of 9 experiment networks, where the accuracy loss is within 1% on average.

Table 4.1: 8-bit quatization evaluation for different networks.

	Classification[a]							Detection[b]	
	VGG16	VGG19	Inception V1	Inception V2	Inception V3	resenet V1	resnet V2	YOLO V2[c]	Tiny YOLO[d]
Float 32 bit	89.8%	85.2%	87.29%	90.30%	93.15%	92.9%	93.7%	85.93%	90.8%
Fixed 8 bit	89.4%	84.3%	85.49%	89.87%	91.41%	92.3%	93.2%	86.19%	89.5%

a: Reported accuracy are top-5 accuracy evaluated on Imagenet ILSVRC2012 validation set

b: Reported value are mAP. c: Evaluated on a private subway x-ray dataset. d: Evaluated on a private traffic view dataset.

4.5.3 Intermediate representation (IR)

The IR is defined based on layers after *p-fusion*, which we call *layer groups*, and each *layer group* gets represented by a set of coefficients, as shown in Table 4.2. The representation is easy to expand by adding more terms to accommodate for new network features. IR contains all the operations included in the current *layer groups*. *Layer index* is the sequential number assigned to each conventional layer. Single layer group may have multiple layer index for input in the case of inception module, where various previous outputted feature maps (FMs) are concatenated to form the input. Meanwhile, multiple intermediate FMs generated during layer group computation can be used as other layer groups' residual or normal input sources. *Dump out location* specifics which set of FM to dump out to the off-chip memories. *Element-wise operation location* is used to flexibly adjust element-wise residual addition position in layer groups, so any combination of pool/activation/element-wise addition can be accommodated.

4.5.4 Slicing and allocation

In this stage, IR generated by *Translation* stage is parsed to get network architecture after operation fusion and quantization. Then, an automatic optimizer is applied to explore optimal slicing scheme that maps current architecture to overlay with maximum throughput.

Suppose an individual layer i is sliced into p^i blocks. One slicing scheme for layer i can be defined as a vector of parameter groups $\vec{P}$: $[(IN^i_{j_n}, IM^i_{j_m}, IC^i_{j_c}, OC^i_{j_c})|j_n \in [0, p^i_n), j_m \in$

Table 4.2: IR content for single layer group.

Layer Type	Fully connected(0), Conv(1), extra pool(2)			
Input idx Output idx	Input/Output layer index of length l_{in}, l_{out} $l_{in} > 1$ for inception case $l_{out} > 1$ for residual case			
Input size Output size	3 Dimensional Triple			
Kernel info	Kernel size		Kernel stride	
Pool info	Pool enable	Pool type	Pool size	Pool stride
Padding info	Pad enable	Pad size	pad bf pool info	
Activation type	Relu and Leaky Relu			
Dump out loc	Intermediate FM from either major or affiliated layer can be dumped out to DRAM			
Element-wise op loc	Residual addition can be performed between any two conventional layers			
Quantization info	Dynamic fix point design for Weights/FM			

$[0, p_m^i), j_c \in [0, p_c^i)]$. Each parameter group $(IN_{j_n}^i, IM_{j_m}^i, IC_{j_c}^i, OC_{j_c}^i)$ decides one round of overlay computation, where $IN_{j_n}^i$, $IM_{j_m}^i$, $IC_{j_c}^i$, and $OC_{j_c}^i$ represent input block width, height, input depth, and output depth, respectively. Then we have:

$$N_{in}^i = \sum_{j_n}^{p_n^i} IN_{j_n}^i, \quad M_{in}^i = \sum_{j_m}^{p_m^i} IM_{j_m}^i$$

$$C_{in}^i = \sum_{j_c}^{p_c^i} \frac{IC_{j_c}^i}{\#C_{out}^i\ slices}, C_{out}^i = \sum_{j_c}^{p_c^i} \frac{IC_{j_c}^i}{\#C_{in}^i\ slices}, \tag{4.3}$$

and

$$p^i = p_n^i \times p_m^i \times p_c^i. \tag{4.4}$$

The inference latency L_j^i of one round computation can be represented by

$$L_j^i = (k_x^i \times k_y^i + 2) \times ON_{j_n}^i \times OM_{j_m}^i, j \in [0, p^i), \tag{4.5}$$

where $ON_{j_n}^i$ and $OM_{j_m}^i$ indicate the output block width and height, and the +2 term accounts for initial memory read and final memory write latency. Then we define the throughput of inferencing a network as

$$T = \frac{\sum_{i}^{\hat{m}} N_{out}^{i} \times M_{out}^{i} \times (2 \times C_{in}^{i} \times k_{x}^{i} \times k_{y}^{i} - 1) \times C_{out}^{i}}{\sum_{i}^{\hat{m}} \sum_{j}^{p^i} L_{j}^{i}}, \tag{4.6}$$

where $\hat{m}$ represents the number of layers after fusion.

Therefore, the slicing optimization can be represented as

$$\begin{aligned} & \max_{P} \ T \\ \text{s.t.} \quad & IN_{j_n}^{i} * IM_{j_m}^{i} <= depth_{thres} \\ & ceil(\frac{IC_{j_c}^{i}}{V_{PE}}) * OC_{j_c}^{i} <= N_{PE} \\ & IC_{j_c}^{i}, OC_{j_c}^{i} <= width_{thres}, \end{aligned} \tag{4.7}$$

where $depth_{thres}$ and $width_{thres}$ stand for on-chip BRAM depth and width limit, respectively.

4.5.5 Extra Efficiency improvements

In most cases, combining two levels of parallelism $IC^i * OC^i$ utilizes a large portion of PE resources. However, rare cases exist where C_{in}^{i} and C_{out}^{i} are too small and offer limited parallelism. This usually happens in the first layer, where C_{in}^{1} is fixed to 3 and C_{out}^{i} is usually below 64. Thereby, our proposed compiler further rearranges input feature maps. Pixels in kernel window are moved to fill the channel dimension to increase the parallelism availability channel-wise, as shown in Fig. 4.15. This rearrangement is able to gain over 10× speedup for the first layer computation on average.

Table 4.3: FPGA Resource Utilization.

	LUT	FF	BRAM	DSP	
OPU1024	XC7K325T	94763(46.50%)	150848(37.01%)	165(37.08%)	516(61.43%)
OPU2048	XC7K325T	129927(63.75%)	233996(57.41%)	165(37.08%)	817(97.26%)
OPU4096	XC7Z100	154516(55.70%)	337651(60.86%)	337(44.64%)	1986(98.32%)

Table 4.4: Network Information

	YOLOv2	tiny-YOLO	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	Resnet-50	Resnet-101
Input size	608×608	416*416	224×224	224×224	224×224	224×224	299×299	224×224	299×299
Kernel size	1×1,3×3	1×1,3×3	3×3	3×3	1×1,3×3,5×5,7×7	1×1,3×3	1×1,3×3,5×5,1×3,3×1,1×7,7×1	1×1,3×3,7×7	1×1,3×3,7×7
Pool size/Pool stride	2×2	2×2	2×2	2×2	3×2,3×1,7×1	3×2,3×1,7×2	3×2,3×1,8×2	3×2,1×2	3×2,1×2 height#Conv layer
21	9	13	16	57	69	90	53	53	
Activation Type	Leaky ReLU	Leaky ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU
Operations (GOP)	54.07	5.36	30.92	39.24	2.99	3.83	11.25	6.65	12.65

Table 4.5: RME of OPU1024 for Different Networks.

	YOLOV2	tiny-YOLO	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	Resnet-50	Resnet-101
Frequency (MHZ)					200				
RME	95.51%	89.21%	97.18% (B)	97.30% (B)	90.38% (B)	90.48% (B)	91.10 (B)%	84.48%	86.92%
Conv RME	95.51%	89.21%	97.01%	97.75%	90.45%	90.75%	90.90%	84.48%	86.92%
Frame/s	7.23	68.32	12.18 (B) / 11.28	9.72 (B) / 9.4	112.48 (B) / 104.47	89.77 (B) / 84.60	30.01 (B) / 27.28	54.36	27.05

B: Evaluated in batch mode with batch size 8.

4.6 Experiment Results

We implement three OPU versions with different MAC numbers on Xilinx XC7K325T FPGA and XC7Z100 FPGA. Corresponding resource utilization is shown in Table 5.3. For OPU1024, all the MACs are implemented with DSP. For OPU2048 and OPU4096, part of the MACs are implemented with LUT since the number of DSPs are not enough. A PC with Xeon® 5600 CPU is used for our compiler program. Result interface and device are shown in Fig. 4.16.

4.6.1 Network Description

To evaluate the performance of OPU, 9 CNNs of different architectures are mapped, including YOLOv2, tiny-YOLO, VGG16, VGG19, Inceptionv1/v2/v3, Resnetv1-50 and Resnetv1-101. Among them YOLOV2 and tiny-YOLO are object detection networks and the rest are image classification networks. Detailed network architectures are shown in Table 4.4. Different Kernel sizes from square kernel (1×1, 3×3, 5×5, 7×7) to sliced kernel (1×7, 7×1) are used, in addition to various pooling (1×2, 2×2, 3×1, 3×2, 7×1, 7×2, 8×2) sizes and activation

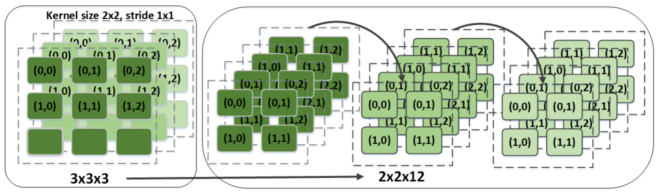


Figure 4.15: Input Rearrangement: channel dimension filling.

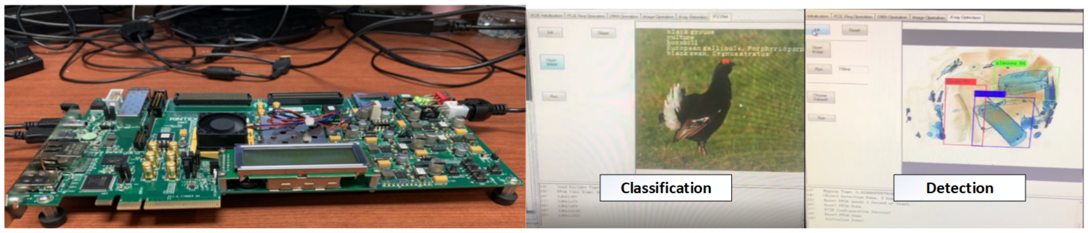


Figure 4.16: Evaluation board and runtime results for classification network VGG16 and detection network YOLO.

types (ReLU and Leaky ReLU). Irregular operations such as inception module and residual module are included. All networks are quantized to 8 bit precision for both kernel weights and feature maps to achieve higher efficiency.

4.6.2 Runtime MAC Efficiency (RME)

OPU is designed to be a domain specific processor for a variety of CNNs. Therefore, Runtime MAC Efficiency (RME) for different CNNs is an important metric for hardware efficiency. RME is calculated by the actual throughput achieved during runtime, divided by the theoretical roof throughput (TTR) of design. The TTR can be calculated by

$$TTR = MAC_{num} \times 2 \times f, \tag{4.8}$$

where MAC_{num} indicates the number of MACs and f represents design frequency. For instance, in our implementation of OPU1024 running with $200MHZ$, 1024 MAC units are utilized for PE array. Therefore, we have $TTR_{opu1024} = 1024 \times 2 \times 200Mhz = 409.6\ GOPS$. The actual throughput achieved by running the convolution part of VGG network on the

Table 4.6: Comparison with customized accelerators (VGG and YOLO).

	[SCD+16]	[QWY+16]	[XLL+17]	OPU1024	[GSQ+18]	OPU1024	
Device	XC7Z045	XC7Z045	XC7Z045	XC7K325T	XC7Z020	XC7K325T	XC7K325T
Network	VGG16				tiny-YOLO		YOLOv2
DSP Utilization	727 (900)	780 (900)	824 (900)	516 (840)	190 (220)	516 (840)	
Data format (bit)	16	16	8	8	200	200	200
Frequency (MHZ)	120	150	100	200	214	200	200
TTR (GOPS)	174	234	329	412	162	412	412
Throughput(GOPS)	118 / 137 (C)	137 / 188 (C)	230	354 / 397 (C)	62.9	366	391
RME	67% / 78 (C)%	58% / 79% (C)	69%	**86% / 97%(C)**	39 %	**89%**	**95 %**

C: Convolutional layer only

OPU is 397 GOPS, so the RME for VGG (CONV) is $\frac{397}{409.6} = 97.79\%$. High RME indicates all computational resources of hardware are well-utilized and processor is efficient for the current network. Note that for fully connected (FC) layers, a high RME is normally difficult to be obtained under non-batch mode due to large number of weights and relatively small computational requirements. Therefore, for networks that contain FC layers, we report the overall RME under batch mode, and frames per second under both non-batch and batch mode for better evaluation. Processor are running under 200 MHZ. As shown in Table 4.5, on average an overall RME of 91.44% on average is achieved for all test networks. This is even higher than the start-of-the-art customized implementations (see 5.6.4).

4.6.3 Comparison with Existing FPGA Accelerators

In this subsection, we compare the performance of OPU with auto-compiler generated network-specific accelerators. Table 4.6 lists out customized accelerators designed for networks VGG16 or YOLO, which are implemented on FPGAs of similar scales for fair comparison. We use throughput and RME as the comparison criteria. The number of MACs and design frequency decide the TTR that can be achieved by certain design. RME shows how well the accelerator utilizes all its available resources actually, which can be directly transformed to real throughput based on TTR. When estimating the TTR of reference design, we also take the data format into consideration. One $25bit \times 18bit$ DSP can be decomposed to handle two $8bit \times 8bit$ multiplications, while it can only handle one $16bit \times 16bit$ multiplication. Therefore, for the same number of DSPs, $8bit$ system has twice overall computational capability compared with $16bit$ system. A coefficient α is used to account for the influence of data format. The RME of different designs can be computed as follows:

$$RME = \frac{T}{\alpha \times DSP_{num} \times f \times 2}, \tag{4.9}$$

where T represents the real throughput achieved by the design, DSP_{num} indicates the total number of DSP utilized. Here $\alpha = 1$ for $16bit$ system, and $\alpha = 2$ for $8bit$ system. The $\times 2$ term is used to compute both multiplication and addition operation. Compiling VGG and YOLO for OPU takes much less time compared to generating one network-specific accelerator by automatic compile. Yet, OPU achieves better RME compared with automatically compiled network-specific accelerators. For example, OPU has 86% RME while the RME for other VGG-specific accelerators ranges from 58% to 69% in table 4.6.

Direct comparison of throughput without taking the number of utilized MACs into consideration is not fair, so we scale our design to match the number of MACs in different reference designs. As shown in Fig. 4.17, the blue dots represent the simulated real performance of OPU implemented with different number of MACs running VGG16. Increasing of the MAC number leads to the improvement of the real throughput. But their relationship is not linear, as MAC number that can not evenly divide the available parallelism of network very well may have low RME. In such cases, extra MACs come with no additional throughput (as indicated by the horizontal lines formed by the blue dots). Purple dots indicate the throughput of reference designs. Most of them locate in the area below OPU "line", which means the similar number of MAC provides lower throughput compared with OPU. Among all the reference designs, implementations in [WYZ+17] and [ZWZ+18] has higher performance compared with OPU. The performance of [WYZ+17] is achieved by its high design frequency (231 MHZ), which is enabled by the systolic PE matrix architecture that shares inputs with neighboring PEs to provide simple routing. However, this architecture is easily constrained by the specific network structure, thus requires the change of PE matrix shape for every new network to achieve claimed performance. (PE matrix shape of [11, 14, 8] and [8, 19, 8] are used for Alexnet and VGG separately [WYZ$^+$17]). Authors of [ZWZ$^+$18] create their own version of pruned VGG to improve performance. Moreover, they use layer-pipelined design which accelerate several layers of one network independently. It requires large modifications of FPGA implementation for different networks ([#DSP, #BRAM] resource of [680, 542] and [808, 303] are used for Alexnet and VGG, respectively).

Normally the design parameters of customized accelerators are tuned specifically to fit the configuration of target network. Therefore, they are able to gain better performance. For example, an accelerator design for VGG can be specifically tuned to fit only the 3×3 kernel size and cannot perform well with YOLO where 1×1 kernel size is included. Moreover, the control flow and data buffer need to be changed to fit different target network. We summarize our performance advantages over customized accelerators into two points. First, input and output channel parallelism has less variation compared with kernel/input feature map pixel parallelism. Combined with the first layer re-arrangement, it can provide enough parallelism to fully utilize the computation resources and gain high throughput. For example, if only paralleling the kernel and input channel, the computation resource can easily fall under-utilized when kernel size is too small. Second, our PE array is designed to fit various types of [input, output] pairs, which cover the majority of the configurations in CNNs. Therefore, we can achieve higher throughput than designs with fixed number of input and output channels.

4.6.4 Power comparison

We compare the power efficiency of OPU with other FPGA designs as well as GPU and CPU. Table 5.13 lists out the comparison results of different hardware platform running VGG16. We measure the power consumption of OPU using a PN2000 electricity usage

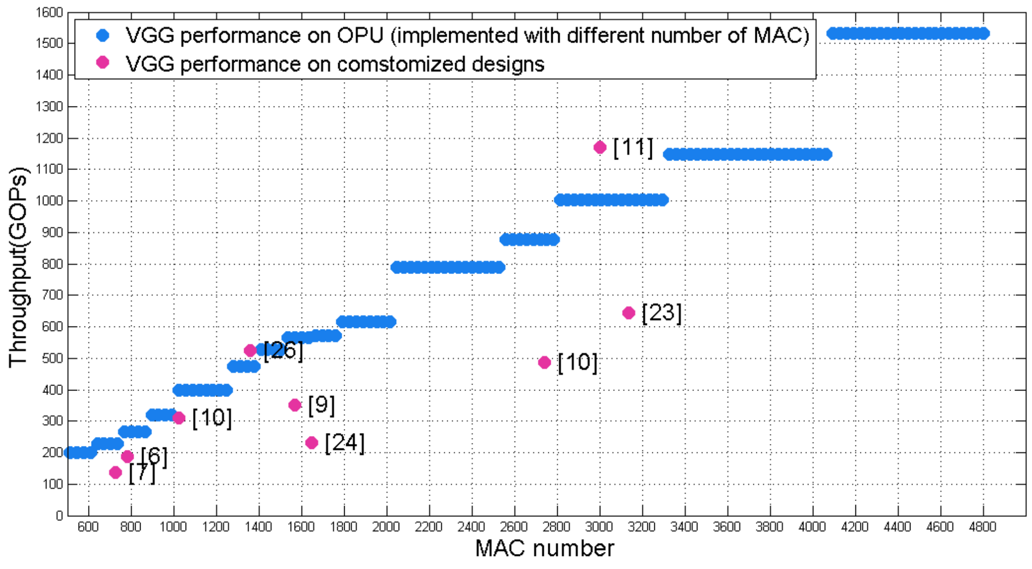


Figure 4.17: Performance comparison of OPU implemented on similar number of MACs with reference designs.

monitor. The reported power includes static and dynamic power consumed by the whole board. FPGA designs generally have better power efficiency compared with CPU, around 2× better power efficiency compared with GPU with batch = 1 and no obvious advantage compared with GPU with larger batch [ZFZ+16][GSQ+18][MSC+16]. Considering the fast development speed of GPU devices, we include the comparison results of three GPUs of different technologies. GTX 780 uses 28nm (same as OPU), GTX 1080 and Titan Xp use 16nm.

We can see From Table 5.13 that the power efficiency of OPU1024 running a VGG16 network is 11.7× better than 14nm technology CPU, on average 4.5× better for batch = 1 GPU devices, and 3.6× to 1.2× better compared with other FPGA designs. The only FPGA design that has better power efficiency is from [ZFZ+16], where they use LUT to implement all the MACs and leave on-chip DSPs idle. They implemented over 4000 MACs (the exact MACs number is not specified by [ZFZ+16]), which brings large throughput advantage compared with OPU1024 with only 1024 MACs implemented. Therefore, we implement

Table 4.7: Power comparison among CPU, GPU and FPGA designs.

	CPU	GPU					FPGA							
Device	i7-8700	GTX 780	GTX 1080 [a]		Titan Xp		[ZFZ+16] KU060	[ZFZ+16] KU060	[ZFZ+16] VX690t	[GSQ+18] XC7Z045	[GSQ+18] XC7Z020	[MSC+16] Stratix-V GXA7	OPU1024 XC7K325t	OPU4096 XC7Z100
Technology	14 nm	28 nm	16 nm		16 nm		20nm	20nm	28nm	28nm	28nm	28nm	28nm	
Power(W)	120	228	180	180		180	25	25	26	9.63	3.5	19.5	16.5	17.7
batch	1	64	1	64	1	64	1	1	1	1	1	1	1	1
Throughput (GOPS)	26	1563	1310	3788	628.53	10620	266	1171	354	137	61.8[b]	114.5[c]	354	1218
Throughput/power (GOPS/W)	0.21	6.85	7.28	21.04	3.49	51.80	10.65	46.84	13.61	14.22	17.65	5.87	21.45	68.81
power efficiency	1×	3.73×	3.96×	11.47×	1.91×	28.25×	5.8×	25.54×	7.42×	7.75×	9.63×	3.20×	**11.7×**	**37.53×**

a Data taken from [ZFZ+16]

b [GSQ+18] only reports convolutional layer throughput(84.3 GOPs), while all the other designs are compared with the overall network throughput. We compute the equivalent overall network throughput based on the throughput relationship of another design implemented by the same paper: $84.3 \times \frac{137}{187} = 61.8GOPs$

c This performance is obtained with running Alexnet

OPU4096 with 4096 MACs on another board with Zynq7z100 for fair comparison. The result shows that OPU4096 has 1.5× better power efficiency compared with [ZFZ+16]. Moreover, OPU4096 shows 3.27× and 1.33× better power efficiency comparing with GTX 1080 and Titan Xp running at batch = 64. This is because static power after bitstream download, which takes a big portion of the overall power, does not tend to increase in proportional with board size. Therefore, using a larger board normally can acquire better power performance, on condition that the performance does not degrade due to design size change.

We also compare the power performance of OPU with GPU/CPU when running other networks such as inception series and residual-net series. We use edge computing targeted GPU platedform Jetson Tx2 and server targeted Titan Xp for evaluation. As shown in Fig. 4.18, OPU1024 with 28 nm has over 15× better power efficiency compared with 14nm i7-8700 CPU. For 16nm Jetson Tx2 and Titan Xp running with batch = 1, the power efficiency of OPU1024 is 2.13× and 5.35× better. For Titan Xp running with batch = 64, OPU1024 is 1.25× better. For Jetson Tx2 running with batch = 16, OPU1024 shows 89% power efficiency. However, big batch size indicates over 6× larger latency compared with batch = 1 mode. For edge computing targeted device the real-time batch = 1 mode with short latency performance is more important.

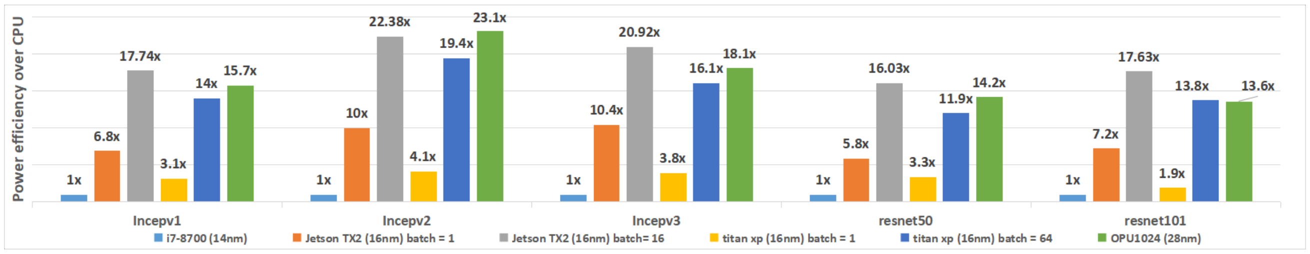


Figure 4.18: Power efficiency (GOPS/W) comparison of CPU/GPU/OPU using CPU as baseline.

4.6.5 Case study of real-time cascaded networks

To further evaluate real-time performance on cascaded networks of OPU, we implement the task on OPU1024 to recognize car license plate from street-view pictures. It is composed of three networks: car-YOLO (YOLO-trained car detection), plate-tiny-YOLO (YOLO-trained plate detection) and a character recognition network (cr-network). For single picture input, the car-YOLO network runs first to label all cars. Then plate-tiny-YOLO and cr-network run to detect the plate numbers or characters for each car.

We compare the performance for OPU1024 and Jetson Tx2. Tx2 is running with batch = 5 and the speed data is computed by total time between input to output divided by 5. Table 4.8 shows that OPU is faster in executing all three networks compared to Jetson.

Table 4.8: Real-time cascaded network evaluation comparison with Jetson

	Frequency (MHZ)	TTR (TOPs)	car-YOLO	plate-tiny-YOLO	cr-network	Speed
Jetson Tx2	845	0.45	188 ms	47 ms	16 ms	1×
OPU1024	200	0.41	64 ms	19 ms	1 ms	2.9×

Overall, OPU is 2.9× faster than Jetson. With similar computation capability, the higher speed achieved by OPU comes from the higher PE utilization rate enabled by our domain specific architecture and compiler.

4.7 Conclusions and Discussions

In this chapter we propose OPU, a domain-specific FPGA overlay processor. We develop a set of instructions with granularity optimized for hardware efficiency. OPU is software programmable and is applicable to a wide range of CNNs without hardware re-configuration. OPU of different scales show $1.2\times$ to $5.35\times$ better power efficiency compared with GPU ($batch = 1$, $batch = 16$, $batch = 64$) and other FPGA designs. Moreover, for cascaded CNN networks to detect car license plate, OPU is $2.9x$ faster compared with edge computing GPU Jetson Tx2 with similar amount of computing resources. Our future work will develop better micro-architecture and more compiler optimization mechanisms, extend OPU to RNN, and also apply and optimize OPU for different deep learning applications, particularly for three dimensional medical images.

CHAPTER 5

Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks

5.1 Introduction

FPGA accelerators for DCNN possess the advantages of high power efficiency, low latency, excellent flexibility and good computational capability. These features make it stand out especially in applications of deep CNNs on edge and embedded devices, *e.g.,* speech recognition on smart phones and visual object recognition in real-time on autonomous driving cars [8], where real-time speed and low power are needed.

With the development of deep learning algorithms, a new group of networks, called LightWeight CNNs (LW-CNNs) [IHM$^+$16, SHZ$^+$18, IMK$^+$14, ZZLS18, Cho17], emerge with the advantages of faster inference time and smaller model size compared with conventional CNNs. While LW-CNNs dramatically shrink down the model size, they also introduce new lightweight operations that cannot be handled well by conventional FPGA CNN accelerators. Moreover, reduction in latency on GPU is also limited. This indicates that lightweight operations do not fit in GPU acceleration architecture (or at least not as nicely as conventional CNN operations do). Therefore, accelerators tuned specifically for LW-CNNs is needed. Several work developed FPGA acceleration for LW-CNNs. [SFL$^+$18] and [ZNL18] designed customized accelerators for MobileNet. However, separated or only partially shared acceleration engines are utilized for conventional convolution and depthwise convolution (DW-CONV). This causes the redundancy in resource utilization and further reduces the runtime efficiency.

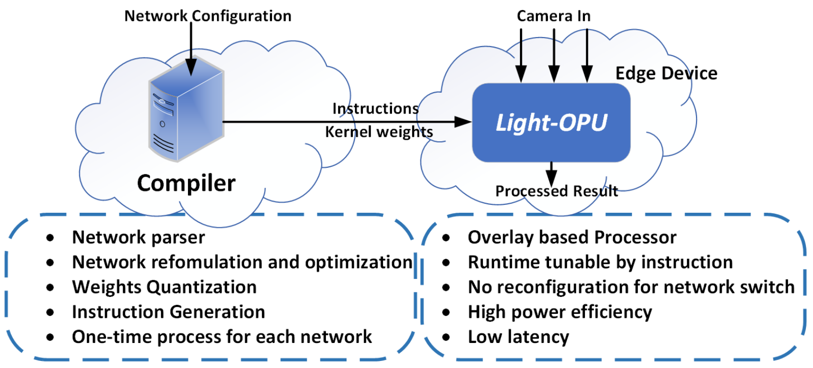


Figure 5.1: *Light-OPU* working flow.

[BZH18] deployed shared acceleration engine for different convolutions, but the architecture is designed for MobileNetV2 specifically. Moreover, some work tried to unify operations by modifying network architectures. [YHW+19] used 1×1 convolution and shift to get rid of DW-CONV, [VB18] and developed network architecture search (NAS) to enhance hardware efficiency for targeted model and dataset. [LDS18] performed NAS with respect to hardware friendly templates and again, targeted dataset. However, modified models are not as universally adaptive to different datasets as the original model, and training cost for NAS is extremely high. In short, existing methods suffer from poor adaptivity to other models, limited operation types, inefficient resource utilization, and high cost of NAS.

To deal with these problems above, we propose *Light-OPU* as an FPGA-based general processor for LW-CNNs acceleration. We adopt part of the instruction and architecture design of our work on conventional CNN acceleration[YWZ+19], then make major improvements to fit the acceleration need of LW-CNNs. More precisely, *Light-OPU* accelerates conventional convolution, DW-CONV and other lightweight operations with one single uniform computation engine. Meanwhile, an automatic compilation framework is provided for the support of general LW-CNNs. As shown in Fig. 5.1, the compiler takes the network architecture configuration from Tensorflow/Keras/ONNX as input, performs the network reformulation and optimization, along with the quantization for compression, then maps network operations to processor modules for instruction generation. Afterwards, the generated instruction sequence is sent to *Light-OPU* for execution. Consequently, fast deployment is enabled for officially published models without any network retraining due to architecture modification.

To be more specific, the features of our proposed *Light-OPU* are listed as follows:

- **Efficient adaptivity to Light-Weight operations**. Taking CNNs as input, *Light-OPU* slices and maps all types of convolutions, including DW-CONV and group convolution to a uniform acceleration framework. Moreover, irregular lightweight operations are either reformulated to fit in the primary computation engine or assigned to the specific acceleration module with low resource cost.

- **Flexible ISA for LW-CNNs**. Our instructions have optimized granularity to guarantee the generality of computation modules. Moreover, instruction based control enables dynamic pipelining of operations. This hides the communication latency and increases the overall efficiency.

- **Acceleration for state-of-the-art LW-CNNs**. We test a set of benchmarks of seven LW-CNNs on *Light-OPU* for performance evaluation. The benchmarks are composed of MobileNet series [HZC+17, SHZ+18, HSC+19], including the newly released MobileNetV3, as well as Xception [Cho17], DenseNet [IMK+14], ShuffleNet [ZZLS18] and SqueezeNet [IHM+16]. All networks can be accelerated without any network architecture modification while achieving 1.3× to 8.4× better power efficiency and up to 172× lower latency compared with state-of-the-art designs [SFL+18, ZNL18, BZH18, VB18, WSWZ19a, MPT+18, PKNP18].

The rest of the chapter is organized as follows. Section 5.2 lists the motivation. Section 5.3 describes the *Light-OPU* instructions. Sections 5.4 and 5.5 explain the *Light-OPU* micro-architecture and the compiler, respectively. Section 5.6 presents our experiment results on various state-of-the-art LW-CNNs. Section 5.7 concludes the chapter.

Table 5.1: Inference time (Batch=1) on NVIDIA Titan Xp GPU, model parameters and number of multiply-add operations. † indicates the ratio compared with that of VGG-19. Input size is 229×229 for Xception and 224×224 for others.

	Inference Time/ms	Speedup†	#Parameter Reduction†	#Operation Reduction†
VGG-19	5.50	1×	1× (138 M)	1× (20G)
SqueezeNetV1.1	1.60	3.43×	74.67×	57.58×
MobileNetV1	2.45	2.24×	32.62×	33.50×
MobileNetV2	3.34	1.65×	40.69×	66.89×
ShuffleNetV1	5.40	1.02×	74.67×	150.57×
Xception	6.44	0.85×	6.05×	4.37×
DenseNet-161	15.50	0.35×	4.85×	2.60×

5.2 Motivation

5.2.1 Non-proportional operation reduction and speedup

Note that when running LW-CONV on GPU platforms, compared with conventional CNNs, the reduction on inference time of LW-CNNs is not proportional to their reduced number of parameters and multiply-add operations. Table 5.1 lists out the comparison of inference time, parameter number and operation number of LW-CNNs with conventional CNN VGG-19 [SZ14b]. It can be seen that the operation number of VGG-19 is 150.57× more than that of ShuffleNetV1, but their inference time on NVIDIA Titan XP GPU is basically the same. Moreover, MobileNetV1 has 33.5× fewer operation number but only gains a speedup of 2.24×. The possible reason is that light-weight operation, *e.g.,* DW-CONV, is more memory bounded than computation bounded. The operations per input element significantly drop compared with conventional convolution. However, CUDA cores are designated for computation-intensive workloads, and they cannot be efficiently utilized in such case. Despite DW-CONV, new lightweight operations still impede acceleration by GPU. As can be seen in Table 5.1, while MobileNetV2 has only 50% of the operation number compared with

MobileNetV1, its execution time on GPU increases by 36%.

Table 5.2: Inference time on CPU with various number of cores.

CPU cores	1	2	4	6	8	10
V1 Latency (ms)	33.56	20.24	13.28	10.31	10.40	9.80
V2 Latency (ms)	27.05	17.27	13.39	11.14	11.35	10.72

As to multi-core CPU, MobileNetV2 has quickly diminished advantage for multi-core execution when compared with V1, because inverted residual operation and higher percentage of DW-CONV employed in V2 require extra memory accesses. This is shown in Table 5.2, where the ratio of $Latency_{v1}/Latency_{v2}$ gradually decreases with the increase of CPU cores. In short, general acceleration platforms (*e.g.,* GPU and multi-core CPU) cannot handle LW operations efficiently. This calls for customized hardware architecture optimized for LW operations, and FPGA acceleration with low non-recurring engineering (NRE) cost is an appropriate candidate.

5.2.2 Uniform support for a variety of Models

Previous work accelerated LW-CNNs via optimizing hardware modules for different operations individually. For instance, [BZH18] and [PKNP18] were specifically designed for MobileNetV2 and SqueezeNet, respectively. [SFL+18] applied separate modules for DW-CONV and conventional convolution without any resource sharing. Moreover, all intermediate feature maps (FMs) are stored on-chip to reduce expensive on-chip off-chip memory traffic, posing constraints on the size of intermediate FMs. DenseNet [IMK+14], with intensive concatenations of previous FMs, can introduce more than 10× on-chip memory overhead and cannot be fit in. Therefore, general support with efficient resource utilization for all special operations in LW-CNNs is required.

Light-OPU accelerates different lightweight operations under a unified hardware architecture. It also optimizes computation efficiency by our compilation framework.

5.3 Instruction Set Architecture

Light-OPU is designed for general LW-CNN inference. We adopt the instruction framework from [YWZ+19] with extra parameter settings for LW-operations specific. Moreover, improvements are made to the instruction execution mechanism for a more flexible and compact run-time execution.

We utilize a complex instruction set architecture, where each instruction can take up several hundreds of cycles to execute. Specifically, each instruction is composed of various number of 32-bit length short instructions. There are two types of short instructions: Conditional instruction (*C-type*) and Unconditional instruction (*U-type*). *C-type* instruction specifies target operations and sets operation trigger conditions. *U-type* instruction delivers corresponding operation parameters for its paired *C-type*. One instruction unit contains one *C-type* instruction with $0 - n$ *U-type* instructions. One instruction block consisting of a number of basic units is fetched together and then distributed to various modules. The least significant bit of instruction indicates the end of current instruction block when its value is 0.

5.3.1 Instruction Types

C-type instruction contains operation (OP) code and trigger condition. OP code indicates the operation type and trigger condition defines the operation execution prerequisite.

We keep six main types of C-instructions defined in [YWZ+19], *i,e.*, *Memory Read*, *Memory Write*, *Data Fetch*, *Compute*, *Post Process* and *Instruction Read*, then add extra control parameters for LW operations. Specifically:

- *Data Fetch* is improved to operate in two modes: (1) *FM reuse* mode corresponds to conventional convolution operation, where only the channel parallelism is explored. Fetched FM is reused for the computations of multiple output channels. The paralleled

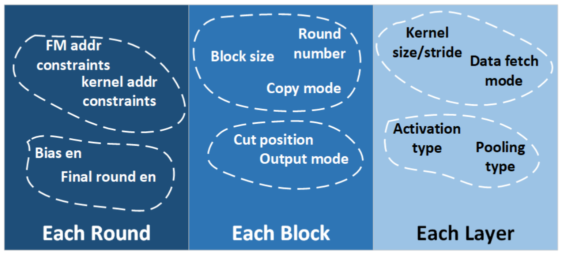


Figure 5.2: Grouping examples of parameters with different updating frequencies.

number of input and output channel can be run-time tuned; (2) Kernel weights reuse mode targets on DW-CONV operations. It explores intra-kernel parallelism to compensate for the limited channel parallelism in DW-CONV. Moreover, kernel weights are reused for computations of multiple FM windows.

- *Compute* controls all processing elements (PEs). One PE computes the inner product of two 1D vectors of length N. $N = 9$ is chosen for our *Light-OPU* (See section 5.4.1), which sufficiently guarantees the space exploration for different networks. Control parameters are added to switch *Compute* between *FM share* mode and *Kernel weights share* mode.

- *Post Process* takes care of extra non-computational-intensive operations, *e.g.,* Squeeze and Excitation (SE) block.

U-type instruction provides operation related parameters. In general, when operation pattern switches, only a subset of parameters are changed accordingly. For a certain *C-type* instruction, its corresponding parameters may have different updating rates. Therefore, as shown in Fig. 5.2, we group parameters with the similar updating rates into the same *U-type* instruction to minimize the total length of instruction sequences, which in turn reduces the memory access time and power consumption.

All the instructions are generated on an updating demand-based scheme, as a set of registers are provided to store the current parameters and trigger conditions until they get updated, which further reduces the length of instruction sequence.

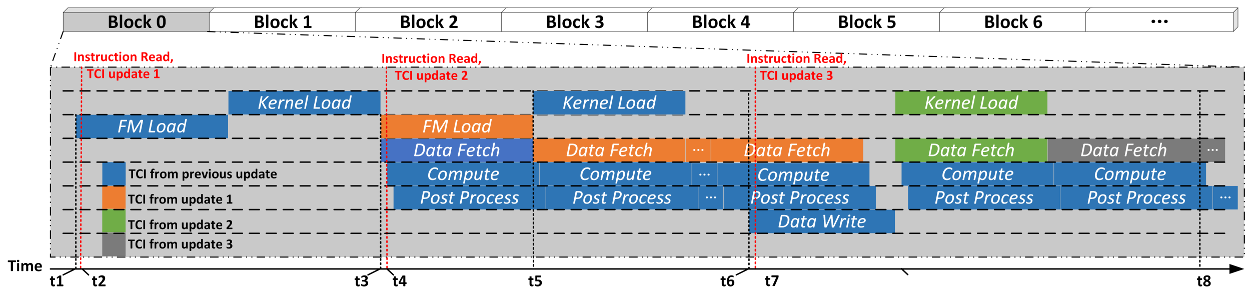


Figure 5.3: Instruction execution and TCI updates of time range $t1$ to $t8$. Red lines indicate TCI updates by *instruction read*. Each colored block shows the execution time range of one triggered instruction.

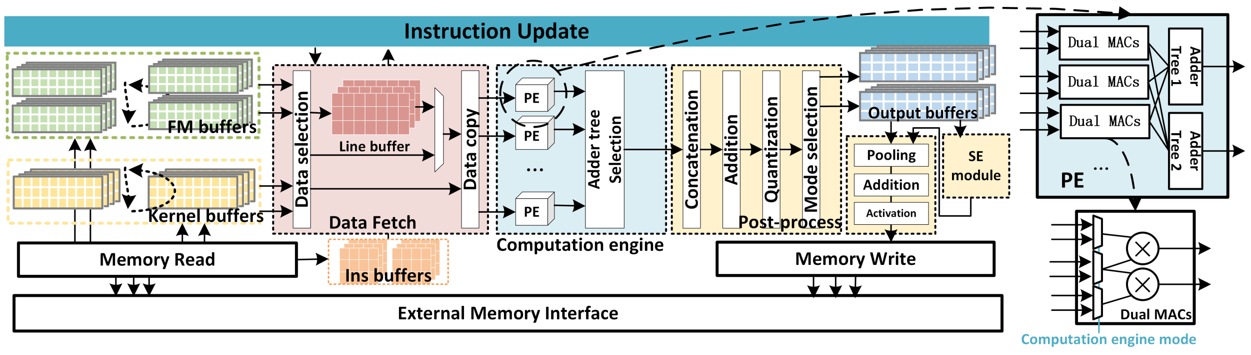


Figure 5.4: Overall micro-architecture and PE structure.

5.3.2 Instruction Execution

We utilize dynamic pipeline fashion to organize our operations. Instead of fixing the instruction order within one layer, the order of our instruction units can be flexibly adjusted for different computation purpose. For efficient instruction control, we design a trigger condition list for each instruction, according to the dependency relationship among different operations under various operating patterns. Modifying the trigger condition index (TCI) by instruction at run-time sets the operation execution prerequisites. Using a dependency based execution strategy relaxes the order enforcement on instruction sequence, leaving enough room for the time uncertainty caused by memory related operations.

For example, Fig. 5.3 shows a fragment of the instruction execution process for one

FM block's computation, where several instructions executed at different time points can be grouped together and read at the same time. Several *Instruction Read* are performed for TCI update, each labeled with one color. The color of instruction during execution process indicates the TCI it currently uses. Note that we update the next TCI right after the trigger of current TCI to make sure the operation will be triggered based on the new TCI next time. For example, *TCI update 1* is performed at time $t2$ right after the trigger of *FM load* at time $t1$. It can be seen that TCIs for *Compute*, *Post Process* and *Data Write* have not been updated within the time range plotted in the figure, where the instructions get executed multiple times whenever the preset condition is satisfied. Another example is *Kernel Load* operation, one mode of the *Memory Read* operations. For *Kernel Load* with TCI color blue, its trigger condition is the completion of *FM Load*. It gets executed twice until *TCI update 3*, labeled with color green, and updates its trigger condition to the completion of *Data Write*. Then at $t6$ *Kernel Load* labeled with green is triggered to pre-load kernel weights for the next round of computation that happens after $t8$.

5.4 Micro-Architecture

Hardware modules in *Light-OPU* are parameter tunable, which switch modes at run-time based on parameter registers updated by instructions. The computation engine is able to operate in different modes according to layer types in order to explore different combinations of parallelism.

As shown in Fig. 5.4, the *Light-OPU* micro-architecture is composed of *Memory Read*, *Memory Write*, *Data Fetch*, *Computation engine*, *Post-Process* and on-chip storage buffers. Each module accepts instruction updates from the *Instruction Update* control module. Micro-architectures only handle the computation of one sub-FM block. If the layer size is larger than the maximum block size allowed by hardware, the layer is sliced into sub-blocks by compiler to fit into hardware (See section 5.5).

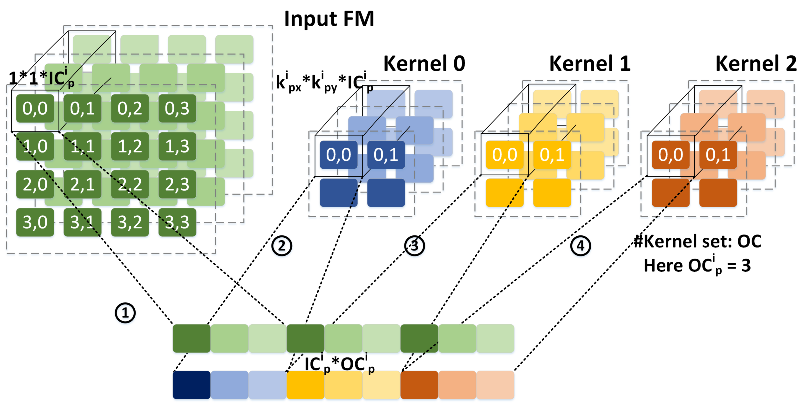


Figure 5.5: Conventional mode: Only the input and output channel parallelisms are explored. Kernel weights of size $k^i_{px} \times k^i_{py} \times IC^i_p$ are decomposed to $k^i_{px} \times k^i_{py}$ $\times 1 \times 1 \times IC^i_p$ point-wise kernels. FM is copied for different output channel calculation.

5.4.1 Computation Engine

For layers in conventional CNN such as YOLO[RF17], GoogLeNet[SVI$^+$16], VGG[SZ14b], ResNet[HZRS16a], and Openpose[CSWS17], flattening the channel level computation guarantees enough parallelism for small to medium FPGA board resources. Moreover, the channel level parallelism is free from the architecture constraints posed by changeable kernel sizes, which ensures the generality of computation engine. However, the emerging LW-CNN comes with the wide application of DW-CONV, bringing challenges to the channel level parallelism based acceleration architectures. For a DW-CONV with n input channels and n output channels, each of the output FM channels is produced by one kernel channel convolving with only one input FM channel. Therefore, the explorable channel parallelism is reduced by $n\times$ compared with conventional convolution layer with the same input and output channel number. Considering the fact that conventional convolution layer is still widely used in LW-CNNs (*e.g.,* DenseNet, SqueezeNet and Xception), we develop two operation modes for the computation engine. With conventional mode targeting at traditional convolutional layers, channel parallelism is explored and FM gets reused. For DW-mode, multiple extra levels of parallelism are explored to handle the DW-CONV layer.

5.4.1.1 Conventional Mode

For conventional convolutional layers, we leverage channel parallelism. Fig. 5.5 explains how it works. For layer i, at each clock cycle, a slice of input channel of depth IC_p^i with width and height as 1×1 is read along with corresponding kernel elements. This fits natural data storage pattern and requires much smaller bandwidth. Parallelism is explored for $IC_p^i \times OC_p^i$. For kernel weights of position $(0, 0)$, input FM channel slice from position $(0, 0)$ to $(2, 2)$ will be fetched out and perform corresponding multiplication. Then we move to kernel weights of position $(0, 1)$. Moreover, our design of the computation unit provides flexible combinations of $[IC_p^i, OC_p^i]$ pairs to accommodate for different layer configurations. By adding selective adder trees after PE array, the computation engine is able to efficiently handle the computation of $[IC_p^i, OC_p^i] = \{[128, 8], [64, 16], [32, 32], [16, 64]\}$. This computation pattern guarantees a uniform data fetching logic for any kernel size or stride, which greatly simplifies the data fetch module, and enables higher design frequency with less resource consumption.

5.4.1.2 DW Mode

For a $[channel_{in}, channel_{out}] = [64, 64]$ DW-CONV, if we use *Conventional mode* for the computation, only 64 multiplications in total can be done in parallel. Therefore, the purpose of introducing *DW mode* for our computation engine is to ensure high run-time resource efficiency of DW-CONV while sharing the same set of PEs with conventional CONV. This can be achieved with an extra data management module. Among all of our target DW-CNNs, the DW layers have a uniform small kernel size of 3×3 (expect for a few layers with kernel size 5×5 in the newly released MobileNetV3). Therefore, we make use of this property and build a typical shift line buffer structure for 3×3 FM window data fetch. The 5×5 kernel can be decomposed into several 3×3 kernels for adaptation. This leads to only less than 3% extra computation time in MobileNetV3 compared with having another line buffer for 5×5 window. As shown in Fig. 5.6, the shift register based line buffer reuses previous

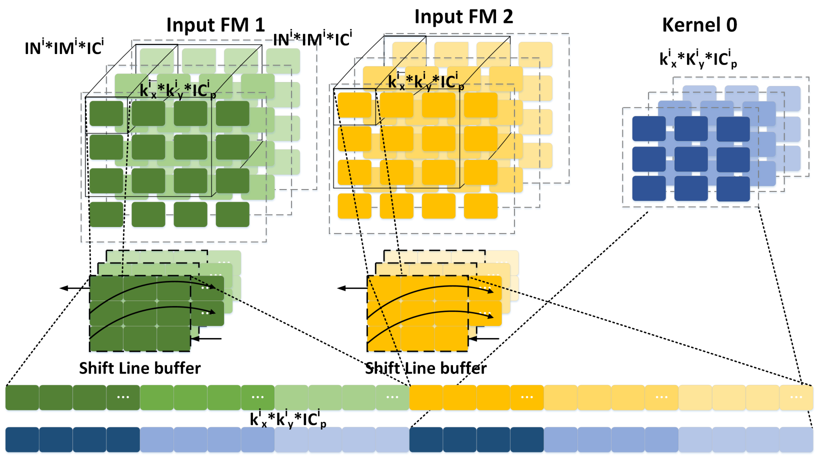


Figure 5.6: DW mode: three levels of parallelisms are explored. (1) Input&output channel level; (2) Intra-kernel level; (3) FM level, as input FMs are fetched from two FM blocks.

values to expand the available FM bandwidth. In this way, the intra-kernel parallelism can be explored and the parallelable multiplications increase to $64 \times 9 = 576$.

Moreover, we decompose each Xilinx DSP48E1 into two 8×8 multipliers to fully utilize computation resources. However, these two decomposed multipliers require sharing of one input due to hardware constraints. For *Conventional mode*, we share the same FM channel data between two different output channels. While for *DW mode*, one input FM channel only corresponds to one output channel. To solve the sharing problem, we fetch FM data from two different FM blocks and share the same kernel weights, as shown in Fig. 5.6.

5.4.2 Other LW operations handling

Apart from DW-CONV, LW-CNNs such as DenseNet [IMK+14] and ShuffleNet [ZZLS18] introduce several other irregular operations which require extra handling.

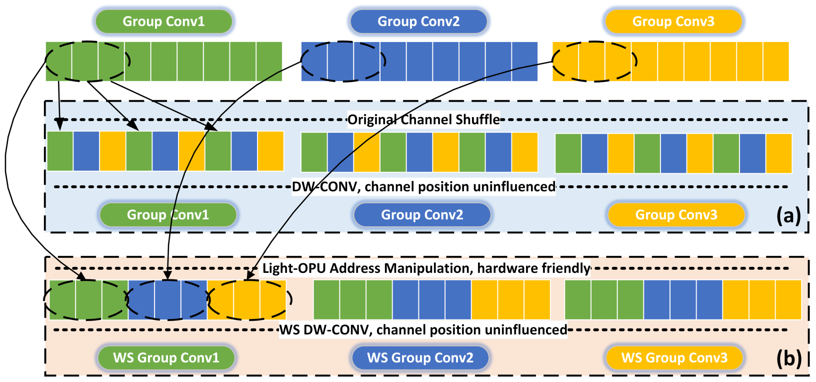


Figure 5.7: (a) *Channel Shuffle* operation and (b) its hardware-friendly implementation.

5.4.2.1 Channel Shuffle

Introduced to increase information sharing among group convolutions, *Channel Shuffle*, explained in Fig. 5.7 (a), performs an important role in ShuffleNet. We label the results from three group convolutions with different colors. The original *Channel Shuffle* operation selects channels separately from each result and recombines channels to form the input of DW-CONV. Then the result of DW-CONV will be fed into another set of group convolutions. This shuffle scheme breaks up the continuous data storage format in memory, thus requiring multiple extra memory read and write operations. To implement the same shuffle scheme in a hardware-friendly way, we reorganize the shuffled results, as shown in Fig. 5.7(b). For each new group, channels from the same original group are put together as smaller groups. Correspondingly, the channel position of kernel weights and biases from following DW-CONVs and following group convolutions gets switched to match the new input order. We label them as Weights-Switched (WS) DW-CONV and WS Group CONV. This reorganization does not change the original shuffle scheme, but greatly simplifies hardware operations. We directly compute the small groups separately and write them to adjacent destination addresses. Therefore, the shuffled results can be formed naturally without any extra memory manipulation operations.

5.4.2.2 Group Convolution

ShuffleNet utilizes *group convolution* to relieve the computation burden from an increased number of channels. *group convolution* slices input FM into separate chucks in channel dimension and conducts individual convolution for each chuck, then concatenates the output FMs. Therefore, performing a *group convolution* can be simplified as calculating several conventional convolutions in sequence with input/output FMs address control, which helps fetch the input channel segments and concatenates output channels. However, as described in subsection 5.4.2.1, *group convolution* in ShuffleNet gets split into smaller convolutions for different output channel groups, which reduces the explorable parallelism and potentially leaves partial PEs idle.

To solve this issue, we fit two *group convolutions* into one round of computation. As shown in Fig. 5.8, we reorganize the kernel weights of *group* $CONV_1^i$ and *group* $CONV_2^i$ into $w1, w2$ and $w3$, each corresponding to one input of next set of three *group convolutions*, respectively. Meanwhile, input FMs for two *group convolutions* are fetched from different FM banks and sent to PE array together with reorganized weights. As a result, the output results get concatenated automatically and can be directly written back to memory. The parallelism of two *group convolutions* cuts the computation time in half without introducing extra memory manipulation operations.

5.4.2.3 Squeeze and Excitation (SE) block

In MobileNetV3 [HSC$^+$19], *SE block* is applied to weight channels for accuracy improvement. The computation increment brought by *SE block* is limited. However, ShuffleNet with *SE blocks* inserted is evaluated in [MZZS18], leading to 26% slow-down in GPU speed compared with original version. This indicates that the irregular structure of *SE block* could degrade computation efficiency of GPU. Therefore, a specific acceleration module is needed. We find that sharing the main computation engine for *SE block* leads to high memory access cost

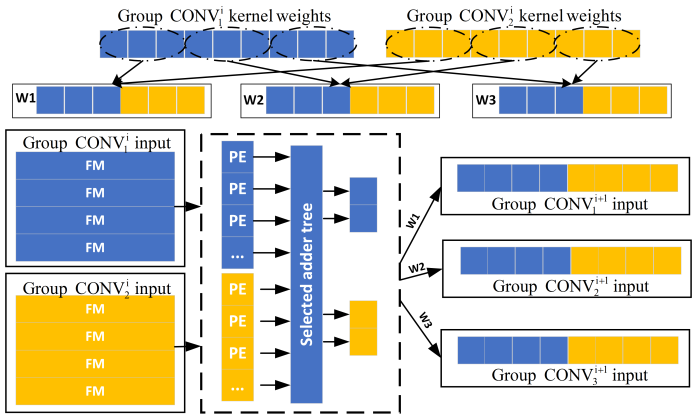

Group CONV$_1^i$ kernel weights
Group CONV$_2^i$ kernel weights
W1
W2
W3
Group CONV$_1^i$ input
FM
Group CONV$_2^i$ input
PE
...
Selected adder tree
Group CONV$_1^{i+1}$ input
Group CONV$_2^{i+1}$ input
Group CONV$_3^{i+1}$ input

Figure 5.8: Calculation of two *group convolutions* in parallel.

due to imbalanced computation cost and data requirement. Therefore, we insert a hardware *SE module* for the computation of *SE block* into the on-chip data flow, which avoids the off-chip data communication with small hardware resource cost. Calculation in *SE block* is shown in Fig. 5.9, where the circled number labels different data sources for different rounds of calculation. For example, when computing *FC (fully connected layer) + ReLU* ①, two inputs for the multiplier array are the results of average pooling and FC weights from the buffer. When computing round ③, one of the inputs is the FC results as the scaling factor, while the other input switches to the intermediate results kept in on-chip BRAM. For one *SE block* with input channel number $ch_{in} = 40$, input FM size $fm_{in} = 56$ and reduction ratio $r = 4$, the calculation takes 6324 cycles with no memory access latency (the weights for FC operation are pre-loaded during previous layer's calculation). Meanwhile, if we calculate the *SE block* using main computation engine, the calculation takes 6324 cycles with 6322 extra memory access latency for intermediate results write and read between rounds. Moreover, the multiplier array in SE module can be shared for the computation of activation function *H-swish* introduced by MobileNetV3, represented as follows:

$$out = \frac{x \times relu6(x+3)}{6}, \tag{5.1}$$

where two arrays of multipliers are needed. Thereby, the multiplier array in the SE module calculates $x \times relu6(x+3)$, and following *activation* module takes care of $\times\frac{1}{6}$.

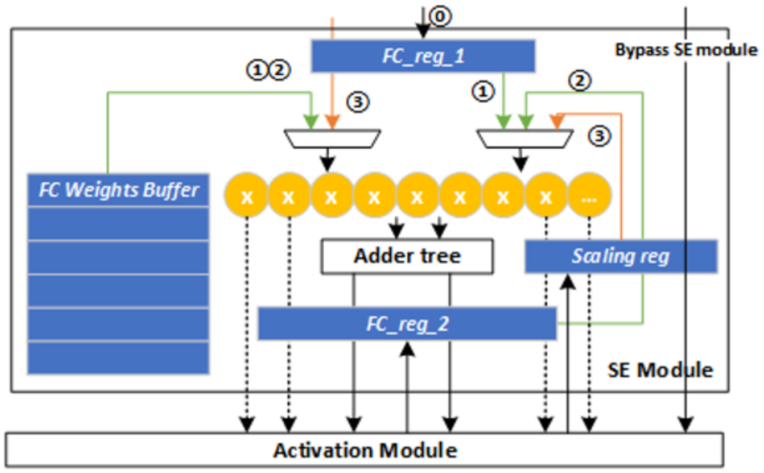


Figure 5.9: Calculation process of *SE block*.

5.5 Compiler

In this section, we propose a compiler as the bridge between network configuration representation and *Light-OPU*'s hardware inference execution. The flow of compiler is shown in Fig. 5.10, mainly accomplishing two goals: (1) **Network Reformulation** that reformulates the network computation into hardware-friendly operations. **Network Reformulation** consists of the steps from *Network Configuration Parsing* to *Operation reordering*; (2) **Hardware Mapping** that maps the reformulated network into hardware with minimum execution latency. **Hardware Mapping** covers the steps from *Network slicing* to *Instruction generation*. The details of each target are discussed as follows.

5.5.1 Network Reformulation

Network Configuration Parsing extracts network structure related information, with input as the frozen model file generated by Tensorflow/Keras/ONNX. Layer parameters and connections are fetched and compressed for easy representation.

Layer Grouping is conducted to link adjacent layers into computation blocks. Each computation block is led by one convolution or fully connected layer, then followed by pooling

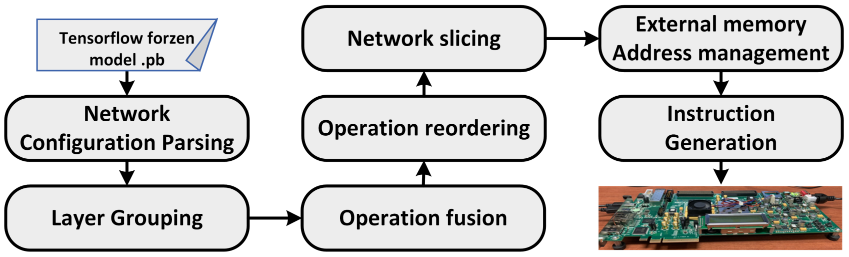

Tensorflow forzen model .pb
Network Configuration Parsing
Layer Grouping
Operation fusion
Operation reordering
Network slicing
External memory Address management
Instruction Generation

Figure 5.10: Compiler Flow.

/ activation / residual layers. External memory access only happens between computation layers to reduce the communication latency.

Operation Fusion is a typical operation in hardware acceleration compilers [CMJ+18]. Layers such as Batch Normalization can be completely merged into preceding convolution layers in some networks. Moreover, we merge the padding operation into the following convolution layer, where padding can be accomplished by zero data selection in *Data Fetch* module.

Operation Reordering is performed in section 5.4.2.1 and 5.4.2.3, where computation order arrangement is sometimes required to make the operation more hardware-friendly. Therefore, kernel weights reorganization and operation order switches are performed to handle the irregular operations introduced by LW-CNNs.

5.5.2 Hardware Mapping

In this stage, an automatic optimizer is applied to explore optimal slicing scheme that maps current architecture to overlay with maximum throughput.

Network Slicing. There are two levels of *Network slicing*, *i.e.,* 2D block slicing and channel slicing. For 2D block size slicing, the block size is constrained by the on-chip buffer size. For channel slicing, the $[Channel_{in}, Channel_{out}]$ combination is limited by the on-chip PE resources. Suppose an individual layer i is sliced into p^i blocks. Then each block is defined as $(IN_j^i, IM_j^i\ , IC_j^i, OC_j^i)$, with $j \in [0, p^i)$, where IN_j^i, IM_j^i, IC_j^i, and OC_j^i represent input block width, height, input channel number, and output channel number, respectively. Note that one sliced block is the FM input for one round of overlay computation, and kernel weights input can be calculated by parameter $[IC_j^i, OC_j^i]$.

If the layer type is conventional convolution, the inference latency L_j^i of one round's computation can be calculated by

$$\begin{aligned} memory_j^i = IN_{j+1}^i \times IM_{j+1}^i \times \left\lceil \frac{IC_{j+1}^i}{Bandwidth} \right\rceil + k_x^i \times k_y^i \times \\ \left\lceil \frac{OC_{j+1}^i \times IC_{j+1}^i}{Bandwidth} \right\rceil + ON_{j+1}^i \times OM_{j+1}^i \times \left\lceil \frac{OC_{j+1}^i}{Bandwidth} \right\rceil \\ compute_j^i = (k_x^i \times k_y^i) \times ON_j^i \times OM_j^i \times \left\lceil \frac{\left\lceil \frac{IC_j^i}{MAC_{PE}} \right\rceil \times OC_j^i}{PE_{num}} \right\rceil, \\ L_j^i = max(memory_j^i, compute_j^i), \end{aligned} \tag{5.2}$$

where ON_j^i and OM_j^i indicate the width and height of output block. *Bandwidth* represents the off-chip memory bandwidth, which takes value 64 under current hardware platform and frequency. MAC_{PE} indicates the number of MACs implemented within one PE unit, which takes 9. PE_{num} indicates the number of PEs, which takes 128. *memory* is the memory access time, including FM data reading and writing as well as kernel weights reading. Note that the kernel weights reading is only required at the first block of the whole layer. *compute* is the computation time required for current block. The overall latency is defined as the maximum of *memory* and *compute*, as we use computation time to hide memory access time.

If the layer type is DW convolution, we modify Eq. (5.2) into

$$\begin{aligned} memory_j^i = (IN_{j+1}^i \times IM_{j+1}^i + k_x^i \times k_y^i + ON_{j+1}^i \times OM_{j+1}^i) \times \left\lceil \frac{IC_{j+1}^i}{Bandwidth} \right\rceil \\ compute_j^i = \alpha \times \left\lceil \frac{ON_j^i \times OM_j^i \times IC_j^i}{2 \times PE_{num}} \right\rceil + IN_j^i \times 2 + 2, \\ L_j^i = max(memory_j^i, compute_j^i), \end{aligned} \tag{5.3}$$

where α represents the kernel size adjustment coefficient, and takes value 1 for 3×3 kernel and 4 for 5×5 kernel. The 2 in the denominator of $compute^i_j$ indicates the two FM banks calculated in parallel (See section 5.4.1.2), and the $IN^i_j \times 2 + 2$ term represents pre-loading time for line buffers.

Therefore, the slicing optimization target can be represented as

$$\begin{aligned} \min_{\omega} \quad & \sum_{i}^{\hat{m}} \left(\sum_{j}^{p^i} L^i_j + memory^i_0 \right) \\ \text{s.t.} \quad & IN^i_j \times IM^i_j <= depth_{thres} \\ & IC^i_j, OC^i_j <= width_{thres}, \end{aligned} \tag{5.4}$$

where $depth_{thres}$ and $width_{thres}$ stand for the depth and width limit of on-chip BRAM, respectively. $memory^i_0$ represents the memory pre-loading time. $\hat{m}$ indicates the total number of layers after *Network reformulation*. ω represents a set of slicing scheme configurations, where each scheme defines p^i sliced block parameters for layer i, including both 2D block slicing and channel slicing.

We use an example to illustrate our slicing strategy. Suppose for a conventional convolution layer, we have $channel_{input} = 96$, $channel_{output} = 48$, $fm_{size} = 48 \times 48$ and $ker_{size} = 2 \times 2$. The constraints are set as $depth_{thres} = 2048$ and $width_{thres} = 64$.

For channel slicing, Fig. 5.11 shows the number of computation rounds required for different slicing schemes. For example, if we slice the layer channel into 2 blocks as $\{[64, 48] \times 2\}$, *i.e.*, reading 64 input channels for each block, we compute partial results of 48 output channels. The computation engine has mode $[64, 16]$, thereby it takes $48/16 = 3$ rounds to finish one block's computation. In total, 6 rounds are required to complete the computation. Similarly, if we slice the channel as $\{[64, 48], [32, 64]\}$, it takes 3 rounds to compute $[64, 48]$ via computation engine mode $[64, 16]$, and 2 rounds to compute $[32, 64]$ using different mode $[32, 32]$. In total, only 5 rounds are needed for the computation, which is the best choice.

For 2D block size slicing, apart from the intuitive rule of filling up the on-chip buffer,

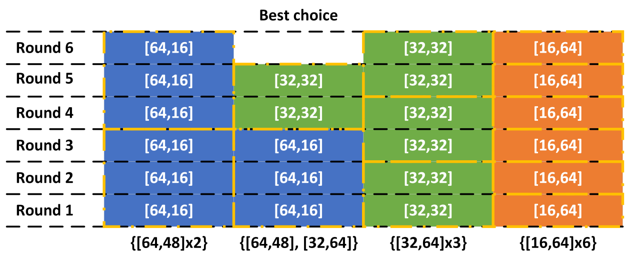


Figure 5.11: Channel slicing example. Each column represents one slicing strategy and each row represents one computation round. Labels on block indicates the number of $[channel_{in}, channel_{out}]$ calculated in this round.

we also need to keep all the block size balanced to hide the memory access latency. As the sum of computation latency stays the same for different slicing strategies, only the memory access time leads to extra latency. From the constraints, we know that at least 4 blocks need to be sliced as FM size $48 \times 48 > 2048$. Suppose we apply the best channel slicing of $\{[64, 48], [32, 64]\}$. This layer will be sliced into $4 \times 2 = 8$ blocks in total, with each block size slicing corresponding to each channel slicing. If we slice the block size as $\{[3 \times 3], [45 \times 3], [3 \times 45], [45 \times 45]\}$, where minimal sized $[3 \times 3]$ block is calculated first since the FM and kernel load latency of the first block cannot be hidden. Then during the computation of block $\{block : [3 \times 3], channel : [64, 48]\}$, we need to load all the data of $\{block : [45 \times 3], channel : [64, 48]\}$. We have $computation = 108$ $cycles$ and $memory = 270$ $cycles$ based on Eq. (5.2). An extra 162 $cycles$ of memory access latency cannot be hidden by the computation. In total, 3033 extra memory latency is induced. However, if we choose a balanced block slicing $\{[22 \times 22], [22 \times 23], [23 \times 22], [23 \times 23]\}$, only the pre-load memory access latency of 916 from the first block is required. Moreover, for balanced

Table 5.3: FPGA resource utilization.

	LUT	FF	BRAM	DSP
Utilization	173522(85.14%)	241175(59.16%)	193.5(43.48%)	704(83.81%)

Table 5.4: Network benchmark statistics.

	MobileNetv1	MobileNetv2	MobileNetv3	SqueezeNetV1.1	DenseNet-161	ShuffleNetV1	Xception
Input size	224×224	224×224	224×224	224×224	224×224	224×224	229×229
Kernel size	including 1×1, 3×3, 5×5, 7×7						
Kernel stride	including 1×1, 2×2						
#Conventional CONV layer	15	36	50	26	160	2	40
#DW-CONV layer	13	17	15	0	0	16	34
#Group CONV layer	0	0	0	0	0	31	0
Activation Type	ReLu	ReLu	H-swish, H-sigmoid	ReLu	ReLu	ReLu	ReLu

slicing scheme, the extra memory access latency stays the same regardless of block number. While for imbalanced slicing scheme, the extra latency increases with the increment of block number. In summary, the minimum latency slicing strategy of this example layer should be $\{[64, 48], [32, 64]\}$ for channel slicing and $\{[22 \times 22], [22 \times 23], [23 \times 22], [23 \times 23]\}$ for 2D block size slicing.

5.6 Experiments

We implement Light-OPU on Xilinx XC7K325T FPGA in a customized board with resource utilization shown in Table 5.3. The power consumption of FPGA board is measured using a PN2000 electricity usage monitor. A PC with Xeon 5600 CPU is used for off-line software compiler. For hardware comparison, we use Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz, with power measured by Stress Terminal UI. We also compare our Light-OPU with edge GPU Jetson TX2. Inferences on GPU use batch = 1 mode for latency dominating evaluation, with power consumption all averaged over 500 runs.

5.6.1 Network Benchmarks

We use seven LW-CNNs including all major lightweight CNNs for a comprehensive evaluation. They are MobileNetV1 [HZC+17], MobileNetV2 [SHZ+18], MobileNetV3 [HSC+19], SqueezeNet [IHM+16], DenseNet [IMK+14], Xception [Cho17] and ShuffleNet [ZZLS18], with

Table 5.5: Network quantization accuracy (Top-1). Performance evaluated on ImageNet LSVRC-2012 dataset.

Network	32*bit* float-point	8*bit* fixed-point
DenseNet-161	77.1%	76.6%
MobileNetV3	68.4%	66.7%

Table 5.6: GPU and FPGA data sheets.

	Jetson TX2	*Light-OPU*
Technology(nm)	16	28
Frequency(MHZ)	1300	200
Peak GOPS	1300	460.8
Bit-width	32*bit* float-point	8*bit* fixed-point

Peak GOPS:Theoretical Peak GOPS when all PEs are used.

statistics shown in Table 5.4. Different kernel sizes (1×1, 3×3, 5×5, 7×7), strides (1×1, 2×2), layer types (Conventional-CONV, DW-CONV, group-CONV) are covered. Irregular operations such as channel shuffle, residual addition and dense block concatenation are also included.

5.6.2 Network Quantization

With existing 8*bit* quantization for SqueezeNet [TM18, PYV18, LHC+18], MobileNetV1 [SFZ+18], MobileNetV2 [PYV18], ShuffleNetV1 [TM18] and Xception [LHC+18], we quantize DenseNet-161 and newly released MobileNetV3 into 8*bit* using typical dynamic fixed-point quantization scheme in this chapter and present accuracy in Table 6.7.2. Below, we use quantized networks for our experiments for FPGA acceleration. Note that CPU and GPU use floating point in our experiments.

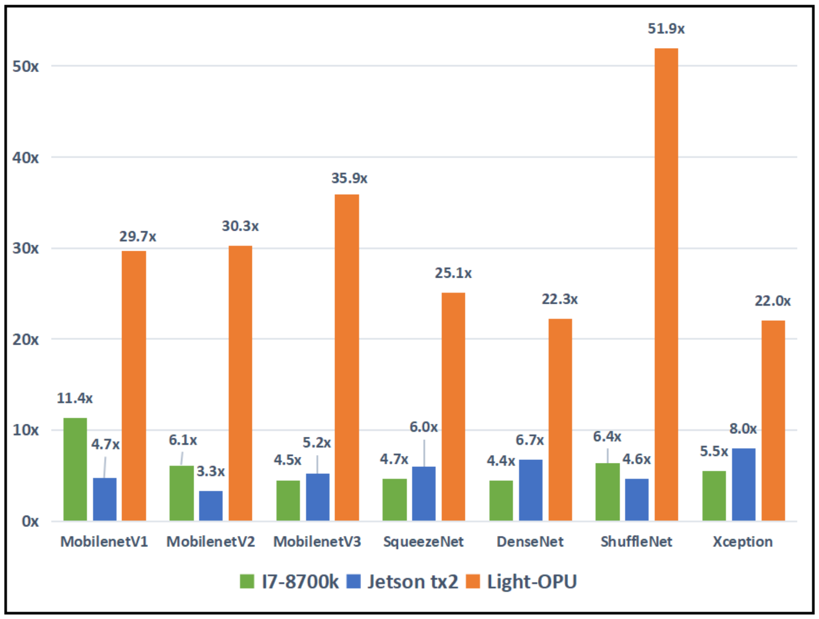


Figure 5.12: Latency comparison (Normalized over embedded CPU ARM Cortex-A57 data).

5.6.3 Comparison with CPU and GPU

With GPU and FPGA information in Table 5.6, we compare the latency in Fig. 5.12 with all latency normalized with respect to ARM A57. Compared with CPUs, *Light-OPU* shows 30.6× and 4.9× speedup over embedded ARM and i7-8700k Intel on average. For GPU comparison, *Light-OPU* has on average a speedup of 5.5× compared with edge GPU TX2, which has 2.8× higher Peak GOPS. The latency advantage of *Light-OPU* over GPUs comes from its domain-specific ISA and micro-architecture tailored for CNN operations, especially LW-CNN operations. For example, *Light-OPU* shows 25.1/6.0 = 4.18× speedup over GPU Jetson TX2 when running SqueezeNet, which contains only conventional 3 × 3 and 1 × 1 convolutions. Meanwhile for MobileNetV2, speedup increases to 30.5/3.3 = 9.18×, because acceleration of DW-CONV layers and residual addition in MobileNetV2 is specifically optimized in *Light-OPU* (See section 5.4.1.2). Moreover, the weights reorganization and *group convolution* parallelism (See section 5.4.2.1 and 5.4.2.2) reduce the memory access latency and the number of operations for ShuffleNet, enabling *Light-OPU* to achieve 11.3× speedup

Table 5.7: Comparison with customized FPGA accelerators.

	[SFL+18]	[ZNL18]	[BZH18]	***Light-OPU***			[VB18]	[WSWZ19a]	***Light-OPU***	[MPT+18]	[PKNP18]	***Light-OPU***
Year	2018	2018	2018	**2019**			2018	2019	**2019**	2018	2018	**2019**
Device	XCZU9EG	Stratix V 5SGSD8	Arria 10 Soc	**XCK325T**			Zynq7045	4×VCU118	**XCK325T**	XC7Z020	DE-10	**XCK325T**
Network	RR-MobileNet	MobileNetV1	MobileNetV2	**MobileNetV1**	**V2**	**V3-Large**	DenseNet-161			SqueezeNetV1.1		
Bit-width	8/4	8	16	**8**			16	16	**8**	32	8 float	**8**
DSP Utilization	1452	1641	1278	**704**			816	4×4993	**704**	192	<336	**704**
Frequency(MHZ)	150	150	133	**200**			125	200	**200**	100	100	**200**
FPS	127.4	231.2	266.2	**264.6**	**313.7**	**323.1**	11.69	153.8	**24.1**	2.6	9	**420.9**
Throughput/MAC (GOPS)	0.03	0.13	0.12	**0.21**	**0.14**	**0.12**	0.17	0.18	**0.25**	0.02	0.02	**0.19**
Power efficiency GOPS/W	NA	NA	4.9	**17.9**	**11.1**	**9.6**	38.9[a]	31.1[a]	**52.5[a]**	1.89	6.49	**15.83**

a: The power data with note (a) is subtracted with Idle power to match the evaluation method of reference power data.

compared with Jetson TX2.

We compare the power efficiency in Fig. 5.13, where the number of useful multiplication/ addition per Watt is utilized to evaluate the **power efficiency** of different networks, and the plotted performance is normalized with respect to ARM A57. *Light-OPU* shows an average of 3.0× better power efficiency compared with GPU Jetson TX2. The superior performance of *Light-OPU* on both latency and power efficiency makes it ideal for various embedded real-time edge computing tasks, *e.g.*, detection, tracking and classification on robotic systems.

5.6.4 Comparison with FPGA Accelerators

To the best of our knowledge, all existing FPGA accelerators are designed specifically for a particular LW-CONV network, therefore we compare our general accelerator *Light-OPU* with customized accelerators. In Table 5.7, we use **frame per second (FPS)** for latency evaluation as all FPGA designs are running at batch = 1 mode. The **Throughput/DSP** is employed for the evaluation of run-time computation resource efficiency, which reflects the percentage of useful computation conducted by DSP on average during run-time. The **Throughput/DSP** is adjusted based on data-width for fair comparison, where values for 8*bit* system are multiplied with 0.5. For MobileNetV1/V2, *Light-OPU* performs 1.6× and 2.3× better in Throughput/DSP compared with existing customized designs, and also gains 2.3× higher power efficiency. For DenseNet-161, compared with [VB18], *Light-OPU* (with

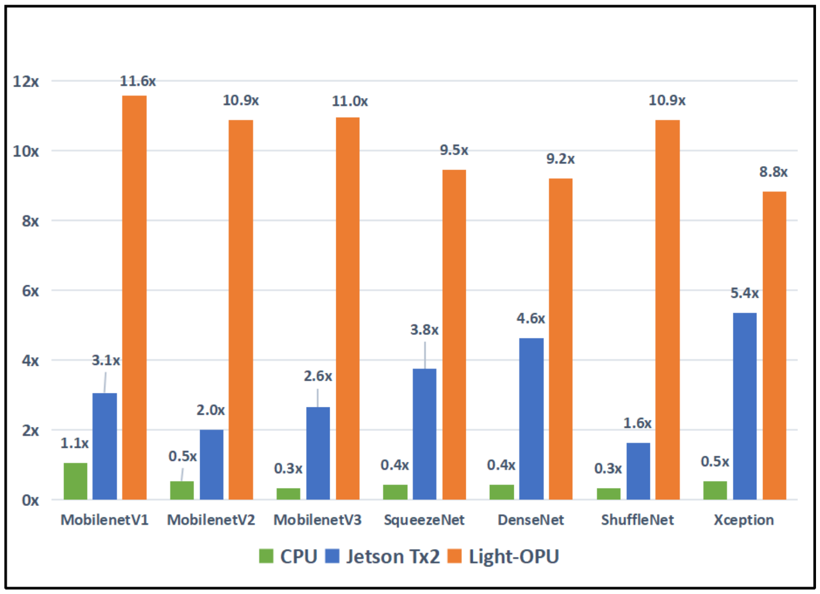


Figure 5.13: Power efficiency comparison (with power normalized over embedded CPU ARM Cortex-A57).

compression of data width) doubles FPS and power efficiency. Multiple-FPGA design in [WSWZ19a] has $153.8/24.1 = 6.4\times$ higher FPS compared with *Light-OPU*, but it utilized $28\times$ more DSP slices. Therefore, it has significantly worse power efficiency and per DSP performance. For SqueezeNet, *Light-OPU* exhibits up to $8.4\times$ higher power efficiency compared with existing designs [MPT$^+$18, PKNP18]. Overall, Light-OPU has $1.39\times$ to $8\times$ improvement in terms of throughput per DSP.

Advantages of *Light-OPU* over existing accelerators are due to the following reasons: (1) Flexible instruction and control enables dynamic pipelining, which reduces off-chip communication time and latency; (2) 8*bit* data representation helps to fully utilize on-chip resources, improve throughput and reduce power consumption; (3) Flexible computation engine design and special handling of various operation in LW-CONV greatly improve the performance and power efficiency.

5.7 Conclusions

We have proposed *Light-OPU*, an FPGA-based overlay processor to accelerate a variety of lightweight CNNs (LW-CNNs). *Light-OPU* performs two levels of optimization: (1) Software-level network reformulation, including layer grouping, operation fusion and operation reordering, eliminates redundant memory access and reduces number of operations in LW-CNN; (2) Hardware-level micro-architecture is specifically designed for LW-CNN operations. Meanwhile, the micro-architecture can be used for conventional convolutional layer computation since it keeps all hardware features such as those from [YWSH19] for conventional CNNs. The flexible acceleration engine guarantees high run-time resource efficiency, and thereby leads to low latency and high power efficiency. *Light-OPU* achieves $5.5\times$ better latency and $3.0\times$ better power efficiency compared with edge computing targeted GPU Jetson TX2, and obtains $1.39\times$ to $8\times$ better throughput per DSP and $5\times$ to $8.4\times$ better power efficiency compared with recent FPGA accelerators for LW-CNNs. Moreover, *Light-OPU* is fully software programmable, and no FPGA reconfiguration is required for network and application switches. In contrast, existing FPGA accelerators are all designed for specific LW-CNNs.

CHAPTER 6

Uni-OPU: An FPGA based Uniform Accelerator for Convolutional and Transposed Convolutional Networks

6.1 Introduction

Transposed convolution (TCONV) — also referred to as fractional strided convolution [RMC15], deconvolution [YBK$^+$18], inverse, up or backward convolution [DV16][SCT$^+$16] — has been proven to be important in various deep learning domains. To begin with, Generative Adversarial Networks (GANs), which are composed of a generator implemented by TCONV and a discriminator by conventional convolution (CONV), have demonstrated remarkable success on style transfer [JAFF16][ZPIE17], synthetic dataset generation [TCAT17][WZX$^+$16] and text-to-image translation [ZXL$^+$17]. Second, a fully convolutional network [LSD15] and its successors that use TCONV to recover input size have shown outstanding performance on biomedical segmentation [RFB15], pixel-wise prediction and content generation tasks [CEE$^+$16][GMAB17]. Moreover, a super-resolution convolutional neural network (SRCNN) [DLHT16][DLT16] equipped with TCONV outperforms traditional image reconstruction approaches in various image super-resolution tasks.

Nevertheless, the efficient execution of TCONV is an issue due to the large number of inefficient operations in its computation process. A TCONV operation typically involves two steps, *i.e.*, up-sampling and CONV. The up-sampling step inserts zeros into the original input feature map (FM) to expand the image size, then the CONV step performs normal convolution on the expanded image. Therefore, a big portion of multiplications in the latter CONV step involve zero values, which wastes hardware computation resources and causes the inefficiency. This phenomenon calls for a specific accelerator for efficient execution and acceleration of TCONV networks. Some literature have developed FPGA accelerators of zero-inserting based TCONV (Zero-TCONV) networks [YBK+18][YSKE18, CKK18, XTW+18, ZDNKD17, LFN+18, WSWZ19b]. Unlike accelerators for CONV networks that focus solely on parallelism exploration and computation-communication balance, Zero-TCONV accelerators require extra designs for inefficient operation elimination. In existing hardware designs, zero-related computations are skipped partially or completely to solve the inefficiency issue. However, as shown in Fig. 6.1, there are mainly three types of existing TCONV accelerators: (1) Accelerators for Zero-TCONV only [ZDNKD17], which cannot process a complete TCONV network containing CONV layers as well; (2) Accelerators that implement Zero-TCONV and CONV engines separately [LFN+18][CKK18], which consumes a large number of hardware resources; (3) Accelerators that implement a combined CONV and Zero-TCONV engine with extra control logic and data buffer overhead [YBK+18][YSKE18][XTW+18][WSWZ19b], as the computing patterns of these two operations are not arranged completely the same. Therefore, developing a uniform accelerator architecture for both CONV and Zero-TCONV with minor resource overhead is still a challenge.

Moreover, recent studies in deep learning algorithms [ODO16] have discovered that Zero-TCONV leads to obvious checkerboard artifacts on generated images. To solve this issue, Nearest-neighbor (NN) interpolation is utilized in the up-sampling step to replace the traditional zero-inserting approach. This new NN-based up-sampling (NN-TCONV) has been widely applied in existing networks to replace the Zero-TCONV. However, while gaining better performance, NN-TCONV renders the existing hardware acceleration architectures invalid, as interpolated elements are no longer zeros and related computations cannot be simply omitted. Meanwhile, if we treat an NN up-sampled image as a regular input and perform a normal convolution, inefficient computations still exist (see Section 6.2.3.2). Therefore,

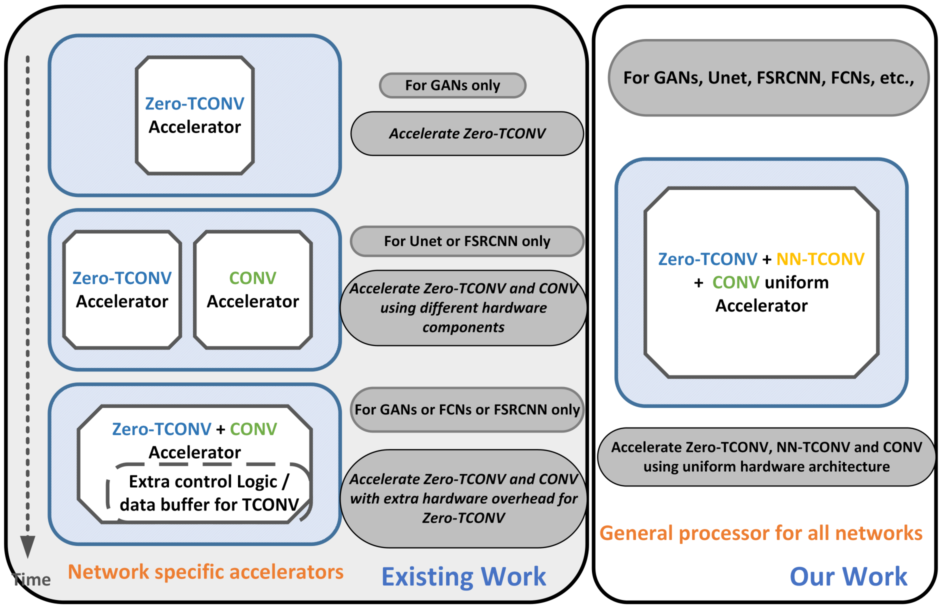

Zero-TCONV Accelerator
For GANs only
Accelerate Zero-TCONV
Zero-TCONV Accelerator
CONV Accelerator
For Unet or FSRCNN only
Accelerate Zero-TCONV and CONV using different hardware components
Zero-TCONV + CONV Accelerator
Extra control Logic / data buffer for TCONV
For GANs or FCNs or FSRCNN only
Accelerate Zero-TCONV and CONV with extra hardware overhead for Zero-TCONV
Time
Network specific accelerators
Existing Work
For GANs, Unet, FSRCNN, FCNs, etc.,
Zero-TCONV + NN-TCONV + CONV uniform Accelerator
Accelerate Zero-TCONV, NN-TCONV and CONV using uniform hardware architecture
General processor for all networks
Our Work

Figure 6.1: Comparison of *Uni-OPU* with existing work.

accelerating NN-TCONV poses another challenge.

In this chapter, we propose an efficient FPGA processor based flow, called *Uni-OPU*, to solve the aforementioned two challenges together. As shown in Fig. 6.1, *Uni-OPU* is able to accelerate Zero-TCONV, NN-TCONV and CONV under a uniform architecture with negligible area overhead. The control flow, data flow and computation engine are shared for all three kinds of layers. Additionally, it is equipped with a complete compilation flow from the high-level network configuration description to an actual FPGA accelerator. We adopt part of the instruction and compiler framework as well as hardware design from our previous work [YWZ$^+$19] for general CONV network acceleration. More precisely, we modify the *architecture parser* for software compilation, which handles the network architecture extraction from Tensorflow [ABC$^+$16] generated configuration files to accommodate different transpose layers. Moreover, a *computation reformulation* is added for kernel conversion and data flow reformulation, which transforms different layers into the same computation pattern. We keep the *scheduling optimizer* for network slicing and scheduling for hardware mapping, as well as the final instruction generation. For instruction architecture, we add several instructions for on-chip data write control and better data fetch control (See section 6.6). For hardware system, we add one address generator and a flag buffer, which leads to negligible area overhead compared with original CONV acceleration design (See section 6.7.1). Overall, *Uni-OPU* is able to perform the inference process of target networks using the instruction sequence generated from software compilation.

We evaluate *Uni-OPU* on various TCONV networks from different application domains, including GANs such as DCGAN [RMC15] for handwritten digits image synthesis, Disco-GAN [KCK+17] for style transfer, and ArtGAN [TCAT17] for artwork style picture synthesis. Other networks such as Unet [RFB15] for biomedical image segmentation, FSRCNN [DLHT16] for image super-resolution and FCN8s [LSD15] for semantic segmentation are also selected as benchmarks.

The major contributions of this work are as follows:

- To the best of our knowledge, we are the first to develop a uniform software/hardware stack for efficient acceleration of traditional Zero-TCONV, newly popular NN-TCONV and regular CONV, with negligible hardware resource overhead compared with regular CONV accelerator.

- A complete compilation flow is provided to automatically analyze the inefficiency of TCONV layers with different types, up-sampling sizes and kernel sizes. Then the computation reformulation, throughput optimization and scheduling will generate instructions accordingly for efficient acceleration execution.

- Evaluation has been performed on six different networks that cover a wide range of applications, while existing accelerators are designed to target only one type of the TCONV networks. On Zero-TCONV networks, *Uni-OPU* (16*bit*) achieves 3.29× to 45.52× speedup compared with conventional CONV FPGA accelerators (CONV-FPGA). On NN-TCONV networks, the speedup is 2.09× to 27.20×. In addition, *Uni-OPU* (16bit) on average outperforms Titan Xp 1.90× and 1.63× on latency and

15.04× and 12.43× on power efficiency, for Zero-TCONV and NN-TCONV respectively. *Uni-OPU* (8bit) on average outperforms Titan Xp 4.01× and 3.41× on latency and 26.58× and 21.38× on power efficiency, for Zero-TCONV and NN-TCONV respectively.

6.2 Background and Motivation

6.2.1 Background

TCONV has been widely utilized in many deep learning network architectures. Popular deep learning platforms such as Tensorflow, Keras [C+15] and Caffe [JSD+14] have specific functions designed to perform TCONV, but many networks implement it using an up-sampling layer followed by a CONV layer, as shown in Fig. 6.2. The relationship between final output FM size and input FM size of TCONV is shown as follows:

$$W_{out} = \frac{(W_{up} - K)}{S} + 1, H_{out} = \frac{(H_{up} - K)}{S} + 1, \tag{6.1}$$

where

$$W_{up} = W_{in} \times up + 2 \times Pad,\ \ H_{up} = H_{in} \times up + 2 \times Pad. \tag{6.2}$$

In Eq. (6.1) and (6.2), we have W_{in} and H_{in} denoting the input FM width and height. S, Pad and K represent the convolution stride, single-side padding size and kernel size [1]. up[2] indicates the up-sampling rate (also known as a scale factor). W_{up} and H_{up} represent the FM size after up-sampling operation. W_{out} and H_{out} indicate the output FM size. Normally we have W_{out} and H_{out} larger than W_{in} and H_{in}. In the example of Fig. 6.2, the size of final output FM is twice that of input FM.

[1]Padding size can be different for all four edges, which is considered in our implementation. Here we use a uniform padding size for simplicity. Similarly, the kernel width and height can be different.

[2]In Tensorflow implementation of TCONV, the up-sampling rate is represented by parameter *strides*.

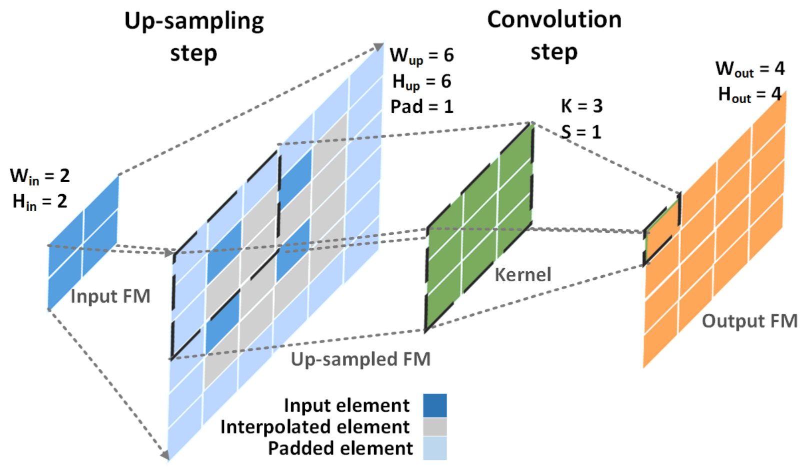


Figure 6.2: A complete two-step process of TCONV.

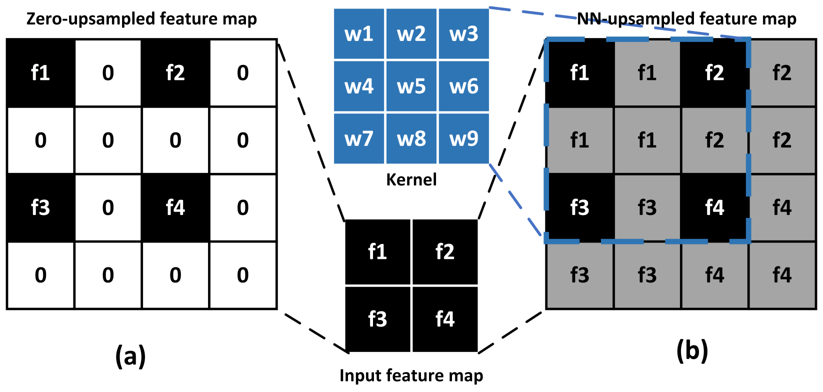


Figure 6.3: (a) up-sampling by padding zeros; (b) up-sampling by NN-based interpolation.

The first up-sampling step can be implemented using different interpolation methods. Conventional TCONV uses a zero-inserting method, as shown in Fig. 6.3(a), where zero padding fills up the up-sampled image. However, it has been proven to cause checkerboard artifacts of various scales in final output image, as can be seen in Fig. 6.4(a). The checkerboard effect comes from the uneven overlapping of non-zeros input values during the second CONV step, which is induced by the zero-inserting operation [ODO16]. To avoid introducing the artifacts, various up-sampling algorithms have been evaluated to encourage weights-tying and reduce the uneven overlap. Among them, nearest-neighbor (NN) interpolation based up-sampling performs the best, as shown in Fig. 6.4(b), where all the checkerboard artifacts

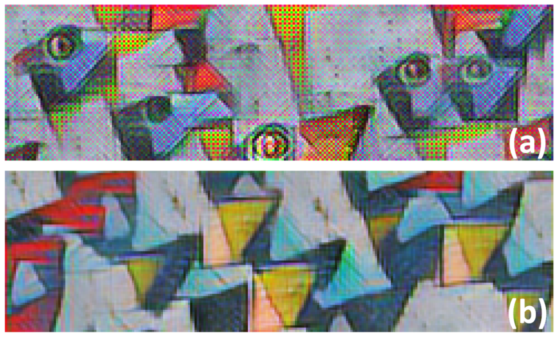


Figure 6.4: (a) Zero-TCONV, heavy checkerboard artifacts; (b) NN-TCONV, no checkerboard artifacts [ODO16].

are eliminated. The up-sampled image based on NN interpolation is shown in Fig. 6.3(b).

6.2.2 Motivation

6.2.2.1 Inefficient operations in Zero-TCONV and NN-TCONV

With regard to hardware acceleration, the inefficiency of TCONV lies in both steps. For the up-sampling step, the interpolated values and their position in the up-sampled output can be inferred directly from input FM. Therefore, we can virtually map out an up-sampled intermediate FM and continue operations afterwards without actually conducting the up-sampling process in hardware.

The latter CONV step in Zero-TCONV owes its inefficiency to the sparse input FM. We count the inefficient operations of Zero-TCONV layers in our benchmark networks, finding that over 70% of the multiplications are wasting the hardware multiplier resources since they have zero as one of the inputs. For the CONV step in NN-TCONV, the inefficiency is not obvious at first glance since we have a dense input FM. However, by taking advantage of the repeated pattern of its intermediate FM, we find that a number of multiplications and additions can be merged and reduced (see Section 6.2.3.2).

6.2.2.2 Lack of a uniform accelerator architecture

Previous work has focused on solving the inefficiency issue for Zero-TCONV. Some work designed specialized hardware components for Zero-TCONV. For example, Zhang et al. [ZDNKD17] only sped up Zero-TCONV. They reverse loop and compute input element positions at run-time, which leads to the hardware resource overhead. Liu et al. [LFN+18] implemented an accelerator for Unet, where different hardware components are used to accelerate CONV and Zero-TCONV. Chang et al. [CKK18] used an individual de-convolutional processor to accelerate Zero-TCONV in the fast super-resolution CNN (FSRCNN).

Other work tried to reuse part of the CONV accelerator for the computation of Zero-TCONV. FlexiGAN [YBK+18] accelerates general GANs, and the design is implemented in ASIC as well [YSKE18]. In this design, the computation is paralleled row-wise. For Zero-TCONV, even and odd rows have different computation patterns, so a MIMD (multiple instruction, multiple data)-SIMD (single instruction, multiple data) combined instruction architecture is used to handle Zero-TCONV while the SIMD part takes care of the CONV. Moreover, extra local instruction buffers are inserted for MIMD control, which leads to higher area overhead. FCN-engine [XTW+18] computes the multiplication of FM elements with one 2D kernel, then adds the overlapped results and crops the edges. It is able to reuse the input/output buffer, weight buffer and multiplication/addition unit of CONV accelerators for Zero-TCONV. Nevertheless, an individual address generator is needed for Zero-TCONV as the data access pattern is not the same as CONV. Moreover, additional distributed on-chip buffers and adders are inserted among PEs only for Zero-TCONV computation. Yan *et al.* [YYT+18] and Wang *et al.* [WSWZ19b] employed similar parallelism methods with different hardware implementations. Additional resources are required for the computation of Zero-TCONV.

All the existing work employs different computing patterns for TCONV and CONV, which comes with extra hardware overhead. Therefore, a uniform accelerator is in need.

6.2.2.3 Lack of hardware accelerator for NN-TCONV

With the advanced NN-TCONV algorithms coming into use, accelerator architectures based on zero-inserting pattern of Zero-TCONV are rendered invalid. Avoiding the zero-related computation no longer works as the up-sampled FM is dense. Hardware accelerators for the new NN-TCONV layers are also in need.

6.2.3 Observation

6.2.3.1 Which level of parallelism should we explore?

The accelerator architecture is decided by the level of parallelism we choose. However, after determining the architecture, the change of hyper parameters (*e.g.*, kernel size, stride, up-sampling rate, FM size, channel number) may require implementation modification. For instance, when the intra-kernel parallelism is explored using the line buffer structure, the line buffer size needs to be changed according to the kernel size [QWY$^+$16]. Moreover, the output FM parallelism based design may need to adjust the size of PE matrix when the FM size varies [DFC$^+$15]. Re-configuration of hardware for each network with different hyper parameters is time-consuming and essentially inefficient for cascaded network designs. Moreover, if layers within the same network have different hyper parameters, the run-time resource utilization may be low due to mismatched hyper parameters and hardware architecture.

We evaluate all the hidden parallelisms within the computation of CONV and TCONV layers, with regard to hyper parameters. As shown in Table 6.1, each column represents one hyper parameter and each row corresponds to one level of hidden parallelism. The red bullet indicates that the implementation for certain parallelism requires adjustment when corresponding hyper parameters change. It can be seen that paralleling the input/output channel is our best choice, because the hardware implementation is only influenced by the channel number, regardless of the change of kernel size, kernel stride, input/output feature map size, up-sampling rate and layer type. Moreover, in our hardware design, we insert logic to reduce the influence of channel number (see Section 6.5.2), making it compatible with all the benchmark networks.

Table 6.1: Implementation influence of hyper parameters when exploring different level of parallelism

	Intra-kernel	Input channel	Output channel	Output FM
Kernel Size	●[c]			●
Up-Sampling Rate	●			●
TConv Type [a]	●			●
Layer Type [b]	●			
Channel number		●	●	
FM Size				●

a Transposed convolution types, including Zero-TCONV and NN-TCONV

b CONV or TCONV

c For example, the change of kernel size will change the hardware architecture when we parallelize the intra-kernel level.

6.2.3.2 How can we accelerate the NN-TCONV?

Unlike Zero-TCONV, NN-TCONV does not introduce any futile computation. However, we can make use of its repeated input patterns to merge the computation and reduce operation number under a properly designed architecture. As shown by the blue kernel and dashed rectangular of Fig. 6.3(b), applying the kernel to the first window of up-sampled FM is shown as follows:

$$\begin{aligned} output = w1 \times f1 + w2 \times f1 + w3 \times f2 + w4 \times f1 \\ +w5 \times f1 + w6 \times f2 + w7 \times f3 + w8 \times f3 + w9 \times f4. \end{aligned} \tag{6.3}$$

Eq. (6.3) involves 9 multiplications and 8 additions, along with the previous up-sampling

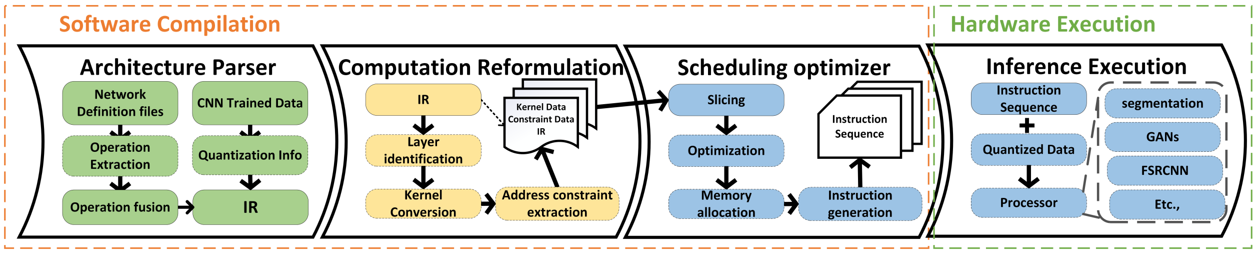


Figure 6.5: Basic components of *Uni-OPU* flow.

operation. However, with a simple rearrangement, Eq. (6.3) can be rewritten as:

$$\begin{aligned} output &= w1' \times f1 + w2' \times f2 + w3' \times f3 + w4' \times f4, \\ with\ w1' &= w1 + w2 + w4 + w5,\ w2' = w3 + w6, \\ w3' &= w7 + w8\ and\ w4' = w9, \end{aligned} \tag{6.4}$$

where only 4 multiplications and 3 additions are required ($w1'$ to $w4'$ can be calculated one-time off-line), and the up-sampling step can be skipped. Therefore, by reformulating the kernel weights and perform pre-adding operations to calculate the value of $w1$ to $w4'$ off-line, we can simplify and accelerate the computation process of NN-TCONV.

6.3 System overview of *Uni-OPU*

Uni-OPU is a complete flow including both software compilation and hardware processor implementation, where two levels of acceleration are performed. The first level is software-level acceleration, where computation reformulation is conducted to eliminate the inefficient operations and unify the computation process. The second level is hardware-level acceleration, where channel level parallelisms are explored and optimized. The overall flow is shown in Fig. 6.5, where a detailed description of each step is described as follows:

- **Architecture parser** parses the .meta configuration file generated by Tensorflow, and extracts the network architecture information. Operation fusion is conducted to merge *Batch Normalization* or *instance Batch Normalization* layer into previous CONV layer.

Parsed operation structures are written into an intermediate graph representation (IR). IR also contains the network quantization information (see Section 6.7.2).

- **Computation reformulation** eliminates inefficient operations, where the kernel conversion is applied to recompute kernel weights. Then the address constraint extraction is conducted to help reformulate the computation pattern. Afterwards, TCONV layer and CONV layer will have the same computation pattern and control flow.
- **Scheduling optimizer** performs the optimization on network slicing, parallelism exploration and scheduling. The optimizer fits the reformulated computation into the processor with the minimum overall latency. Memory allocation is performed along the process. After the scheduling, instruction sequence is generated.
- **Inference execution** takes previous quantized data and instruction sequence as input, and executes the target network inference process on the hardware processor.

In the following section, we will introduce the software components of the *Uni-OPU* flow first, then move to the hardware implementation of the processor.

6.4 Software Compilation

The software compilation includes *Architecture parser*, *Computation Reformulation* and *Scheduling optimizer*. It transforms a network configuration file into an instruction sequence that is processor-executable. Here we focus on the introduction of *Computation Reformulation* and *Scheduling optimizer*.

6.4.1 Computation reformulation

In this step, we reformulate the computation data and operation order for two purposes: (1) Reducing the inefficiency in TCONV; (2) Unifying the computation patterns of three

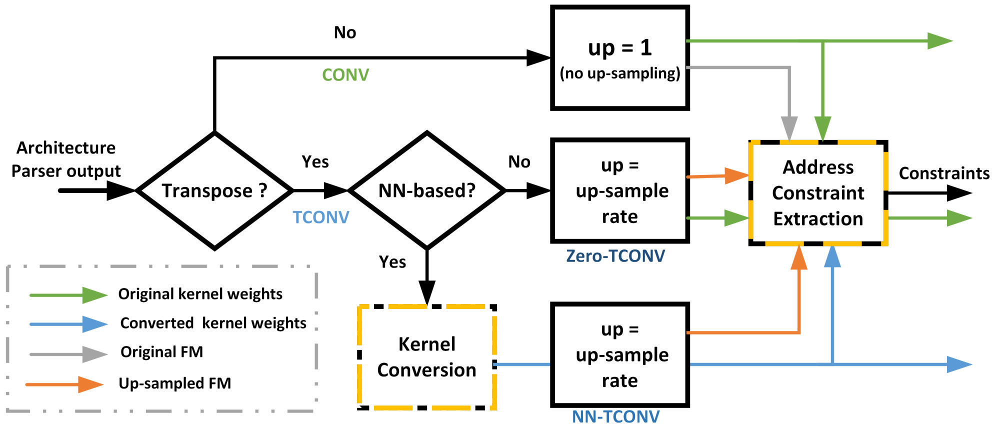


Figure 6.6: Processing flow of computation reformulation.

different layers, i.e, NN-TCONV layer, Zero-TCONV layer and CONV layer. The overall flow for the *computation reformulation* is shown in Fig. 6.6. There are two main functional blocks: *kernel conversion* and *address constraint extraction*. *kernel conversion* performs pre-processing on kernel weights to reduce redundant operations (See Section 6.4.1.1). *address constraint extraction* calculates the set of required constraints for the control of unified computation pattern (See Section 6.4.1.2). It can be seen that we first check whether a layer is a TCONV or not based on *architecture parser* result. If current layer is a regular CONV layer, we go through the top path and perform *address constraint extraction* based on CONV layer's original kernel weights and original FM. If current layer is a TCONV layer, we would further decide the type of TCONV. For Zero-TCONV layer, we would perform *address constraint extraction* on its original kernel weights and up-sampled FM. For NN-TCONV layer, we would first conduct *kernel conversion* to calculate all pre-addable weight combinations and form a new set of kernel weights. Then *address constraint extraction* is performed on converted kernel weights and up-sampled FM. The final output of computation reformulation is a set of address constraint parameters and converted kernel weights, which can be directly applied to original input FM. We discuss each step in detail in the following subsections.

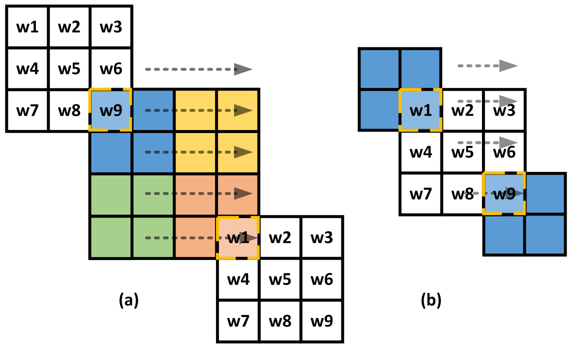


Figure 6.7: Searching for all pre-addable combinations of original kernel weights. (a). Naive method; (b). Improved method for higher efficiency.

6.4.1.1 Kernel conversion

pre-adds weights to skip the up-sampling process and reduce the computation requirement, as indicated in section 6.2.3.2. This requires the identification of all pre-addable combinations of one kernel. A straightforward method is applying the kernel across the up-sampled FM, and recording all the occurred combinations, as shown in Fig. 6.7(a). However, this method produces repeated combinations and is thus not efficient. Instead, we create an auxiliary kernel with size $up \times up$, and slide it across the original kernel weights. As shown in Fig. 6.7(b), all the weights within one kernel window form one pre-addable combination, which takes only $16/36 = 44\%$ searching time compared with Fig. 6.7(a) in this specific case. Then we pre-add the elements in each extracted combination to create converted kernel weights. We can merge these two steps into a CONV operation, where the FM is original kernel weight, and kernel is the $up \times up$ auxiliary kernel with all elements assigned as 1. The output FM of CONV process is the reformulated kernel.

The process of producing the new kernel weights ker_{new} of one layer is described in Algorithm 1. Suppose C_{in} and C_{out} represent the input and output channel number, respectively. We extract each 2D kernel in the original kernel set, and perform a CONV on it with the auxiliary kernel. The padding size is set as $up - 1$. The 2D output gets inserted to ker_{new} as part of the converted kernel weights. Fig. 6.8 shows an example. Suppose we have kernel

Algorithm 1 Kernel Conversion

```
1: function KERNEL CONVERSION($ker_{ori}$, $up$)
2:     $ker_{new} \leftarrow empty, aux_{ker} \leftarrow ones(up, up), pad \leftarrow up - 1$
3:     for $i \leftarrow C_{out}$ do
4:         for $j \leftarrow C_{in}$ do
5:             $tmp\_ker = CONV(aux_{ker}, ker_{ori}, pad)$
6:             $ker_{new}.push\_back(tmp\_ker)$
7:         end for
8:     end for
9: end function
```

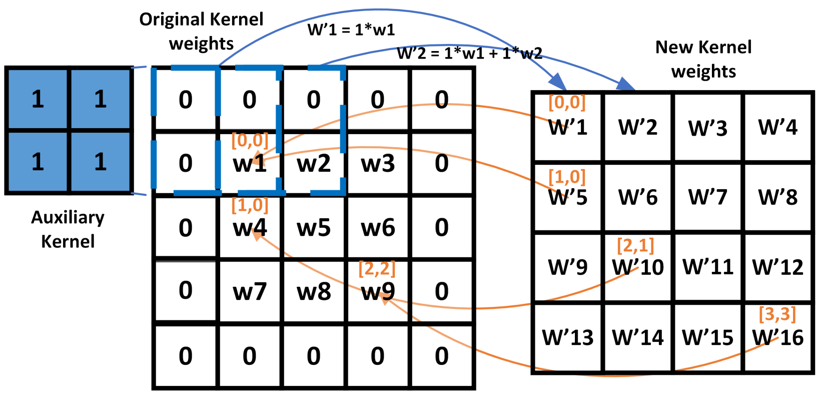


Figure 6.8: Reformulation of the new kernel weights.

size = 3 and $up = 2$. Then the padding size is $pad = 2 - 1 = 1$. The blue arrows illustrate the convolution process, and the reformulated kernel weight has size 4×4. The new weights are listed in Table 6.2.

6.4.1.2 Address constraint extraction

transforms CONV and Zero/NN-TCONV into the same computation pattern, which is described in Algorithm 2. The computation pattern parallels the input/output channel. For a block of input FM and kernel weights, we take out each kernel weight $ker[k, j, i]$, and calcu-

Table 6.2: The value of reformulated kernel.

new kernel	value	new kernel	value
w'1	w1	w'9	w4+w7
w'2	w1+w2	w'10	w4+w5+w7+w8
w'3	w2+w3	w'11	w5+w6+w8+w9
w'4	w3	w'12	w6+w9
w'5	w1+w4	w'13	w7
w'6	w1+w2+w4+w5	w'14	w7+w8
w'7	w2+w3+w5+w6	w'15	w8+w9
w'8	w3+w6	w'16	w9

late its multiplications with selected set of elements from input feature map. The selection is based on the constraints of starting and ending address $[x_{is}, x_{ie}], [y_{is}, y_{ie}]$, as well as strides s_{ix}, s_{iy}. Each multiplication result is added to the corresponding output in address x_o, y_o, which can be calculated by the starting and ending constraints $[x_{os}, x_{oe}], [y_{os}, y_{oe}]$ and strides s_{ox}, s_{oy} of output FM.

Algorithm 3 describes the process of address constraint extraction. If kernel is reformulated, we use the function $MAP\_KORI(r)$ first to map the position of each kernel weight element in 2D kernel (regardless of C_{in} and C_{out}) back to original kernel. Examples are shown by the orange arrows and position labels in Fig. 6.8. $w'1$ maps back to $w1$ in original kernel with position $(0, 0)$, and $w'16$ maps back to $w9$ with position $(2, 2)$. For reformulated kernel weights that have more than one original kernel weight components, we use the position of top left corner weights (see $w'10$ and $w'5$).

Then the constraints are calculated by functions CON_IN_S and CON_IN_E from Algorithm 4. Examples are shown in Fig. 6.9. Suppose we have $up = 2$, $pad.left = pad.right = pad.up = pad.down = 1$. For kernel weights $w1$ at position $[0, 0]$ (mapped back from reformulated kernel weights $w'1$), when we slide it through the up-sampled FM, $w'1$s

Algorithm 2 Unified Computation pattern

```
1: function COMPUTATION($fm$, $ker$)
2:     for $i \leftarrow 0, (C_{out}/P_{out}) - 1$ do                      ▷ $P_{out}$ parallelism
3:         for $j \leftarrow 0, (C_{in}/P_{in}) - 1$ do                     ▷ $P_{in}$ parallelism
4:             for $k \leftarrow 0, K_x \times K_y - 1$ do
5:                 $cur_k = ker[k, j, i],\ x_o = x_{os}[k],\ y_o = y_{os}[k]$
6:                 for $x_i \leftarrow (x_{is}[k], x_{ie}[k])$ with step $s_{ix}$ do
7:                     for $y_i \leftarrow (y_{is}[k], y_{ie}[k])$ with step $s_{iy}$ do
8:                         $out[x_o, y_o, i] += input[x_i, y_i, j] \times cur_k$
9:                         $x_o += s_{ox},\ x_o = (x_o > x_{oe})?x_{os} : x_o$
10:                        $y_o += s_{oy},\ y_o = (y_o > y_{oe})?y_{os} : y_o$
11:                    end for
12:                end for
13:            end for
14:        end for
15:    end for
16: end function
```

Algorithm 3 Address constraint Extraction

```
1: function MAP_KORI($r$)
2:     $r_{ori} = r - (up - 1),\ r_{ori} = r_{ori} < 0?\ 0 : r_{ori}$
3:     Return $r_{ori}$
4: end function
5: function CONSTRAINT($fm$, $ker$, $pad$, $s$)
6:     $In\_addr\_con,\ out\_addr\_con \leftarrow empty$
7:     for $x_k \leftarrow 0, K_x - 1$ do
8:         for $y_k \leftarrow 0, K_y - 1$ do
9:             $x_{kori}, y_{kori}$ = (MAP_KORI($x_k$), MAP_KORI($y_k$))
10:            $x_{is}, y_{is}$=CON_IN_S($x_{kori}, y_{kori}, pad$)
11:            $x_{ie}, y_{ie}$=CON_IN_E($x_{kori}, y_{kori}, pad$)
12:            $s_{xi} = s_{yi} = ceil(s/up)$
13:            $x_{os}, y_{os}$=CON_OUT($x_{is}, y_{is}, x_{kori}, y_{kori}, s$)
14:            $x_{oe}, y_{oe}$=CON_OUT($x_{ie}, y_{ie}, x_{kori}, y_{kori}, s$)
15:            $s_{xo} = s_{yo} = 1$
16:        end for
17:    end for
18: end function
```

applicable range starts at address $[0, 0]$ and ends at address $[3, 3]$, which covers elements from $[0, 0]$ to $[1, 1]$ in the original FM. Note that we calculate the covering range in the original FM directly using CON_IN_S and CON_IN_E, then we have $x_s = y_s = ceil((0-1)/2) = 0$. Furthermore, $W_{up} = H_{up} = 2\times2+1+1 = 6$, then $x_e = y_e = floor((6-(3-0)-1)/2) = 1$. The same range of $[0, 0]$ to $[1, 1]$ is gained. Moreover, we have the strides $s_{xi} = s_{yi} = ceil(1/2) = 1$. Another example is $w'14$, which equals to $w7 + w8$ and maps back to position $[2, 0]$ based on $MAP\_KORI(r)$. Then we can get $x_s = ceil((2 - 1)/2) = 1$, $y_s = ceil((0 - 1)/2) = 0$ as well as $x_e = floor((6 - (3 - 2) - 1)/2) = 1$, $floor((6 - (3 - 0) - 1)/2) = 1$. Therefore, $w'14$ will be applied to input FM address between $[1, 0]$ to $[1, 1]$ with strides $[1, 1]$.

Algorithm 4 Address mapping between original FM and up-sampled FM.

```
1: function CON_IN_S($x$, $y$, $pad$)
2:     $x_{ori} = ceil((x - pad.left)/up)$
3:     $y_{ori} = ceil((y - pad.up)/up)$
4:     Return $x_{ori}, y_{ori}$
5: end function
6: function CON_IN_E($x$, $y$, $pad$)
7:     $W_{up} = W_{fm} \times up + pad.left + pad.right$
8:     $H_{up} = H_{fm} \times up + pad.up + pad.down$
9:     $x_{ori} = floor((W_{up} - (K_{orix} - x) - pad.left)/up)$
10:    $y_{ori} = floor((H_{up} - (K_{oriy} - y) - pad.up)/up)$
11:    Return $x_{ori}, y_{ori}$
12: end function
```

After obtaining the address constraints on input FM, corresponding address constraints on output FM $[x_{os}, y_{os}]$ and $[x_{oe}, y_{oe}]$ can be calculated by Algorithm 5, which simply deduces the output position based on general convolution rules.

The computation reformulation thoroughly eliminates the inefficient operations, thus achieving the software-level acceleration. Fig.6.10 shows the operation reduction rate of

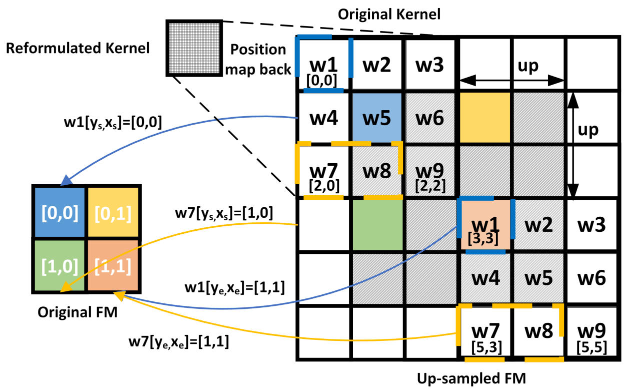


Figure 6.9: An example of address constraint extraction

Algorithm 5 Address mapping to output FM.

```
1: function CON_OUT($x$, $y$, $x_k$, $y_k$, $pad$, $s$)
2:     $x_{out} = (x \times up + pad.left - x_k)/s$
3:     $y_{out} = (y \times up + pad.up - y_k)/s$
4:     Return $(x_{out}, y_{out})$
5: end function
```

TCONV layers gained by the computation reformulation. Both types of TCONV are tested on each network. For instance, we reduce 2.2× operation number for NN-TCONV based DCGAN and 3.6× for Zero-TCONV based DCGAN. Moreover, it can be seen that FCN8s has much higher operation reduction rate compared with rest of the networks, due to its higher percentage of redundant operations caused by larger up-sampling rate. It should be noted that the input and output channel number as well as their computation process remain the same after computation reformulation, and therefore can be parallelized. Then the scheduling optimizer section takes care of the parallelism exploration, which performs the hardware level acceleration. We will discuss it in the following section.

6.4.2 Scheduling Optimizer

Taking the reformulated network as input, we perform the optimization to fit the current network into the hardware processor, aiming for the maximum throughput. We calculate

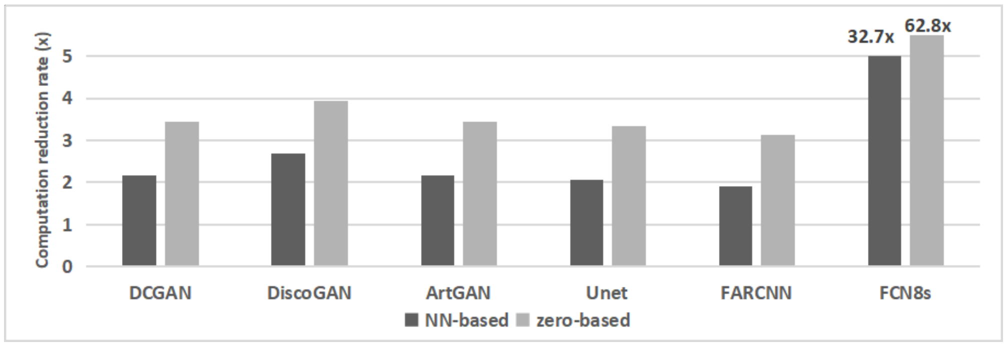


Figure 6.10: Operation reduction by computation reformulation.

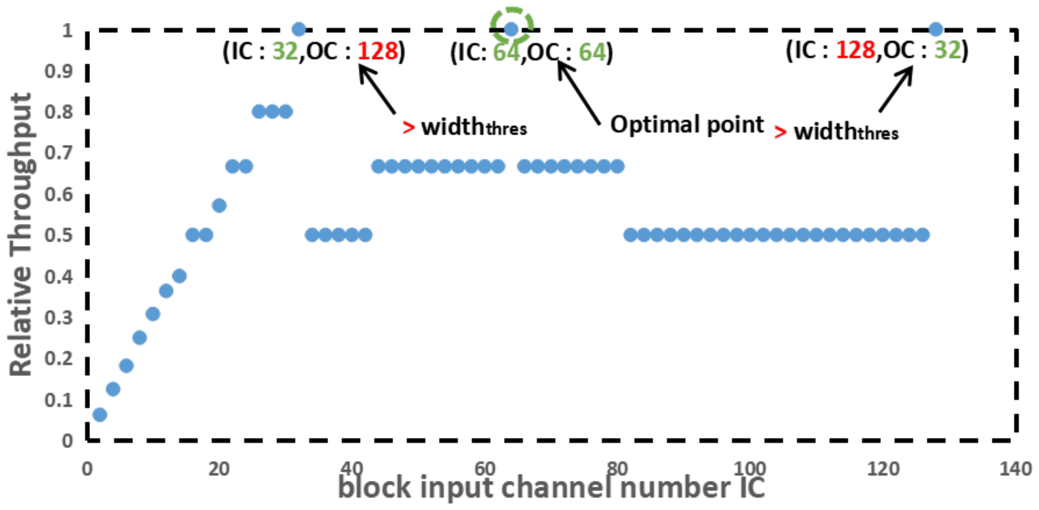


Figure 6.11: Throughput under different choices of $[IC, OC]$.

the network layer by layer. For each layer, the optimization process includes two dimensions of slicing, *i.e.*, FM slicing and channel slicing. Suppose an individual layer i is sliced into p^i blocks, we can define the slicing scheme for layer i as a vector of parameter groups $\vec{P}$: $[(IN^i_{j_n}, IM^i_{j_m}, IC^i_{j_c}, OC^i_{j_c})|j_n \in [0, p^i_n), j_m \in [0, p^i_m), j_c \in [0, p^i_c)]$. Each parameter group $(IN^i_{j_n}, IM^i_{j_m}, IC^i_{j_c}, OC^i_{j_c})$ decides one round of processor computation, where $IN^i_{j_n}$, $IM^i_{j_m}$, $IC^i_{j_c}$, and $OC^i_{j_c}$ represent the input block width, height, the input channel number, and the output channel number, respectively. Then we have:

$$N_{in}^i \approx \sum_{j_n}^{p_n^i} IN_{j_n}^i, \quad M_{in}^i \approx \sum_{j_m}^{p_m^i} IM_{j_m}^i$$

$$C_{in}^i = \sum_{j_c}^{p_c^i} \frac{IC_{j_c}^i}{\#C_{out}^i \ slices}, C_{out}^i = \sum_{j_c}^{p_c^i} \frac{IC_{j_c}^i}{\#C_{in}^i \ slices}, \tag{6.5}$$

and

$$p^i = p_n^i \times p_m^i \times p_c^i. \tag{6.6}$$

The use of $\approx$ for N_{in}^i and M_{in}^i is due to the overlapping area required in the process of FM slicing, so the summation of block sizes will be slightly larger than the input FM width N_{in}^i and height M_{in}^i. Moreover, C_{in}^i and C_{out}^i represent the input and output channel number of the layer.

Suppose layer i has kernel weights $k^i \times C_{in}^i \times C_{out}^i$, we require k^i sets of constraints $\{x_{is_p}, x_{ie_p}, s_{ix_p}\}$, $\{y_{is_p}, y_{ie_p}, s_{iy_p}\}$, $\{x_{os_p}, x_{oe_p}, s_{ox_p}\}$ and $\{y_{os_p}, y_{oe_p}, y_{ox_p}\}$, where $p \in [0, k^i - 1]$. The inference latency L_j^i of one block computation is represented by

$$comp = ON_{j_n}^i \times OM_{j_n}^i + \sum_{p=0}^{k_i-1} \frac{x_{oe_p} - x_{os_p}}{s_{ox_p}} \times \frac{y_{oe_p} - y_{os_p}}{s_{oy_p}}, \tag{6.7}$$

$$load = IN_{j_n}^i \times IM_{j_n}^i + k^i \times IC_{jc}^i \times OC_{jc}^i, \tag{6.8}$$

$$L_j^i = max(comp, load), \tag{6.9}$$

where $ON_{j_n}^i$ and $OM_{j_m}^i$ indicate the width and height of the output block, $ON_{j_n}^i \times OM_{j_m}^i$ accounts for the latency of the output memory write. *load* summarizes the loading time of the input FM block and kernel weights block. Then we define the throughput of inferencing a network as

$$T = \frac{\sum_i^{\hat{m}} N_{out}^i \times M_{out}^i \times (2 \times C_{in}^i \times k^i - 1) \times C_{out}^i}{\sum_i^{\hat{m}} \sum_j^{p^i} L_j^i}, \tag{6.10}$$

where $\hat{m}$ represents the number of layers.

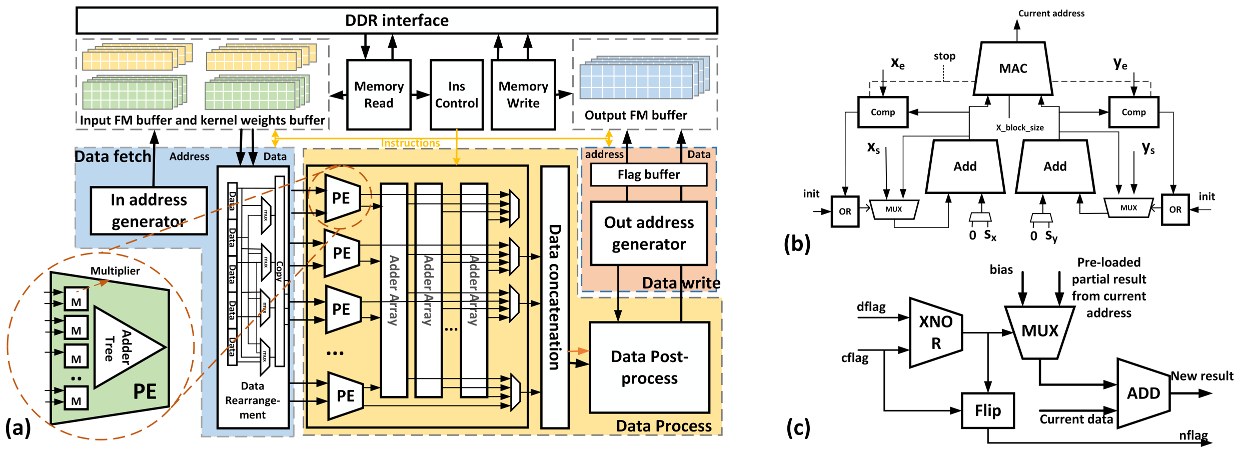


Figure 6.12: (a) Overview of the hardware accelerator architecture; (b) Strided address generator; (c) *first visit identification* module

Therefore, the target of slicing optimization can be represented as

$$\begin{aligned} &\max_{P} \quad T \\ \text{s.t.} \quad & IN^i_{jn} * IM^i_{jm} <= depth_{thres} \\ & ceil(\frac{IC^i_{jc}}{MAC_{PE}}) * OC^i_{jc} <= PE_{num} \\ & IC^i_{jc}, OC^i_{jc} <= width_{thres}, \end{aligned} \tag{6.11}$$

where $depth_{thres}$ and $width_{thres}$ stand for on-chip BRAM depth and width limit, respectively. MAC_{PE} stands for the number of MACs contained by one processing element (PE) unit, and PE_{num} indicates the number of PEs. We have $MAC_{PE} \times PE_{num} = MAC_{num}$, which is the total number of MACs implemented in the design. Here we use an example to show the distribution of achievable throughput under different choices of $[IC, OC]$. Suppose we take $MAC_{PE} = 16$, $MAC_{num} = 4096$ and $width_{thres} = 64$. For one layer with $C_{in} = 128$ and $C_{out} = 128$, the results are shown in Fig. 6.11. Note that the throughput is normalized by the theoretical maximum throughput of the computation engine. It can be seen that there are three $[IC, OC]$ locations with the maximum throughput. However, $[IC : 32, OC : 128]$ and $[IC : 128, OC : 32]$ violate the $IC, OC \leq width_{thres} = 64$ constraint and we therefore have $[IC : 64, OC : 64]$ as the final solution.

6.5 Hardware Micro-Architecture

In this section, we present the hardware micro-architecture of *Uni-OPU*, which is adopted from [YWZ$^+$19] and essentially implements Algorithm 2 as well as the rest of the computation non-intensive layers, such as pooling and activation. Fig. 6.12(a) shows an overview of the accelerator. We use ping-pong structure for both *input FM buffer* and *kernel weights buffer* to hide the off-chip memory access latency. *Ins Control* sends instructions to the rest of modules. *Data Fetch* reads and rearranges the on-chip data. *Data Write* writes back on-chip data. Then *Data Process* includes the computation engine and a post process block.

6.5.1 Address generator

We use address generators in *Data fetch* and *Data write* for on-chip memory access. Benefiting from our computation reformulation, parallelism strategy and data storage pattern, only one address generator is required for each of the data access modules. This greatly simplifies the control logic and provides easy routing. The structure of the address generator is shown in Fig. 6.12(b). For our uniform computation pattern, the memory access of FM data follows a strided pattern. Specifically, the generated address is decided by two strided parameters for x and y dimensions with constraints. Therefore, the address generator is implemented with two adders for the increments of two strided parameters and one MAC for final address computation. When *init* signal is valid, the outputs of adders x_{out} and y_{out} are assigned x_s and y_s. Then for each clock cycle, the x_{out} gets incremented by step s_x until it reaches the ending value x_e, which will trigger the increment of y_{out} by s_y. Each pair of $[x_{out}, y_{out}]$ will generate one address of $y_{out} \times x\_block\_size + x_{out}$ using the MAC unit, until the stop condition ($y_{out} = y_e$ and $x_{out} = x_e$) is reached. Then parameters will get updated and the next round is initiated.

6.5.2 Data Process

As shown in Fig. 6.12, we use a multiplier array followed by an adder tree structure to build the basic PE. MAC_{PE} is the number of multipliers in each PE. Here we use $MAC_{PE} = 16$, which decides the minimum unit of input channel that can be efficiently handled by the computation engine (as described by the second condition of Eq. (6.11)). The number of PEs (PE_{num}) can be scaled according to available computation resource. PE_{num} decides the maximum number of output channels that can be produced by one round of computation. The computation engine is designed to efficiently handle a set of channel number pairs $[In, Out]$ with $In < min(MAC_{num}, width_{thres})$ and $Out < min(PE_{num}, width_{thres})$. $[In, Out]$ for certain round of the computation can be set by the parameters In_mode and Out_mode from instruction. Specifically, a 1024-multiplier array is able to handle the$[16, 64]$, $[32, 32]$, $[64, 16]$ pairs with full efficiency. *Data Post-process* considers the summation of partial results, as well as pooling and activation operations. It is implemented in a $width_{thres}$ data batch SIMD mode for higher throughput. A data concatenation unit is implemented to collect data until the number of output channel reaches $width_{thres}$. Moreover, data quantization is performed by data shift, cut and round operations to save on-chip memory.

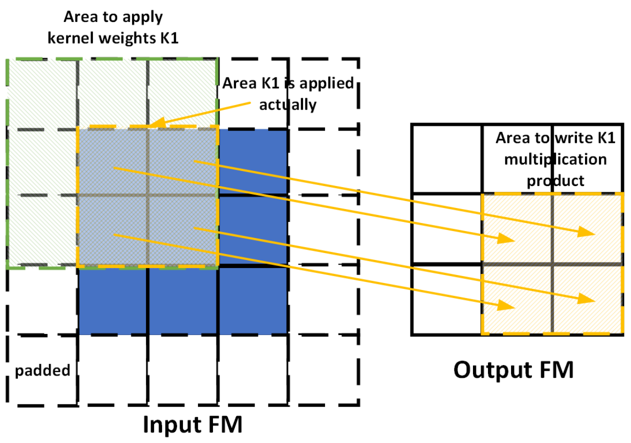
6.5.3 Data fetch and Data write

Apart from the address generator, a data rearrangement unit is implemented in *Data fetch* module for data copy, selection and concatenation. Within each round of computation (line $4 - 13$ in Algorithm 2), one position of kernel weights from different channels will be shared for all multiplications. Meanwhile, FM data will get updated in each clock cycle based on the generated address. We will select out the required bits from FM data and perform the data copy to form fixed length input of *Data Process* based on the $[In, Out]$ operating mode. For instance, if the operation mode is $[32, 32]$, and BRAM bandwidth is set to $64 \times DATA\_WIDTH$, it indicates only the first $32 \times DATA\_WIDTH$ data is valid and should be selected out. Then the selected data will be copied 32 times to fit the 32 computation requirement of the output channel.

In *Data write* module, a flag buffer is utilized for *first visit identification.* As shown in line 8 of Algorithm 2, each output FM element is the result of multiple additions, thereby its initial value needs to be set to zero. This requires us to 'refresh' the content of output buffer BRAM back to zero for each increment of i, which takes $BRAM\_size$ clock cycles due to FPGA hardware constraints. To avoid the 'refreshing' time overhead, we employ a small flag buffer with the same depth as output buffer. Elements stored in the flag buffer take two values, *i.e.*, 0 and 1. For each i iteration computation round, we set a default flag value ($dflag$). When we access one output buffer address, corresponding flag value $cflag$ for that address will be pre-loaded and performs $dflag \odot xflag$, as shown in Fig. 6.12(c). If $XNOR$ result is $True$, we add current computation value with bias and store the results to output buffer, then flip the $cflag$ to its opposite value $nflag$ and store $cflag$ to flag buffer. If the result is $False$, we add the current computation value with existing value in the output buffer, then store the results back. Then $dflag$ will be flipped for next i iteration. Benefiting from this improved writing architecture, we get rid of the padding step (Note that the padding step indicates the general padding at the edge of feature map instead of the up-sampling padding) at the end of *Memory Write* for each layer. Instead the padding effect will be reflected during the computation stage. As shown in Fig. 6.13, we only compute 4 multiplications with valid input FM elements and write to 4 corresponding *Output FM* positions using the write address generator.

6.5.4 Memory Arrangement

Data arrangement in memory influences the achievable bandwidth and the efficiency of data access. As shown in Fig. 6.14, we employ a channel-wise storage pattern that groups elements from different channels under the same address. Therefore, during the data access process, data from different channels can be fetched within a single clock cycle, which satisfies the



Area to apply kernel weights K1
Area K1 is applied actually
Area to write K1 multiplication product
padded
Input FM
Output FM

Figure 6.13: Eliminating regular padding by write address manipulation. The green area covers the 9 *Input FM* elements that need to be multiplied with kernel weights $K1$. However, 5 out of 9 multiplications are redundant. Therefore, we only conduct 4 multiplications in the yellow area and write to the corresponding position of *Output FM*.

requirement of paralleling the input channel.

6.6 Instruction Set Architecture

We utilize a complex instruction set structure for the execution of networks on processor (details of the instruction architecture and execution scheme can be found in section III of our previous work [YWZ$^+$19]). Each instruction takes several hundreds cycles to complete. We define the granularity of our instruction set to be block-level for high flexibility. To be specific, each instruction only controls a part of operations for a single block of input FM data, instead of the complete network layer. Six main instructions are shown in Fig. 6.15. Parameters of different instructions have different lengths. Therefore, in actual implementation, we decompose each long instruction into several short instructions with $32bit$ uniform length. One short instruction contains parameters with the same updating frequency. For example, $[s_{ix}, x_{is}, x_{ie}, s_{iy}, y_{is}, y_{ie}]$ are generally updated for each switch of kernel weights. In contrast, *Kernel size* and *Kernel round* will not get updated until current layer is finished. Therefore,

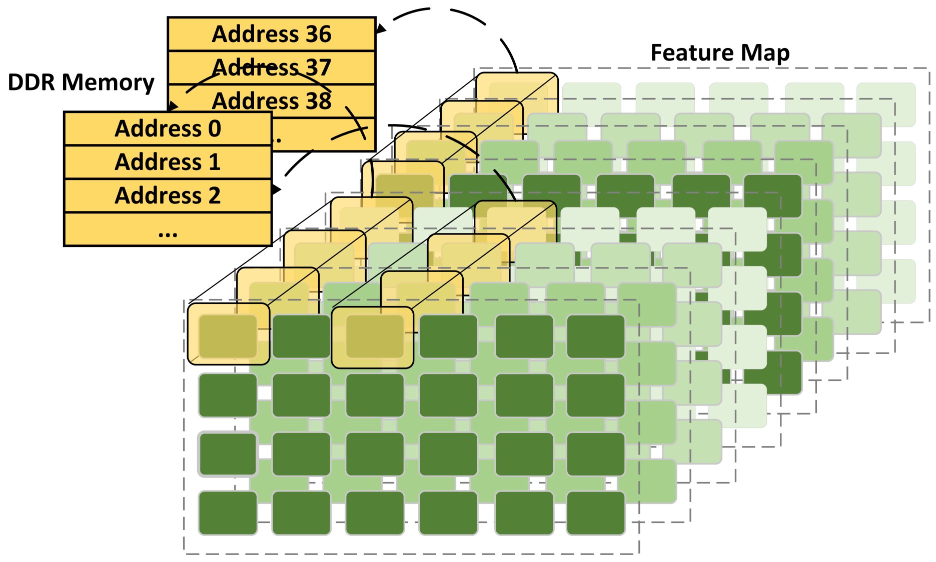


Figure 6.14: Data storage format arrangement.

Instruction	Parameters			
Memory Read	Type	Mode	Start address	Read Number
Memory Write	Mode	Start address	Read Number	
Data Fetch	$s_{ix}, x_{is}, x_{ie}, s_{iy}, y_{is}, y_{ie}$	Kernel size	Kernel round	Block size
Data Write	$s_{ox}, x_{os}, x_{oe}, s_{oy}, y_{os}, y_{oe}$	Default Flag		
Compute	Compute Mode			
Post Process	Pooling en	Activation en	Concat mode	

Figure 6.15: Instruction set.

we only generate short instructions for parameters that require updating at current stage, which reduces the length and loading time of the instruction sequence. Extra parameters for start/end address and stride for on-chip write step are added in *Post process* instruction due to the additional write address generator.

6.7 Experiment

6.7.1 Experiment Setting

We evaluate the acceleration performance of *Uni-OPU* on FPGA on different TCONV networks, which cover a wide range of hyper parameter choices and different application areas. The description for network benchmarks is shown in Table 6.3. A Xilinx Zynq7z100 is

Table 6.3: Information of Network Benchmarks.

Network	TCONV number	CONV number	Up-sample size	kernel size	kernel stride	application
DCGAN	2	1	2	3	1	image synthesis
ArtGAN	5	1	2	3	1	art style image synthesis
DiscoGAN	7	7	2	4	(1,2)	style transfer
Unet	4	19	2	(1,3)	1	segmentation
FSRCNN	1	7	2	(1,3,5,9)	1	super resolution
FCN8s	3	18	(2,8)	(1,3,4,16)	1	segmentation

used to implement 8*bit* and 16*bit* versions of the *Uni-OPU* by hand-coded Verilog, with resource utilization shown in Table 6.4. Compared with conventional convolution network accelerator [YWZ+19], which is a 8*bit* system, the resource overhead of 8*bit Uni-OPU* is $339276/337651 - 1 = 0.48\%$ for FF, $337.5/337 - 1 = 0.14\%$ for BRAM and $1987/1986 - 1 = 0.05\%$ for DSP. Benefiting from the regular padding eliminating, we get an $154389/154516 - 1 = -0.08\%$ LUT utilization reduction. An Intel i7-7700k CPU is used for off-line software compilation. FPGA power is measured using a PN2000 electricity usage monitor, where the power consumption of the complete board is included. For GPU baseline, we use a Nvidia Titan Xp with 3840 CUDA cores running at 1481MHZ. We evaluate the GPU performance of each network using two batch choices (batch = 1 and batch = 64) for more comprehensive comparison. Specifically, batch = 1 evaluates the best single image inference latency for real-time applications. Batch = 64 evaluates the largest throughput as big batch size guarantees high resource utilization. Each test is averaged over 500 runs.

For the rest of the experiment parts, section 6.7.2 shows our network quantization results. Section 6.7.3 shows the acceleration performance on Zero-TCONV implemented networks and their comparison with existing FPGA accelerators. Section 6.7.4 shows the acceleration performance on NN-TCONV implemented networks.

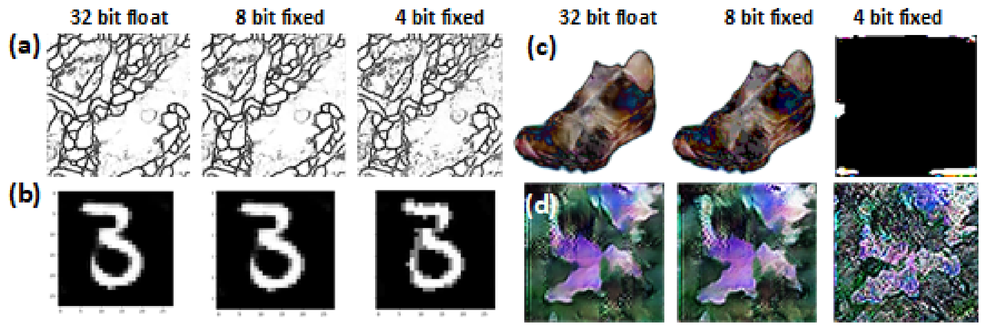


Figure 6.16: (a) Unet; (b) DCGAN; (c) DiscoGAN; (D) ArtGAN.

6.7.2 Network Quantization

Network quantization boosts the performance of hardware accelerators, as lower width representation means more implementable multipliers and adders with regard to the same hardware resource constraints. In our design, one Xilinx DSP48E1 [dsp] is able to implement one $16bit \times 16bit$ fixed-point multiplier or two $8bit \times 8bit$ fixed-point multipliers. Quantization is well-studied in conventional CNN, but fewer related researches are found for TCONV networks. Only Wang et. al. in [WWJ+19] studied the quantization of GANs. Meanwhile, all the previous acceleration work employed the quantized implementation on FPGA, but limited information is provided on the quantization influence of network performance.

Here we employ a dynamic fixed-point quantization scheme, which utilizes different fraction lengths for each layer's weights and FM. The fraction length is chosen so the sum of least square error of quantized value and original value is minimized. The performance of four quantized networks is evaluated visually in Fig. 6.16, where we find $8bit$ fixed representation for both kernel weights and FM is able to maintain the quality of their generated images. For FSRCNN, we calculate the PSNR(dB) [DLT16] of 32 bit float (37.0013 dB), $16bit$ fixed(37.0012 dB) and $8bit$ fixed (32.5965 dB) networks for accuracy evaluation, where 16 bits are required for high FSRCNN accuracy. FCN8s shows obvious segmentation error at $8bit$ quantization, so we only employ $16bit$ version.

Table 6.4: FPGA Resource Utilization.

	LUT	FF	BRAM	DSP
Zynq7z100 8bit	154389(55.66%)	339276(61.15%)	337.5(44.7%)	1987(98.37%)
Zynq7z100 16bit	117971(42.53%)	247238(44.56%)	494.5(65.50%)	1987(98.37%)

Table 6.5: Performance/Watt over GPU compared with other TCONV accelerators.

Network	DCGAN				DiscoGAN			Unet		
Design	[YBK+18]	[WSWZ19b]	*Uni-OPU*		[YBK+18]	*Uni-OPU*		[LFN+18]	*Uni-OPU*	
Bit-width	16	16	16	8	16	16	8	16	16	8
Device	XCVU13P	XC7VX690T	XC7Z100		XCVU13P	XC7Z100		XC7Z045	XC7Z100	
Utilized DSP	1560 (13%)	2048 (64%)	1987 (98%)		NaN	1987 (98%)		640 (71%)	1987 (98%)	
Frequency(MHz)	190	200	200		190	200		200	200	
Performance/Watt over GPU (batch = 1)	3.3×[a]	3.2×[b]	**24.91×**	**37.63×**	3.0×	**18.65×**	**31.89×**	7.85×	**13.95×**	**28.89×**
Performance/Watt over GPU (batch = 64)			**4.81×**	**7.26×**		**4.22×**	**7.22×**	NA	**1.89×**	**4.05×**

a. Compared with Nvidia Titan X without the clarification of batch size b. Compared with Nvidia 1080ti without the clarification of batch size

6.7.3 Acceleration performance of Zero-TCONV networks

We first evaluate all six networks with Zero-TCONV for better comparison with existing acceleration work.

Comparison with conventional convolution accelerator. We use our general accelerator in [YWZ+19] which is optimized only for conventional convolutional layers and has state-of-the-art performance. We address it as CONV-FPGA, indicating that no optimization for TCONV is considered. CONV-FPGA treats TCONV as a regular up-sampled CONV, and we ignore the up-sampling time for simplicity. The speedup performance of *Uni-OPU* over CONV-FPGA is shown in Fig. 6.17. Benefiting from inefficient operation reduction, *Uni-OPU* is 3.29× to 45.52× faster than CONV-FPGA on Zero-TCONV layers. For CONV layers, *Uni-OPU* has the same performance as CONV-FPGA. The speedup of the whole network depends on the percentage of TCONV vs. CONV. More TCONV computation indicates higher network speedup. In summary, *Uni-OPU* is able to speedup the TCONV layer without sacrificing its performance on the CONV layer.

Comparison with GPU platform. The performance comparison of *Uni-OPU* and

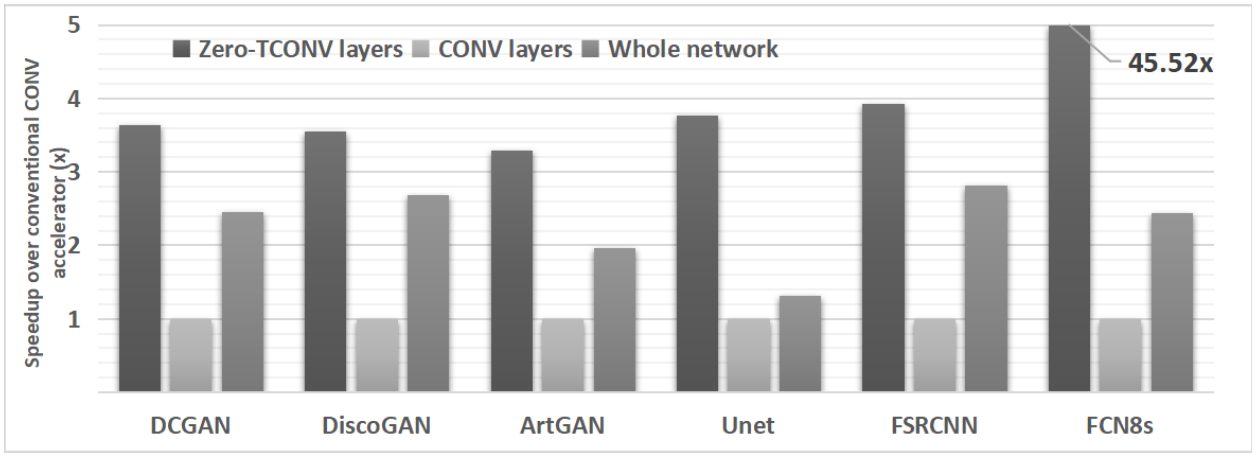


Figure 6.17: *Uni-OPU* vs. CONV-FPGA (Zero-TCONV).

GPU baseline is shown in Fig. 6.18. We have the top graph showing the speed evaluation results of GPU (batch = 1), GPU (batch = 64), *Uni-OPU* (16*bit*) and *Uni-OPU* (8*bit*) (excluding FSRCNN and FCN8s). All the performance is normalized using the GPU (batch = 1) performance. We can find that *Uni-OPU* obtains an average 4.01× (8*bit*) and 1.90×(16*bit*) speedup compared with GPU (batch = 1). When GPU is running at batch = 64 mode, the CUDA units have a higher utilization and the normalized single image speed (not latency) is 5.03× faster than *Uni-OPU* (16*bit*) on average. However, Titan Xp has 1.875× more computation units and 7.4× higher frequency compared with *Uni-OPU*, which adds up to a 13.88× higher computation capability. The performance advantage of *Uni-OPU* is gained from two aspects: (1) The domain specific acceleration architecture we build in FPGA; (2) The TCONV computation reduction by our software compilation. The bottom graph of Fig. 6.18 exhibits the results of performance/Watt, which indicates the energy efficiency of different platforms. *Uni-OPU* (16*bit*) and (8*bit*) have 2.91× to 24.92× and 11.91× to 37.63× higher performance/Watt compared with GPU (batch = 1). Even for GPU (batch = 64), *Uni-OPU* still has 1.14× to 4.81× (16*bit*) and 2.91× to 7.26× (8*bit*) better power efficiency. It can be seen that networks with higher TCONV percentage tend to gain more power efficiency compared with GPU, as a large amount of computation is eliminated.

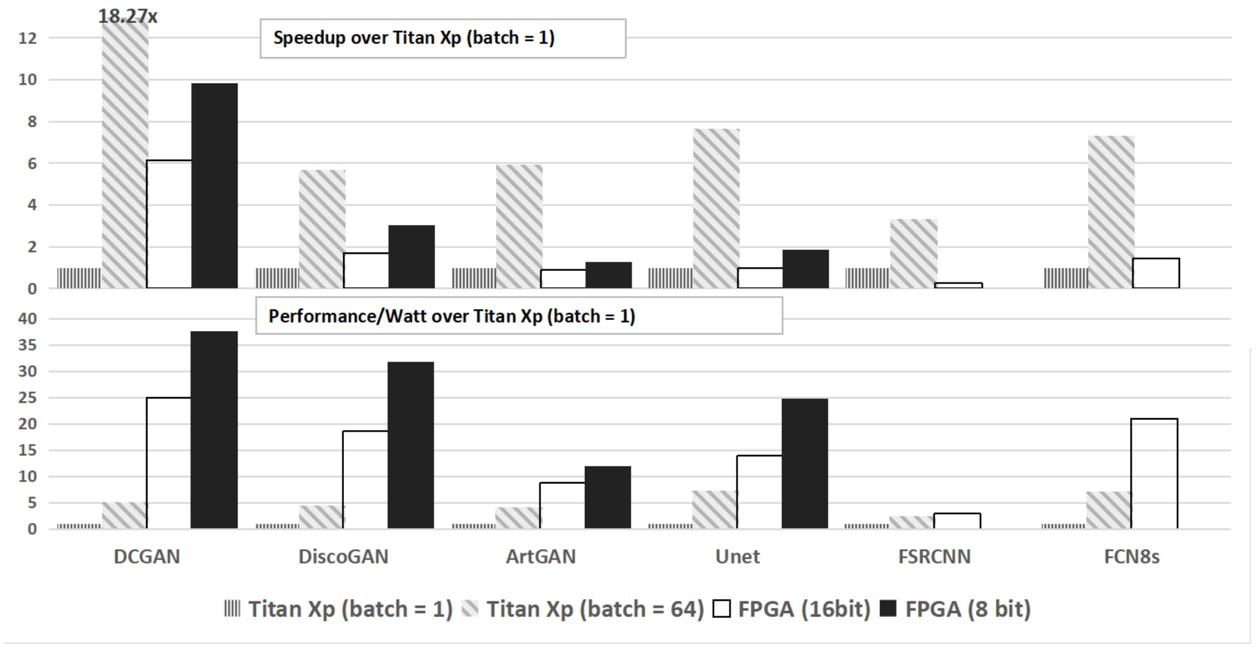


Figure 6.18: *Uni-OPU* vs. GPU baseline (Zero-TCONV).

Comparison with other FPGA TCONV accelerators. We also compare *Uni-OPU* with state-of-the-art Zero-TCONV accelerators, with the results shown in Table 6.5. Note that we employ the performance/Watt over GPU as the comparison criterion, as these benchmarks do not have the network input size and absolute throughput information, which renders direct throughput comparison difficult. On the other hand, power efficiency comparison with GPU is able to reflect both the speed and power consumption performance of the design, thus making it a reliable criterion. These comparison networks [YBK+18] [LFN+18] [WSWZ19b] are implemented via 16 bit quantization. However, as indicated by section 6.7.2, we are able to quantize DCGAN/DiscoGAN/Unet to 8*bit* without the performance degradation of the generated image. Therefore, both 8*bit* and 16*bit* versions are evaluated. Results show that for DCGAN, *Uni-OPU* achieves 4.81× and 7.26× power efficiency over Titan Xp (batch = 64). Meanwhile, previous design has only 3.3× and 3.2× compared with less powerful GPU Titan X and 1080ti. Similarly, for DiscoGAN, 4.22× and 7.22× better power efficiency over Titan Xp is achieved, while previous work shows only 3.0×. For Unet,

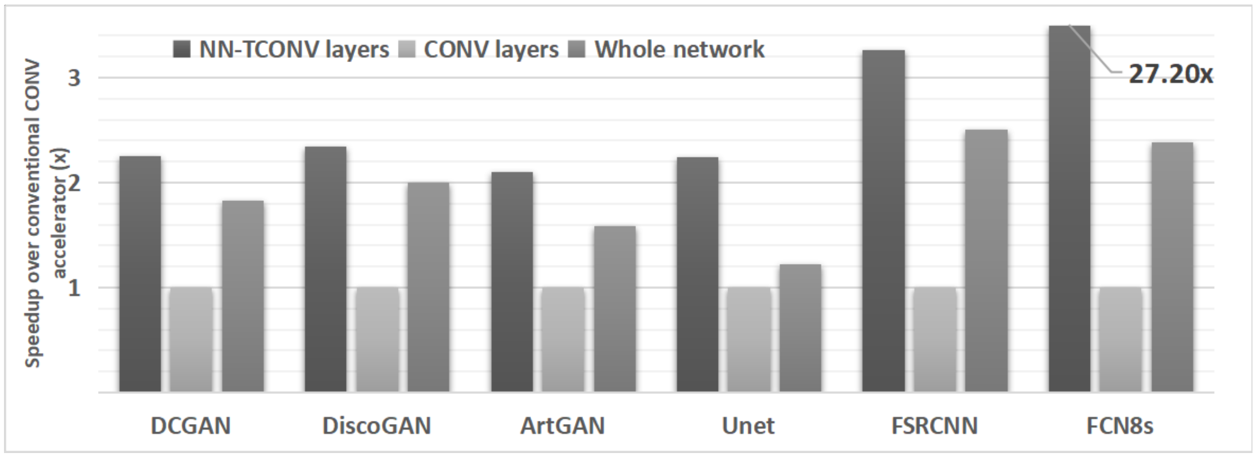


Figure 6.19: *Uni-OPU* vs. CONV-FPGA (NN-TCONV).

Uni-OPU obtains $\frac{13.95}{7.85} \approx 1.8\times$ and $\frac{28.89}{7.85} \approx 3.8\times$ higher power efficiency compared with [LFN+18]. It should be noted that work [YBK+18][WSWZ19b][LFN+18] have individual accelerators with specifically tuned parameters for each network. Meanwhile, *Uni-OPU* is a uniform processor that can execute different TCONV networks by switching instruction sequence without FPGA reconfiguration. The performance gain of *Uni-OPU* comes from the uniform architecture we employ for both TCONV and CONV layers. The complete sharing of control logic, data buffer and computation engine guarantees the high run-time computation resource efficiency, which in turn improves the power efficiency.

6.7.4 Acceleration performance of NN-TCONV networks

The hardware acceleration of NN-TCONV has not been explored before, therefore we compare our acceleration performance with CONV-FPGA and the GPU baseline only.

Comparison with conventional convolution accelerator: NN-TCONV performs poorly in inefficiency reduction since no zero multiplication is introduced by the up-sampling step. However, based on the kernel conversion (see Section 6.4.1.1), we are able to make use of the NN-up-sampling property, which effectively reduces the required number of computations. As shown in Fig. 6.19, for NN-TCONV layers, 2.09× to 27.20× speedup can be

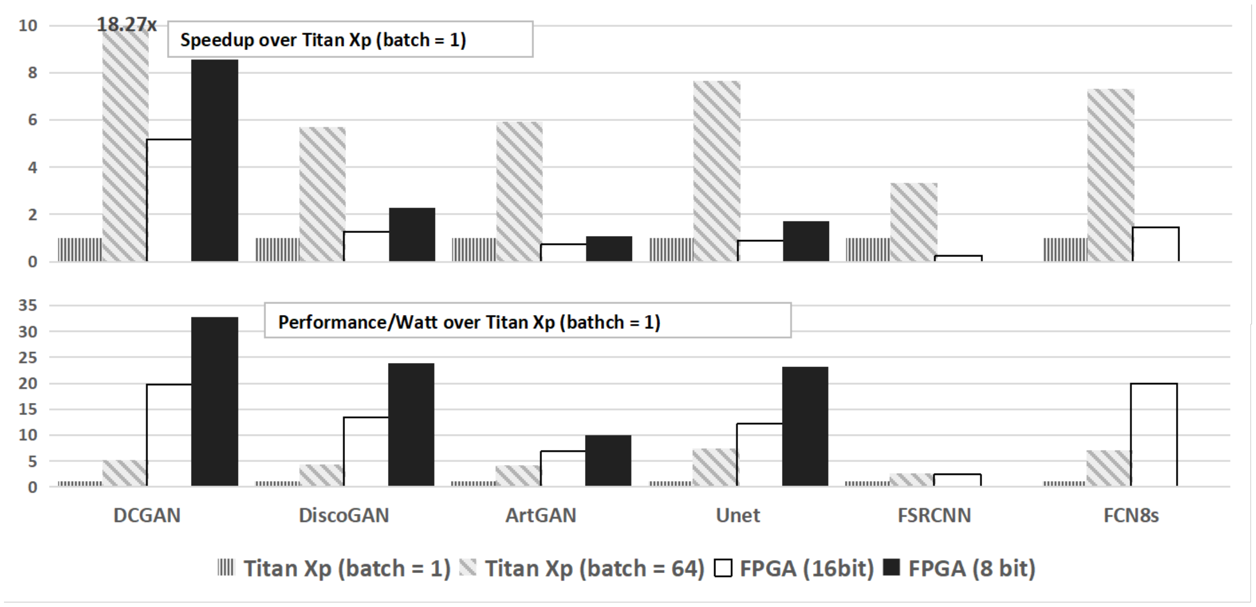


Figure 6.20: *Uni-OPU* vs. GPU baseline (NN-TCONV).

achieved by *Uni-OPU* compared with CONV-FPGA.

Comparison with GPU platform. We also evaluate the performance of speed and power efficiency against the GPU baseline. As shown in Fig. 6.20, *Uni-OPU* is able to gain 1.63× (16*bit*) and 3.41× (8*bit*) speedup on average compared with Titan Xp (batch = 1). As for the power efficiency, the performance of *Uni-OPU* is 2.46× to 19.91× (16*bit*) and 10.04× to 32.73× (8*bit*) better. Moreover, compared with Titan Xp (batch = 64), *Uni-OPU* still performs 2.32× (16*bit*) and 4.33× (8*bit*) better on average for the power efficiency.

6.8 Conclusion

In this chapter, we have presented the first full software/hardware stack solution that utilizes a uniform processor to accelerate both CONV and different types of TCONV layers. In *Uni-OPU* flow, *computation reformulation* and *address constraint extraction* are performed for the reduction of run-time operations and unification of computation patterns. This two-step process reduces over 70% of required operations on average. Moreover, the hardware processor focuses on channel-wise parallelism, making it capable of accelerating networks with a wide range of hyper parameters without FPGA reconfiguration. Evaluation results show that *Uni-OPU* is 1.45× to 3.68× more power efficient compared with state-of-the-art Zero-TCONV accelerators. For NN-TCONV, whose acceleration has not been explored before, *Uni-OPU* obtains 12.43× (16*bit*) and 21.38× (8*bit*) higher power efficiency compared with Titan Xp, with 1.63× and 3.41× lower latency.

CHAPTER 7

Summary

Deep learning is getting employed in our everyday life, hardware accelerator helps improve DCNNs inference performance and enable them to be applied to real-time scenarios. In this thesis we propose a series of work on FPGA acceleration for DCNNs. The development of deep learning algorithm and corresponding hardware acceleration is at a very high speed. The first typical acceleration work only appears at 2015 [ZLS+15b], and in less then five years there are many accelerator designs proposed.

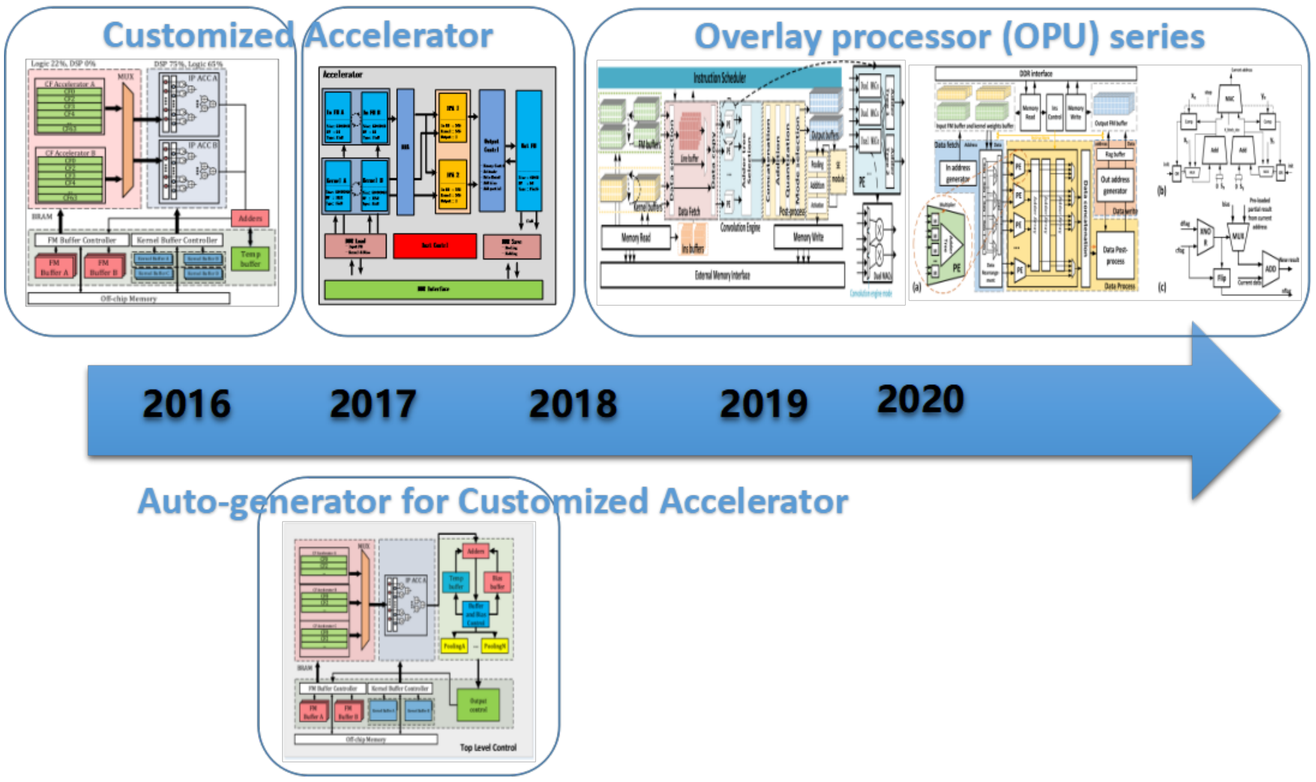

Customized Accelerator
Overlay processor (OPU) series
2016
2017
2018
2019
2020
Auto-generator for Customized Accelerator

Figure 7.1: Timeline of our work

Fig 7.1 briefly explain the timeline of our work. In late 2015 and early 2016 we proposed FPGA based customized DCNN using uniform Convention-Flattening(CF) Accelerators and Inner-Product (IP) accelerators shared across multiple convolutional layers. The uniform CF and IP accelerator maximizes throughput and minimizes the off-chip memory access. Baesd on this framework, we designed customized accelerator for VGGnet and YOLOv2, with different optimization choices and data management module. Specifically, for YOLOv2 our FPGA accelerator was productized and applied to subway X-ray auto-hazard detection facilities.

Starting from late 2016 we worked on the auto-generator for customized accelerator that can generate working HDL code for given CNN accelerator on given FPGA platform, based on our improved architecture template. The parameterized template can conduct convolution with different sizes and strides using similar structure, and the optimization problem was simplified into finding the best IPA length. The experimental results shown that, despite the fact the implementation is generated by compiler, the performance is still comparable to network specific accelerators. For example, our automatically generated Alexnet design on VX485T performs 2.1 times better than the state-of-the-art implementation [MGAG16].

In early 2018 we started our work on the development of the OPU (Overlay processing unit) for the FPGA based DCNN acceleration processor. We develop a set of instructions with granularity optimized for hardware efficiency. OPU is software programmable and is applicable to a wide range of CNNs without hardware re-configuration. OPU of different scales show 1.2× to 5.35× better power efficiency compared with GPU (batch = 1, batch = 16, batch = 64) and other FPGA designs. Moreover, for cascaded CNN networks to detect car license plate, OPU is 2.9x faster compared with edge computing GPU Jetson Tx2 with similar amount of computing resources.

Later we extended our ISA and OPU to cover more newly emerged DCNN architectures. *Light-OPU* was proposed to handle the light-weights DCNNs, which shares the computation engine among conventional convolution layer and depth-wise convolutional layer.The software compiler also conducts network optimization, including layer grouping, operation fusion and operation reordering, eliminates redundant memory access and reduces number of operations in LW-CNN. *Uni-OPU* was proposed for as the first full software/hardware stack solution that utilizes a uniform processor to accelerate both CONV and different types of TCONV layers. In *Uni-OPU* flow, *computation reformulation* and *address constraint extraction* are performed for the reduction of run-time operations and unification of computation patterns. This two-step process reduces over 70% of required operations on average. Moreover, the hardware processor focuses on channel-wise parallelism, making it capable of accelerating networks with a wide range of hyper parameters without FPGA reconfiguration. Evaluation results show that *Uni-OPU* is 1.45× to 3.68× more power efficient compared with state-of-the-art Zero-TCONV accelerators. For NN-TCONV, whose acceleration has not been explored before, *Uni-OPU* obtains 12.43× (16*bit*) and 21.38× (8*bit*) higher power efficiency compared with Titan Xp, with 1.63× and 3.41× lower latency.

In summary, Fig 7.2 depicts different stages of our work on FPGA DCNN acceleration.

At stage A we need human FPGA designer to analyze the given specific network (calculate memory usage, bandwidth requirement, multiplier requirement, etc.) and target FPGA board resources (DSP, Memory, LUT, DDR, PCIE , etc.), then design and implement the accelerator architecture from scratch. Moreover, a complete accelerator system requires more than just the accelerator itself, the designer most also consider the implementation of communication with a CPU to handle the input data acquisition, result output and user GUI. It could take months for to implement a customized FPGA accelerator, while it may have outstanding performance with regard to one specific network, the architecture is hard to be ported for other networks.

Therefore, at stage B we were trying to simply the FPGA accelerator development process. We implemented an end-to-end auto-compiler to generate RTL designs based on predefined RTL library. As shown in Fig. 7.2, we would specify the network configuration and targeted FPGA board and send to auto-compiler as inputs. The auto-compiler would ana-

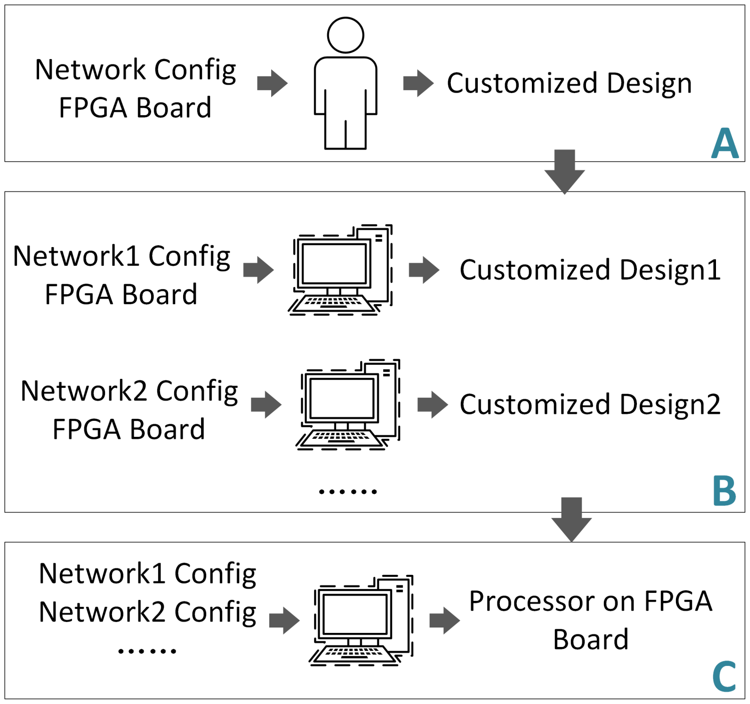


Figure 7.2: Different stages of our work

lyze and optimize design parameters, then generate a complete set of Verilog files to form the accelerator. This simplifies FPGA accelerator design process and avoid tedious implementation step. However, there a still underlying issues in auto-generators. For example, different generation of the FPGA boards have different DDR/PICE, which requires different IPs and interface on the accelerator side. Moreover, the auto-generator cannot guarantee that the generated design will satisfy the timing requirement. If the design cannot achieve desired frequency, we need to lower the frequency and degrade performance. Furthermore, the generated design can only execute one network, which may not suit growing DCNN application scenarios which require several DCNNs to execute in a chain for complex AI task.

In the end at stage C we propose OPU, an overlay processor on FPGA for general DCNN accelerations. From Fig 7.2 it can be seen that we will fix the FPGA board and all the peripherals including CPU/DDR. For different networks, we would use OPU compiler to perform a one-time compilation and generated instruction.bin and weights.bin. Then we would store the instruction and weights files of different networks into DDR connected to FPGA board. During the execution, CPU will send corresponding input image / start signal and collect result for further processing. We also have our own board fabricated as OPU test-platform. Using OPU platform for cascaded CNN networks to detect car license plate, OPU is $2.9x$ faster compared with edge computing GPU Jetson Tx2 with similar amount of computing resources. This project has been productized and applied in a roadside curb-parking system.

[ABC$^+$16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[AHB$^+$18] Mohamed S Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C Ling, et al. Dla: Compiler and fpga overlay for neural network inference acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 411–4117. IEEE, 2018.

[BWL20] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[BZH18] Lin Bai, Yiming Zhao, and Xinming Huang. A cnn accelerator on fpga using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(10):1415–1419, 2018.

[C$^+$15] François Chollet et al. Keras, 2015.

[CEE$^+$16] Patrick Ferdinand Christ, Mohamed Ezzeldin A Elshaer, Florian Ettlinger, Sunil Tatavarty, Marc Bickel, Patrick Bilic, Markus Rempfler, Marco Armbruster, Felix Hofmann, Melvin D'Anastasi, et al. Automatic liver and lesion segmentation in ct using cascaded fully convolutional neural networks and 3d conditional random fields. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 415–423. Springer, 2016.

[Cho17] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[CKK18] Jung-Woo Chang, Keon-Woo Kang, and Suk-Ju Kang. An energy-efficient fpga-based deconvolutional neural networks accelerator for single image super-resolution. *IEEE Transactions on Circuits and Systems for Video Technology*, 2018.

[CMB$^+$10] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. A programmable parallel accelerator for learning and classification. In *PACT*, pages 273–284. ACM, 2010.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[CSJC10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Arch. News*, volume 38, pages 247–257, 2010.

[CSWS17] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7291–7299, 2017.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advance in Neural Info. Processing Systems*, pages 1223–1231, 2012.

[DFC+15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[DLHT16] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence*, 38(2):295–307, 2016.

[DLT16] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In *European conference on computer vision*, pages 391–407. Springer, 2016.

[dsp] Performance and resource utilization for floating-point v7.1. https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[FF10] Li Fei-Fei. Imagenet: crowdsourcing, benchmarking & other cool things. In *CMU VASC Seminar*, 2010.

[FPHL09] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. Cnp: An fpga-based processor for convolutional networks. In *FPL*, pages 32–37. IEEE, 2009.

[Gir15] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[GMAB17] Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. Unsupervised monocular depth estimation with left-right consistency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 270–279, 2017.

[GSQ$^+$18] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[HGDG17] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[HSC$^+$19] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*, 2019.

[HZC$^+$17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[HZRS16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[HZRS16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[IHM$^+$16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[IMK$^+$14] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JAFF16] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*, pages 694–711. Springer, 2016.

[JSD$^+$14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[JYP$^+$17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[KCK$^+$17] Taeksoo Kim, Moonsu Cha, Hyunsoo Kim, Jung Kwon Lee, and Jiwon Kim. Learning to discover cross-domain relations with generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1857–1865. JMLR. org, 2017.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LCL$^+$15] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM, 2015.

[LDS18] Mohammad Loni, Masoud Daneshtalab, and Mikael Sjödin. Adonn: adaptive design of optimized deep neural networks for embedded systems. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 397–404. IEEE, 2018.

[LFN$^+$18] Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-cheung Ng, Yang Chu, and Wayne LUK. Optimizing cnn-based segmentation with deeply customized convolutional and deconvolutional architectures on fpga. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):19, 2018.

[LHC$^+$18] Jun Haeng Lee, Sangwon Ha, Saerom Choi, Won-Jo Lee, and Seungwon Lee. Quantization for rapid deployment of deep neural networks. *arXiv preprint arXiv:1810.05488*, 2018.

[LM13] Steve Leibson and Nick Mehta. Xilinx ultrascale: The next-generation architecture for your next-generation architecture. *Xilinx White Paper WP435*, 2013.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[MCVS17a] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–8. IEEE, 2017.

[MCVS17b] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54. ACM, 2017.

[MGAG16] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 575–580. IEEE, 2016.

[MPT$^+$18] Panagiotis G Mousouliotis, Konstantinos L Panayiotou, Emmanouil G Tsardoulias, Loukas P Petrou, and Andreas L Symeonidis. Expanding a robot's life: Low power object recognition via fpga-based dcnn deployment. In *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4. IEEE, 2018.

[MSC$^+$16] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2016.

[MZZS18] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[ODO16] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 1(10):e3, 2016.

[ORK$^+$15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[PKNP18] Kathirgamaraja Pradeep, Kamalakkannan Kamalavasan, Ratnasegar Natheesan, and Ajith Pasqual. Edgenet: Squeezenet like convolution neural network on embedded fpga. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 81–84. IEEE, 2018.

[PSM$^+$13] Maurice Peemen, Arnaud Setio, Bart Mesman, Henk Corporaal, et al. Memory-centric accelerator design for convolutional neural networks. In *ICCD*, pages 13–19. IEEE, 2013.

[PYV18] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 580–595, 2018.

[QWY$^+$16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[RDS$^+$15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211–252, 2015.

[RF17] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[SCD$^+$16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25. ACM, 2016.

[SCT$^+$16] Wenzhe Shi, Jose Caballero, Lucas Theis, Ferenc Huszar, Andrew Aitken, Christian Ledig, and Zehan Wang. Is the deconvolution layer the same as a convolutional layer? *arXiv preprint arXiv:1609.07009*, 2016.

[SFL$^+$18] Jiang Su, Julian Faraone, Junyi Liu, Yiren Zhao, David B Thomas, Philip HW Leong, and Peter YK Cheung. Redundancy-reduced mobilenet acceleration on reconfigurable logic for imagenet classification. In *International Symposium on Applied Reconfigurable Computing*, pages 16–28. Springer, 2018.

[SFZ$^+$18] Tao Sheng, Chen Feng, Shaojie Zhuo, Xiaopeng Zhang, Liang Shen, and Mickey Aleksic. A quantization-friendly separable convolution for mobilenets. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 14–18. IEEE, 2018.

[SHZ$^+$18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[SLJ$^+$15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR 2015*, 2015.

[SPM$^+$16] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 17. IEEE Press, 2016.

[SVI$^+$16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[SZ14a] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[SZ14b] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TCAT17] Wei Ren Tan, Chee Seng Chan, Hernán E Aguirre, and Kiyoshi Tanaka. Artgan: Artwork synthesis with conditional categorical gans. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3760–3764. IEEE, 2017.

[TM18] Frederick Tung and Greg Mori. Deep neural network compression by in-parallel pruning-quantization. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[VB18] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: mapping regular and irregular convolutional neural networks on fpgas. *IEEE transactions on neural networks and learning systems*, 30(2):326–342, 2018.

[WSWZ19a] Deguang Wang, Junzhong Shen, Mei Wen, and Chunyuan Zhang. An efficient design flow for accelerating complicated-connected cnns on a multi-fpga platform. In *Proceedings of the 48th International Conference on Parallel Processing*, page 98. ACM, 2019.

[WSWZ19b] Deguang Wang, Junzhong Shen, Mei Wen, and Chunyuan Zhang. Towards a uniform architecture for the efficient implementation of 2d and 3d deconvolutional neural networks on fpgas. *arXiv preprint arXiv:1903.02550*, 2019.

[WWJ$^+$19] Peiqi Wang, Dongsheng Wang, Yu Ji, Xinfeng Xie, Haoxuan Song, XuXin Liu, Yongqiang Lyu, and Yuan Xie. Qgan: Quantized generative adversarial networks. *arXiv preprint arXiv:1901.08263*, 2019.

[WYZ$^+$17] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 29. ACM, 2017.

[WZX$^+$16] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in neural information processing systems*, pages 82–90, 2016.

[WZZH17] Huikai Wu, Shuai Zheng, Junge Zhang, and Kaiqi Huang. Gp-gan: Towards realistic high-resolution image blending. *arXiv preprint arXiv:1703.07195*, 2017.

[XLL$^+$17] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 62. ACM, 2017.

[XTW$^+$18] Dawen Xu, Kaijie Tu, Ying Wang, Cheng Liu, Bingsheng He, and Huawei Li. Fcn-engine: accelerating deconvolutional layers in classic cnn processors. In *Proceedings of the International Conference on Computer-Aided Design*, page 22. ACM, 2018.

[YBK$^+$18] Amir Yazdanbakhsh, Michael Brzozowski, Behnam Khaleghi, Soroush Ghodrati, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*, 2018.

[YHW$^+$19] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, et al. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 23–32. ACM, 2019.

[YSKE18] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Ganax: A unified mimd-simd acceleration for generative adversarial networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 650–661. IEEE Press, 2018.

[YWSH19] Yunxuan Yu, Chen Wu, Xiao Shi, and Lei He. Overview of a fpga-based overlay processor. In *2019 China Semiconductor Technology International Conference (CSTIC)*, pages 1–3, March 2019.

[YWZ$^+$19] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. Opu: An fpga-based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[YYT$^+$18] Jiale Yan, Shouyi Yin, Fengbin Tu, Leibo Liu, and Shaojun Wei. Gna: Reconfigurable and efficient architecture for generative network acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2519–2529, 2018.

[YZW$^+$20] Yunxuan Yu, Tiandong Zhao, Mingyu Wang, Kun Wang, and Lei He. Uni-OPU: An FPGA-based uniform accelerator for convolutional and transposed convolutional networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020.

[YZWH20] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. Light-OPU: An FPGA-based overlay processor for lightweight convolutional neural networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 122–132, 2020.

[ZDNKD17] Xinyu Zhang, Srinjoy Das, Ojash Neopane, and Ken Kreutz-Delgado. A design methodology for efficient implementation of deconvolutional neural networks on an fpga. *arXiv preprint arXiv:1705.02583*, 2017.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[ZFZ^{+}16] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 12. ACM, 2016.

[ZLS^{+}15a] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, pages 161–170. ACM, 2015.

[ZLS^{+}15b] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[ZNL18] Ruizhe Zhao, Xinyu Niu, and Wayne Luk. Automatic optimising cnn with depthwise separable convolution on fpga:(abstact only). In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 285–285. ACM, 2018.

[ZPIE17] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

[ZWZ^{+}18] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas. In *Proceedings of the International Conference on Computer-Aided Design*, page 56. ACM, 2018.

[ZXL^{+}17] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5907–5915, 2017.

[ZZLS18] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

APPENDIX A

OPU ISA

We utilize a complex instruction set architecture, where each instruction performs specific task in the compute workload for deep learning applications and CPI(Cycles per instruction) is usually several hundreds. To be more specific, each complex instruction can be decomposed to multiple 32-bit short instructions, where number of short instructions per complex instruction varies.

Short instructions fall into two categories: Conditional instruction (C-type) and Unconditional instruction (U-type). C-type instruction specifies target operations and sets operation trigger conditions. U-type instruction delivers corresponding operation parameters for its paired C-type instruction. One complex instruction consists of one C-type instruction followed by N U-type instructions. One instruction block, which includes all short instructions for one complex instruction, is fetched at one time and then dispatched to corresponding hardware modules.

31	30 6	5 0
immi	parameters	opcode

- immi: as instruction fetch indicator, immi=1 indicates that next instruction should be fetched, while immi=0 indicates the end of instruction fetch for one instruction block.
- opcode: operation code, indicating different specific purpose for the parameters section.

A.1 Instruction Set Listings

Ubiquitous **immi** and **opcode** sections are explained above and will be skipped in this paragraph, which will focus on the parameters section.

31	30 … 21	20	19 … 13	12 … 6	5 … 0
immi	reserved	dw_flag	dma_block_y_size	dma_block_x_size	0

- opcode=0
- U-type
- dw_flag: indicates dw-mode for compute, differences to conventional mode are listed as follows
  - shift line buffers are used for more memory-efficient data fetch, where feature map data reuse is exploited to increase available bandwidth of feature map buffers.
  - two line buffers fetch input feature map blocks simultaneously to input buffer of compute engine so that two input feature map data share one weight data for one decomposed Xilinx DSP48E1.
- dma_block_y_size,dma_block_x_size: height and width, in the context of the first two dimensions of a (H,W,C) tensor, of a data block, which is going to be fetched from DDR to on-chip input feature map buffer in pipeline for one compute round. Since one DDR address refers to 64 8-bit data, the fetched DMA block corresponds to $dma\_block\_y\_size \times dma\_block\_x\_size \times 64$ feature map data.

31	30 26	25 16	15 6	5 0
immi	reserved	fm_in_y_size	fm_in_x_size	1

- opcode=1
- U-type
- fm_in_y_size,fm_in_x_size: original height and width of input feature map. For FC layer, input tensor is reshaped to (H,W,64).

31	30 28	27 17	16 6	5 0
immi	reserved	channel_out	channel_in	2

- opcode=2
- U-type
- channel_out,channel_in: number of output channels and input channels of current layer.

31	30 6	5 0
immi	ddr_fm_addr_ini	3
immi	ddr_ker_addr_ini	4
immi	ddr_bias_addr_ini	5
immi	ddr_res_addr_ini	6

- opcode=3,4,5,6

- U-type
- ddr_fm_addr_ini: starting address for feature maps in DDR address space for current ddr load operation
- ddr_ker_addr_ini: starting address for kernel/weight in DDR address space for current ddr load operation
- ddr_bias_addr_ini: starting address for bias in DDR address space for current ddr load operation
- ddr_res_addr_ini: starting address for feature maps specifically for residual add in DDR address space for current ddr load operation

31	30 21	20 6	5 0
immi	reserved	ddr_fm_read_num	7
immi	reserved	ddr_ker_read_num	8
immi	reserved	ddr_bias_read_num	9
immi	reserved	ddr_save_fm_num	10

- opcode=7,8,9,10
- U-type
- ddr_fm_read_num: number of DDR addresses to be fetched as input feature maps for current compute round. Say the input tensor block is in the shape (H,W,C). $ddr\_fm\_read\_num = H \times W \times \lceil \frac{C}{64} \rceil$.
- ddr_ker_read_num: number of DDR addresses to be fetched as kernel/weight for current compute round.

- ddr_bias_read_num: number of DDR addresses to be fetched as bias for current compute round.
- ddr_save_fm_num: number of DDR addresses to be written to DDR as output feature maps from current compute round.

31	30	29	28 23	22 20	19 11	10 6	5 0
immi	reserved	ddr_load_single	ker_on_board	ddr_load_start_trig	ddr_load_start_dma_num	ddr_load_type	11

- opcode=11
- C-type
- ddr_load_single: the single load of DDR that happens for the extra loading of multiple residual blocks
- ker_on_board: number of kernels on board
- ddr_load_start_trig: trigger condition of DDR load, indicating the precondition to start DDR load. To be specific, if certain trigger condition, configured by this section, is satisfied, next DDR load will start. Preconditions corresponding to each trigger condition index (TCI) are listed as follows
  - 000: whenever last DDR load finishes
  - 001: whenever both last DDR load and DMA finishes
  - 010: whenever layer_start signal arrives (ddr_load is usually the first instruction to execute, therefore the initial layer_start signal can trigger it.)
  - 011: whenever last DDR write finishes
  - 100: ddr_load will not be triggered again (until TCI changes)

  - 101: whenever last DDR_write back to bram finishes and the last ddr_load finishes (situation where DDR_write doesn't write to DDR, but to the on-chip BRAM).
  - 110: whenever last output_control bram write finishes and last ddr_load finishes (situation where no DDR_write is paired with this round of the computation).
  - 111: whenever last DDR_write finishes and last DDR_load finishes.

- ddr_load_start_dma_num: number of DMA rounds that need to be finished before next DDR load can start

- ddr_load_type: one-hot encoding, where each bit indicates loading different data. Bit semantics are as follows, where x stands for arbitrary value. In hardware module, bits are checked starting from least significant bit.

  - $xxx1$: loading input feature maps is included in this DDR load.
  - $xx1x$: loading kernel/weight is included in this DDR load.
  - $x1xx$: loading bias is included in this DDR load.
  - $1xxx$: loading residual feature maps is included in this DDR load. Notice residual feature maps share the same on-chip buffer with input feature map.

31	30 28	27 21	20 17	16 10	9 6	5 0
immi	dma_start_trig	y_max	y_min	x_max	x_min	12

- opcode=12
- C-type
- dma_start_trig: next DMA starts upon the completion of certain preconditions, which are listed as follows

– 000: whenever last DMA finishes

– 001: whenever last DDR load finishes

– 010: whenever last DMA and DDR load finish

– 011: whenever last DDR write finishes, and current TCI for DDR write is not 1 or 6. If current TCI for DDR write is 1, it means DDR write needs to run multiple round in a row, so we cannot start another DMA until all the DDR write finish. If current TCI for DDR write is 6, it means we are in the residual addition mode, current DDR write is waiting for the DDR load to finish so it can start to write, no new DMA should be triggered under this circumstance.

– 100: DMA will not be triggered again (until TCI changes)

– 101: whenever last DDR write finishes

– 111: reserved, currently indicates that DMA will not be triggered again (until TCI changes)

- x_min, x_max, y_min, y_max: DMA read constraints when fetching data from on-chip BRAM.

31	30 25	24 21	20 17	16 14	13 11	10 6	5 0
immi	reserved	ker_y_size	ker_x_size	read_y_stride	read_x_stride	ker_repeat	13

- opcode=13
- U-type
- ker_y_size,ker_x_size: K_h and K_w dimension in the shape of (K_h, K_w, C_i, C_o)
- read_y_stride,read_x_stride: kernel stride in height and width

- ker_repeat: the repeating times of one kernel data (currently $1024 * 8bit$) in DMA. It is utilized in Fully connected mode, where each time the IPA would only require $64 * 8bit$ length of the kernel, therefore after reading one complete row of kernel, we would keep it in the register for 16 cycles and left shift it $64 * 8bit$ every cycle to get the new data.

31	30 18	17 15	14 9	8 6	5 0
immi	reserved	type	ker_round	output_num	14

- opcode=14
- U-type
- type: layer type. Different layer types are listed as follows
  - 0: fully-connected layer
  - 1: convolution, including regular convolution, group convolution and depthwise convolution
  - 2: 3D convolution
  - 3: single pooling layer
  - 4: elementwise scaling in SE(Squeeze-and-Excitation) block
  - 5: zero-based transposed convolution
  - 6: nearest-neighbor-based transposed convolution
- ker_round: The reuse time of one FM data. For example, if for each cycle IPA is able to compute 16 output channels, and current output channel block is 64. Then we would set the ker_round to 4 and use each FM data 4 times to get $16 * 4 = 64$ output channels before we switch to next FM data.

- output_num: number of output channels that can be generated by current IPA mode within one round = 2^{output_num+1}

31	30	29 28	27 23	22 18	17 12	11 6	5 0
immi	reserved	copy_mode	ker_repeat_last	reserved	ker_addr_e	ker_addr_s	15

- opcode=15
- U-type
- copy_mode: indicate how input feature map gets replicated to feed IPA input. Say IPA mode is [64,16], which indicates 64 input channels are to be computed in parallel and 16 output channels are to be computed in parallel. Also assume that two kernel share one DSP with one input feature map data. In such case, input feature map data need 8 replications to fill IPA input buffer. Bit semantics are listed as follows:
  - 0: 8 replications
  - 1: 16 replications
  - 2: 32 replications
  - 3: reserved
- ker_repeat_last: the repeating times of kernel weights for the last row of kernel (In fully connected layer mode)
- ker_addr_e: the end address of current DMA round when reading from the on-chip kernel RAM
- ker_addr_s: the start address of current DMA round when reading from the on-chip kernel RAM

31	30 24	23	22	21	20	19 15	14 10	9 6	5 0
immi	output_channel	output_final_block	final_output	add_temp	add_bias	shift_num	cut_pos	trig_output_start	16

- opcode=16
- C-type
- output_channel: number of output channels of current layer. Since output channels need to be split and computed in multiple round and then get concatenated, this parameter is used to check if all output channels have been computed.
- output_final_block: indicate that this is the last block of current layer, namely the completion of current layer.
- final_output: indicate if this is final result of accumulation of partial sums.
  - *final_output*=0: partial sum result should be written to temporary buffer in 16-bit and would be accumulated in the next round.
  - *final_output*=1: final sum should be truncated to 8-bit and written to DDR. Most significant 8 bits in 16 bits are kept.
- add_temp: indicate if we should accumulate IPA outputs with results in temporary buffer
- add_bias: indicate if we should accumulate IPA outputs with biases in temporary buffer
- shift_num: the number of bits to be left shifted for bias to match decimal position with IPA output
- cut_pos: decimal position when 27-bit IPA output is cut and written to 16-bit temporary buffer after addition with bias/temporary result.

- trig_output_start: indicate the start of post-IPA processing in the output_control hardware module, including accumulation of partial sum, bias addition, results concatenation, data cut to 8-bit for further DDR store, etc.

31	30 26	25 16	15 6	5 0
immi	shift_num_fm	fm_out_y_size	fm_out_x_size	17

- opcode=17
- U-type
- shift_num_fm: the number of bits to be left shifted for IPA output to match decimal position with bias
- fm_out_y_size,fm_out_x_size: height and width of output feature map block

31	30	29 27	26 23	22 20	19 17	16 15	14 11	10 7	6	5 0
immi	ddr_write_choice	padding_size	activation_type	pooling_y_stride	pooling_x_stride	pooling_type	pooling_y_size	pooling_x_size	reserved	18

- opcode=18
- U-type
- ddr_write_choice: Indicates the location of DDR write. 1'b1, write to on-chip BRAM. 1'b0, write to DDR.
- padding_size: post-padding size(must be symmetric for now)
- activation_type: Bit semantics for different activation types are listed as follows
  - 0: no activation

- 1: ReLU
- 2: leaky ReLU
- 3: H-sigmoid
- 4: H-swish
- 5: LUT-based non-linear function
- others: N/A

- pooling_y_stride,pooling_x_stride: stride in height and width dimension for pooling
- pooling_type: Bit semantics for different pooling types are listed as follows
  - 0: max pooling
  - 1: average pooling
  - 2: global average pooling
  - 3: N/A
- pooling_y_size, pooling_x_size: height and width of pooling window

31	30 29	28	27 25	24 18	17 11	10	9	8	7	6	5 0
immi	ddr_save_pos	ddr_save_des	ddr_save_start_trig	block_pool_y_size	block_pool_x_size	residual	upsample_output	activation	pooling	padding	19

- opcode=19
- C-type
- ddr_save_pos: indicate four computation patterns on hardware in the post-IPA stage. Bit semantics are listed as follows
  - 00: data$\longrightarrow$ residual add$\longrightarrow$ activation$\longrightarrow$ pooling

  - 01: data$\longrightarrow$ activation$\longrightarrow$ residual add$\longrightarrow$ pooling
  - 10: data$\longrightarrow$ activation$\longrightarrow$ pooling$\longrightarrow$ residual add
  - 11: data$\longrightarrow$ activation$\longrightarrow$ pooling

- ddr_save_des: write to ofm BRAM or DDR
  - 0(DDR): the final results and padding data, if necessary, of each layer
  - 1(ofm BRAM): for some residual layers, we need to do residual addition multiple times. So the temporary partial sums are stored in ofm BRAM.

- ddr_save_start_trig: next DDR write starts upon the completion of certain preconditions
  - 000: whenever last BRAM write and DDR load finishes
  - 001: whenever last DDR write finishes
  - 010: DDR write not be triggered (until TCI changes)
  - 011: last DDR write back tp on-chip bram finishes and last DDR load finishes
  - 100: whenever last DDR load finishes
  - 101: whenever last DDR write and load finishes
  - 110: whenever last extra single load for residual feature maps finishes
  - 111: reserved, currently indicates DDR write not be triggered (until TCI changes)

- block_pool_y_size, block_pool_x_size: height and width of the block for pooling

- residual: indicate if residual addition is applied in the post-processing in current layer

- upsample_output: indicate if upsampling is applied in current layer, which is realized by writing extra 0s or replications to DDR.

- activation: indicate if activation is applied in the post-processing in current layer

- pooling: indicate if pooling is applied in the post-processing in current layer
- padding: indicate if post-padding is applied in current layer

31	30 6	5 0
immi	fm_output_addr_ini	20
immi	ddr_ins_addr_ini	21

- opcode=20,21
- U-type
- fm_output_addr_ini: base address when write output feature map to DDR for current DDR write
- ddr_ins_addr_ini: starting address of instruction block in DDR for current instruction load

31	30 7	6	5 0
immi	reserved	network_done	22

- opcode=22
- U-type
- network_done: indicate the completion of target model

31	30 20	19 13	12 6	5 0
immi	reserved	output_block_y_size	output_block_x_size	23

- opcode=23
- U-type
- output_block_y_size, output_block_x_size: block size of result before pooling

31	30 20	19 6	5 0
immi	ddr_save_mask[63:39]		24
immi	ddr_save_mask[38:14]		25
immi	reserved	ddr_save_mask[13:0]	26

- opcode=24,25,26
- U-type
- ddr_save_mask: 64-bit mask corresponds to 64 8-bit data when written to DDR

31	30 18	17 15	14 12	11 9	8 6	5 0
immi	reserved	pooling_padding_l	pooling_padding_r	pooling_padding_u	pooling_padding_d	27

- opcode=27
- U-type
- pooling_padding_l, pooling_padding_r, pooling_padding_u, pooling_padding_d: padding to left,right,top,bottom edge in pooling stage

31	30 20	19 13	12 6	5 0
immi	reserved	ddr_load_block_y_size	ddr_load_block_x_size	28

- opcode=28
- U-type
- ddr_load_block_y_size, ddr_load_block_x_size: block size for DDR load. Load address number=$ddr\_load\_block\_y\_size \times ddr\_load\_block\_x\_size$

31	30 28	27 20	19 13	12 6	5 0
immi	reserved	average_pool_para	ddr_save_block_y_size	ddr_save_block_x_size	29

- opcode=29
- U-type
- ddr_save_block_y_size, ddr_save_block_x_size: block size for DDR store.
- average_pool_para:parameter for average pooling multiplication

31	30 29	28	27 21	20 17	16 10	9 6	5 0
immi	reserved	default_flag	out_y_max	out_y_min	out_x_max	out_x_min	30

- opcode=30
- default_flag:
- out_y_max, out_y_min: selective output height range (used in transposed convolution mode)
- out_x_max, out_x_min: selective output width range (used in transposed convolution mode)

31	30 … 20	19 … 13	12 … 6	5 … 0
immi	reserved	out_y_stride	out_x_stride	31

- opcode=31
- out_y_stride, out_x_stride: stride to store output (used in transposed convolution)

31	30 … 20	19 … 17	16 … 14	13 … 6	5 … 0
immi	reserved	SE_stage	SE_fc_in_num	SE_fc_out_num	32

- opcode=32
- SE_stage: We define 3 stages for the computation of SE block. In the first stage we conduct post-process for regular convolution, and collect the result after average pooling as scaling factors. In the second stage we would fc operations for scaling factor computing. In the third stage we would perform the output scaling operation.
- SE_fc_in_num, SE_fc_out_num: Input and output size for the two fc operations.