RESOURCE-EFFICIENT FPGA ACCELERATION FOR MACHINE LEARNING APPLICATIONS THROUGH HLS

BY

XINHENG LIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Deming Chen, Chair
Assistant Professor Jian Huang
Associate Professor Steven Lumetta
Teaching Assistant Professor Zuofu Cheng

ABSTRACT

The rapidly growing machine learning development has demonstrated its great capability and effectiveness in handling complicated real-world problems such as computer vision and natural language processing. However, normal CPU-based implementations cannot deliver sufficient performance for deep neural networks (DNNs) that are used in many machine learning applications due to their intensive computation and memory bandwidth requirements. As a result, application developers seek other hardware platforms to boost up the performance of deep learning workloads. Field programmable gate arrays (FPGAs), famous for their ability to maximize parallelism, flexibility to explore different hardware architectures, and high energy efficiency, have been widely employed to accelerate the DNN applications. Meanwhile, the higher productivity and better design space exploration features of High-Level Synthesis (HLS) have granted this design methodology wider acceptance for hardware design. In recent years, HLS techniques and design flows have also advanced significantly, and many new FPGA designs are developed with the HLS design flow. In this dissertation, we present several novel design methodologies for high-performance and resource-efficient DNN accelerator designs and implementations on FPGAs leveraging commercial HLS design flows. Summarizing the design methodologies explored in these works, we conclude that designing high-performance and resource-efficient FPGA-based DNN accelerators requires both novel architectural design honoring resource and bandwidth constraints and the algorithmic optimization for the DNN computation.

To Professor Chen, for his patience and guidance.
To my friend, for their love and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Recently, the area of deep learning has regained popularity due to various factors, such as affordable high-performance computing, new breakthroughs in deep learning algorithms and a massive amount of credible data for training. Deep learning, a subfield of machine learning, is a class of multi-layer algorithms to extract high-level features from the raw data input. The multi-layer algorithms are typically in the format of neural networks with weight parameters learned from standard training process such as backward propagation. The input entry of deep learning algorithms is usually raw real-world data such as image, text, or sound waveform. Each layer in a specific deep learning algorithm extracts features from its input and forms the output with a higher abstraction level [1]. The features relevant to concept discrimination are enhanced as the algorithm proceeds to the deeper layers, and irrelevant variations are suppressed.

Deep neural networks (DNNs), the major architecture of deep learning algorithms, have been adopted by designers and researchers to deal with problems which were considered as challenging in the past. This class of problems covers many aspects of the real world including image classification [2], automated driving [3], object detection [4], face recognization [5] and natural language processing [6], etc.

The exploration of the DNN model can be traced back to the last century. Early in 1990, Y. Lechun et al. proposed a neural-network-based training algorithm for handwritten digit recognition, which can achieve 99% accuracy [7]. Later on, after 2000, the establishment of public datasets contributes to the emerging of DNN models. In 2009, the famous image classification dataset ImageNet [8] was first presented in the Conference on Computer Vision and Pattern Recognition (CVPR). The dataset contains more than 14 million labeled images and is adopted by many DNN models as the training, testing, and evaluation dataset. In 2012, Krizhevsky et al. proposed the

AlexNet [9] and won the ImageNet large-scale vision recognition challenge (ILSVRC) [10] with a top-5 accuracy of 84.7%. The accuracy of image classification on ImageNet was pushed up to 96.4% [11] by the newly proposed DNN models in the next few years.

FPGAs, well known with both high computation efficiency and low power consumption, are the ideal platforms to implement and accelerate machine-learning applications. Much work has been done on the exploration of suitable FPGA implementations of DNN algorithms. In 2018, Zhang et al. implemented efficient inter-layer pipe-lined architecture to enable the parallelism of execution among different layers [12]. Meanwhile, more and more FPGA designers started to choose High-Level Synthesis (HLS) as a primary design methodology in their hardware development procedure. The automation in HLS design flow sharply reduces the length of the development cycle in hardware design and allows the FPGA designers to explore the efficient and effective implementation for state-of-the-art deep learning algorithms. With the consideration of the development efficiency and the platform customizability, the machine-learning hardware design through HLS for FPGA is an important research area.

1.1 Motivation

DNN algorithms have been proven to be difficult for hardware acceleration. The number of operations required in a typical DNN algorithm is usually at the level of several Giga floating point operations (GFLOP) and the number of data parameters can reach tens of millions. The VGGnet [13] DNN model, for instance, contains 19.6 GFLOPs and 138 million trainable parameters. Meanwhile, the real-time DNN applications usually demand high throughput and low latency solutions to fulfill the time requirement. The fast changing DNN models also increase the difficulty of designing flexible hardware accelerators. Various DNN micro-structures such as residual layers and inception layes have been proposed. Such micro-structures require unique optimization strategies in hardware accelerator design. Although FPGAs are suitable hardware platforms of DNN applications to accommodate the changing requirement of the fast evolving DNN applications, the fixed hardware resources and off-chip data communication bandwidth restrict the performance

of FPGA-based DNN accelerators. Considering the above-mentioned challenges in hardware accelerator design for DNN applications, it is of great interests to explore the methodology of resource-efficient FPGA acceleration for DNN applications.

1.2 Problem Definition and Research Objectives

In this dissertation, we focus on the exploration of various design methodologies to maximize the resource efficiency and performance for FPGA-based DNN accelerators honoring the memory bandwidth and the hardware resource constraints. Specifically, we target to achieve the following research goals:

1. Finding the resource allocation strategy in DNN hardware accelerator design targeting optimal performance.

2. Developing the scheduling algorithm of DNN IPs to minimize the idle time of computation units.

3. Exploring the DNN-specific computation algorithm to reduce the number of computations and increase the computation efficiency.

To achieve the above goals, we explored several efficient architectures and algorithms to accelerate DNNs on FPGA platforms through HLS design flow and the related optimization. Our works are summarized below:

1. We implemented a high-performance deep-learning-based video content recognition application. Meanwhile, we proposed a resource balancing algorithm to allocate proper computation resources to different DNN layers and achieved minimized latency.

2. We developed a fine-grained inter-layer pipeline architecture to enable the overlapping of execution of DNN layer operations.

3. We proposed a high-efficiency Winograd systolic architecture for convolution layers and explored the memory banking optimization in the Winograd systolic architecture.

4. We proposed a novel algorithm that maximizes the computation unit efficiency of the hardware platform in DNN-related applications.

1.3 Thesis Outline

This thesis is organized in the following way. In Chapter 2, we will briefly introduce the background information about the FPGA architecture and the DNN structures. In Chapter 3, we discuss the resource allocation algorithm for FPGA-based DNN applications. In Chapter 4, we discuss the scheduling algorithm for DNN hardware IPs with pipelining. In Chapter 5 and Chapter 6, we explore several algorithmic and architectural optimizations to increase the effective efficiency of computation units. Chapter 7 summarizes the above-mentioned works and draws the conclusion on optimization strategies of FPGA-based DNN accelerators.

CHAPTER 2

BACKGROUND

This chapter provides background information for our proposed works. Firstly, an overview of FPGA with details in architecture and components is demonstrated. Secondly, traditional HLS design methodology is introduced. The third part reviews fundamentals of the major layers in DNNs.

2.1 FPGA Architecture

FPGAs are programmable silicon devices that can be configured into a logic circuit with required functionality. The overall architecture of FPGAs is generally composed of functional blocks, inter-block routing fabric, and input/output pins, as shown in Figure 2.1. The functional blocks usually contain general logic, arithmetic logic, or on-chip memories depending on the type of the block. The routing fabric allows programmable interconnection among the functional blocks. The I/O pins connect the on-chip logic into peripheral hardware such as off-chip DRAM memory, PCIe interface, or host processing system.

The functional blocks for the logic circuit are implemented as Configurable Logic Blocks (CLB), which provide the primary resource for implementing combinatorial circuits and registers. The CLB contains basic logic elements that are connected into slices by the internal route. The main basic logic elements include Look Up Tables (LUT), Multiplexers (MUX), and Flip-Flop registers (FF). The CLBs can be configured as simple combinatorial logic circuits, registers, or small volume memories for required purposes. The arithmetic logic blocks in FPGA are usually instantiated as DSP slices. Typical DSP slices contain fabricated integer multipliers and accumulators, and these DSP slices can perform complex arithmetic operations such as integer/floatpoint multiplication, logarithmic operation, and exponential operations. The

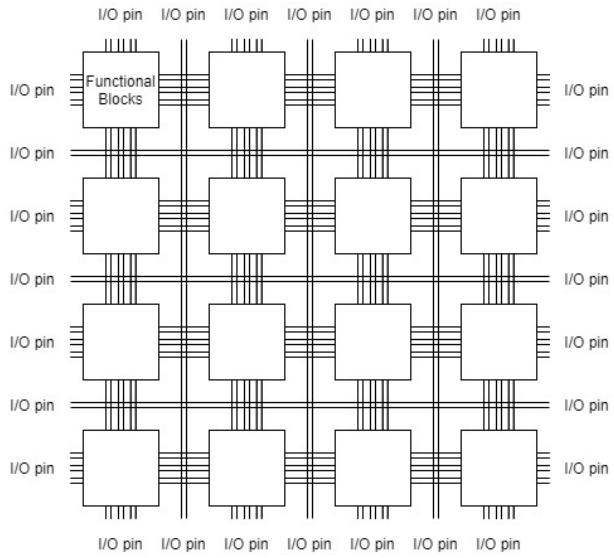


Figure 2.1: FPGA architecture

memory function blocks are Block Random Access Memory (BRAM) slices, which can be configured as on-chip address-access RAMs. Specifically, the BRAMs in typical modern FPGAs can be configured as dual-port RAMs, where data from two addresses can be accessed simultaneously. The pre-fabricated functional blocks usually have different hardware configurations such as register size, BRAM depth, and bit-width according to vendors and FPGA series.

The reconfigurability of FPGA devices entitle them with compelling advantages: on the one hand, compared with the fixed-function Application Specific Integrated Circuit (ASIC) which usually requires long time and high cost for design and fabrication, the FPGA devices can be reconfigured within a short time for many times and the cost can be much lower for suitable applications; on the other hand, compared with CPU, the flexibility of FPGA devices allows the hardware designer to implement efficient task-specific logic circuits which usually have much better performance and much lower power consumption. However, FPGA designers must consider the configuration and limitations of the functional blocks and corresponding resources on FPGA to generate the optimal and efficient implementation of applications.

2.2 HLS Design Flow

High-Level Synthesis (HLS) is the automated process that synthesizes the high-level, untimed behavior specification (C or SystemC for example) into efficient application-specific hardware designs in the format of low-level registration transfer language (RTL) specification which can be implemented in ASICs and FPGAs. The HLS flow usually contains three main steps: Control and Data Flow Graph (CDFG) extraction, Optimization and RTL generation. The CDFG is a directed acyclic graph in which a node can be either an operation node or a control node (such as a branch and loop). Each directed edge in a CDFG represents the transfer of value or control from one node to another. With the control flow and data transfer information embeded in the CDFG, the HLS tool can perform resource and latency optimization and finally generate the RTL hardware design. In the step of CDFG extraction, the source code from high-level language is converted into intermediate representation (IR) and the CDFG is generated based on the IR. The loops in the original source code are often unrolled or auto pipelined to accommodate the resource and latency optimiztion step. HLS will perform three steps of optimization: allocation, scheduling and binding. Allocation specifies the necessary functional unit. Scheduling determines the occurring cycle of each operation while honoring the dependence and timing constraint. Binding maps each operation and variable from the source code to a specific function unit or storage device respectively. In the last step, the RTL code with specified behavior is generated based on the allocation, scheduling and binding result. The datapath is composed by the function units and registers specified by allocation and binding. The control flow is turned into finite state machine specified by the scheduling.

Many previous works have discussed the optimization algorithms. The traditional "As Soon as Possible" (ASAP) [14] scheduling algorithm incrementally assigns each operation instantly when its data dependency is fulfilled. The "As Late as Possible" (ALAP) [14] scheduling works similarly but in a reverse order. The "Force Directed Scheduling" (FDS) [15] is a heuristic aiming at better scheduling solutions by distributing the operations into scheduling steps in a balanced way. The "Integer Liner Programming" (ILP) [16] converts the scheduling problem into an ILP optimization problem and can find solutions with the minimum scheduling latency and minimum re-

quired resources. Also, module libraries can be used for resource allocation. The choice of the optimization and module library have different effects regarding quality of solution thus providing opportunities for domain-specific HLS.

2.3 DNN Structures

2.3.1 Basic Neural Network

The basic building block of a neural network is termed as *neuron*. Figure 2.2 shows the typical structure of a neuron which accepts a group of input signals x_1-x_m and generates the output y_k. The neuron has different levels of sensitivity from each input signal x_j which are characterized by the parameters termed as *weight* (denoted by w_{kj} in the figure). The weighted signals are combined linearly in a summing junction and with one additional *bias* parameter. The activation function φ limits the neuron's output within a finite range. In summary, a neuron k may be described using the following equation:

$$y_k = \varphi(\sum_{j=1}^{m} w_{kj}x_j + b_k) \tag{2.1}$$

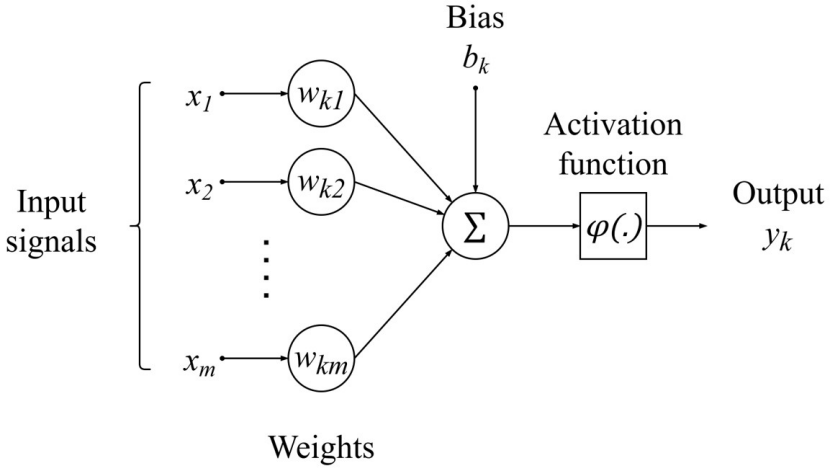


Figure 2.2: Nonlinear model of a neuron

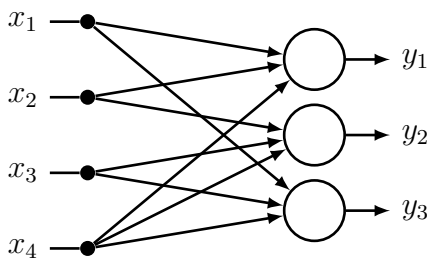


Figure 2.3: A feed forward-layer with neurons

In the DNN structure, the neurons are organized in the form of *layers*. Figure 2.3 shows the structure of a neuron layer. All the layer neurons use the same group of input signals, and the output signals of the neurons in the layer form an output signal group. The input and output group may be organized as tensors and neurons may have different patterns of selection of inputs, forming layers with different functionalities, such as convolution layer and fully connected layer.

A DNN model consists of a sequence of neuron layers with different functionalities. For example, the structure of AlexNet is show in Figure 2.4. The model of AlexNet includes five convolution layers, three max-pooling layers and three fully connected layers. The input and output of a layer are usually multidimensional tensors. Specifically, the input and output of convolutional neural networks are usually tensors of three dimensions of channels (or depth), height and weight. We provide brief introductions to major layer types in DNN models next.

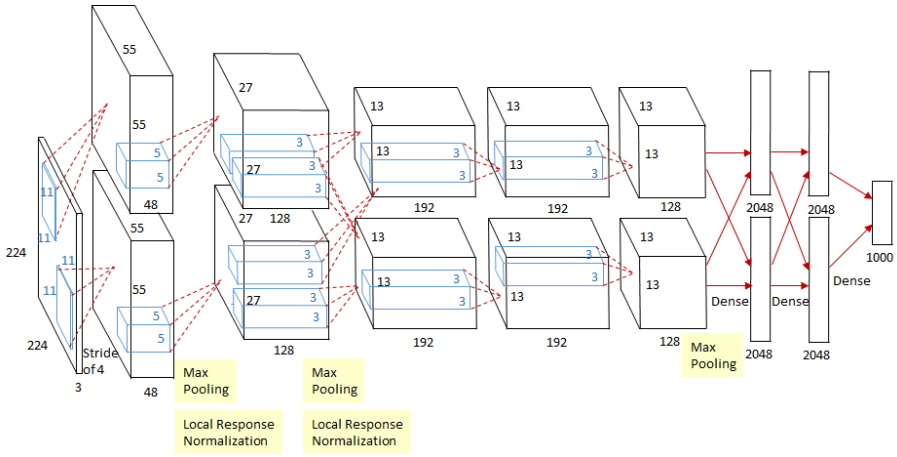


Figure 2.4: AlexNet structure [2]

2.3.2 Activation Layer

The activation function φ in the neurons is usually a monotonous differentiable function which limits the output signals into a certain value range and may form an independent layer. Equation 2.2 shows several commonly used activation functions. One of the function types is the sigmoid function. The sigmoid function limits the output into the range of $(0, 1)$ and is used for DNN models with probability prediction as the output. The hyperbolic tangent function, or tanh for short, is a nonlinear activation function that outputs values between -1.0 and 1.0 and is mainly used in classification between two classes. The tanh function was preferred over the sigmoid activation function previously, as models that used it were easier to train and often had better predictive performance. However, both the sigmoid and tanh function require the exponential operation which is computationally expensive. Also, the gradient of sigmoid and tanh functions lies among $(0, 1]$ and the gradient decreases exponentially as the neural network becomes deeper, which causes the low convergence rate with low gradient error feedback in the back propagation training process. The Rectified Linear Unit (ReLU) activation function has a much simpler form with only comparison and linear operations which is far more hardware friendly compared with *sigmoid* and *tanh* functions. The ReLU activation function also suffers less from the vanishing gradient problem [17]. The gradients of the ReLU activation function at the activation region (positive input interval) become constant and do not vanish as the network goes deeper. The ReLU activation function is commonly used in modern DNN models.

$$\begin{aligned} \text{sigmoid: } f(x) &= \frac{1}{1+e^{-x}} \\ \text{hyperbolic tangent: } f(x) &= \frac{e^{2x}-1}{e^{2x}+1} \\ \text{ReLU: } f(x) &= max(x, 0) \end{aligned} \tag{2.2}$$

2.3.3 Fully Connected Layer

If all the neurons in a neural network layer are connected to all the input signals, then the layer is termed as a *fully connected layer*. Denoting the weight from input x_i to output y_j as w_{ij} with the assumption that there

are m input signals and n output signals, we may represent all the weight parameters as an $m \times n$ matrix $W^{m \times n}$. According to Equation 2.1, the fully connected layer may be expressed as Equation 2.3 where $\vec{x}$, $\vec{b}$ and $\vec{y}$ denote the input signals, neuron bias and output signals respectively. As mentioned before, the activation function usually forms an independent activation layer, leaving the fully connected layer in the form of matrix-vector multiplication together with the bias vector summation shown in Equation 2.4. Fully connected layers are usually appended at the end of typical DNN models to extract the final classification scores.

$$\vec{y} = \varphi(W\vec{x} + \vec{b}) \tag{2.3}$$

$$\vec{y} = W\vec{x} + \vec{b} \tag{2.4}$$

2.3.4 Convolution Layer

DNN application uses convolution layers to deal with pattern recognition and classification problems. The input and output signals are organized as feature-maps which are 3D tensors that can be viewed as many channels of 2D arrays. The convolution layer convolves the input feature-maps with the weight parameters to generate the output feature-map. The weight parameters are organized as a sequence of $K \times K$ filters.

Following the conventional notation of tensors, we represent an N-dimension tensor t as $t^{X_1 \times X_2 .. X_N}$ with the n-th dimension size equal to X_n. A certain element with dimension index $x_1, x_2 ... x_N$ in the tensor is denoted by $t_{x_1, x_2 ... x_N}$. Then we use the following symbols to represent the involved tensors in a convolution layer.

- $x^{C_i \times H_i \times W_i}$ for input signal tensor with C_i channels of $H_i \times W_i$ feature-maps.
- $y^{C_o \times H_o \times W_o}$ for output signal tensor with C_o channels of $H_o \times W_o$ feature-maps.
- $WT^{C_o \times C_i \times K \times K}$ for weight tensor with filter size $K \times K$.
- b^{C_o} for bias tensor.

And the computation of the output signal tensor is specified in Equation 2.5.

$$y_{c_o,h,w}=\sum_{c_i=0}^{C_i\text{-}1}\sum_{k_h=0}^{K\text{-}1}\sum_{k_w=0}^{K\text{-}1} x_{c_i,h+k_h,w+k_w}\cdot WT_{c_o,c_i,k_h,k_w}+b_{c_o} \tag{2.5}$$

During the convolution, each channel of the input feature-map is convolved by one of the $k \times k$ filters, and all the filtering results are summed up in-place to form one channel of the output feature-map. The convolution layer aims to extract and augment specific feature patterns represented by the filters in the weight parameters.

2.3.5 Pooling Layer

The pooling layer computes the maximum or average value of the region covered by a sliding window on each channel of the input feature-map to generate a output feature-map. The pooling layers down sample the feature-maps by summarizing the regional information in the feature-map so that the convolution kernel may cover a larger region of feature information in deeper layers after each pooling. DNN models include pooling layers at periodic positions to reduce the size of the feature-map and thereby to reduce the number of computations and weight parameters.

CHAPTER 3

HLS-BASED OPTIMIZATION FOR VIDEO CONTENT ANALYSIS ACCELERATOR[1]

FPGA is a promising candidate for the acceleration of Deep Neural Networks (DNN) with improved latency and energy consumption compared to CPU-based and GPU-based implementations. DNNs use sequences of layers of regular computation that are well suited for HLS-based design for FPGA which features convenient and configurable IP-based design flow. However, accelerating large neural networks under resource constraints is still a key challenge. HLS must manage on-chip computation, buffering resources, and off-chip memory accesses to minimize the total latency. In this chapter, we present a design framework for DNNs that uses highly configurable IPs for neural network layers together with a new design space exploration engine for Resource Allocation Management (REALM). We also carry out efficient memory subsystem design and fixed-point weight re-training to further improve our FPGA solution. We demonstrate our design framework on the Longterm Recurrent Convolution Network for video inputs. We are the first to implement the LRCN DNN accelerator on the FPGA platform. Our implementation on a Xilinx VC709 board achieves 3.1× speedup compared to the NVIDIA K80 and 4.75× speedup compared to the Intel Xeon with 17.5× lower energy per image.

3.1 Introduction

Recent years have seen rapid development of DNNs deployed on FPGAs [18, 19, 20, 21, 22, 23, 24]. DNNs are composed of layers of regular computations such as convolution and pooling. High-level synthesis (HLS) is well suited to optimize the regular computations of network layers. How-

[1] This is a joint work with equal contribution by co-author Xinheng Liu and Xiaofan Zhang.

ever, there are significant challenges in managing computational complexity, on-chip memory limitation, and external memory bottlenecks. Each layer in a DNN features different computational and memory bandwidth demand; effective design of a network demands both different optimization strategies based on layer type as well as different optimization parameters between different instances of the same layer. To produce optimal network implementations under resource constraints, we must determine best on-chip memory usage and external memory access patterns, explore layer implementation options and determine how to best allocate limited FPGA resources among the layers in order to minimize overall latency. In this chapter, we develop the Resource Allocation Management (REALM) framework to analyze resource requirement and perform resource allocation among the layers in order to minimize total network latency. We demonstrate our framework using the Long-term Recurrent Convolutional Network (LRCN) [25]. LRCN is among the most complex tools available today aiming to achieve cognitive intelligence in the context of video/image analysis. To summarize, the main features of this chapter are:

1. A flexible HLS IP for designing Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) optimally for a range of IP parameterizations. We use instances of this IP to implement the LRCN.

2. A resource partitioning solution that provides guidelines for resource allocation per layer for minimum overall latency. We call this solution, REALM.

3. An implementation of LRCN on the Xilinx VC709 board. We demonstrate the effectiveness of REALM and our HLS IP in the design process and achieve better performance than GPU and CPU solutions.

4. We demonstrate an efficient implementation of LRCN on the Xilinx Virtex-7 VC709 evaluation platform and achieve significant acceleration. Also, comprehensive comparison study among FPGA, CPU, and GPU is provided.

The rest of this chapter is organized as follows. In Section 3.2, the LRCN is briefly introduced, and existing FPGA-based acceleration schemes for DNNs are discussed. In Section 3.3, the main challenges encountered in mapping

the LRCN to the FPGA are presented. Section 3.4 describes our REALM framework, the HLS IP, and additional techniques for optimizing the design. The overall system implementation and comparison study are presented in Section 3.5. In Section 3.6, we conclude this chapter.

3.2 Background

3.2.1 LRCN

A typical LRCN is implemented using AlexNet [9] for CNN and LSTM [26] for RNN. A LRCN uses 2.22 billion floating-point operations and 86.56M synapse weights for processing just one video frame. Video frames are entered sequentially into the system and first processed by CNN (the left side in Figure 3.1) for the extraction of visual features and the output is a vector with 1000 dimensions with each dimension representing a category of objects. This vector is passed to the RNN module (the right side in Figure 3.1) to generate proper descriptions (RNN produces a separate word in each iteration). By combining these different neural networks together, LRCN becomes an end-to-end model for video content description.

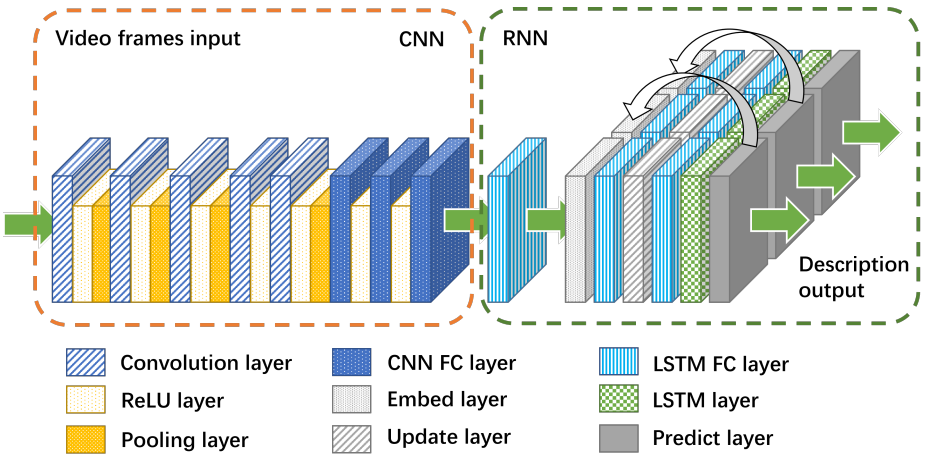


Figure 3.1: LRCN structure

3.2.2 Related Work

FPGA-based acceleration has achieved very high performance for DNNs. The work presented in [18] explores the design space of loop optimizations in a CNN implementation. In [19], an efficient design is presented with a weight quantization method. An OpenCL-based design method and an associated design-space exploration for system-level throughput optimization is carried out in [20] to produce a CNN implementation. Memory access times are considered in [21] to achieve comprehensive optimization goals. The paper [22] exhaustively analyzes loop optimizations and data movement patterns in CNN loops. An FPGA accelerator for LSTM is designed in [23] which explores both computation and communication optimizations. In [24], LSTM model compression and an associated accelerator design are presented and the results surpass CPU and GPU solutions. Compared to these methods, we design a parameterized HLS IP for implementing neural networks and we introduce a resource allocation strategy called REALM, for achieving minimizing overall latency.

3.3 Design Challenges

The memory space and computational complexity of LRCN are very high. Table 3.1 summarizes the detailed requirement with CNN (AlexNet) and 15 iterations of RNN representing 15 output words. In total, 2.22 Giga (billion) floating-point operations are necessary during inference while 411970 inputs are distributed to different layers and 659290 outputs are generated. The video description requires 86.56 million weight parameters which occupy 346.24MB of the memory. Layers in LRCN show different characteristics regarding computation and memory requirements. The computational demand of convolutional layers, fully connected layers and RNN are respectively 60.06%, 5.29% and 34.65% while the memory space requirements are 2.69%, 67.73% and 29.58% respectively. The complex structure and large variation in the computation and communication characteristics of different layers in LRCN present the following challenges.

1. **Resource allocation and partitioning.** Currently, HLS relies heavily on pragmas. Although using HLS pragmas can improve the perfor-

mance of loops of DNN layers, it is not straightforward to relate such HLS pragmas directly to the performance because of the possible data dependencies. Second, LRCN consists of multiple loop structures across different layers, a homogeneous resource allocation for these layers will not produce the best results.

2. **Memory limitation.** The large size of the LRCN weight data forces us to use external memory to store these weights. The frequent access of external memory easily becomes a bottleneck, which means that the performance of a critical loop is not affected so much by its computation demand and resource allocation, but by how frequently it needs to access weights from the memory. While modeling the performance, it is much simpler if this memory bottleneck can be dealt with so as to reduce its impact on the system's performance. In this chapter, we explore techniques in order to improve memory performance, simplify the overall performance model, and enable an effective resource allocation scheme for minimizing overall latency.

Table 3.1: Space and computation complexity of LRCN

Layers	# of Weights	GFLOP	# of Input data	# of Output data
Conv1	0.03 M	0.21	150.53 K	290.40 K
Conv2	0.31 M	0.45	69.98 K	186.62 K
Conv3	0.89 M	0.30	43.26 K	64.90 K
Conv4	0.66 M	0.22	64.90 K	64.90 K
Conv5	0.44 M	0.15	64.90 K	43.26 K
FC1	37.76 M	0.08	9.22 K	4.10 K
FC2	16.79 M	0.03	4.10 K	4.10 K
FC3	4.10 M	0.01	4.10 K	1.00 v
RNN 15 iterations	25.61 M	0.77	1.00 K	0.015 K
Total	86.56 M	2.22	411.99 K	659.29 K

Algorithm 1 Loop structure for convolution

```
1: for h → OutHeight, h++ do
2:     for w → OutWidth, w++ do
3:         for C_o → OutChannel, C_o++ do
4:             for C_i → InChannel, C_i++ do
5:                 for i → KernelHeight, i++ do
6:                     for j → KernelWidth, j++ do
7:                         Out[C_o][h][w]+=
8:                         weight[C_o][C_i] × In[C_i][h+i][w+j]
9:                     end for
10:                end for
11:            end for
12:        end for
13:    end for
14: end for
```

3.4 Design Methodology

3.4.1 IP for LRCN Design

The IP-based design methodology provides the opportunity to quickly implement a high-quality FPGA design with customized IPs. In order to leverage its benefits, the proposed HLS IP covers the most critical and universal operations (multiply-accumulations) in DNNs. With this parameterized IP, we can fit it into the LRCN implementation.

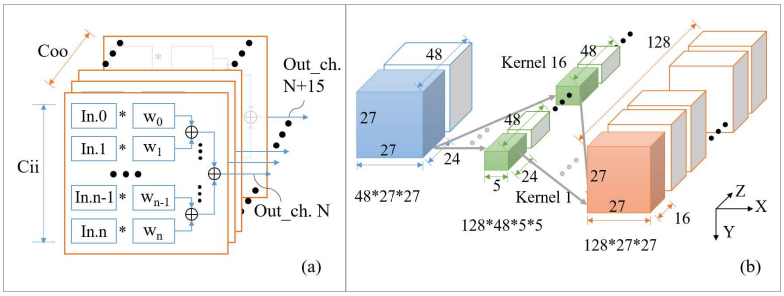


Figure 3.2: (a) The parameterizable HLS IP design and (b) computation in blue, green, and orange blocks carried out by a single IP

The HLS flow automatically turns high-level language code describing the behavior of the target hardware design into hardware descriptive language code. In this work we adopt the Xilinx's Vivado HLS flow which accepts C++ code as the input. Since the optimization strategy of FPGA-based design differs greatly from the one of CPU-based design, it is more efficient to start from the original code instead of the code specifically optimized

for CPU. Algorithm 1 shows the pseudo-code for the convolution algorithm without any optimization. Although the HLS tool is able to perform elegant register-level resource allocation and operation scheduling for the code, the designer still needs to provide guiding information such as pragmas and loop reconstruction to achieve the desired hardware architecture. The original convolution loop structures in Algorithm 1 are inefficient for the FPGA device considering the on-chip resource constraint and off-chip bandwidth limit. For the purpose of hardware efficiency, we perform the following optimization for the original code:

1. We tile the loop 3 and loop 4 in Algorithm 1 with configurable factor C_{oo} and C_{ii}. In the critical loops representing the LRCN layers, we moved the loop iterations with minimal dependency inwards, so that the inner loops in the transformed source code may be unrolled for maximum parallelization and resource utilization.

2. We instantiate double buffers by declaring ping-pong arrays and use pragmas to guide the HLS tool to sytheisize the double buffer architecture. In such a way, the computation and data loading processes run concurrently.

We abstracted the optimized loop structure shown in Algorithm 1 as an HLS IP, and use it to construct the network. As shown in Figure 3.2a, the IP consists of C_{oo} multiply accumulate units of dimension C_{ii} each. It can represent a 2D, unrolled, loop tile of multiply-accumulate operations. The proposed IP source code is shown in Algorithm 2. Figure 3.2 helps visualize how a complete convolutional layer is built using the IP. One blue block and sixteen green blocks are processed by the IP which generates partial sum of the orange block (one eighth of the layer's output). In Figure 3.2b, $C_{ii} = 24$ and $C_{oo} = 16$, and this tile is reused $27 \times 27 \times 8 \times 2$ times to obtain all the outputs of the layer. The optimized code following Algorithm 2 is fed into the HLS flow for automated hardware IP generation and FPGA implementation.

3.4.2 Resource Allocation for Minimal Latency

To design a resource allocation strategy targeting minimal latency, we propose a latency model with the following assumptions:

Algorithm 2 Pseudocode of proposed IP

```
1: for C_i → InChannel, C_i+=C_ii do
2:     for C_o → OutChannel, C_o+=C_oo do
3:         for i → KernelHeight, i++ do
4:             for j → KernelWidth, j++ do
5:                 #pragma HLS dataflow //Load from ping-pong buffer
6:                 Load_Data_Func()
7:                 for h → OutHeight, h++ do
8:                     for w → OutWidth, w++ do
9:                         #pragma HLS pipeline // HLS IP starts below
10:                        for coo → C_00, coo++ do // Output traversal
11:                            for Sel → 1, 2 do //Select ping-pong buffer
12:                                for cii → C_ii, cii++ do // Input traversal
13:                                    Out[Sel][C_o+coo][h][w]+=
14:                                    weight[Sel][coo][cii] × In[SelBuf][C_i+cii][h+i][w+j]
15:                                end for
16:                            end for
17:                        end for
18:                    end for
19:                end for
20:            end for
21:        end for
22:    end for
23: end for
```

1. Each layer in the DNN has a fixed computation amount C_i.

2. The proposed accelerator architecture instantiates a hardware IP for each layer with allocated resource R_i. And the total computation resource is constrained by on-board computation resource R_{total}.

3. The latency of each layer is proportional to C_i/R_i, the ratio between the layer computation amount and the allocated resource of the layer IP.

4. All the IPs perform layer computation in serial and the total latency is the sum of the latency of each layer.

With the above assumptions, we may conclude the proper choice of resource for each layer to achieve minimal latency. Equation 3.1 and Equation 3.2 demonstrate the calculation for latency and resource usage for the design with α as a constant obtained empirically.

$$latency = \alpha \sum_i \frac{C_i}{R_i} \tag{3.1}$$

$$R_{total} = \sum_i R_i \tag{3.2}$$

According to Cauchy inequality, Equation 3.3 holds.

$$\begin{aligned}
\left[\sum_i (\sqrt{\frac{C_i}{R_i}})^2\right]\left[\sum_i (\sqrt{R_i})^2\right] &\geq \left[\sum_i \sqrt{C_i}\right]^2 \\
\left[\sum_i \frac{C_i}{R_i}\right]\left[\sum_i R_i\right] &\geq \left[\sum_i \sqrt{C_i}\right]^2 \\
\left[\sum_i \frac{C_i}{R_i}\right] &\geq \frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i} \\
\alpha\left[\sum_i \frac{C_i}{R_i}\right] &\geq \alpha\frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i} \\
latency &\geq \alpha\frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i}
\end{aligned} \tag{3.3}$$

Also, the Cauchy inequality achieves equality under the condition specified in Equation 3.4 which indicates that the minimal latency is achieved.

$$\textbf{REALM: } \frac{R_i}{R_j} = \frac{\sqrt{C_i}}{\sqrt{C_j}} \tag{3.4}$$

Equation 3.4 is hence the essence of REALM (Resource Allocation Management), which can be used to budget resources among different layers in the network. Once we obtain the ratio of resource allocation per layer from REALM, we can set the tile-sizes of the HLS IP appropriately to reflect this ratio and reach minimum latency. Table 3.2 shows the allocated DSP for convolution layers in our LRCN application guided by REALM with R_{total} set as 20%, 40% and 60% of total on-chip DSP resources.

3.4.3 Network Pruning and Quantization

One of the conditions that can invalidate the assumptions behind REALM is that memory access time dominates the layer latency instead of computations. This can happen in the case of FC layers and LSTM which have a very low computation-to-communication ratio. Second, FPGAs perform favorably with fixed-point DSP units but not as well when using floating-point

Table 3.2: DSP allocation scheme generated by REALM

Layer	Operations	20% DSP#	40% DSP#	60% DSP#
Conv1	105415200	130	261	391
Conv2	223948800	190	380	570
Conv3	149520384	155	311	466
Conv4	112140288	134	269	403
Conv5	74760192	110	220	329
Total#	665784864	720	1440	2160

data.

To address these concerns, we prune the original LRCN network to reduce the number of output nodes in the fully connected layers, FC1 and FC2, from 4096 to 256. In addition, two LSTM layers with 1000 hidden units each are converted to one LSTM layer with 256 hidden units. We then change the weights, bias and intermediate data to use fixed point numbers (shown in Table 3.3) to improve DSP utilization and reduce the memory bandwidth pressure. We re-train the modified LRCN network using Caffe for maintaining accuracy. The accuracies of the networks are summarized in Table 3.4. After pruning and quantization, the LRCN network occupies 11.08 million weight and requires 1.45 G operations. These updated numbers are used for setting up REALM.

Table 3.3: Layer-wise bit-width quantization

Layers	Output and intermediate data (total bits, frac. bits)	Weight and bias data (total bits, frac. bits)
Conv1	16, 4	12, 11
Conv2	16, 7	12, 11
Conv3	16, 8	12, 11
Conv4	16, 9	12, 11
Conv5	16, 10	12, 11
FC1-FC3	16, 11	12, 11
RNN	16, 12	12, 11

Table 3.4: Accuracy after re-training

Network	Accuracy
LRCN - original (AlexNet + 2 LSTM layers)	43.0%
LRCN pruned (AlexNet + 1 LSTM layer)	41.8%
LRCN pruned, fixed-point (AlexNet + 1 LSTM layer) *Implemented on FPGA*	42.0%

3.4.4 Memory Management

In addition to network pruning and quantization, we take further action to maintain the validity of the assumptions under REALM by reducing the impact of communication costs. Using a 12-bit format means that for a bus-width of 512 bits (this applies also to other bus-widths which are usually power-of-2), a few bits in each access may have to be discarded (512 is not divisible by 12). To prevent this, we collect bits from three bus accesses for regrouping into weights. The scheme is shown in Figure 3.3. To further improve the memory access efficiency, we need to ensure that the data access patterns exploit the maximum DDR memory bandwidth. We re-order the multidimensional weight data into a linear sequence that follows the order of computation. This ensures that data locality is exploited when the HLS IP instance accesses weight data thus to improve the throughput of access. With such a weight loading interface, the hardware accelerator is able to fetch 128 12-bit weight parameters in every three cycles without additional buffers inside the computation IP. To further reduce the memory access latency, we instantiate FIFOs outside the LRCN layers in the path connecting the layers to external memory. In addition, we use Vivado HLS to synthesize ping-pong buffers at the input of each layer. This is intended to hide memory latency between the layer's computational unit and the FIFO feeding the layer.

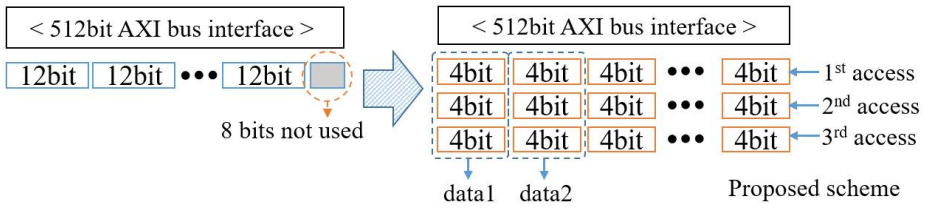


Figure 3.3: Memory-bus organization for minimizing wastage

3.5 Implementation and Comparison

3.5.1 Overall System Setup

Xilinx Virtex-7 VC709 evaluation platform with XC7VX690T FPGA is used for our design with optimizations mentioned in Section 3.4 , and Vivado HLS 2016.2 is used for high-level synthesis.

We build an end-to-end, real-time, video content description system that can directly process frames from a commercial webcam. We use Tegra TK1 and a Logitech C920 full-HD webcam as the front-end. We down-sample the captured frames to the size that fits the LRCN network and stream the image frame over the internet. On the backend side, the host PC receives the frames and pre-process the image, including re-ordering of pixels and fixed-point conversion, before off-loading the data to our LRCN kernel implemented on the FPGA. After computation, the kernel sends back an index vector which is used for dictionary lookup to produce a sentence. The complete system is shown in Figure 3.4.

Figure 3.4: Left: Front-end (Tegra TK1 and webcam); Right: Back-end (Xilinx VC709 FPGA board, host PC)

3.5.2 Comparison

Resource consumption of proposed LRCN is shown in Table 3.5, and the maximum frequency is 100 MHz in our board level implementation. We compared the performance of the pruned LRCN model on CPU, GPU, and FPGA. The floating-point version of the pruned LRCN was run on GPU and CPU and the fixed-point version of the pruned LRCN was run on the FPGA.

Since there is no previous works that recorded the performance of the LRCN application on GPU or CPU platform, we implemented LRCN on GPU and CPU under the Caffe [27] framework which provides optimized DNN libraries for both CPU (based on BLAS [28]) and GPU (based on cuDNN [29]) as our comparison target. We map the GPU solution into two Nvidia K80 graphic cards and the CPU solution into an Intel XeonE5-2630 processor.

Table 3.5: Resource consumption

BRAM	DSP	Flip-flop	LUT
1508	3130	321165	316250
51%	87%	37%	73%

The performance and power comparisons are provided in Table 3.6. For the FPGA version, a power meter was used to measure the consumption of the entire evaluation board during the execution of the kernel. For the GPU version, power was measured using the command nvidia-smi, and for the CPU version, power was measured using a power meter.

Table 3.6: LRCN performance comparison

	Frequency	Latency	Speedup	Power	Efficiency
This work	100 MHz	0.040 s	4.75×	23.6 W	0.94 J/frame
Nvidia K80	562 MHz	0.124 s	1.53×	133 W	16.49 J/frame
Intel Xeon E5-2630	2.6 GHz	0.19 s	1.00×	88 W	16.72 J/frame

3.6 Conclusion

In this chapter, we presented an implementation of LRCN and explored the methodology in HLS-based design flow for FPGA. We introduce a resource allocation strategy called REALM, which drove theoretical guidelines for per-layer resource allocation for minimum overall latency. We implemented methods including network pruning, weight quantization, and retraining, as well as efficient memory system design. Using our resource allocation guidelines, we tuned the parameters of the proposed HLS IP instances to implement the LRCN to obtain a design whose power and latency performance surpassed those of GPU and CPU implementations. Our implementation

proves to be efficient in LRCN-related applications which require low power consumption and high FPS throughput.

CHAPTER 4

RESOURCE AND DATA OPTIMIZATION FOR HARDWARE IMPLEMENTATION OF DEEP NEURAL NETWORKS TARGETING FPGA-BASED EDGE DEVICES

Targeting convolutional neural networks (CNNs), we adopt the high-level synthesis (HLS) design methodology and explore various optimization and synthesis techniques to optimize design on an FPGA. Our motivation is to target embedded devices that operate as edge devices. Recently, as machine learning algorithms have become more practical, there have been much effort to implement them on devices that can be used in our daily lives. However, unlike server devices, edge devices are relatively small and thus have much more limited resources and performance. Therefore, control of resource usage and optimization plays an important role when we implement machine learning algorithms on an edge device. The key idea explored in this chapter is backward pipeline scheduling which optimizes the pipeline between CNN layers. This optimization technique is especially useful to utilize the limited on-chip memory resource for classifying an image on an edge device. We have achieved latency of 175.7 μs for classifying one image in the MNIST dataset using the LeNet and 653.5 μs for classifying one image in the Cifar-10 dataset using the CifarNet. For the LeNet we were able to maintain high accuracy of 97.6% for the MNIST dataset and 83.4% for the Cifar-10 dataset. We achieved the best single-image latency, 5.2× faster for the LeNet and 1.95× faster for the CifarNet, compared with NVIDIA Jetson TX1.

4.1 Introduction

In recent years, we see the booming of deep convolutional neural networks in solving artificial intelligence tasks. Some of these deep learning methods have surpassed human-level performance and enabled new applications, such as machine translation, AI medical diagnosis, and autonomous driving. In

order to deliver machine intelligence to more people, we need to find ways to deploy such well-trained highly accurate deep learning models to Internet of Things (IoT) devices, which require edge computing platforms. Edge devices usually denote mobile or embedded systems, including phones, drones, security cameras, or any other computing or sensing devices that connect to a network and transfer data. These devices have tight energy/thermal constraints and offer limited hardware resources/computing power, but are often required to accomplish latency-critical tasks such as object detection tracking for unmanned vehicles, facial recognition for security cameras, and control mechanism for smart manufacturing.

Advances in high-level synthesis (HLS) during the last decade have led to its increased adoption as a primary design methodology. HLS offers important advantages in higher design productivity, better design space exploration, friendly debugging of high-level specifications, and automation of test generation infrastructure. There are many active academic and commercial HLS projects and tools that continue to improve both design quality and productivity [30, 31, 32, 33, 34, 35]. Due to HLS, practical applications are embedable on IoT devices easily and quickly. In [36], several design solutions including long-term recurrent convolution network (LRCN) for video captioning, inception module for the FaceNet face recognition, as well as long short-term memory (LSTM) for sound recognition are discussed. These and other similar design solutions are ideal implementations to be deployed in vision or sound based IoT applications.

Although HLS provides various advantages for FPGA designs, optimizing the FPGA performance through HLS remains challenging. Applying the right set of HLS techniques can prove complicated. The work in [37] demonstrates that the HLS solution quality can range from very slow all the way to 200× speedup compared to the CPU solution. Optimizing CNN through HLS faces similar challenges because there are many parameters that can be designed and controlled within CNN.

In this chapter, we explore different strategies and methodologies to optimize CNN on an FPGA. As some of dataset does not require a large CNN structure, it is efficient to use smaller CNN architecture, and replicate the CNN many times to improve both latency and throughput of the application. However, since on-chip memory is very precious for FPGA, we need to develop techniques to share the weight data among replicated CNNs while

they are processing different images in the same batch. Since all the images involve the same weight data, data sharing between the same CNN tasks on the loop-level is implemented to avoid replicated weight data storage.

As CNN is a sequential architecture in which the output of one layer becomes the input for the next layer, it is very important to pipeline between each layer to reduce the waiting time for the next layer. In order to achieve efficient pipelining, we apply our novel method, backward pipeline scheduling, resulting in dramatic latency improvement of processing an image, which is considered to be critical for an edge device. Due to the backward pipeline scheduling algorithm, data that is computed in one layer does not have to wait for all other data in the same layer to be computed. As data is computed from one layer, the data is used immediately for the next layer and this process propagates in the pipeline. Furthermore, along with the backward pipelining, to increase the throughput, we applied batch processing to our work. We process 10 K images, where each time we process 5 or 25 images as a batch, and complete the whole application with hundreds of batches. Since all the images that are in the same batch involve the same filters, computation can be further optimized. In summary, our work makes the following contributions:

•We propose backward pipeline scheduling in designing the CNN accelerator to achieve deep pipelining among layers in the neural network.

•We propose an implementation of CNN handwriting digits and Cifar-10 object recognition through HLS for embedded FPGAs as edge devices. The single image performance is 5.2× faster than NVIDIA Jetson TX1 for the LeNet and 1.9× faster for the CifarNet.

•We implement a data sharing method to save limited on-chip memory resource on the FPGA while enabling effective batch parallel processing.

The rest of the chapter is organized as follows. Section 4.2 presents the background of CNN and HLS. In Section 4.3, our algorithm and methodology for optimizing CNN through HLS on an FPGA are discussed in detail. Section 4.4 discusses hardware architecture implementation of the design. Section 4.5 presents and analyzes our experimental results, and Section 4.6

concludes the chapter.

4.2 Background

4.2.1 FPGA-based CNN Optimization

FPGA has become a promising platform for hardware acceleration because of its high performance, low power consumption, shorter development cycle and the reconfiguration flexibility compared to ASIC solutions. With such advantages, recently several research works have used FPGAs to accelerate CNN computations [19, 38, 39, 40, 18, 41]. Specifically discussed in [40] is a multistage data-flow implementation of CNN, which takes efficient utilization of the computation resources to achieve high performance in object classification. In [18], an FPGA-based CNN network accelerator is proposed. The paper discussed two main types of constraints of CNN designs: communication rate and computation capacity. Their design faced the constraint of limited communication rate between the FPGA and the external memory. In our work, we are able to overcome such a limitation through effectively reducing external data transfers and layer combinations.

4.2.2 High-level Synthesis Design Flow

HLS brings about such advantages by providing automated code transformations from high-level languages (such as C, C++, SystemC, etc.) to hardware description languages (HDL). HLS also provides automated optimization options through compiler pragmas, which can control the HLS engine to generate the RTL code following specific implementation styles. For example, these pragmas can guide the generation of loop and tiling structures, function interfaces, pipelining and inlining, and various resource instantiations. In this chapter, we leverage Xilinx Vivado HLS to facilitate our CNN design and report unique design techniques with the HLS design methodology. This automated code transformation provided by HLS enables designers to implement more delicate work easily onto FPGAs [42, 43]. Y. Guan et al. proposed an efficient FPGA-based LSTM-RNN accelerator with

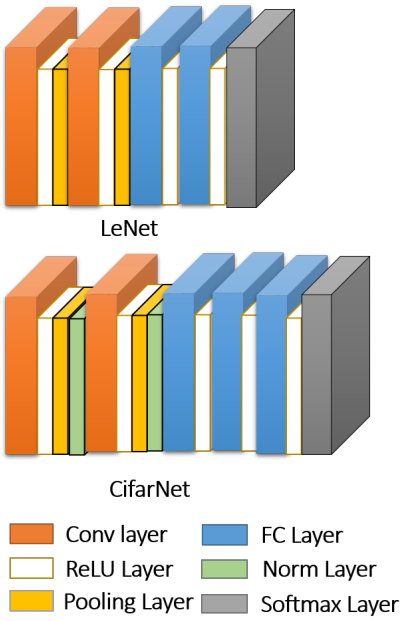


Figure 4.1: LeNet & CifarNet architecture

the HLS tool in 2017 [44]. R. Zhao et al. also adopted HLS as the design tool in their work of binarized convolutional neural network in 2017 [45].

4.2.3 CNN Structure

Figure 4.1 shows the CNN architectures we used to classify handwritten digits in the MNIST dataset [46] and 10 objects in the Cifar-10 dataset [2]. For the Cifar-10 dataset, we have pre-processed the images to train better and faster. While pre-processing the images, we discard their boundaries in order to make the network focus more on actual pixels that display the object to classify. Also, by distorting the images, by for example rotating and re-scaling, we can have more input data than given by the Cifar-10 dataset. Therefore, the network can learn fast as it can converge faster and generate higher accuracy. The inputs are pre-processed to have size of $24 \times 24 \times 3$ instead of the original size of $32 \times 32 \times 3$. The detailed layer configurations of the LeNet and the CifarNet are shown in Table 4.1 and Table 4.2.

Table 4.1: LeNet configuration

Type	Input size	Output size	# Params
Convolution	28×28	24×24×8	5×5×8
ReLU	24×24×8	24×24×8	NA
Pooling	24×24×8	12×12×8	NA
Convolution	12×12×8	8×8×16	5×5×16
ReLU	8×8×16	8×8×16	NA
Pooling	8×8×16	4×4×16	NA
Fully connected	256	128	256×128
ReLU	128	128	NA
Fully connected	128	10	128×10
Softmax	10	10	NA

Table 4.2: CifarNet configuration

Type	Input size	Output size	# Params
Convolution	24×24×3	24×24×32	5×5×3×32
ReLU	24×24×32	24×24×32	NA
Pooling	24×24×32	12×12×32	NA
Convolution	12×12×32	12×12×32	5×5×32×32
ReLU	12×12×32	12×12×32	NA
Pooling	12×12×32	6×6×32	NA
Fully connected	1152	192	1152×192
ReLU	192	192	NA
Fully connected	192	48	192×48
ReLU	48	48	NA
Fully connected	48	10	48×10
Softmax	10	10	NA

4.3 Algorithm and Methodology

In this section, we introduce our algorithm to perform the backward pipeline scheduling which achieves optimal data dependency relation among consecutive 2D-window operation layers. We use the LeNet and the CifarNet as examples to demonstrate the algorithm and method. However, the methodology can be applied to any other neural networks.

4.3.1 Data Dependency Analysis

A regular CNN network usually consists of convolutional layers, activation layers, and pooling layers. Such layers typically have a mesh-like layout and have a window-structured data dependency on the output from their previous layers. The input and output of a typical CNN network layer are configured in the format of feature-map with multiple channels. The output of a CNN layer is obtained through a particular type operation based on a 2D window of size F on the feature-map with fixed moving stride S. We define input to be feature-maps of size $H \times W$ and C channels. We use $I_{i,x,y}$ and $O_{i,x,y}$ to denote the pixel value in the ith channel and location (x, y) of input and out array.

Equation 4.1 gives the computation of output in the convolutional layer with $K_{i,o,h,w}$ representing the the filter element.

$$O_{i,x,y} = \sum_{o=1}^{C} \sum_{h=1}^{F} \sum_{w=1}^{F} K_{i,o,h,w} I_{o,xS+h,yS+w} \tag{4.1}$$

Equation 4.2 provides the computation of a max-pooling layer with window size W and stride S.

$$O_{i,x,y} = max(\{I_{i,xS+h,yS+w} | h \in [0, F), w \in [0, F)\}) \tag{4.2}$$

For simplicity, we consider all the data with the same location in feature-map through the channel dimensions to be combined in one data chunk. To compute a certain data chunk in output with feature-map coordinate $\langle x, y \rangle$, we need a set of data chunks from the input array. We define the set of coordinates of required data chunks to be the dependency set $Dep(\langle x, y \rangle)$ of coordinate $\langle x, y \rangle$. We can write the data dependency set as Equation 4.3. The equation works for all the layers with 2D window-structured on feature-map dimensions. Considering the factor of padding, we improve Equation 4.3 to Equation 4.4 with Z as the padding size.

$$Dep(\langle x, y \rangle) = \{\langle i, j \rangle | xS < i \leq xS + F, yS < j \leq yS + F\} \tag{4.3}$$

$$Dep(\langle x, y\rangle) = \{\langle i, j\rangle | xS - Z \leq i < xS + F - Z, 0 \leq i < H, \\ yS - Z \leq j < yS + F - Z, 0 \leq j < W\} \tag{4.4}$$

4.3.2 Backward Pipeline Scheduling

To achieve optimal pipeline and reduce the waiting time caused by data dependency, we develop an algorithm to arrange the order of data request in the current layer to fulfill the data requests in the following layer. We implement this algorithm by finding the data dependency set of each pixel coordinate in the computation order of the next layer.

Figure 4.2 illustrates an example of generating data request list for a max-pooling layer of window size 2 and stride 2. The output feature-map size is 2×2 with the order of data request labeled in the corresponding mesh block in the Figure 4.2.

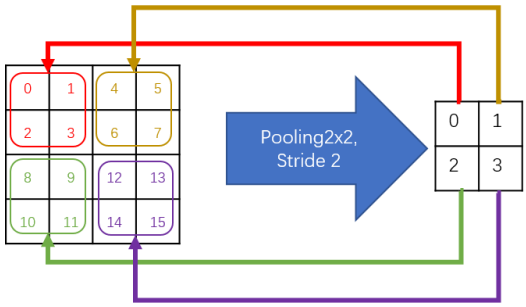


Figure 4.2: Data request list generation

The algorithm is described in Algorithm 3. For each coordinate in the data request list of the next layer (**nextList**), the data dependency set is computed. Then following the order of data request list, the coordinates in each dependency set are scheduled to a new data request list of the current layer (**curList**). Interleaved coordinates are only scheduled once during the first dependency set to which they belong. This (**curList**) becomes (**nextList**) of the previous layer. The current layer shall assume that its preceding layer feeds output data chunks following the order specified by the data request list of the current layer (**curList** or **nextList** of the previous layer) and the current layer is implemented to compute its output data following the data request list of its next layer (**nextList** or **curList** of the next layer).

Algorithm 3 Algorithm for data request list generation

```
1: function(nextList)
2: Input nextList: the data request coordinate ⟨x, y⟩ list of the next layer
3: Output curList: the data request coordinate ⟨x, y⟩ list of the current
   layer
4: Output curCompList: the list which stores the index of data after the
   transmission of which the data dependency is fulfilled
5: curList=[], curCompList=[], Outputindex=0
6: for i = 0 to nextList.length-1
7:       ⟨x, y⟩ = nextList[i]
8:       for all ⟨m, n⟩ in Dep(⟨x, y⟩)
9:            if ⟨m, n⟩ ∉ curList
10:                curList.append(⟨m, n⟩)
11:                Outputindex++
12:      curCompList.append(Outputindex);
13: return curList, curCompList
```

Algorithm 3 generates the data request list and computation index list for the current layer using the data request list of the next layer. Therefore, the overall scheduling algorithm proceeds in a backward manner: we initialize the output order of the last layer in the row-major order and perform Algorithm 1 in reverse order to the first layer. After the scheduling process, each layer will have its own request list and computation index list which stores the required number of inputs needed to calculate the output (**curCompList**). A typical CNN structure usually consists of several convolutional layers and pooling/activation layers followed by fully connected layers. Since the algorithm only works for layers based on 2D-window operations, we only schedule the pipeline behavior of layers before the last several fully connected layers. Figure 4.3 provides the comparison between the non-pipelined design and the design with our algorithm. The example in the figure includes one 3×3 stride 1 convolution layer and one 2×2 stride 2 pooling layer. For a non-pipelined design, the pooling layer can only start working after the input of convolution and pooling layers is entirely computed. For the backward pipeline scheduled design, the pooling layer can begin computing its first output pixel when the first 16 pixels of the convolution layer's input are calculated. The waiting latency for the pooling layer is reduced to the computation time for the dependent data in the input of the convolution layer and the pooling layer.

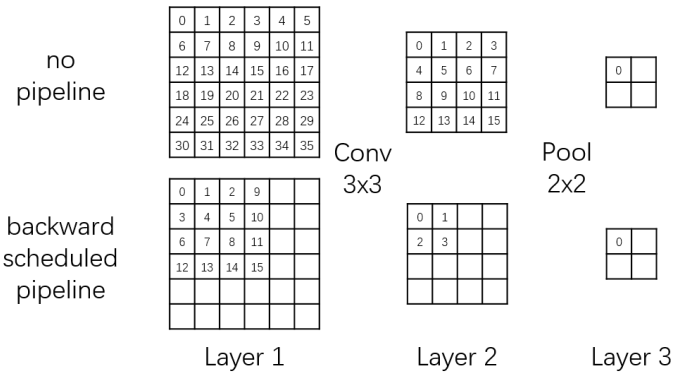


Figure 4.3: Backward pipeline scheduling flow

4.3.3 Layer Behavior

The data request list and the computation index list provide a fixed schedule that each layer must follow to compute its output data. Each layer holds a buffer matrix to store its input data generated by its previous layer. The data is sent in the unit of data chunks following the current data request list. Each time a layer receives a data chunk from the previous layer, it stores the data chunk in the buffer matrix at the coordinates indicated by the current data request list. Also, the computation index list monitors whether the data in the buffer matrix is enough to compute the output that is requested by next layer. The current layer enqueues the generated data chunk into the FIFO connecting the current layer and next layer. The algorithm is specified in Algorithm 4. In the algorithm, the function *window_operation* denotes the compute process of the 2D-window operation of the current layer such as Equation 4.1 or Equation 4.2. We denote the enqueue operation as symbol $\ll$ and the dequeue operation as symbol $\gg$.

4.3.4 Latency Balancing

Considering the application environment as edge devices, we balance the latency for each layer to achieve optimal resource utilization under the same performance. We assume the computation resource area to be proportional to the computation capability of the module *window_operation*. The data consumption rate of the current layer should match the data production rate of the previous layer. We conclude the average data consumption rate (R) for one layer as Equation 4.5 where F and L represent the total latency to

Algorithm 4 Algorithm for layer behavior

```
1: module(fifo_in, fifo_out)
2: Input Port fifo_in: the FIFO port to which the previous layer feed data
   chunks
3: Output Port fifo_out: the FIFO port to which the current layer feed
   computed data chunks
4: const curList, const curCompList, const nextList
5: matrix buffer
6: ComputeIndex = 0
7: for i = 0 to curList.length-1
8:     ⟨x, y⟩ = curList[i]
9:     fifo_in ≫ buffer[x][y]
10:    while( i = curCompList[ComputeIndex])
11:        ⟨i, j⟩=nextList[ComputeIndex]
12:        fifo_out ≪ window_operation(⟨i, j⟩,buffer)
13:        ComputeIndex++
```

complete lines 8-10 and lines 11-13 in Algorithm 4 respectively. With the same notation, the average data production rate (P) is shown in Equation 4.6.

$$R = \frac{curList.length}{curList.length \cdot F + curCompList.length \cdot L} \tag{4.5}$$

$$P = \frac{nextList.length}{curList.length \cdot F + curCompList.length \cdot L} \tag{4.6}$$

To achieve an efficient pipeline between two consecutive layers A and B, we need to set the production rate of A to match the consumption rate of B. Note that $curList.length$ equals the size of input feature-map $HI \times WI$ while $nextList.length$ and $curCompList.length$ equal the size of input feature-map $HO \times WO$. We have Equation 4.7 to constrain the latency F and latency L and eliminate the bottleneck effect in the pipeline.

$$\begin{aligned} Constant &\approx HI_A \cdot WI_A \cdot F_A + HO_A \cdot WO_A. \cdot L_A \\ &\approx HI_B \cdot WI_B \cdot F_B + HO_B \cdot WO_B \cdot L_B \end{aligned} \tag{4.7}$$

4.4 Hardware Implementation

This section introduces the hardware architecture details for implementing CNN structure with backward pipeline scheduling. We use the LeNet-5 and

the CifarNet as our benchmark to test our design methods. However, the method is general and can be applied to other types of DNNs as well.

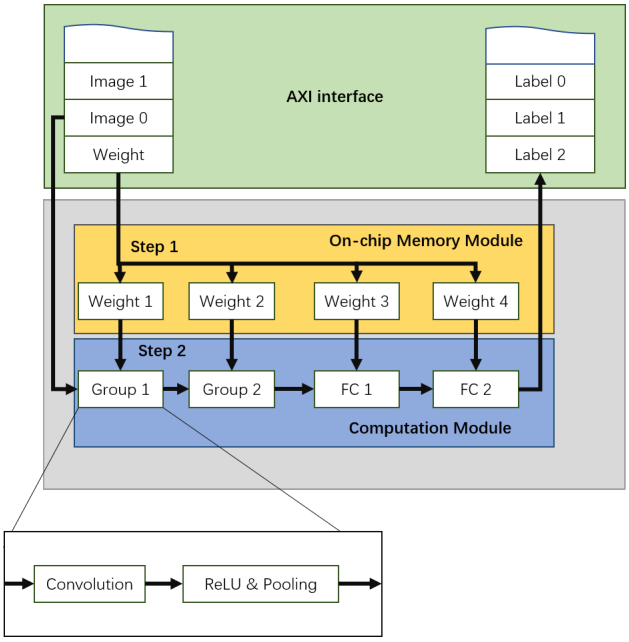


Figure 4.4: Block structure of the design

4.4.1 Architecture Overview

The CNN design consists of two main convolution layer groups and several fully connected layers. Each convolution layer group contains one convolution layer, one ReLU layer and one max-pooling layer as shown in Figure 4.4. These groups are instantiated as 2D-window modules which will be further discussed in Section 4.4.2. The fully connected layers are implemented using paralleled matrix multiplication module. Apart from the computation modules, the CNN accelerator also contains an on-chip memory module, which stores the weight data frequently requested by the modules while processing. The hardware design communicates with external memory through the AXI4 stream DMA interface. The AXI4 stream interface is a FIFO streaming interface which transfers data from or to external memory sequentially. Weight and image data will be fed into the FPGA hardware through the AXI4 stream interfaces while the result label sequence from classification computed by computation module is sent out to the external memory by the AXI4 stream interfaces. Each AXI4 stream interface contains a FIFO

buffer which continuously reads data from external memory. In this design, we use two AXI4 stream interfaces for input and output streaming. The overall structure is shown in Figure 4.4. The AXI4 interface first streams in the weight data as shown in step 1 in Figure 4.4. Then the image data is fed in frame by frame to the computation module and goes through convolution groups and fully connected layers to perform the corresponding computation as shown in step 2. Meanwhile, the output labels will be sent back to the external memory.

This design is built in such a way for the following purposes. First, it avoids the transfer operations of weight and inter-layer data between FPGA and external memory compared to conventional CNN hardware implementation. According to the calculation in [18], the bottleneck for CNN designs is usually the communication rate instead of computation capacity. The communication rate refers to how fast the FPGA can communicate with external memory. By reducing the data transfer operations, the limitation of performance caused by the bandwidth is removed. Therefore, better performance can be achieved by full usage of computation resources. Second, pipelining requires modules accessing weight data simultaneously. By storing weight data on-chip, the computation modules can access corresponding weight value independently without interfering with other computation modules.

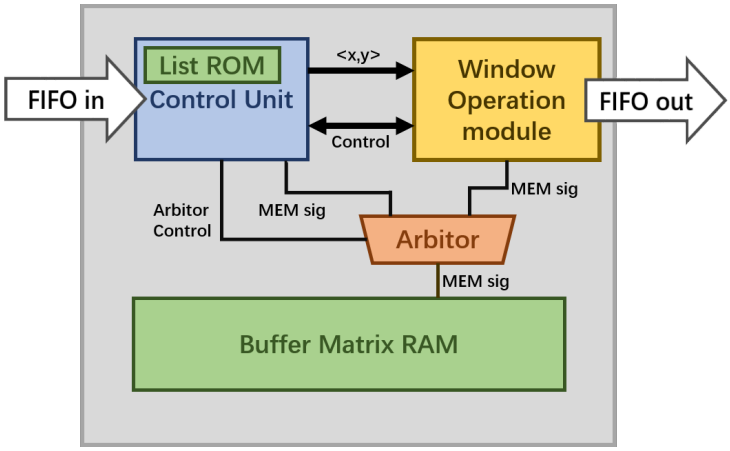


Figure 4.5: General block structure of 2D-window operation modules

4.4.2 2D-Window Modules

The convolution and pooling layers are implemented as 2D-window operation modules. The 2D-Window operation modules are designed to behave as described in Algorithm 4. Figure 4.5 shows the general structure of a 2D window. The structure consists of three major parts: the control unit, the memory block group, and window operation module.

The control unit includes the state-machine that controls the loop iteration and condition flow. All the scheduling lists are instantiated as constant ROMs inside the control unit for quick index access. The control unit also handles the input data fetching and arbitrates the service of the RAM which acts as the buffer matrix. The control unit fetches data from the input stream port and stores the data at the address referenced from the List ROM. If the counter matches the current output of computation index list ROM, the control unit passes the output coordinate $\langle x, y \rangle$, transfers the RAM service and initializes the operation of the window operation module.

In the pooling layers, the 2D-window module is instantiated as a max-pooling module. The pooling module decodes the vector $\langle x, y \rangle$ into RAM addresses mapped by coordinates in $Dep(\langle x, y \rangle)$. Through the decoded address, the max-pooling module loads in the data chunk from the RAM buffer. The data chunk is unpacked into a partitioned array of feature-map value along channel dimension. The pooling module then performs pooling and ReLU operations on the data arrays to generate pooling and ReLU results. The pooling results are repacked to a data chunk and fed into the output FIFO.

In the convolution layers, the 2D-window module is instantiated as a convolution module. The data chunk in dependency set is read in and unpacked in the same way as in the pooling module. A paralleled and pipelined convolution is performed with the weight and bias data fetched from the on-chip memory modules. This process is shown in Listing 4.1. The pseudo code performs the computation process in Equation 4.1 with a fixed $\langle x, y \rangle$ pair and varying index i. The two innermost loops are unrolled to achieve parallel computation. The convolution result *Oarray* is packed back to a data chunk and fed into output FIFO.

```
conv_tile( xS, yS,
Buffer[HI][WI],weight[CO][CI][F][F]){
  Iarray[CI];//ARRAY_PARTITION
  Oarray[CO];//ARRAY_PARTITION
  //clear array Oarray
  for(int h=0; h<F,h++){
  for(int w=0; w<F; w++){
  #pragma HLS pipeline
    unpack(Buffer[iS+h][jS+w], Iarray);
    for(int co=0;co<CO;co++){
    #pragma HLS unroll
      for(int ci=0;ci<CI;ci++){
      #pragma HLS unroll
             Oarray[co]+=
    weight[co][ci][h][w]* Iarray[ci];
           }}}}
    return pack(Oarray);
}
```

Listing 4.1: Tiled convolutional layer pseudo code

4.4.3 Batch Processing

In [18], the authors discuss how to use loop unrolling and loop pipelining to achieve better performance for a single convolution layer. However, the method discussed in [18] does not give much performance increase for smaller CNN due to the smaller number of filters in those CNNs. Therefore, even though the maximum unrolling factor has been chosen, the computation resource is still not fully utilized. In order to take full advantage of computation resources and achieve much better performance, batch processing methods can be applied.

With streaming input data, the batch processing will fetch a set of images and complete their processing simultaneously. Similar techniques are used in GPU domain as well [47]. We first stream in the images and store them in the image batch which is instantiated using on-chip memory. Then each image in the batch goes through an independent computation module. Finally, the generated labels are also stored in batch and then streamed out. This procedure is shown in Figure 4.6 with N as the batch size.

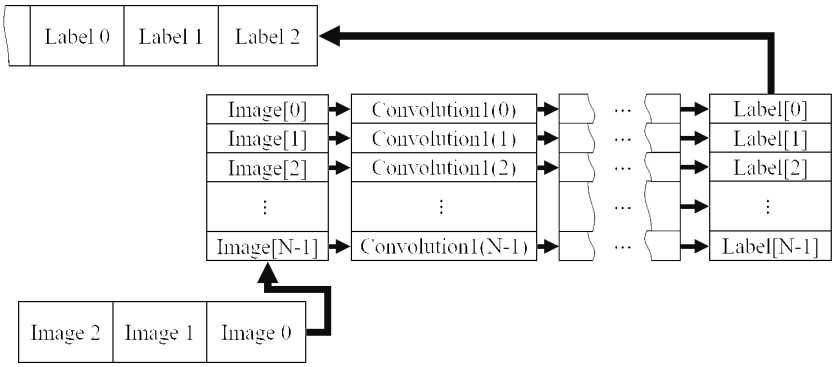


Figure 4.6: Batch processing computation module

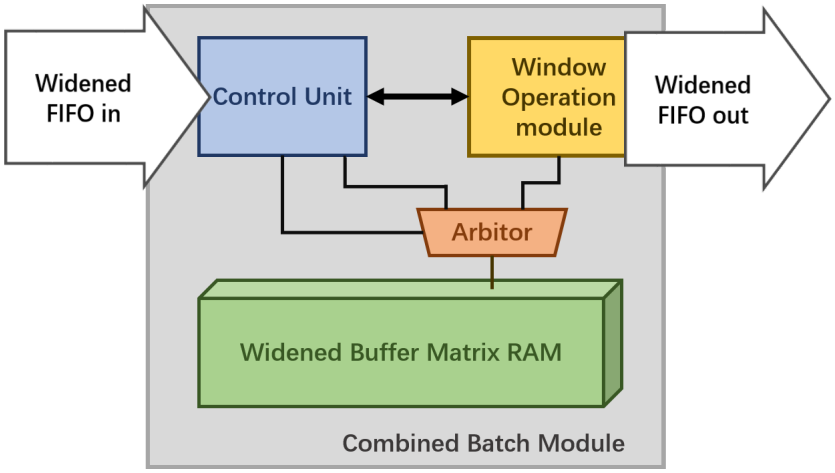


Figure 4.7: Structure for batch mode 2D-window array

However, naively duplicating the modules is inefficient and wastes resources. All the computation module copies require access to the same weight data stored in on-chip memory while the on-chip memory port can only serve one module at the same time, which causes racing and latency of the waiting time for RAM service among modules. Also, the control logic of the modules in the same batch has the same behavior pattern: multiple copies cause unnecessary resource occupations. To avoid the problem caused by direct duplication, we only copy the necessary components. We combine the module batch into one single module with only one copy of the control module as shown in Figure 4.7. Since different images generate different input/output feature-maps, the FIFOs between layers and matrix buffers are widened by N times to transmit and store N data chunks at the same time. For the pooling layer, the widened data chunk goes through the same process to generate a widened pooling result chunk in the window operation module. For the

convolutional layer, the behavior of the convolution module is modified to accommodate batch mode as shown in Listing 4.2. Variable *BufferWIDE* represents the widened matrix buffer. The modified unpack/pack function transfers the widened data chunk from or to N data arrays along the channel dimensions. The N input arrays are processed in parallel to generate results on the N output arrays with share weight data from on-chip memory module as shown in the fully unrolled for-loop in lines 15-19 of the code listing.

```
1  conv_tile( xS, yS,
2  BufferWIDE[HI][WI],//ARRAY_PARTITION dim=1,2
3  weight[CO][CI][F][F]//ARRAY_PARTITION dim=1,2
4  ){
5    Iarray[N][CI];
6    Oarray[N][CO];
7    //clear array Oarray
8    for(int h=0; h<F,h++){
9    for(int w=0; w<F; w++){
10   #pragma HLS pipeline
11     unpack(BufferWIDE[iS+h][jS+w], Iarray);
12     for(int co=0;co<CO;co++){
13     #pragma HLS unroll
14       for(int ci=0;ci<CI;ci++){
15         for(int cb=0; cb<N; cb++){
16       #pragma HLS unroll
17              Oarray[cb][co]+=
18     weight[co][ci][h][w]*Iarray[cb][ci];
19            }}}}}
20     return pack(Oarray);
21 }
```

Listing 4.2: Batched convolutional layer pseudo code

4.5 Experiment Result and Analysis

To get experimental results for our algorithm we have implemented both the LeNet and the CifarNet on an FPGA and a GPU. Since we want to target embedded devices, we have selected NVIDIA Jetson TX1 and Xilinx ZYNQ-7000 SOC ZC706. The platform specifications are shown in Tables 4.3 and 4.4. All the computations are fully parallelized to effectively and quickly generate output.

Table 4.3: Xilinx ZC706 device spec

LUT	218600
Flip-Flop	437200
BRAM	1090
DSP	900
Clock Sources	Fixed 200 MHz LVDS oscillator

Table 4.4: Jetson TX1 device spec

Global memory	3995 MBytes
GPU Max Clock rate	72 MHz
Max constant memory	65536 bytes
Max shared memory	49152 bytes
Max Block Dimension	(1024, 1024, 64)
Max Grid Dimension	(2147483647, 65335, 65335)

4.5.1 Statistical Analysis

In the experiment, we use the design optimized by naive pipeline and unroll pragmas as the baseline to illustrate the effectiveness of our algorithm and strategy discussed above. Table 4.5 lists the latency, throughput and resource utilization of designs after each optimization method. The backward pipeline scheduling improves the latency by 1.6X by enabling interaction of request lists among layers. After the backward pipeline scheduling, we notice that convolution layer 1 has prominent latency among the pipelined layers as shown in Figure 4.8. Meanwhile, convolution layer 2 occupies extra DSPs and LUTs resources but makes little contribution to the performance. We alter the unrolling factors in the operation module of convolution layer 1 and convolution layer 2 and reduce the bottleneck latency to 16,034 clock cycles. The overall performance is improved by 1.53X after latency balancing. Also, the DSP, flip flop and LUT usage are reduced by 3.24X, 2.22X and 2.25X respectively. Then we optimize our design with the batch method which improves the throughput but makes no improvement on single image latency. We can observe that the throughput increases proportionally to the batch size while the latency remains almost the same. Overall, we implement the LeNet digit classifier with the highest throughput of 130871.9 images/s and best single image latency of 175.7 μs. We implement and optimize the CifarNet using backward pipeline scheduling and latency balancing. Due to

the device constraint, we did not apply batch optimization on the CifarNet. The resource and performance result of our final version of the CifarNet are listed in Table 4.6. We achieve single image latency of 653.4 μs and throughput of 1530.3 image/s in our implementation of the CifarNet.

Table 4.5: Resource utilization and performance statistic of LeNet

Version	BRAM	DSP	FF	LUT	Latency	Clock Period	Throughput (img/s)
Baseline	144	162	27325	30056	447.3 μs	8.02 ns	2099.5
Backward Pipeline Schedule	144	162	28432	32467	278.4 μs	8.54 ns	3591.9
Latency Balancing	144	50	12793	14392	175.7 μs	8.54 ns	5660.6
Batch(5)	247	170	41403	46573	176.4 μs	8.60 ns	28472.5
Batch(25)	**762**	**850**	**202777**	**208612**	**191.0 μs**	**9.09 ns**	**130871.9**
Resource @ZC 706	1090	900	437200	218600	NA	NA	NA

Table 4.6: Area and performance for CifarNet

	CifarNet	Resource @ZC 706	Utiliazation
BRAM	492	1080	45%
DSP	162	900	18%
FF	59925	437200	13%
LUT	54017	218600	24%
Latency	653.4 μs	NA	NA
Throughput	1530.3 FPs	NA	NA

Table 4.7: Performance comparison for LeNet

Version	Latency	Throughput	Accuracy
TX1(Batch 1)	0.91 ms	1098.9 img/s	98.8%
TX1(Batch 25)	0.945 ms	26455.0 img/s	98.8%
Ours. (Batch 1)	175.7 μs	5660.6 img/s	97.6%
Ours. (Batch 25)	191.0 μs	130871.9 img/s	97.6%

4.5.2 Performance Comparison

The tables comparing the performance of NVIDIA Jetson TX1 and our design are shown in Table 4.7 and Table 4.8. For an image from the MNIST

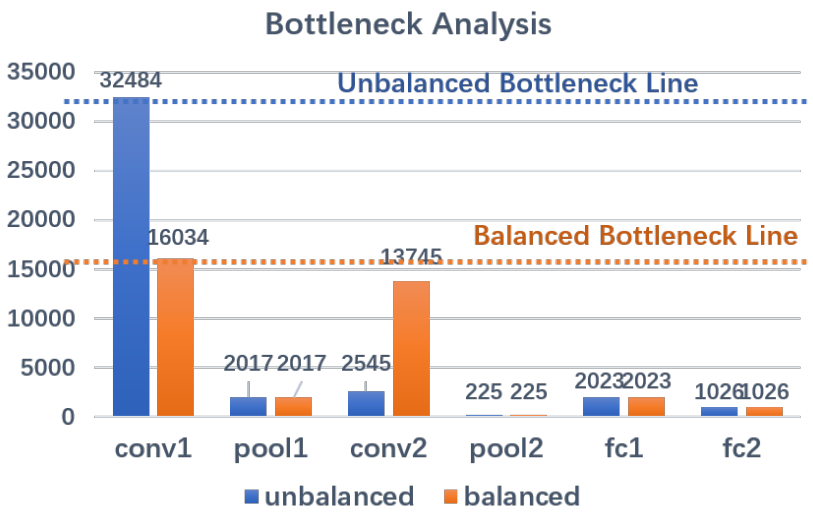


Figure 4.8: Latency comparison for latency balancing

dataset, TX1 takes 0.91 ms to classify. The throughput of GPU for the MNIST dataset is 26455 image/s if we set the batch size to be 25. For an image from the Cifar-10 dataset, it takes 1.27 ms to classify and the throughput is 787.4 image/s. Based on our experimental result we can see that FPGA processes one image 5.2× faster than TX1 does. We see that even if the algorithms are fully parallelized for the LeNet on TX1, the resources of TX1 cannot be fully utilized for small batch size. As batch size gets much larger, TX1 will be able to start processing many more images in parallel, beating the speed of FPGA. However, since we are targeting edge devices, we focus on the latency of classifying one image or a small batch of images. The latency of classifying one or a small batch of images is more important for an edge device as it has to process the input in real time and it usually is not in a large batch mode as used in cloud computing [48]. We also compare our LeNet single image latency with that of [49] and [50]. The result is listed in Table 4.9. We achieve 11× and 7.5× speedup by enabling proper pipeline among layers and further optimization in parallel computation architecture.

Table 4.8: Performance comparison for CifarNet

Version	Latency	Throughput	Accuracy
TX1(Batch 1)	1.27 ms	787.4 img/s	86.7%
Our Design(Batch 1)	653.4 μs	1530.3 img/s	83.6%

Table 4.9: Performance comparison with previous work for LeNet

	[49]	[50]	Our work
CNN model	LeNet-5	LeNet-5	LeNet-5
platform	ZC706	VC709	ZC706
Precision	fixed(25)	fixed(8-16)	fixed(16)
Latency	2 ms	1.318 ms	175.7 μs

4.6 Conclusion

When applying machine learning algorithms on IoT devices, implementing the algorithms on limited resources is important. The algorithm must work fast on the embedded devices to achieve practicality. In this chapter, we optimized the CNN structure for high-accuracy handwriting digits and Cifar object recognition through a novel scheduling algorithm and high-level synthesis. We explored methodologies such as parallel classifying operations with batch processing, backward pipelining and latency balancing. We achieved 5.2× speedup compared to the GPU version. We believe the techniques proposed and the HLS design methodology used should be applicable to other types of convolutional neural networks and enable FPGAs to become strong candidates for high throughput, high speed, yet low power/energy accelerators for various types of IoT applications, which can lead to far-reaching impact.

CHAPTER 5

WINOCNN: KERNEL SHARING WINOGRAD SYSTOLIC ARRAY FOR EFFICIENT CONVOLUTIONAL NEURAL NETWORK ACCELERATION ON FPGAS

The combination of Winograd's algorithm and systolic array architecture has demonstrated the capability of improving DSP efficiency in accelerating convolutional neural networks (CNNs) on FPGA platforms. However, handling arbitrary convolution kernel sizes in FPGA-based Winograd processing elements and supporting efficient data access remain underexplored. In this chapter, we are the first to propose an optimized Winograd processing element (WinoPE), which can naturally support multiple convolution kernel sizes with the same amount of computing resources and maintains high runtime DSP efficiency. Using the proposed WinoPE, we construct a highly efficient systolic array accelerator, termed WinoCNN. We also propose a dedicated memory subsystem to optimize the data access. Based on the accelerator architecture, we build accurate resource and performance modeling to explore optimal accelerator configurations under different resource constraints. We implement our proposed accelerator on multiple FPGAs, which outperforms the state-of-the-art designs in terms of both throughput and DSP efficiency. Our implementation achieves DSP efficiency up to 1.33 GOPS/DSP and throughput up to 3.1 TOPS with the Xilinx ZCU102 FPGA. These are 29.1% and 20.0% better than the best solutions reported previously, respectively.

5.1 Introduction

Convolution neural networks (CNN) have been playing an essential role in solving practical applications, and FPGAs have demonstrated their flexibility, efficiency, and reconfigurability as an ideal platform for CNN acceleration [36, 51, 52, 53, 41]. Many previous works have proposed different

algorithms and architectures to achieve high performance for CNN acceleration on FPGAs [52, 53, 12, 54, 41, 4]. Since DSPs in FPGAs are usually the major computational resource, the runtime DSP efficiency, defined as the average amount of effective convolution operations executed per DSP per second (GOPS/DSP), is crucial for FPGA design performance and is one of the most important factors to evaluate the quality of FPGA designs [52, 41, 55, 56].

Meanwhile, Winograd's minimal filtering algorithm has been widely adopted in CNN acceleration [57]. It trades multiplications with additions to save computational resources [57]. In FPGA, such a trade-off saves DSP resources from massive number of multiplications in CNNs, and hence improves the concurrency and efficiency of acceleration. However, due to the inherent characteristics of the algorithm, existing Winograd convolution algorithms are usually specifically designed for a fixed convolution kernel size, e.g., 3×3 [58, 59]. When applied to other popular kernel sizes, i.e., 1×1 in lightweight CNNs, it becomes inefficient due to the overhead of kernel padding [59]. In addition, the tile-based data pattern required by the Winograd algorithm together with the concurrent processing requirement usually result in high data transmission overhead [59].

Systolic array-based accelerator architectures are considered compelling to deal with the massive number of computations and communications required by CNNs [60, 61, 59], delivering the state-of-the-art performance. However, the performance of the systolic array-based architecture largely relies on the efficiency of the processing elements (PEs) inside the array, the data transmission among PEs, as well as the data access from external memory, which are all non-trivial to optimize.

In this chapter, to address the aforementioned issues and improve system performance and DSP efficiency for Winograd-based CNN acceleration, we make the following contributions:

- We design a novel Winograd-based processing element, WinoPE, using our generalized resource sharing mechanism that supports flexible convolution kernel sizes with high DSP efficiency.
- Using the proposed WinoPEs, we construct a scalable systolic array-based accelerator WinoCNN, which supports flexible configurations with different parallelism levels honoring FPGA resource constraints.

- We design a fine-grained and highly efficient memory control system that can deal with different memory access patterns and provide tile-based data to our WinoPEs with high efficiency and throughput.
- We propose accurate models for resource and performance estimation, which guide the design space exploration for the configurable parameters of our WinoCNN accelerator.

5.2 Background and Design Challenges

5.2.1 Winograd Convolution on FPGA

Winograd convolution is based on the Winograd minimal filtering algorithm that computes an $m \times m$ output matrix Y by convolving a $(m+k-1) \times (m+k-1)$ input matrix d with a $k \times k$ kernel g as described in Figure 5.1. The input size is also treated as the Winograd filter size. It reduces the number of multiplications at the cost of additions [57]. A 2D Winograd algorithm $F(m \times m, k \times k)$ includes a consecutive sequence of matrix transformation and element-wise multiplication (represented as $\odot$). The G, B, and A are constant transform matrices generated by Cook-Toom algorithm [57].

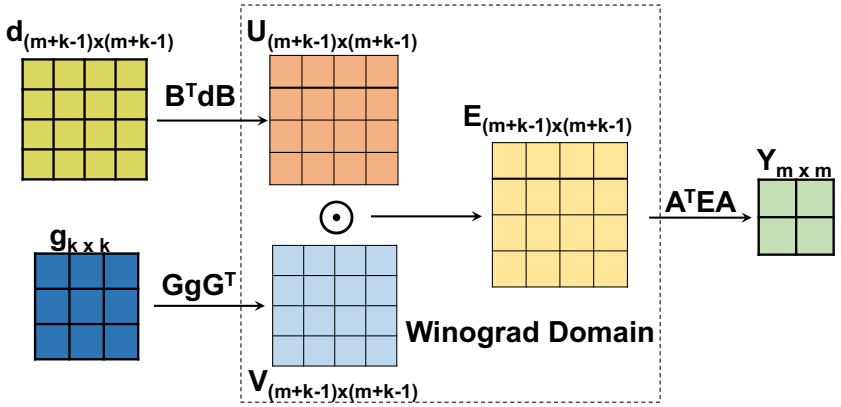


Figure 5.1: $F(m \times m, k \times k)$ Winograd convolution

A convolution layer in CNN with $k \times k$ convolution kernel can be computed using Winograd algorithm with a configuration of $F(m \times m, k \times k)$. The computation of each output feature-map O with size $H_o \times W_o$ is divided into tiles with size $m \times m$, resulting in $\lceil H_o/m \rceil \lceil W_o/m \rceil$ tiles in each output

channel. The computation of the output tile O_{o,x_o,y_o} starting at pixel (x_o, y_o) in channel o can be completed by applying Winograd algorithm on input tiles I_{i,x_i,y_i} starting at pixel (x_i, y_i) in all the C input feature-map channels with kernel $K_{o,i}$ and summing the results up, as shown in Equation 5.1.

$$O_{o,x_o,y_o} = A^T(\sum_{i=1}^{C}[(B^T \cdot I_{i,x_i,y_i} \cdot B) \odot (G \cdot K_{o,i} \cdot G^T)])A \tag{5.1}$$

The transformation operations with B, G, A are matrix multiplications with constant element values that can be completed by add/shifting operations. So the total number of multiplications equals to the number of element-wise multiplications of U and V, which is less than the required multiplications in the conventional convolution [62]. Since the multiplications on FPGAs are conducted by DSPs, reducing required multiplications in convolution helps to improve parallelism with a given number of DSPs and hence improves computation performance.

However, there is a critical problem: The constant transformation matrices (B, G, A) for a given convolution kernel size have fixed patterns; this results in inefficient DSP utilization when using the hardware designed for one kernel size to a different kernel size, where it has to either split/pad the input data/kernels or to instantiate a new accelerator. For example, to compute 1×1 convolution kernel with a Winograd-based PE designed for 3×3 kernel, we need to pad 1×1 convolution to 3×3 with zeros, which can only achieve $\frac{1}{9}$ of the DSP efficiency of executing 3×3 convolution; or alternatively, instantiating a dedicated accelerators for 1×1 kernel only, which occupies additional resources. Hence, designing a Winograd-based PE with flexible support for different kernel sizes while maintaining high DSP efficiency is essential but remains unexplored.

5.2.2 Systolic Architecture

A systolic array [63] is typically composed of many interconnected identical PEs, where the intermediate data is computed by PEs and passed to adjacent PEs. Systolic array architectures are efficient for parallel computing and is widely adopted by FPGA accelerators for matrix multiplications and convolutions [41, 64]. One previous design [59] proposes a systolic ar-

ray architecture with Winograd algorithm to accelerate sparse convolution, which achieves 5× higher performance compared to the normal dense convolution accelerator. Another work [58] proposes a systolic array architecture specifically designed for ResNet units.

In general, mapping the application to systolic array requires the data buffering in the PEs and the short PE-to-PE data transmission pattern. CNNs are not naturally providing such buffering and connections patterns, which requires careful refinement of the orders of the operations and buffering of the data.

5.2.3 Efficient Memory Access

Inefficient data access of the PEs downgrades the overall performance [65, 41, 66]. To support efficient data access with limited off-chip memory bandwidth, the memory subsystem for the accelerator must be carefully designed for specific data re-arrangements and access patterns [58], e.g., using multiple line-buffers [66]. However, it is difficult to create a universal design that would be compatible with different CNN layer configurations. In addition, the systolic array of PEs requires the memory subsystem to provide concurrent off-chip memory access and on-chip data reuse to fully utilize the computational capacity of all PEs. Winograd algorithm further complicates the memory access requirements due to the varied planar data access patterns of the PEs.

5.3 Design Principles

To resolve the challenges discussed in Section 5.2, we design our WinoCNN accelerator system with the following design principles.

5.3.1 Sharing in Winograd Algorithm

As discussed in Section 5.2, the low DSP efficiency of the Winograd algorithm for varying convolutional kernel sizes is caused by the constant transformation matrices. The key solution is to provide flexible kernel size support within the Winograd convolution PE without reloading the transformation

matrix and reorganizing the computation procedure. For a Winograd convolution $F(m \times m, k \times k)$, the transformation matrices B, A, G and the intermediate Winograd filter sizes are fixed, as shown in Figure 5.1. The required number of element-wise multiplications equals to the size of U and V, which is $(m + k - 1) \times (m + k - 1)$.

The input transformation matrix B depends on the size of input tile d for the input transformation ($U = B^T dB$). For a set of Winograd algorithm configurations with a Winograd filter size ω, denoted as $F_\omega(m \times m, k \times k)$, where $\omega = m + k - 1$ ($\omega \geq k$). As long as ω values are the same, the computation patterns of input transformation and element-wise multiplication are exactly the same. Matrices $B_\omega^T(m \times m, k \times k)$ with same ω are identical. An example for $\omega = 4$ is shown in Figure 5.2. Meanwhile, U and V are all $\omega \times \omega$ matrices. Therefore, the hardware resource to process $U = B_\omega^T dB_\omega$ and $E = U \odot V$ can be shared among all $F_\omega(m \times m, k \times k)$.

The transformation matrices G and A will be different for different convolutional kernel sizes under the same ω. We observe that there are numerous repeated values for the G and A matrices across different m and k values when ω is the same, and the different element(s) could be used as identifier(s) for different kernel sizes and output sizes. As shown in Figure 5.2, a single element s could be used to identify $G_4(4 \times 4, 1 \times 1)$ and $G_4(2 \times 2, 3 \times 3)$.

Also, this sharing property of the transformation matrix A_ω and G_ω can be generalized to larger Winograd filter size ω such as F_8 and F_{10} for larger convolution kernel sizes such as 5×5 and 7×7 with multiple identifiers. As shown in Figure 5.3, the transformation matrices G_6 and A_6 with three identifiers s_0, s_1 and s_2 can be shared for the convolution kernel sizes 1×1, 3×3 and 5×5. This provides us a unique opportunity to reuse the same computation resource (DSP) for different input kernel sizes using a unified PE for Winograd convolution. The design details of the PE and resource sharing are presented in Section 5.4.1.

5.3.2 Task Mapping for PEs

We assume a PE can perform the element-wise multiplication and output transformation $PE(U, V) = A^T(U \odot V)A$ for $F_\omega(m \times m, k \times k)$ Winograd convolution in one cycle. To properly map the convolution task into PEs, we partition the computation process of a convolution layer into several it-

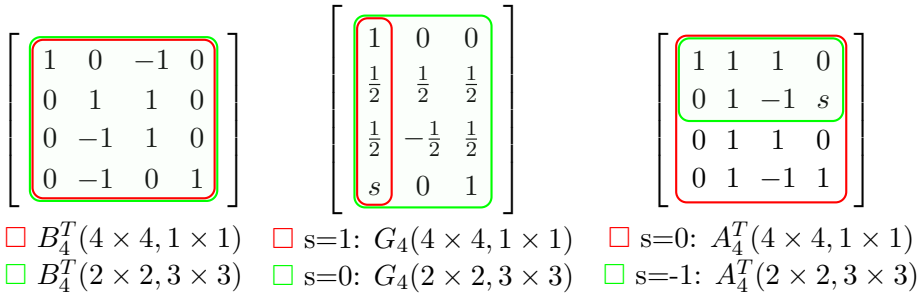


Figure 5.2: Winograd transformation matrix for F_4

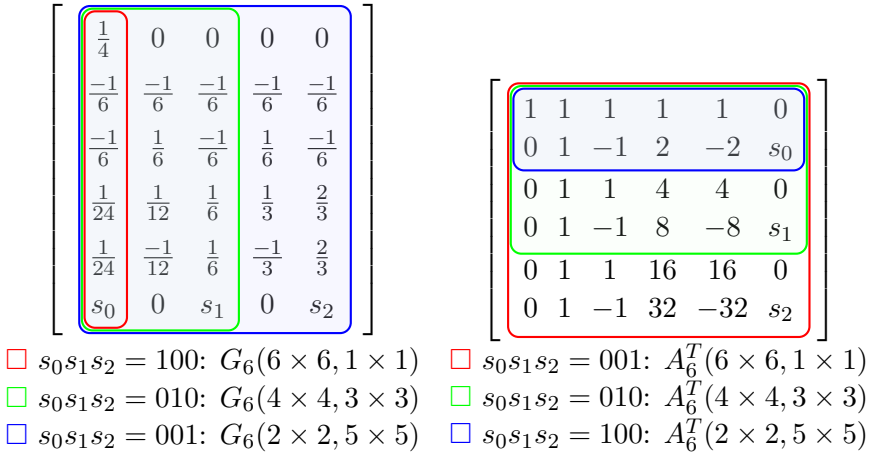


Figure 5.3: Winograd transformation matrix for F_6

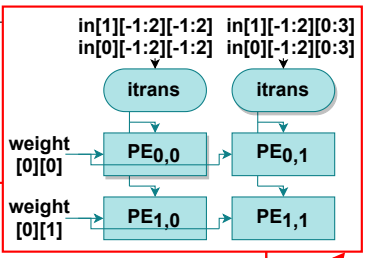
erations. In each iteration, RS consecutive rows of output feature-map are computed. Figure 5.4a shows the pseudo-code to compute the output feature-map of a convolution layer with $k \times k$ kernel size using one PE. The input, weight and output are represented as C-style array `in[ID][IH][IW]`, `w[ID][OD][k][k]`, and `out[OD][OH][OW]`, respectively. However, loops shown in the Figure 5.4a do not have the tiled structure to target the 2D PE array. In order to map the computation to the 2D PE array and increase the parallelism of data processing, we rearrange the loop as shown in Figure 5.4b and introduces two levels of tiling for the computation. Loop `L0` iterates through the output rows with a step of RS. Loop `L1` iterates through the output depth with a tile size of M. Loop `L2` iterates through the input depth. Loop `L3` segments the RS output rows into Winograd output tile of size m. Loop `L4` partitions the output columns into segments containing N size-m output tiles. After unrolling of `L5` and `L6`, $M \times N$ tiles of data will be processed by an $M \times N$ PE array in one cycle (as shown in Figure 5.4b for

a 2×2 array). In this way, all WinoPEs with the same row index or column index share the same weights or the same input tile, respectively.

```
L0: for (r_c=0;r_c<OH;r_c+=RS)
L1: for (od=0;od<OD;od++)
L2: for (id=0;id<ID;id++)
L3: for (r=r_c;r<r_c+RS;r+=m)
L4: for (c=0;c<OW;c+=m){
    V=wtrans(w[id][od][0:k-1][0:k-1]);
    U=itrans(in[id][r:r+ω-1][c:c+ω-1]);
    Y=PE(U,V);
    out[od][r:r+m-1][c:c+m-1]+=Y; }
```

(a)

```
L0: for (r_c=0;r_c<OH;r_c+=RS)
L1: for (od=0;od<OD;od+=M)
L2: for (id=0;id<ID;id++)
L3: for (r=r_c;r<r_c+RS;r+=m)
L4: for (c=0;c<OW;c+=m*N)
L5: for (i=0;i<M;i++)
L6: for (j=0;j<N;j++) {
        cn=c+j*m;
        V=wtrans(w[id][od+i][0:k-1][0:k-1]);
        U=itrans(in[id][r:r+ω-1][cn:cn+ω-1]);
        Y=PE<i,j>(U,V);
        out[od+i][r:r+m-1][cn:cn+m-1]+=Y; }
```



(b)

Figure 5.4: Loops of computation process

Note the direct mapping of the tiled computation to the PE array will generate high fanout (as shown in the embedded figure for the 2×2 array) and worsens the timing of the implementation. In order to address this issue, we schedule the computation of the PEs following the structure of an $M \times N$ systolic array. The detailed design will be presented in the Section 5.4.2.

5.3.3 Efficient Memory Access

Efficient execution of Winograd-based PEs requires simultaneous data access within a tile, as shown in Figure 5.5. This planar data access pattern (data tiles) brings in a challenge for efficient memory control and data supply for the PEs. When multiple PEs are instantiated as an array to process different tiles of a CNN layer, there are also overlaps among the data tiles required by

the PEs. As shown in the example in Figure 5.5, the data tiles required by adjacent PEs overlap with each other (marked with purple circle). Simply assigning input buffers for all the PEs would cause high on-chip memory usage [61]. However, line buffer based design [66] faces difficulties when supplying multiple tiles for Winograd-based PEs under systolic architecture that requires varied memory access patterns, i.e., varied window moving steps. As shown in Figure 5.4b `L4`, the PE array requires N input tiles with a horizontal moving step size of $N \cdot m$ each cycle. Meanwhile, the output tile size m differs according to kernel size k, leading to a varied window moving step. These motivate us to design a specialized memory system for our WinoCNN architecture.

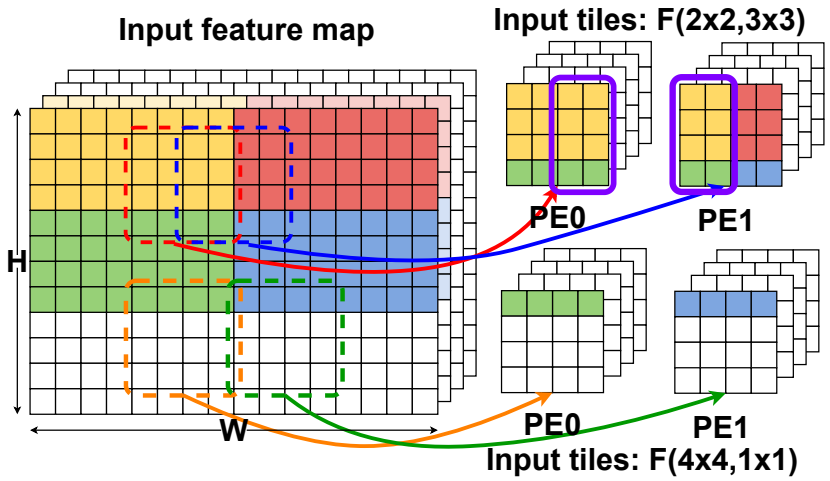


Figure 5.5: Planar data access pattern of PEs

To design an efficient PE array to work with such data access patterns, we draw three design principles: First, to improve computation efficiency with high parallelism, the data elements inside one tile must be fetched in parallel and provided to the computational unit simultaneously; and second, the overlapped data across tiles shall be fetched from external memory only once and then reused to reduce memory access overhead. Third, the memory system should be able to supply data for Winograd convolutions with different kernel sizes efficiently.

5.4 Implementation

We implement our WinoPE, systolic array and memory subsystem based on the design principles from Section 5.3 to build our WinoCNN acceleration system.

5.4.1 WinoPE: PE with Multiple Kernel Support

Our WinoPE is the basic processing unit of the Winograd systolic convolution accelerator system (WinoCNN), where each WinoPE is able to complete the computation of an input kernel and a set of feature tiles in a single clock cycle. WinoPE is featured with the flexible support for different convolution kernel sizes without the DSP overhead in a unified architecture. As discussed in Section 5.3, the Winograd algorithm with the same Winograd filter size ω can share the corresponding transformation matrices as well as the expensive dot product module. We choose the design of sharing between $F_4(4\times4, 1\times1)$ and $F_4(2\times2, 3\times3)$ as the example to present our kernel sharing mechanism. Figure 5.6 shows the unified architecture to process a single tile in our WinoPE. It contains an input tile register array (red block), a weight tile register array (blue block), a matrix of multipliers (purple circle), an output transformation module (green block), and an output tile (dark and light yellow) toward an output buffer.

In each working cycle, the WinoPE reads in a tile of input data and a tile of weights in parallel. Note here, the input tiles are transformed on-chip when they are fetched from the input buffer and the convolution kernel weights are transformed before they are stored into the on-chip memory to reduce the resource usage for transformation logic. After fetching the input and weight, the element-wise multiplication $U \odot V$ is performed. The output transformation module takes the results of $U \odot V$ and generates the output tile. The data fetching and element-wise multiplication modules can be directly shared and fully utilized by different convolution kernel sizes. To handle the different output caused by different kernel sizes, we design a selectable output transformation matrix A^T_{sel}, in which the **selection bit** s in the matrix A is used as a matrix identifier, as shown in Figure 5.2 (Section 5.3). As an instance in F_4, when s is set to 0, the WinoPE performs $F_4(4 \times 4, 1 \times 1)$ algorithm, where the whole 4×4 matrix is the output of

the WinoPE (light yellow block in Figure 5.6). When s is set to -1, the WinoPE performs $F_4(2\times 2, 3\times 3)$ algorithm, where the top left four elements of the result matrix is the output (dark yellow block). In this way, our WinoPE processes convolution layers with different kernel sizes without DSP overhead. Finally, the computed outputs are stored in the output buffer constructed with BRAMs. Note that such a selection bit design can be easily extended to the Winograd algorithm with larger Winograd filter sizes for larger convolution kernel sizes.

Furthermore, we partition the input tile register U and weight tile register V into individual registers that each contains a single value from the input channels, so that the multiplication for an entire tile is finished in a single clock cycle. The processing efficiency of WinoPE is further increased by instantiating Q input channels of batch size B of tile registers with the corresponding number of $U \odot V$ multiplier matrices. An adder tree constructed with LUT is used to accumulate the outputs from the multiplier matrices.

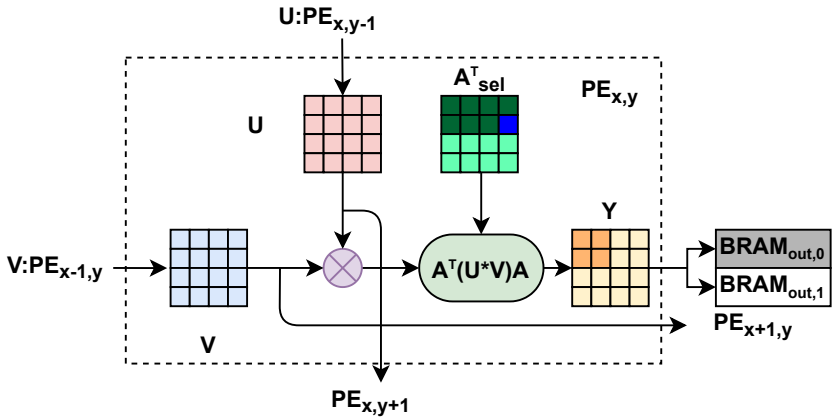


Figure 5.6: WinoPE processing logic

The selection bit design allows us to share the computation resources without wasting the DSPs when processing convolutions with different kernel sizes. However, it remains challenging to use Winograd convolution algorithm for large kernel convolution and irregular kernel convolution. A practical limitation is the larger Winograd filter size requires more LUT resources to conduct the addition operation during the constant matrix multiplication for Winograd convolution algorithm. Also, recent DNN models adopt irregular convolution kernels such as 1×7 and 7×1 sizes that are not well supported by the Winograd algorithm.

To handle the large kernel convolution and irregular kernel convolution, we design a split mechanism that splits the target convolution kernel into supported kernel sizes as shown in the Equation 5.2 and Equation 5.3.

$$K_s^{i,j}[h][w] = \begin{cases} 0, \;\; ik + h \geq H_t \text{ or } jk + w \geq W_t \\ K_t[ik + h][jk + w], otherwise \end{cases} \tag{5.2}$$

$$Output_{target} = \sum_{i,j} FM^{ik,jk} * K_s^{i,j} \tag{5.3}$$

K_t represents the target convolution kernel with size $H_t \times W_t$ and K_s represents the supported convolution kernel with size $k \times k$. The target kernel is split into $\lceil \frac{H_t}{k} \rceil \times \lceil \frac{W_t}{k} \rceil$ supported kernels with unaligned elements padded with zeros. The split kernel $K_s^{i,j}$ is segmented from the target kernel with ik, jk offset from the top left element. The targeted convolution (denoted as $*$) is completed by applying convolution for each supported kernel $K_s^{i,j}$ on input features with the same 2D pixel offset (denoted as $FM^{ik,jk}$) and summing up the split results as shown in Equation 5.3.

5.4.2 Parameterized Systolic Array

Instead of sharing a single set of data tiles among different WinoPEs in the same clock cycle (as shown in the Figure 5.4b), we construct the WinoPEs as an $M \times N$ systolic array that shares the weight and input data among WinoPEs by shifting them PE-to-PE to further utilize the on-chip registers and reduce the high fanout and long connection caused by the flattened implementation. To achieve this, we take advantage of the insensitivity of the loop order and assign FIFOs to WinoPEs (as shown in Figure 5.7). Note here, the PEs are called by an outside loop as `L0` and the loops `L1-L4` are the same as the ones in Figure 5.4b; however, the input Winograd tile and the weight tile are fetched from the top and left FIFO interfaces which connect to the top and left neighbors of the WinoPEs, respectively. In the same iteration, the input and weight data are pushed into bottom and right FIFO interfaces and passed to the bottom and right neighbors after one clock cycle due to the blocking mechanism of the FIFO. Therefore, the WinoPEs are constructed as a systolic array.

With the assigned FIFOs for our WinoPEs, we could easily instantiate the

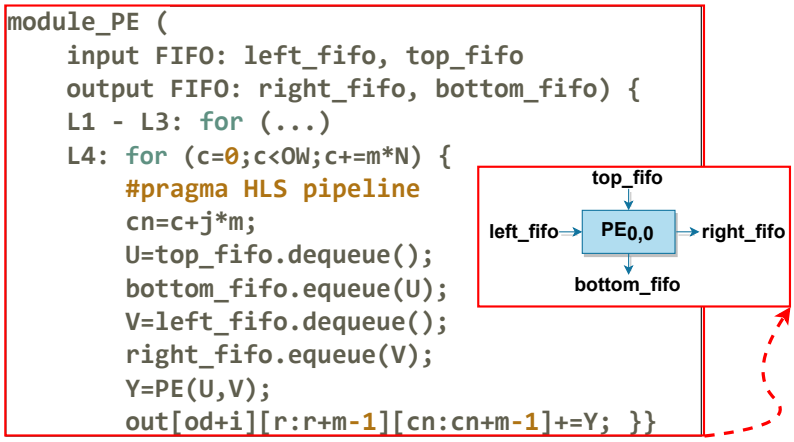


Figure 5.7: Loop behavior for $PE_{i,j}$

systolic WinoPE array by organizing the row and column FIFOs, denoted as $row\_fifo[M][N]$ and $col\_fifo[N][M]$. The M, N parameters are configurable during the WinoCNN system generation.

5.4.3 Hierarchical Memory Subsystem

To provide the required data to the WinoPE array efficiently, we propose: (1) a BRAM buffer matrix that has a unique addressing mechanism to support efficient parallel data access, and (2) a pipelined planar data control and scheduling to provide efficient on-chip data reuse and support the flexible input tile access pattern.

BRAM buffer matrix To guarantee the parallel access of the input tiles, we fold the input feature-maps into a matrix of BRAM buffers, denoted as $BRAM_{in}[H_b][W_b][D_b]$, which consists of $H_b \times W_b$ BRAM buffer instances (or BRAM bank) of depth D_b, as shown in Figure 5.8. Each BRAM buffer instance has its individual address port and data port, hence total $H_b \times W_b$ entries can be accessed from the BRAM buffer matrix in each cycle at different addresses. An address mapping mechanism is designed as shown in Equation 5.4 to decide the location in the buffer as $BRAM_{in}[h][w][addr]$ for a certain input pixel in the feature-map $in[id][r][c]$, where ID, id, r, c represent the number of input channel for a layer, channel index, row index and column

index for the pixel in the input feature-map:

$$\begin{aligned} h = r\%H_b,\ w = c\%W_b \\ addr = concat(\lfloor r/H_b \rfloor, \lfloor c/W_b \rfloor \cdot ID + id) \end{aligned} \tag{5.4}$$

The BRAM banks in the same row share the same high address bits, while the BRAM banks in the same column share the same low address bits. The concatenation function ensures that the entries are accurately located with the given index. With the sequence of input Winograd tiles denoted as T_N,

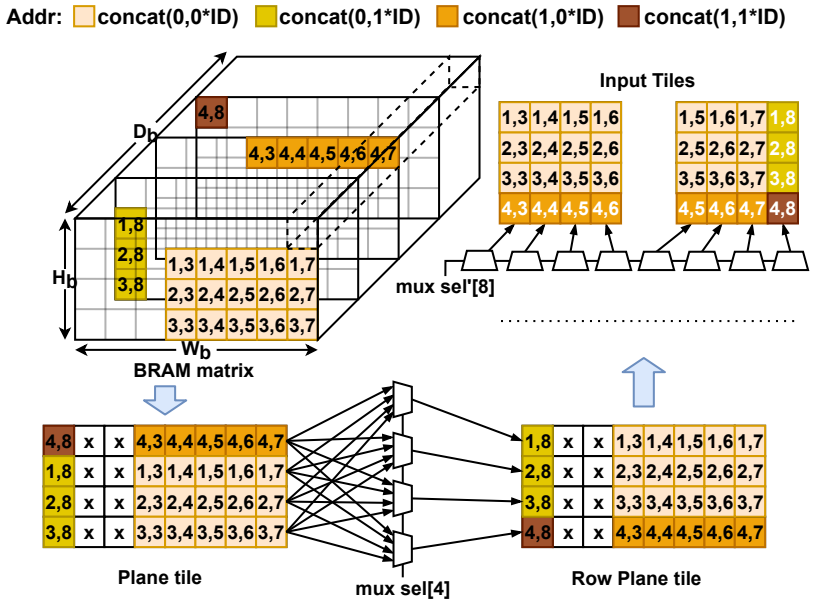


Figure 5.8: Hierarchical memory subsystem

where T_N consists of N tiles with $\omega \times \omega$ size, defined in Equation 5.5, all the elements in T_N forms a continuous input data block, denoted as T_U, with size $\omega \times ((N-1)m + \omega)$:

$$T_N[n] = in[id][r : r + \omega - 1][c + nm : c + nm + \omega - 1] \tag{5.5}$$

$$T_U = \bigcup_{n=0}^{N-1} T_N[n] = in[id][r : r + \omega - 1][c : c + (N-1)m + \omega - 1] \tag{5.6}$$

As an instance in Figure 5.8, $H_b = 4$, $W_b = 8$, $\omega = 4$, $m = 2$ and $N = 2$ and two data tiles are required to be accessed within input feature-maps $in\_data[0][1:4][3:6]$ and $in\_data[0][1:4][5:8]$. The union of the two input tiles can be represented as $T_U = in\_data[0][1:4][3:8]$. According to the

address mapping defined in Equation 5.4, the pixels in $in\_data[0][1:4][3:8]$ are accessed from four different regions of BRAM buffer matrix in one clock cycle with different high address bits and low address bits.

Planar data access The input tiles are then passed through a three-stage pipeline to ensure the data reuse and to provide the planar data to the WinoPEs. As shown in Figure 5.8, The first stage stores the $H_b \times W_b$ output from BRAM matrix buffer into registers. The second stage ensures the row order of the planar data with a row plane multiplexer array. The third stage splits the plane to tiles by a column multiplexer array. The mux selection bits are generated on-the-fly regarding the values of r, c, and id. Since both the BRAM bank addresses and the mux selection are generated on-the-fly, the memory architecture is able to supply input tiles with varied window moving steps regarding the kernel size for the current convolution layer.

5.5 System Architecture and Modeling

We construct our WinoCNN system and build the performance and resource models for easy exploration of the architectural configurations.

5.5.1 WinoCNN Architecture Overview

The overall architecture of our WinoCNN accelerator system is shown in Figure 5.9. Note here, the flexible convolution kernel size support is provided by our WinoPEs. The convolution layers of the input models are computed in output row stationary. The input reading, computation, and output data offloading are scheduled to run in parallel.

5.5.2 System Modeling

As shown in Section 5.4, our system is built with performance and resource sensitive architectural parameters, the corresponding models are built for design space exploration.

DSP usage model The major instances of DSPs are occupied by the WinoPEs. Each WinoPE computes the element-wise sum of the product along Q input channels for input tiles $U^{\omega\times\omega}$ of batch size B and weight tiles $V^{\omega\times\omega}$. Thus, the total number of DSPs required by a WinoPE is $\omega^2 \cdot B \cdot Q$. The systolic WinoPE array contains $M \times N$ WinoPEs, so the total DSPs required by our WinoCNN accelerator is:

$$DSP_{use} = \omega^2 \cdot M \cdot N \cdot B \cdot Q \tag{5.7}$$

BRAM usage model The BRAM resource is mainly occupied by the input, weight and output buffers. The BRAMs are in the form of 18-bit width and 1024 depth blocks.

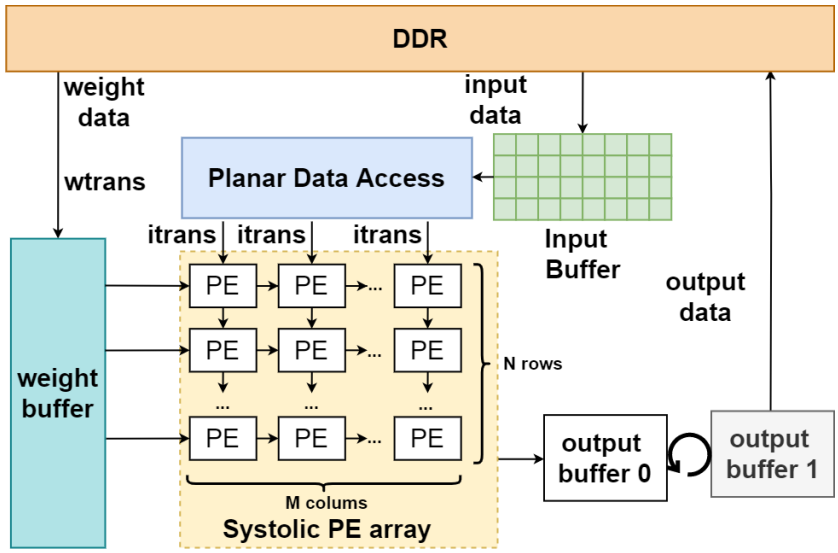


Figure 5.9: Overall architecture

The input buffer is a buffer matrix of size $H_b \times W_b$ with buffer depth as D_{in}. Each bank should be capable of storing B input data with 8 bits. The number of BRAMs for input is $H_b \cdot W_b \cdot \lceil 8 \cdot B/18 \rceil \cdot \lceil D_{in}/1024 \rceil$.

Each row of the systolic array requires $\omega^2 \cdot Q$ transformed and quantized 16-bit weight data, thus requires $\lceil \omega^2 \cdot Q \cdot 16/18 \rceil$ BRAM blocks with a fixed depth of 1024 to provide enough weight access bandwidth. The total BRAM required for weight buffer is $M \cdot \lceil 16 \cdot \omega^2 \cdot Q/18 \rceil$.

Each WinoPE has a $\omega \times \omega$ buffer matrix to store 18-bit temporary output data of batch B and buffer depth D_{out}. With the requirement of two buffers for ping-pong access, each WinoPE needs $2 \cdot \lceil \omega^2 \cdot B \cdot 18/18 \rceil \cdot \lceil D_{out}/1024 \rceil$

BRAMs as output buffer.

The total number of BRAM is the sum of the above:

$$\begin{aligned} BRAM_{use} =& H_b W_b \lceil \frac{8 \cdot B}{18} \rceil \cdot \lceil \frac{D_{in}}{1024} \rceil \\ &+ M \cdot \lceil \frac{16 \cdot \omega^2 Q}{18} \rceil + 2 \cdot M \cdot N \cdot \omega^2 B \lceil \frac{D_{out}}{1024} \rceil \end{aligned} \tag{5.8}$$

Latency model Communication latency t_{comm} and computation latency t_{comp} in each phase of the convolution procedure are used to build the latency model. The maximum value between these two in each phase dominates the overall latency.

Since all weights are required once within each loop iteration, we have $D_{weight} = k^2 \cdot ID \cdot OD$. Each loop iteration includes a read input process and a write output process with the corresponding data transmission amount of $D_{input} = RS \cdot ID \cdot IW \cdot B$ and $D_{output} = RS \cdot OD \cdot OW \cdot B$. Neglecting the absence of output writing in the first iteration and the input reading in the last iteration, we estimate the communication latency as:

$$t_{comm} = (D_{weight} + D_{input} + D_{output})/BW \tag{5.9}$$

To compute RS rows of outputs in each computation process, the WinoPE array needs to sum up the convolution results through ID input planes to generate $RS \times OW$ output data for OD output planes. In each cycle, $M \times N$ WinoPEs sum up the convolution results of Q input planes for $N \times m \times m$ output pixels along M depth. Considering implementation frequency f, the computation latency for each iteration is:

$$t_{comp} = \lceil ID/Q \rceil \lceil OD/M \rceil \lceil RS/m \rceil \lceil OW/(N \cdot m) \rceil / f \tag{5.10}$$

The overall latency is estimated as:

$$t_{loop} = \lceil OH/RS \rceil \cdot max(t_{comm}, t_{comp}) \tag{5.11}$$

Parameter exploration For the convenience of hardware implementation, we fix the batch size at $B = 2$. To guarantee the access of the planar data, we set H_b as 4 or 8 for F_4 or F_6 respectively and $W_b = \min 2^k$, s.t. $W_b \geq 2\omega$. Note here, the row step RS is a variable during the processing and is

chosen as large as possible so that the input and output rows can fully utilize the on-chip buffers. For a given CNN model, the M, N, Q, D_{in} and D_{out} are explored targeting $min(\sum_{l \in layers} t_{loop,l})$ with the given DSP and BRAM resources on the platform.

5.6 Evaluations

To validate the effectiveness of our design, we use Xilinx ZCU102 and Ultra96 boards for evaluation, where both platforms are equipped with a quad-core ARM Cortex-A53. The detailed resource specifications are shown in Table 5.2. We use Vivado HLS design suit 2019.2 for accelerator implementation using C++.

5.6.1 WinoPE Evaluation

Resource effectiveness We first compare the resource utilization of our WinoPE with the PEs without multiple kernel support, as shown in Table 5.1. All PEs are configured with $Q = 2$ and $B = 2$. The same DSP utilization in each PE type ensures that the maximum parallelism of the PEs is the same. Using the same amount of DSP resources, our WinoPE consumes more LUT and FF resources than each dedicated PE but with the benefit of supporting different convolution kernel sizes without effecting the runtime efficiency.

Table 5.1: Resource utilization of different PEs

PE type	LUT	FF	DSP	PE type	LUT	FF	DSP
$F_4(2 \times 2, 3 \times 3)$	5328	2430	128	$F_6(4 \times 4, 3 \times 3)$	21542	19235	288
$F_4(4 \times 4, 1 \times 1)$	6495	9831	128	$F_6(6 \times 6, 1 \times 1)$	24056	39126	288
WinoPE-F_4	7852	10501	128	WinoPE-F_6	33959	42793	288

Performance effectiveness Since the DSP efficiency without efficient data supply will be lower than the theoretical performance, we evaluate our WinoPE together with our memory subsystem and compare the results to other PEs theoretical performance with the assumption of the data supply is perfect. We first conduct experiments of synthetic convolution layers with

different kernel sizes and compare them to the theoretical performance values using the same configuration as shown in Figure 5.10. We measure the DSP efficiency to exclude the impact of different platforms with a system frequency at 100 MHz. The DSP efficiency of WinoPEs for different kernel sizes is measured on board, and the maximum performance for other PEs are calculated theoretically; both are shown in Figure 5.10. Compared to the theoretical performance of F_4 and F_6, our implementations of WinoPE-F_4 and WinoPE-F_6 under all kernel sizes achieve near-maximal theoretical performance with the proposed memory subsystem.

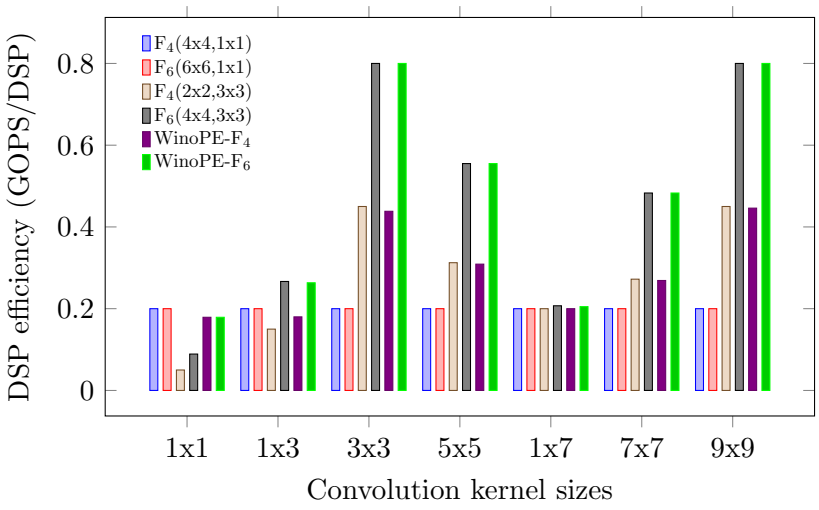


Figure 5.10: DSP efficiency of WinoPE to theoretical values

5.6.2 WinoCNN Evaluation

We adopt the most representative CNN models as benchmarks to demonstrate the effectiveness of our WinoCNN system design, including VGG-16, Inception-V4 (denoted as INet-V4), and YoloV2. The non-convolution layers are executed in the processors with multi-thread optimization for end-to-end model execution. All convolutional layers are executed on the WinoCNN accelerator.

WinoCNN configuration We explore the optimal WinoCNN system configurations for different platforms using our analytical model. The selected values are shown in Table 5.2 together with resource utilization and runtime performance on the different platforms for different models.

Table 5.2: WinoCNN configuration and performance on different platforms

Platform		Ultra96 (WinoPE-F_4)	ZCU102 (WinoPE-F_4)	ZCU102 (WinoPE-F_6)
PE Config.	M	2	8	4
	N	1	2	2
	Q	4	4	4
	D_{in}	4096	8192	4096
	D_{out}	1024	1024	1024
Resource Util. %	DSP	77.8(360)	82.8(2520)	93(2520)
	BRAM	85.9(432)	95.5(1824)	87(1824)
	LUT	60.8(70K)	76.3(274K)	81(274K)
	FF	43.2(141K)	43.4(548K)	48(548K)
	Freq. (MHz)	250	250	214
Throughput (GOPs)	VGG-16	265	1862	3120.3
	INet-V4	127.2	820.3	857.2
	YoloV2	157	1241	1717.7

The WinoCNN accelerator configurations for different platforms and different Winograd kernel sizes vary significantly because of the different DSP and BRAM capacity of the platforms, where all configurations target to fully utilize the on-chip DSP and BRAM resources. The achievable frequency under each configuration for a certain platform is also shown together with the final performance. Our WinoCNN system naturally supports better timing due to the timing-friendly shorter data path between the WinoPEs. Notably, for the networks with homogeneous convolutional layers, i.e., VGG-16, our design achieves 3.12 TOPS throughput at 214 MHz clock frequency while the performance drops to 857.23 GOPS when there are multiple divergent convolutional layer configurations in Inception-V4, i.e., 1×7 kernel. This is because of the varied efficiency of the Winograd algorithm for different convolution kernel sizes.

Comparison with state-of-the-art designs We then measure the execution latency, throughput, and DSP efficiency of our implemented models and compare them with the state-of-the-art implementations, as shown in Table 5.3.

Since all the convolution layers are executed by our WinoCNN accelerator, the DSP efficiency and latency data are calculated for the convolution layers.

Table 5.3: Comparison with state-of-the-art designs

	Platform	Model	Freq. (MHz)	Prec.	Batch size	DSP Usage	Thro. (GOPS)	Lat. (ms)	DSP effi. (GOPS/DSP)
[66]	ZCU102	VGG-16	200	16 bit fixed	32	2520	2601.3[1]	10.43[1]	1.03[1]
[61]	Arria10 GT1150	VGG-16	232	8-16 bit fixed	-	1500	1171.3	26.85	0.780
[58]	Stratix V GSMD8	Resnet-18	160	16 bit fixed	-	576	233	7.23	0.405
[59]	XCVU095	VGG-16	150	8-16 bit fixed	-	768	460	-	0.599
[67]	Arria-10	VGG-16	250	16 bit fixed	-	1344	1642	-	1.22
[68]	ZCU102	VGG-16	281	8 bit	2	1926	1225.2	57.53	0.636
		INet-V4					1390	35.26	0.722
		YoloV2					1008	16	0.523
Ours.	ZCU102	VGG-16	214	8-16 bit	2	2345	**3120.3**	**19.67**	**1.33**
		INet-V4					857.23	49.7	0.388
		YoloV2					**1717.7**	**13.9**	**0.73**

DSP efficiency of our design is 1.71× of the design in [61], which does not use Winograd transformation. When compared to the designs with Winograd algorithm [59] [66] together with additional model-specific optimizations, our design shows a 1.2× and a 6.78× improvement of throughput compared to that of [66] and [59], respectively. Notably, the design in [66] adopts a 32 batch size for FC layers, which is much larger than ours (fixed at 2) and leads to a long latency for a single image to be processed completely. In the comparisons, all the previous architectures containing model-specific designs can not support flexible kernel sizes, while our WinoCNN supports multiple convolution kernels without effecting the DSP efficiency. Our design also provides slightly better achievable frequency due to the efficient systolic array architecture on Xilinx platforms.

When comparing to the Vitis-AI implementations [68], our WinoCNN shows better throughput and latency for both VGG-16 and YoloV2 even with a lower clock frequency and without DSP double pumping. For the Inception-V4 model which contains unique kernel shapes, i.e., 1×7, 7×1, 3×1 and 1×3, we use the less efficient $F(4 \times 4, 1 \times 1)$ or $F(6 \times 6, 1 \times 1)$ to process them, which lead to a worse performance than the specially optimized Vitis-AI processing cores.

Also, we compare the performance between Google's Coral edge TPU USB accelerator and our implementation. Table 5.4 shows the board parameter comparison between our test platform and the edge TPU. We use the maximum achievable computation throughput expressed in TOPs to denote the computation capability of the device. For edge TPU, this throughput represents the maximum 8-bit integer operations that can be achieved per-second. For FPGA device, this throughput is calculated by the on-board DSP num-

bers and the achievable frequency of the proposed WinCNN architecture. The edge TPU has a larger computation capability than our FPGA platform due to its high working frequency and customized design. Table 5.5 lists the performance comparison of VGG-16 and Inception-V4 models on edge TPU device and the FPGA. For VGG-16 DNN model, both our solutions perform better than the edge TPU with a 15.06× speed up for ZCU102 and a 1.28× speed up for Ultra96 in terms of inference latency. For Inception-V4 DNN model, our zcu102 solution still performs 1.74× faster than the TPU solution. However, due to the lower computation capability, our Ultra96 solution has a 3.87× slower performance than the edge TPU solution.

Table 5.4: Board parameter comparison between FPGA and edge TPU

Device	Frequency	Computation Capability
Edge TPU	500 MHz	4 TOPs
ZCU102	214 MHz	1.095 TOPs
Ultra96	214 MHz	0.154 TOPs

Table 5.5: Performance comparison between FPGA and edge TPU

Model	Device	batch size	Freq. (Mhz)	Prec.	Lat. (ms)	Speed Comp.
VGG-16	Edge TPU	2	500 MHz	8-bit	296.2	baseline
	ZCU102	2	214 MHz	8-bit	19.67	15.06×
	Ultra96	2	214 MHz	8-bit	231.6	1.28×
INet-V4	Edge TPU	2	500 MHz	8-bit	86.5	baseline
	ZCU102	2	214 MHz	8-bit	49.7	1.74×
	Ultra96	2	214 MHz	8-bit	334.9	0.258x

5.7 Conclusion

In this chapter, we present a systolic array-based convolution accelerator design targeting the Winograd algorithm. Our accelerator, WinoCNN, is constructed by unique Winograd convolution PEs (WinoPE) which support flexible convolution kernel sizes without sacrificing DSP efficiency. WinoCNN also has an efficient memory subsystem that is suitable for planar data access for the array of WinoPEs. Our accelerator system is configurable for different

FPGA platforms with accurate resource and performance models. Overall, our accelerator delivers high throughput and state-of-the-art DSP efficiency comparing with previous accelerator implementations.

CHAPTER 6

HIKONV: HIGH THROUGHPUT QUANTIZED CONVOLUTION WITH NOVEL BIT-WISE MANAGEMENT AND COMPUTATION

Quantization for Convolutional Neural Network (CNN) has shown significant progress with the intention of reducing the cost of computation and storage with low-bitwidth data inputs. There are, however, no systematic studies on how an existing full-bitwidth processing unit, such as CPUs and DSPs, can be better utilized to carry out significantly higher computation throughput for convolution under various quantized bitwidths. In this study, we propose HiKonv, a unified solution that maximizes the compute throughput of a given underlying processing unit to process low-bitwidth quantized data inputs through novel bit-wise parallel computation. We establish theoretical performance bounds using a full-bitwidth multiplier for highly parallelized low-bitwidth convolution, and demonstrate new breakthroughs for high-performance computing in this critical domain. For example, a single 32-bit processing unit can deliver 128 binarized convolution operations (multiplications and additions) under one CPU instruction, and a single 27×18 DSP core can deliver eight convolution operations with 4-bit inputs in one cycle. We demonstrate the effectiveness of HiKonv on CPU and FPGA for both convolutional layers or a complete DNN model. For a convolutional layer quantized to 4-bit, HiKonv achieves a 3.17× latency improvement over the baseline implementation using C++ on CPU. Compared to the DAC-SDC 2020 champion model for FPGA, HiKonv achieves a 2.37× throughput improvement and 2.61× DSP efficiency improvement, respectively.

6.1 Introduction

Quantization is a frequently used technique in hardware implementation of Deep Neural Network (DNN) models in order to reduce both the memory

consumption and execution time [69, 70, 71, 4, 12, 52]. It is typically done by approximating high-precision floating-point numbers to low-bitwidth integers or fixed-point numbers. This is particularly important for modern DNN models as many of them employ convolutional layers, which contain intensive multiplication and accumulation (MAC) operations [53, 12, 52, 55]. Therefore, many novel quantization methods have been proposed in the literature to reduce the precision of weights, activations, and even gradients for DNNs while retaining their high accuracy [70, 71, 52, 72].

The current hardware implementation of quantized DNNs is, however, not ideal as there is no general support for quantized MACs without changing the underlying hardware [73, 74, 75, 76, 77, 78]. Most hardware units have a high-bitwidth (such as 32 or 64 bits) MAC for either floating-point numbers or integers [79]. When they are used for quantized MACs, most of the bitwidths are left underutilized, wasting precious computing resources. Even with the 8-bit multi-data processing of the Advanced Vector Extensions (AVX) support in X86_64 architecture, processing a single 4-bit multiplication would still occupy the 8-bit data width with the remaining 4 bits simply wasted [80]. The waste becomes even more severe when either processing lower bitwidth (such as binary) data or utilizing a hardware unit with higher built-in bitwidths.

Reconfigurable hardware such as FPGA may alleviate some of the waste because of its bit-level flexibility for configuration, but it exhibits similar drawbacks, especially for FPGAs with high-precision Digital Signal Processing (DSP) units [79, 55, 51]. Without a careful bit-wise management of inputs and outputs, deploying quantized DNNs onto FPGAs with the given DSPs would still waste much of their computation capacity.

In this work, we propose a novel solution, HiKonv, that can significantly improve the existing arithmetic units' utilization efficiency when conducting quantized convolution, thus improving throughput for convolution and reducing end-to-end DNN computation latency. Our solution is based on a careful management of bitwidths used for quantized MACs and novel mapping of multiple parallelized MACs onto an existing arithmetic unit, such that the arithmetic unit's computation capacity is fully utilized. We further show theoretically that such a management and mapping strategy is universal in the sense that it can be applied to arbitrarily quantized bitwidths and high-bitwidth arithmetic units. For example, a single 32-bit processing unit

can deliver 128 binarized convolution operations using one instruction for CPU, and a single 27×18 DSP core can deliver eight convolution operations with 4-bit inputs in one cycle. Based on such a theoretical analysis, we show that there are different optimal design points in choosing the quantization bitwidth for a given arithmetic processing unit. Our experimental results further validate our analysis and HiKonv's general applicability. For example, our CPU-based implementation of HiKonv achieves up to 3.17× latency improvement for quantized convolution over existing methods on the same CPU. We also apply HiKonv to an end-to-end quantized DNN model, UltraNet [81], in an FPGA setting, and the measured on-board result outperforms the state-of-the-art in terms of throughput and DSP efficiency by 2.37× and 2.61×, respectively.

Because of its generality, we believe HiKonv opens up a new venue for further improving the hardware efficiency of DNN based inferences. It not only improves the throughput and latency for existing quantized DNN models on existing hardware, but also offers new opportunities for designing new hardware-friendly quantized DNN models or co-designing both the hardware and quantized DNN models.

6.2 Preliminary

Before we present our proposed HiKonv solution, we first review (1) input slicing and data packing for concurrency improvement and (2) 1D convolution.

6.2.1 Input-Slicing for Concurrency Improvement

Input slicing and data packing are generally used by the current hardware units to process low-bitwidth inputs [82, 83, 80]. The input bitwidth is split into different pieces, each of which is called a slice to hold a low-bitwidth data. It uses the available bitwidth space to hold a number of data slices to improve the parallelism while still preserving the correct output. An example of INT4 optimization for Xilinx DSP48E2 unit is shown in Figure 6.1, each of the input contains two slices. It takes advantage of the multiple input ports and the internal addition operation of the DSP to enable four multiplications

of data slices simultaneously.

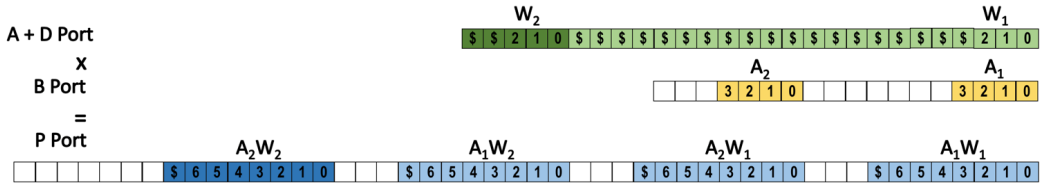


Figure 6.1: INT4 optimization on DSP48E2 [82]

Since an INT4-UINT4 multiplication generates a result that needs at least an 8-bit space, guard bits are added during the packing of the low-bitwidth data to guarantee the correctness of the result. Here, we define the term *guard bit* as the filling 1s or 0s between the packed data in the multiplicand for the purpose of preventing computation overflow. The multiplication with the sliced and packed inputs is represented as:

$$\begin{aligned}&(A_2 \cdot 2^{11} + A_1) \cdot (W_2 \cdot 2^{22} + W_1) \\ &= A_2W_2 \cdot 2^{33} + A_1W_2 \cdot 2^{22} + A_2W_1 \cdot 2^{11} + A_1W_1\end{aligned} \tag{6.1}$$

The output of Equation 6.1 is the concatenation of four multiplication results with zeros between them due to the guard bits. This process accomplishes four multiplications within one operation cycle.

6.2.2 1D Convolution

The conventional 1D discrete convolution between an N-element sequence f and a K-element kernel g (denoted as $F_{N,K}(f, g)$) can be represented as Equation 6.3. Here, we define the infinite length sequence h as the zero extension of f with the index range of $(-\infty, \infty)$. Meanwhile, y is the output with $N + K - 1$ non-zero elements. Alternatively, y can be represented as an $(N + K - 1)$-element sequence with Equation 6.4. Each of the $y[m]$ involves a sequence of multiplication and addition operations.

$$h[n] = \begin{cases} f[n] & , 0 \leq n < N \\ 0 & , n < 0 \text{ or } n \geq N \end{cases} \tag{6.2}$$

$$y[m] = (h * g)[m] = \sum_{k=0}^{K-1} h[m-k]g[k] \tag{6.3}$$

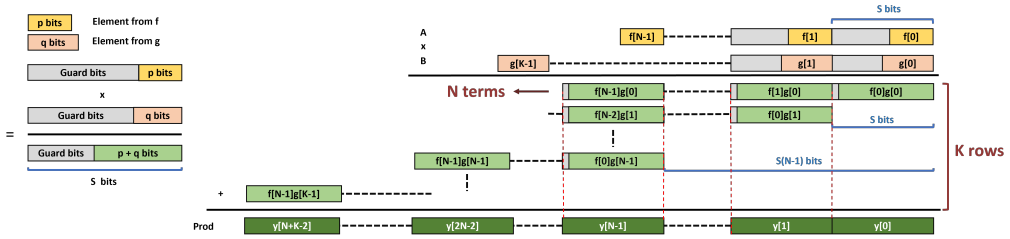


Figure 6.2: Ideal process of $Prod = A \cdot B$

$$y[m] = \sum_{k+n=m} h[n]g[k] \tag{6.4}$$

We may also derive the total convolution operation number for both the multiplications and the accumulations in each $F_{N,K}(f, g)$ convolution. For the multiplication, there are total K rows of N product terms summed up vertically to form the elements in output sequence y as illustrated by the light green blocks in Figure 6.2. So the total number of multiplications performed in the 1D convolution is $N \cdot K$. Meanwhile, each column of product terms are summed up vertically to form one element in the output sequence and the number of columns equals the length of the output sequence, which is $N+K-1$. Each product term is associated with one accumulation operation except for the product term at the bottom of each column. So the total number of accumulations equals the number of product terms minus the number of accumulation columns. Then we may conclude that the total convolution operation number in one $F_{N,K}(f, g)$ convolution is shown in Equation 6.5.

$$\begin{aligned} \#\text{ of Ops} &= \#\text{ of Multiplications} + \#\text{ of Accumulations} \\ &= N \cdot K + (N \cdot K - (N + K - 1)) \\ &= N \cdot K + (N - 1) \cdot (K - 1) \end{aligned} \tag{6.5}$$

6.3 Multiplier for Convolution

Inspired by input slicing and data packing for novel bit management and high processing concurrency, we generalize the solution for using a given hardware unit to process the maximum amount of low-bitwidth convolution operations concurrently with theoretical guarantees.

We first define the variables related to our exploration. As shown in Figure 6.2, we assume a given high-precision hardware unit that can multiply

Bit_A-bit integer input A with Bit_B-bit integer input B and generate the product $Prod$. The bitwidths of A and B define the computation capability of the hardware unit, or more specifically, the multiplier, and thus determine the design setting of HiKonv specific to this unit. Convolution input f and kernel g are the two sequences of low-bitwidth integer values quantized to p and q bits, respectively.

To determine how to load A and B with multiple convolution operands from f and g and perform the convolution between these operands, we define an additional variable S to be the size of a slice of inputs for both A and B as demonstrated on the left in Figure 6.2. The lower bits of each slice contain one operand from f or g. To simplify the problem, we assume the N and K are the maximum numbers of operands from f and g that can fit into A and B, respectively. Hence, the polynomial representations of A and B are:

$$A = \sum_{n=0}^{N-1} f[n] \cdot 2^{S \cdot n}, \quad B = \sum_{k=0}^{K-1} g[k] \cdot 2^{S \cdot k} \tag{6.6}$$

Although the intermediate results of the multiplication are invisible to us, we assume the processing unit takes the most ideal way for the multiplication of two inputs, as shown in Figure 6.2. The entire multiplication is treated as the multiplication of slices in A with the corresponding slices in B followed by shifting the product left by S bits and accumulating the shifted results to the previous result. There are always $N \cdot K$ products between elements from f and g that are computed and accumulated to form the output $Prod$.

6.3.1 From Multiplication to Convolution

To use the product $Prod = A \cdot B$, we need to segment the output into an effective format for convolution during the process. In order to segment the intermediate results, we extend the guard bits G_b in [82]. The guard bits are not only to prevent overlaps between the effective product of two adjacent intermediate partial products but also to segment out the partial accumulations of vertically stacked segments. Its length varies according to the maximum number of multiplication terms $f[n]g[k]$ that are summed together. According to our settings for A and B, a maximum of $min(K, N)$ terms are summed together for each output segments. Therefore, to ensure

the correctness of the final result, each of the slicing should contain both the guard bits $G_b = \lceil log_2 min(K, N) \rceil$ and the bits of the p-bit and q-bit inputs for the production, respectively.

Theorem 1. *Assuming guard bits, G_b, are properly decided according to the specific multiplier setting, with given A and B constructed from N-element sequence f and K-element sequence g, where f and g are quantized respectively to p and q bits, we can obtain $N + K - 1$ segments from the product $Prod = A \cdot B$ which are all short partial convolutions in the form of 1D convolution.*

Proof. Considering the guard bits, we can obtain:

$$S = \begin{cases} q + G_b, & p = 1, q \geq 1 \\ p + G_b, & q = 1, p \geq 1 \\ p + q + G_b, & otherwise \end{cases} \tag{6.7}$$

$$p + (N-1)S \leq Bit_A \tag{6.8}$$

$$q + (K-1)S \leq Bit_B \tag{6.9}$$

Thereby, the intermediate stages are shifted left by S bits for every stage, and the effective vertical accumulation of the partial products in the segments from all the stages stacked together would not exceed the length of S bits, as shown in Figure 6.2. Then, the multiplication is represented as:

$$\begin{aligned} Prod = A \cdot B &= (\sum_{n=0}^{N-1} f[n] \cdot 2^{S \cdot n}) \cdot (\sum_{k=0}^{K-1} g[k] \cdot 2^{S \cdot k}) \\ &= \sum_{m=0}^{N+K-2} (\sum_{n+k=m} (f[n] \cdot 2^{S \cdot n} \cdot g[k] \cdot 2^{S \cdot k})) \\ &= \sum_{m=0}^{N+K-2} (\sum_{n+k=m} (f[n] \cdot g[k]) \cdot 2^{S \cdot m}) \end{aligned} \tag{6.10}$$

Different from general multiplications, convolution consists of a sequence of multiplications and accumulations. Referring to the form of 1D convolution in Equation 6.4, the result of $Prod$ can be represented as:

$$Prod = \sum_{m=0}^{N+K-2} y[m] \cdot 2^{S \cdot m} \tag{6.11}$$

where the intermediate accumulations form a 1D convolution of two sequences in each of the output segments, and the total number of convolution segments is $N + K - 1$. □

Per the above, we can use a high-bitwidth multiplier to process two integers A and B to form $N + K - 1$ convolutions of short sequences. However, due to the two's complement representation of signed values, directly packing negative values into A or B leads to wrong results for the intermediate products. To guarantee the correctness of the products as the intermediate results, we must consider the sign bit during the packing of the elements from f and g into A and B as well as segmenting the result $Prod$.

If f and g are all unsigned integers, we can construct A and B as integers with bit-wise assignments and the zero extension without additional operations:

$$\begin{aligned} A[S(n+1)-1:Sn] &= f[n] \\ B[S(k+1)-1:Sk] &= g[k] \end{aligned} \tag{6.12}$$

Meanwhile, each $y[m]$ can be segmented out from $Prod$ with:

$$y[m] = Prod[S(m+1)-1:Sm] \tag{6.13}$$

However, if f and g contain signed integers, we need additional bit management.

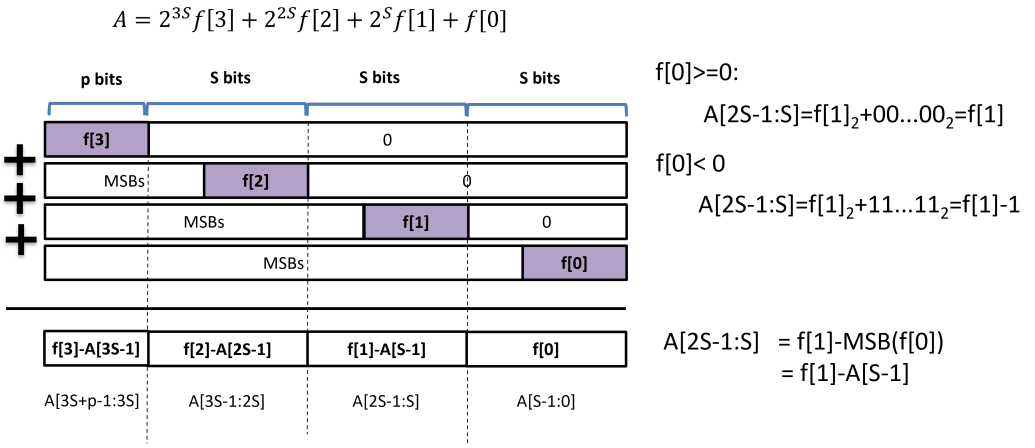


Figure 6.3: Input packing for signed integer f sequence

Figure 6.3 shows the packing of four elements of f into multiplicand A. Taking the second S-bit segment as an instance, in 2's complement expression, if $f[0]$ is positive, the MSB is 0, and the sign extension part are all zeros.

On the other hand, if $f[0]$ is negative, the sign extension part are all 1s in binary expression and represents -1 in 2's complementary representation. In such condition, we decrement 1 from $f[1]$ to form the second S-bit and perform the packing process with concatenation and 1-bit incrementer instead of using a larger bitwidth adder. The packing process works recursively for all the slices while slicing of the output works in a reversed manner. Equation 6.14 shows the packing and slicing formula for signed integer f and g. With this bit management technique, we obtain $N + K - 1$ partial convolutions from $N \cdot K$ segments of intermediate results for a single multiplier to process signed input data.

$$
\begin{aligned}
A[S(n+1)-1:Sn] &= \begin{cases} f[0] & ,n=0 \\ f[n]-A[Sn-1] & ,n>0 \end{cases} \\
B[S(k+1)-1:Sk] &= \begin{cases} g[0] & ,k=0 \\ g[k]-B[Sk-1] & ,k>0 \end{cases} \\
y[m] &= \begin{cases} Prod[S-1:0] & ,m=0 \\ Prod[S(m+1)-1:Sm]+Prod[Sm-1] & ,m>0 \end{cases}
\end{aligned}
\tag{6.14}
$$

6.3.2 Convolution Extension

Now we have presented an efficient algorithm to use the multiplication unit on a hardware platform to perform the $F_{N,K}$ 1D convolution. However, the size of N are limited by the bitwidth of the hardware multiplier whereas most real-world applications have much larger input sizes. Moreover, the $F_{N,K}$ 1D convolution is often used as a unit building block for other larger-scale convolution operations. Thus, we design a new algorithm to use the $F_{N,K}$ 1D convolution to complete arbitrarily large size 1D convolutions and any arbitrary convolutions. As shown in Figure 6.2, the order of the elements for these intermediate production is controlled by the order of elements packed into the slices in A and B; it allows us to devise different accumulation methods to provide flexibility to construct different convolutions beyond 1D convolution.

1D Convolution Extension Regarding the $F_{N,K}$ as a basic operation, we extend it to $F_{X \cdot N,K}$ convolution of a longer sequence by summing up the

elements in output sequences of different $F_{N,K}$ convolutions.

Theorem 2. *The output sequence $y = F_{X \cdot N,K}$ of a 1D convolution between an $(X \cdot N)$-element sequence f and a K-element filter g can be represented as the sum of index-shifted output sequences $y_x = F_{N,K}(f_x, g)$, as shown in Equation 6.17. Here, $f_x = f[xN : (x+1)N - 1](x \in [0, X-1])$.*

Proof. Following Equation 6.2, we extend f and f_x sequences into zero extension sequences h and h_x. Then h is represented as the sum of index-shifted h_x sequence:

$$h[n] = \sum_{x=0}^{X-1} h_x[n - xN] \tag{6.15}$$

According to Equation 6.3, the convolution output y is calculated with:

$$\begin{aligned} y[n] &= \sum_{k=0}^{K-1} h[n-k]g[k] \\ &= \sum_{k=0}^{K-1} (\sum_{x=0}^{X-1} h_x[n - xN - k])g[k] \\ &= \sum_{x=0}^{X-1} (\sum_{k=0}^{K-1} h_x[n - xN - k]g[k]) \end{aligned} \tag{6.16}$$

Given that $y_x[n] = \sum_{k=0}^{K-1} h_x[n-k]g[k]$, we can represent sequence y as the sum of index-shifted y_x sequences.

$$y[n] = \sum_{x=0}^{X-1} y_x[n - xN] \tag{6.17}$$

□

Equation 6.17 reveals that the extended $F_{X \cdot N,K}$ 1D convolution is computed by a shift-accumulation pattern with $F_{N,K}$ base operation results. Figure 6.4 demonstrates how the elements in different y_x sum up to the elements in y. Each computed y_x sequence is shifted xN indices and then summed up to form the element of y, which is marked by the red square. Still, we use existing adder unit of the given platform to complete multiple additions by adding the bit slices from the product *Prod* as mentioned in Section 6.3.1 as marked by the blue square. In such a case, the guard bit of $G_b = \lceil log_2 K \rceil$ is also adjusted with additional bits to prevent the partial results from overflow.

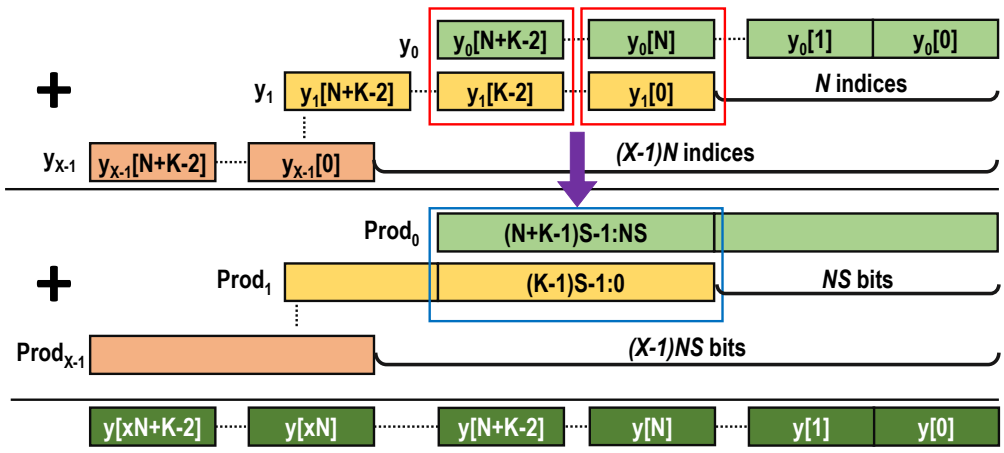


Figure 6.4: Computation of $F_{XN,K}$ 1D convolution

DNN Convolution The convolution layer in DNN computes a feature-map array $I[C_i][H_i][W_i]$ and a kernel array $WT[C_o][C_i][K][K]$ for output feature-map array $O[C_o][H_o][W_o]$ which can be represented as:

$$O[c_o][h][w]= \sum_{c_i=0}^{C_i-1} \sum_{k_h=0}^{K-1} \sum_{k_w=0}^{K-1} I[c_i][h+k_h][w+k_w]WT[c_o][c_i][k_h][k_w] \tag{6.18}$$

Here, we neglect the bias term and assume that the output feature-map is computed with non-padding convolution. For padded convolution, we may first merge the padding pixels into the original input feature-map to form a new feature-map and then perform the non-padding convolution. In such condition, the output feature-map size is related to the input feature-map size and kernel size as shown in Equation 6.19:

$$\begin{aligned} H_o &= H_i - K + 1 \\ W_o &= W_i - K + 1 \end{aligned} \tag{6.19}$$

With the inherent convolution computation pattern, we may also compute a DNN convolution layer with $F_{N,K}$ 1D convolution as the base operation, as shown in Theorem 3.

Theorem 3. *For a DNN convolution, the output feature-map can be computed by $F_{X\cdot N,K}$ 1D convolution with the following equation:*

$$O[c_o][h][w] = \sum_{c_i=0}^{C_i-1} \sum_{k_h=0}^{K-1} y_{c_i,c_o,h,k_h}[w+K-1] \tag{6.20}$$

Here, the term y_{c_i,c_o,h,k_h} is a 1D convolution result with $X = \lceil \frac{W_i}{N} \rceil - 1$:

$$y_{c_i,c_o,h,k_h} = F_{X \cdot N,K}(f, g) \tag{6.21}$$

where f and g are defined as:

$$\begin{aligned} f[w] &= \begin{cases} I[c_i][h + k_h][w], & 0 \le h < H_i, 0 \le w < W_i \\ 0, & otherwise \end{cases} \\ g &= WT[c_o][c_i][k_h][K - 1 : 0] \end{aligned} \tag{6.22}$$

Proof. For abbreviation, we denote sequence y_{c_i,c_o,h,k_h} as y'. According to the definition of 1D convolution, sequence y' can be computed by Equation 6.23. Here, we use k as the summation index counter and the notation of h, c_i, c_o, k_h and k_w are the same as the ones in Equation 6.18.

$$\begin{aligned} y'[n] &= \sum_{k=0}^{K-1} f[n\text{-}k]g[k] \\ &= \sum_{k=0}^{K\text{-}1} I[c_i][h{+}k_h][n\text{-}k]WT[c_o][c_i][k_h][K\text{-}1\text{-}k] \\ &= \sum_{k=0}^{K\text{-}1} I[c_i][h{+}k_h][n{+}(K\text{-}1\text{-}k)\text{-}K{+}1]WT[c_o][c_i][k_h][(K\text{-}1\text{-}k)] \\ &= \sum_{(K\text{-}1\text{-}k)=K\text{-}1}^{0} I[c_i][h{+}k_h][n{+}(K\text{-}1\text{-}k)\text{-}K{+}1]WT[c_o][c_i][k_h][(K\text{-}1\text{-}k)] \\ &= \sum_{k_w=K\text{-}1}^{0} I[c_i][h{+}k_h][n{+}k_w\text{-}K{+}1]WT[c_o][c_i][k_h][k_w] \\ &= \sum_{k_w=0}^{K\text{-}1} I[c_i][h{+}k_h][n{+}k_w\text{-}K{+}1]WT[c_o][c_i][k_h][k_w] \end{aligned} \tag{6.23}$$

Then by replacing n with $w + K - 1$ in Equation 6.23, we obtain Equation 6.24.

$$\begin{aligned} y'[w{+}K\text{-}1] &= \sum_{k_w=0}^{K\text{-}1} I[c_i][h{+}k_h][w{+}k_w]WT[c_o][c_i][k_h][k_w] \\ y_{c_i,c_o,h,k_h}[w{+}K\text{-}1] &= \sum_{k_w=0}^{K\text{-}1} I[c_i][h{+}k_h][w{+}k_w]WT[c_o][c_i][k_h][k_w] \end{aligned} \tag{6.24}$$

With Equation 6.24, Equation 6.18 could be represented as:

$$
\begin{aligned}
O[c_o][h][w] &= \sum_{c_i=0}^{C_i\text{-}1} \sum_{k_h=0}^{K\text{-}1} \sum_{k_w=0}^{K\text{-}1} I[c_i][h+k_h][w+k_w] WT[c_o][c_i][k_h][k_w] \\
&= \sum_{c_i=0}^{C_i\text{-}1} \sum_{k_h=0}^{K\text{-}1} \left(\sum_{k_w=0}^{K\text{-}1} I[c_i][h+k_h][w+k_w] WT[c_o][c_i][k_h][k_w] \right) \\
&= \sum_{c_i=0}^{C_i\text{-}1} \sum_{k_h=0}^{K\text{-}1} y_{c_i,c_o,h,k_h}[w+K\text{-}1]
\end{aligned} \tag{6.25}
$$

□

A convolution layer in DNN has multiple input and output channels, which require accumulation of channel-wise features to form the final output. By grouping the $F_{N,K}$ output sequences with different c_i but same c_o,h,k_h and x indices, and accumulating the corresponding product $Prod$ with adders, we can perform the channel-wise accumulation of the feature-maps. In this case, the required number of guard bits is $G_b = \lceil log_2(M \cdot min(K, N)) \rceil$ for the accumulation of M feature-maps along input channel in a convolution.

Specifically, in the condition of DNN applications, the weight data can be pre-processed prior to the inference. A certain $K \times K$ DNN convolution filter may be packed into K multiplicands following the Equation 6.12 and Equation 6.14. Furthermore, the pre-processed weight multiplicands may be compressed by discarding the guard bits which are always all 1s or all 0s as shown in Figure 6.5. The compressed multiplicands can be decompressed by filling up the guard bits without introducing additional computation resources. Table 6.1 shows the number of LUTs saved by pre-processing the weight under different weight bitwidths.

Table 6.1: LUTs reduction with pre-processed weight

Weight Bits	1	2	3	4	5	6	7	8
LUTs	226	100	78	21	57	26	29	32

6.3.3 Throughput Analysis

Based on the above discussions, the equivalent achievable throughput for convolution of inputs with p and q bits quantized data by a given processing

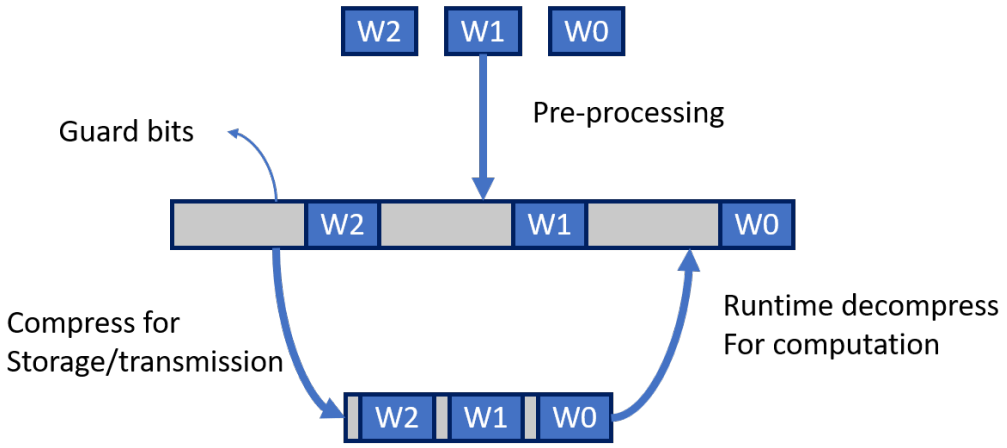


Figure 6.5: Weight pre-processing and compression

unit is a function of both the supported bitwidth of A and B by the hardware and the given bitwidth of the elements in f and g.

According to Equation 6.5 in Section 6.2.2, the total convolution operation number in a $F_{N,K}$ 1D convolution is $N \cdot K + (N-1)(K-1)$. For each set of convolution and multiplier configuration parameters including p, q, Bit_A and Bit_B, we can derive the N, K pairs that may achieve the maximum number of equivalent convolution operations performed by each multiplication within the constraint specified by Equation 6.8 and Equation 6.9. Also, in the single multiplier condition, we have the $G_b = \lceil log_2 min(K, N) \rceil$ as discussed before. In order to maximize the effective number of convolution operations for HiKonv, we formulate it as a discrete optimization problem as follows.

$$\begin{aligned} &maximize\ N \cdot K + (N-1) \cdot (K-1) \\ &\text{subject to } 0 < N \\ &\qquad 0 < K \\ &\qquad p + (N-1)(p + q + \lceil log_2 min(K, N) \rceil) \leq Bit_A \\ &\qquad q + (K-1)(p + q + \lceil log_2 min(K, N) \rceil) \leq Bit_B \end{aligned} \tag{6.26}$$

We may easily conclude the upper bound of the N and K as:

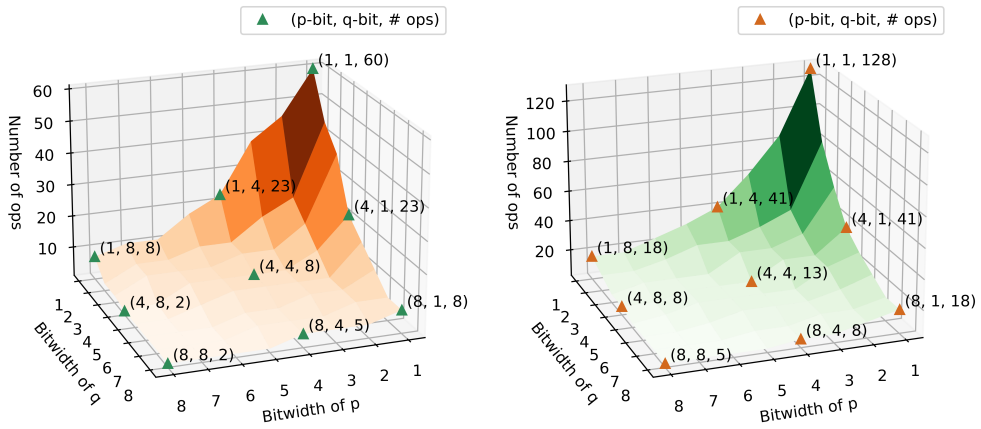
$$\begin{aligned} N &\leq \frac{Bit_A - p}{p + q + \lceil log_2 min(K, N) \rceil} + 1 < \frac{Bit_A - p}{p + q} + 1 \\ K &\leq \frac{Bit_B - q}{p + q + \lceil log_2 min(K, N) \rceil} + 1 < \frac{Bit_B - q}{p + q} + 1 \end{aligned} \tag{6.27}$$

Then we may solve this optimization problem with a straightforward search algorithm shown in Algorithm 5 by iterating all the possible N, K pairs to find the optimal solution.

Algorithm 5 Optimal Throughput Search

$Max_N = (Bit_A - p)/(p+q) + 1, Opt_K = 0$
$Max_K = (Bit_B - q)/(p+q) + 1, Opt_N = 0$
$Max_{NK} = 0$
for $k = 1, k < Max_K, k$++ **do**
    for $n = 1, n < Max_N, n$++ **do**
        $Cond_1 = p + (n-1)(p+q+\lceil log_2 min(k,n) \rceil) \leq Bit_A$
        $Cond_2 = q + (k-1)(p+q+\lceil log_2 min(k,n) \rceil) \leq Bit_B$
        $Cur_{NK} = n \cdot k + (n-1) \cdot (k-1)$
        if $Cond_1$ & $Cond_2$ & $Cur_{NK} > Max_{NK}$ **then**
            $Opt_N = n, Opt_K = k, Max_{NK} = Cur_{NK}$
        end if
    end for
end for
Return Opt_N, Opt_K, Max_{NK}

Figure 6.6 shows two examples of multipliers with different bitwidth configurations. For a given high bitwidth processing unit and given p, q values, we may obtain the maximum supported throughput (highest number of effective multiplications and additions) for a processing unit following Algorithm 5. For instance, when the bitwidths of the two inputs of a multiplier are 27 and 18 bits (Figure 6.6a), according to Equations 6.7, 6.8, and 6.9 and the required guard bits, we could obtain $S = 4, N = 9, K = 4$ when the p and q are both 1-bit binary values. The maximum supported throughput for this set of parameters is equivalent to 60 ops per cycle, which include 36 multiplications and 24 additions that would be required for computing the 1D convolution if all the computation is carried out in a conventional way without HiKonv. With HiKonv, all we need is one multiplication by the high bitwidth multiplier with our specific slicing/packing solution. As another example, when p and q are both four bits, the multiplier provides 8 equivalent ops per cycle (6 multiplications and 2 additions). In Figure 6.6, we show the configurations and equivalent number of operations for p and q from 1-bit to 8-bit, which are the common bitwidths of low-precision quantization. The HiKonv design principle generally applies to all bitwidths. When the inputs for the multiplier are both 32 bits, these values are further increased to 128 ops per cycle and 13 ops per cycle for 1-bit and 4-bit p and q respectively, as shown in Figure 6.6b.



(a) A = 27 bits, B = 18 bits (b) A = 32 bits, B = 32 bits

Figure 6.6: Throughput of processing units with different bitwidth settings. The number of operations (z-axis) can be calculated by solving for an optimization problem specific to values of p, q, and the input bitwidths of the processing unit (e.g., the high-bitwidth multiplier)

6.4 Evaluations

HiKonv is a general technique that can be adopted for both the general purpose processor and reconfigurable hardware platform. We demonstrate its efficacy on both platforms.

6.4.1 General-purpose Processors

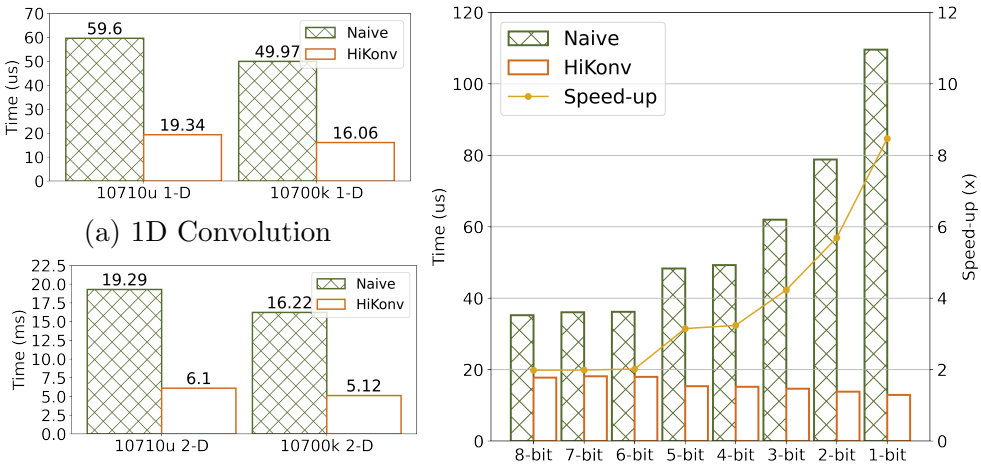
We first evaluate HiKonv on both CPU-based desktop and mobile platforms with an Intel Core i7-10700K CPU and i7-10710U CPU, respectively. We measure the performance of both 1D convolution and a DNN convolution layer. For 1D convolution, the baseline implementation has two-level nested loops. The outer loop scans through the input vector, whereas the inner loop scans through the kernel vector. We adopt the horizontal stacking strategy proposed by HiKonv 1D convolution. The features are packed during run-time, and kernels are packed offline before the processing starts. For the output, we first shift the previous partial result to the right by $S \cdot 2$ bits and the current partial result to the left by $S \cdot (N-2)$ bits. Then, we add them together to form the whole result for this loop. In the end, we take the last $S \cdot N$ bits as the N outputs with the corresponding indices.

For a quantified analysis with a DNN layer, we pick the final layer of Ul-

traNet [81], which is the champion model for the DAC-SDC contest 2020 and randomly generate feature and kernel vectors. We implement the DNN layer by embedding the 1D convolution algorithm into the six-level nested loops that scan through the input channel, output channel, output height, output width, kernel height, and kernel width according to Theorem 3. Since CPU hardware lacks bit-wise management capability, dealing with signed values can cause unnecessary overhead from intricate bit operations. While we can deploy the HiKonv solution with signed values, the hardware constraint makes such optimization less efficient than unsigned values. Since modern CPUs are equipped with 32-bit multipliers, without loss of generality, we use $A = B = 32$ bits as the multiplication bitwidth, and pack $p = q = 4$-bit unsigned values in each of the operands. According to Theorem 2, we obtain $N = 3, K = 3, G_b = 2$, and $S = 10$ bits. Figures 6.7a and 6.7b show the 1D and 2D convolution latency results, respectively. Both are compared to the baseline implementation with nested loops without our HiKonv solution.

It is clear that our HiKonv solution is about three times faster than the baseline implementations under all four combinations. The experimental results are slightly slower than the theoretical speed-up shown in Section 6.3 because of the processing overhead. Despite the reduction in loop counts and thus the total number of multiplications to generate all outputs, HiKonv has additional bit-shifting and gating operations to prepare the operands and segment the output. Since the ALU handles both multiplications and bit-wise operations, the latency of bit-wise operations are not much faster than multiplications and lead to extra overhead.

We further test the performance with low-precision bitwidths from one to eight bits. Assuming $p = q$, we calculate the corresponding N, K, and G_b and pack the quantized values into 32-bit accordingly. Figure 6.7c shows the result of the 1D convolution with the same setting as before. It is clear that when the bitwidth of the processed data reduces, the performance of our HiKonv solution increases because of the increased slice number. When the bitwidth is 1 bit, HiKonv solution provides a 8.6× performance improvement.



(a) 1D Convolution

(b) 2D Convolution

(c) Speed-up for different bitwidths

Figure 6.7: HiKonv evaluation on CPU

6.4.2 Reconfigurable Hardware

We also conduct the evaluation of our HiKonv solution on the Xilinx Ultra96 MPSoC platform, which is equipped with 360 DSP48E2 units and a quad-core ARM processor. Each of the DSP48E2 has one 27-bit, one 18-bit and one 45-bit input port. It can perform one $(M_0 \cdot M_1 + Acc)$ MAC operation in one clock cycle, where M_0, M_1, Acc are the inputs at the 27-bit, 18-bit, and 45-bit port, respectively. Different from software implementation of HiKonv for general-purpose processors, with reconfigurable hardware, the input packing is conducted with small adders for each of the slices, and output segmentation is conducted by bit-wise operations. These advantages on the hardware can fully benefit the performance of our HiKonv solution.

Binary convolution layer We first evaluate the extreme case of quantized convolution which is the Binary Neural Networks (BNN). A convolutional layer in a BNN takes the binary inputs for both feature-maps and kernel weights, processes the convolution between them, and generates the outputs. Note that the outputs may not be binary due to the channel-wise accumulation. We first implement a binary convolution layer with 4-bit outputs without using the DSP resources, denoted as BNN-LUT; we then replace the binary computations with our HiKonv solution with DSP, denoted as BNN-HiKonv. In comparison, we evaluate the resource utilization of these two designs under the setting of the same concurrency and same clock frequency,

as shown in Table 6.2.

Table 6.2: Comparison of resource util. of binary convolution

# of Concurrent MACs		336	576	960	1536	3072
BNN-LUT	LUT	3371	4987	7764	12078	23607
BNN-HiKonv	LUT	2672	2536	3369	3587	9319
	DSP	16	32	64	128	256
	DSP Thro.	21	18	15	12	12
	LUT/DSP	43.7	76.6	68.7	65.4	55.8

Clearly, compared to BNN-LUT, the LUT usage of BNN-HiKonv is reduced. However, the throughput for each DSP reduces when the concurrency increases due to the reason that there is more vertical stacking, and it takes more guard bits when the concurrency increases. The equivalent number of LUTs replaced by one DSP (LUT/DSP) varies from 43.7 to 76.6 due to the accumulation logic in the convolution operation. HiKonv creates opportunities to leverage DSPs for high-throughput BNN (or other low-bitwidth models) convolution computations that would help map a larger BNN with high concurrency into the same FPGA. It can also potentially increase the design's clock frequency since DSPs can run at a higher frequency than LUTs.

Complete model We apply our HiKonv solution to the entire UltraNet model [81] on the Xilinx Ultra96 MPSoC FPGA. The weight and activation of this model are quantized to 4-bit. We execute all the convolution layers on the programmable logic and the other layers on the ARM processor in the FPGA platform. We follow the same layer architecture and system architecture as the original UltraNet design and only change the computation for convolution with our HiKonv solution. Besides using DSPs, we also use small adders and shifters constructed by LUTs, taking advantage of the flexible configuration features of the FPGA.

In addition to resources utilization, we also measure the throughput in frame-per-second (fps) and calculate the DSP efficiency in terms of Giga-operations-per-second-per-DSP (Gops/DSP) for comparison as shown in Table 6.3. All the testing data is first loaded into the DDR to leverage the full capacity of the accelerators in our evaluations.

UltraNet-HiKonv uses more LUT resources than the original implementation due to the shifting and adding logic; however, it reduces the DSP

Table 6.3: UltraNet resource and performance

	LUT	DSP	fps	DSP Eff. (Gops/DSP)
UltraNet	4.3 K	360	248	0.289
UltraNet-HiKonv	4.8 K	327	401/588	0.514/0.753

utilization thanks to the dramatic improvements of the efficiency and the throughput of the DSPs. The original implementation of the UltraNet uses one DSP for two 4-bit MACs that is natively supported by the synthesis tool. It only achieves 248 fps with a 0.289 Gops/DSP efficiency. With our HiKonv solution, the on-board implementation of UltraNet achieves 401 fps with a 0.514 Gops/DSP DSP efficiency. This significant improvement is achieved under the constraint that the software execution on the ARM core is not fast enough to feed the input data to the FPGA accelerator to process, even with our best software optimization of multi-threading and data buffering. If this ARM core bottleneck is removed, the UltraNet-HiKonv accelerator can reach an even higher performance of 588 fps with the DSP efficiency of 0.753 Gops/DSP.

6.5 Related Works

Existing solutions for low-bitwidth arithmetic [84] build their own computation units based on the inputs [73, 74, 75, 76, 77, 78] and benefit from the control flexibility down to a single bit. Prior work for accelerating DNN inference has also incorporated low-bitwidth computations. Tensor processing units (TPUs) introduce 16-bit *bfloats* [85], and mobile GPUs and other edge devices now support 8-bit computations. However, these improvements focus only on homogeneous arithmetic requirements and do not allow flexible arithmetic computations with varied bitwidths. Therefore, when processing data with a bitwidth different from its targeted bitwidth, it either leaves some precision unused with wasted resources or hurts the efficiency of the overall process.

There are methods that simply pack short bitwidth values into longer words and attempt to incorporate additional computations using the existing unit through bit shifting and packing [80, 82, 86, 87, 88] to further improve processing efficiency. However, those studies are ad-hoc and do not

fully leverage the hardware's capability as HiKonv does. Moreover, there are no theoretical studies to guide the flexible management of low-bitwidth quantized data. Our work fills the gap of processing low-bitwidth data under theoretical guidance for the best computation efficiency and throughput on either existing hardware architecture or any bit-efficient processing units in the future.

6.6 Conclusion and Discussion

In this work, we present HiKonv, a general technique with theoretical guarantees for using a single multiplier unit to process multiple low-bitwidth convolution operations in parallel for significantly higher computation throughput with flexible bitwidths. It is able to support convolutions in DNNs and achieves the highest possible throughput for quantized convolution with novel bit-wise management and computation.

As a demonstration of its general applicability and benefits, we show that HiKonv has achieved 3.17× throughput improvement on CPU and 2.37× and 2.61× throughput and DSP efficiency improvements for the DAC-SDC 2020 champion model on FPGA. HiKonv suits for both software and hardware optimizations and provides new opportunities for future hardware designs for efficient DNN processing.

CHAPTER 7

CONCLUSION

In this concluding chapter we summarize the contributions of this thesis and the possible impact. The key topics explored in this thesis are the methods of delivering high-performance architecture for DNN applications on FPGA platforms with fixed and limited resources.

In Chapter 3, we demonstrate the detailed implementation of a recurrent neural network for video content recognition: LRCN. In this chapter, we introduce a novel resource allocation strategy, REALM, which provides a guideline for computation resource allocation for multi-layer DNNs. We design parameterized IPs for each layer in LRCN and configure the parameters for each layer following the REALM strategy. Finally, we use the Xilinx's VC709 board as the hardware platform to demonstrate the effectiveness of REALM and our HLS IP in the design process and achieve better performance than GPU and CPU solutions.

In Chapter 4, we propose a novel backward pipeline scheduling algorithm to schedule the computation tasks in DNN layers for DNN hardware IPs. Through the scheduling algorithm, we achieve deep pipelining among layers in the DNNs. Meanwhile, we provide the algorithm to balance the computation resources for DNN IPs to achieve optimal throughput and latency. We test our scheduling algorithm and DNN IPs on Xilinx ZYNQ-7000 SOC ZC706 and demonstrate better performance than GPU solutions.

In Chapter 5, we present a systolic array based convolution accelerator design working together with the Winograd algorithm. In this chapter, we propose a novel Winograd-based processing element, WinoPE, that supports flexible convolution kernel sizes with high DSP efficiency, following a resource sharing mechanism we explored in the Winograd algorithm. With the proposed WinoPE as the basic computation element, we implement a scalable systolic array-based accelerator WinoCNN. Also, we design a fine-grained and highly efficient memory control system to handle the complex and in-

tensive data access pattern for the WinoCNN architecture. In this work, our implementation demonstrates state-of-the-art performance and resource efficiency in different DNN models compared with previous work.

In Chapter 6, we propose a novel solution, Hikonv, that can dramatically improve the existing arithmetic units' utilization efficiency in quantized DNN applications. Our solution fully utilizes the computation capacity of existing arithmetic unit in hardware platforms by enabling parallelized MACs for low bit-width integers. Our experimental results further validate our analysis and HiKonv's general applicability. Both our CPU-based and FPGA-based implementations present drastic performance improvement on DNN application over the existing methods with the same testing platform.

After the summary of the specific contribution of each method, we may draw several main conclusions as follows:

- The central factor that decides the final performance of a hardware accelerator architecture is the average utilization rate of the on-chip computation resources. The computation tasks in DNN applications should be properly scheduled and assigned with balanced computation resources to minimize the total idle time of each computation unit.

- The high-performance DNN hardware accelerator architectures usually demand efficient data transmission management which includes the management of both the off-chip data loading/off-loading and the on-chip data transmission. The characteristics of on-chip memory elements and the data access patterns of the computation architecture need to be carefully considered to design the memory system that can fulfill the data requirement of a DNN accelerator.

- The DNN applications have specific computation patterns. Specifically, the convolution-like computation pattern allows us to boost up the performance efficiency of on-chip computation resources through algorithmic optimizations. This field is not fully explored and there are still potential opportunities for performance improvement.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[2] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Tech. Rep., April 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[3] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3D object detection network for autonomous driving," *Computing Research Repository*, vol. abs/1611.07759, 2016. [Online]. Available: http://arxiv.org/abs/1611.07759

[4] X. Zhang, C. Hao, H. Lu, J. Li, Y. Li, Y. Fan, K. Rupnow, J. Xiong, T. S. Huang, H. Shi, W. Hwu, and D. Chen, "Skynet: A champion model for DAC-SDC on low power object detection," *Computing Research Repository*, vol. abs/1906.10327, 2019. [Online]. Available: http://arxiv.org/abs/1906.10327

[5] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," *Computing Research Repository*, vol. abs/1503.03832, 2015. [Online]. Available: http://arxiv.org/abs/1503.03832

[6] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 160–167. [Online]. Available: https://doi.org/10.1145/1390156.1390177

[7] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 396–404. [Online]. Available: http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.pdf

[8] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "Imagenet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[10] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and F.-F. Li, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Computing Research Repository*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[12] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[13] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *Computing Research Repository*, vol. abs/1409.1556, 2015.

[14] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[15] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[16] J. H. Lee, Y. C. Hsu, and Y. L. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," in *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, 1989, pp. 20–23.

[17] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *International Conference on Artificial Intelligence and Statistics*, vol. 15, Apr 2011, pp. 315–323. [Online]. Available: https://proceedings.mlr.press/v15/glorot11a.html

[18] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Intl. Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[19] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network," in *International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[20] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[21] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *International Conference on Computer Design*, 2013, pp. 13–19.

[22] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *International Symposium on Field-Programmable Gate Arrays*, 2017, p. 45–54. [Online]. Available: https://doi.org/10.1145/3020078.3021736

[23] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 629–634.

[24] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *International Symposium on Field-Programmable Gate Arrays*, 2017, p. 75–84. [Online]. Available: https://doi.org/10.1145/3020078.3021745

[25] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," *Computing Research Repository*, vol. abs/1411.4389, 2014. [Online]. Available: http://arxiv.org/abs/1411.4389

[26] Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, C. Laurent, Y. Bengio, and A. C. Courville, "Towards end-to-end speech recognition with deep convolutional neural networks," *Computing Research Repository*, vol. abs/1701.02720, 2017. [Online]. Available: http://arxiv.org/abs/1701.02720

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 2014.

[28] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. Whaley, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Transaction on Mathematical Software*, 2002.

[29] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," 2014.

[30] Xilinx, "Vivado high-level synthesis." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[31] CALYPTO, "Catapult C synthesis." [Online]. Available: http://www.calypto.com/catapult-c-synthesis.php

[32] Altera, "OpenCL SDK." [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html

[33] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LEGUP: High-level synthesis for FPGA-based processor/accelerator systems," in *International Symposium on Field Programmable Gate Arrays*, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[34] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Symposium on Application Specific Processors*, July 2009, pp. 35–42.

[35] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and effective placement and routing directed high-level synthesis for FPGAs," in *International Symposium on Field-programmable Gate Arrays*, 2014, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554775

[36] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen, "Machine learning on FPGAs to face the IoT revolution," in *International Conference on Computer-Aided Design*, Nov 2017, pp. 819–826.

[37] Z. Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *Asia and South Pacific Design Automation Conference*, Jan 2016, pp. 218–225.

[38] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 247–257, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815993

[39] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional networks," in *International Conference on Field Programmable Logic and Applications*, 2009, pp. 32–37.

[40] N. Li, S. Takaki, Y. Tomiokay, and H. Kitazawa, "A multistage dataflow implementation of a deep convolutional neural network based on FPGA for high-speed object recognition," in *IEEE Southwest Symposium on Image Analysis and Interpretation*, March 2016, pp. 165–168.

[41] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *International Conference on Field Programmable Logic and Applications*, Sept 2017, pp. 1–4.

[42] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *International Conference on Field-Programmable Technology*, Dec 2011, pp. 1–8.

[43] G. Lucas, S. Cromar, and D. Chen, "Fastyield: Variation-aware, layout-driven simultaneous binding and module selection for performance yield optimization," in *Asia and South Pacific Design Automation Conference*, Jan 2009, pp. 61–66.

[44] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Asia and South Pacific Design Automation Conference*, Jan 2017, pp. 629–634.

[45] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021741

[46] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[47] L. B. Costa, S. Al-Kiswany, and M. Ripeanu, "GPU support for batch oriented workloads," in *IEEE Intl. Performance Computing and Communications Conference*, Dec 2009, pp. 231–238.

[48] S. Liu, A. Papakonstantinou, H. Wang, and D. Chen, "Real-time object tracking system on FPGAs," in *Symposium on Application Accelerators in High-Performance Computing*, July 2011, pp. 1–7.

[49] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," in *International Conference of Signal Processing and Intelligent Systems*, Dec 2016, pp. 1–6.

[50] Z. Liu, Y. Dou, J. Jiang, and J. Xu, "Automatic code generation of convolutional neural networks in FPGA implementation," in *International Conference on Field-Programmable Technology*, Dec 2016, pp. 61–68.

[51] D. Chen, J. Cong, S. Gurumani, W. Hwu, K. Rupnow, and Z. Zhang, "Platform choices and design demands for IoT platforms: Cost, power, and performance tradeoffs," *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 70–77, 2016.

[52] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for iot intelligence on the edge," 2019. [Online]. Available: http://arxiv.org/abs/1904.04421

[53] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An open framework for mapping dnn models to cloud FPGAs," in *International Symposium on Field Programmable Gate Arrays*, 2019.

[54] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *International Conference on Field-Programmable Logic and Applications*, 2016.

[55] Y. Chen, K. Zhang, C. Gong, C. Hao, X. Zhang, T. Li, and D. Chen, "T-DLA: An open-source deep learning accelerator for ternarized DNN models on embedded FPGA," in *IEEE Computer Society Annual Symposium on VLSI*, 2019.

[56] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "High level synthesis of complex applications: An h.264 video decoder," in *International Symposium on Field-Programmable Gate Arrays*, 2016.

[57] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980.

[58] X. Xie, F. Sun, J. Lin, and Z. Wang, "Fast-ABC: A fast architecture for bottleneck-like based convolutional neural networks," in *IEEE Computer Society Annual Symposium on VLSI*, 2019.

[59] F. Shi, H. Li, Y. Gao, B. Kuschner, and S. Zhu, "Sparse Winograd convolutional neural networks on small-scale systolic arrays," in *International Symposium on Field-Programmable Gate Arrays*, 2019.

[60] J. Cong and J. Wang, "Polysa: Polyhedral-based systolic array auto-compilation," in *ICCAD*, 2018.

[61] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Design Automation Conference*, 2017.

[62] A. Lavin, "Fast algorithms for convolutional neural networks," *Computing Research Repository*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[63] H. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, vol. 1, 1979, pp. 256–282.

[64] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™ deep learning accelerator on Arria 10," in *International Symposium on Field-Programmable Gate Arrays*, 2017.

[65] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *International Symposium on Field-Programmable Custom Computing Machines*, 2017.

[66] Y. Liang, L. Lu, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 857–870, 2020.

[67] J. Yepez and S. Ko, "Stride 2 1-D, 2-D, and 3-D Winograd for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 28, no. 4, pp. 853–863, 2020.

[68] Xilinx. (2021) Vitis-AI model zoo. [Online]. Available: https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo

[69] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *Computing Research Repository*, 2021.

[70] C. Gong, T. Li, Y. Lu, C. Hao, X. Zhang, D. Chen, and Y. Chen, "µl2q: An ultra-low loss quantization method for DNN compression," in *International Joint Conference on Neural Networks*, 2019.

[71] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "VecQ: Minimal loss DNN model compression with vectorized weight quantization," in *Computing Research Repository*, 2020.

[72] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *Computing Research Repository*, 2016.

[73] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks," in *International Symposium on Computer Architecture*, 2018.

[74] S. Ryu, H. Kim, W. Yi, and J.-J. Kim, "BitBlade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *Design Automation Conference*, 2019.

[75] D. Shin, J. Lee, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An energy-efficient deep-learning processor with heterogeneous multi-core architecture," in *IEEE Micro*, 2018.

[76] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," in *IEEE Journal of Solid-State Circuits*, 2018.

[77] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Design Automation Conference*, 2018.

[78] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. Leong, "PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks," in *International Symposium on Field-Programmable Custom Computing Machines*.

[79] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen, "Machine learning on FPGAs to face the IoT revolution," in *IEEE/ACM International Conference on Computer-Aided Design*, 2017.

[80] A. Stojanov, T. M. Smith, D. Alistarh, and M. Püschel, "Fast quantized arithmetic on x86: Trading compute for data movement," in *International Workshop on Signal Processing Systems*, 2018.

[81] Z. Kang, "Ultranet: A FPGA-based object detection for the DAC-SDC 2020," 2020. [Online]. Available: https://github.com/heheda365/ultra_net

[82] Xilinx, "Convolutional neural network with INT4 optimization on Xilinx devices," 2020, accessed: 2020-06-24. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp521-4bit-optimization.pdf

[83] ——, "Deep learning with INT8 optimization on Xilinx devices," 2017, accessed: 2017-04-24. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[84] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," *Computation Research Repository*, 2020. [Online]. Available: https://arxiv.org/abs/2009.00993

[85] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud TPUs." [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[86] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, "A mixed-precision RISC-V processor for extreme-edge DNN inference," in *IEEE Computer Society Annual Symposium on VLSI*, 2020.

[87] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors," *Philosophical Transactions of the Royal Society A*, 2020.

[88] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for ARM Cortex-M CPUs," *Computing Research Repository*, 2018.