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

BY

XINHENG LIU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Deming Chen

ABSTRACT

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 play an important role when we want to implement machine learning algorithms on an edge device. The key idea explored in this thesis 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 data set using the LeNet and 653.5 µs for classifying one image in the Cifar-10 data set using the CifarNet. For the LeNet we were able to maintain high accuracy of 97.6% for the MNIST data set and 83.4% for the Cifar-10 data set. We achieved the best single-image latency, 5.2x faster for the LeNet and 1.95x faster for the CifarNet, compared with NVIDIA Jetson TX1.

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

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Chen for the continuous support of my master's study and related research, and for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I cannot imagine having a better advisor and mentor for my study in the past three years.

I also thank my fellow lab-mate Dae Hee Kim for the assistance in the experiment, and for his company during the sleepless nights when we were working together before deadlines.

Last but not the least, I would like to thank my parents for their spiritual support from far over the ocean in my homeland.

TABLE OF CONTENTS

CHAPTER 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 [1, 2, 3, 4, 5, 6]. Due to HLS, practical applications are embedable on IoT devices easily and quickly. In [7], 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 [8] demonstrates that the HLS solution quality can range from very slow all the way to 200x 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 thesis, we explore different strategies and methodologies to optimize CNN on an FPGA. As some of data set 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 10k 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 5.2x faster than NVIDIA Jetson TX1 for the LeNet and 1.9x 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 thesis is organized as follows. Chapter 2 presents the background of CNN and HLS. In Chapter 3, our algorithm and methodology for optimizing CNN through HLS on an FPGA are discussed in detail. Chapter 4 discusses hardware architecture implementation of the design. Chapter 5 presents and analyzes our experimental results, and Chapter 6 concludes the thesis.

CHAPTER 2

BACKGROUND

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 [9, 10, 11, 12, 13, 14]. Specifically, [12] discussed a multistage data-flow implementation of CNN, which takes efficient utilization of the computation resources to achieve high performance in object classification. In [13], 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.

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 thesis, we leverage Xilinx Vivado HLS to facilitate our CNN

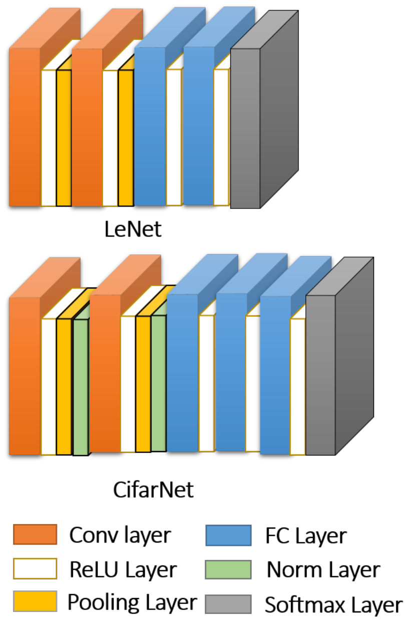

Figure 2.1: LeNet & CifarNet Architecture

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 [15, 16]. Y. Guan et al. proposed an efficient FPGA based LSTM-RNN accelerator with HLS tool in 2017 [17]. R. Zhao et al. also adopted HLS as design tool in their work of binarized convolutional neural network in 2017 [18].

2.2.1 CNN Structure

Figure 2.1 shows the CNN architectures we used to classify handwritten digits in the MNIST data set [19] and 10 objects in the Cifar-10 data set [20]. For the Cifar-10 data set, 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 data set. Therefore, the network can learn fast as it can converge faster and generate higher accuracy. The inputs are pre-processed to have size of 24x24x3 instead of the original size of 32x32x3. The detailed layer configurations of the LeNet and the CifarNet are shown in Tables 2.1 and 2.2.

Table 2.1: LeNet Configuration

Type	Input Size	Output Size	# Params
Convolution	28x28	24x24x8	5x5x8
ReLU	24x24x8	24x24x8	NA
Pooling	24x24x8	12x12x8	NA
Convolution	12x12x8	8x8x16	5x5x16
ReLU	8x8x16	8x8x16	NA
Pooling	8x8x16	4x4x16	NA
Fully Connected	256	128	256x128
ReLU	128	128	NA
FullyConnected	128	10	128x10
Softmax	10	10	NA

Table 2.2: CifarNet Configuration

Type	Input Size	Output Size	# Params
Convolution	24x24x3	24x24x32	5x5x3x32
ReLU	24x24x32	24x24x32	NA
Pooling	24x24x32	12x12x32	NA
Normalization	12x12x32	12x12x32	NA
Convolution	12x12x32	12x12x32	5x5x32x32
ReLU	12x12x32	12x12x32	NA
Pooling	12x12x32	6x6x32	NA
Normalization	6x6x32	6x6x32	NA
Fully Connected	1152	192	1152x192
ReLU	192	192	NA
FullyConnected	192	48	192x48
ReLU	48	48	NA
FullyConnected	48	10	48x10
Softmax	10	10	NA

CHAPTER 3

ALGORITHM AND METHODOLOGY

In this chapter, 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 Cifar-Net as examples to demonstrate the algorithm and method. However, the methodology can be applied to any other neural networks.

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

$$O_{i,x,y} = \sum_{o=1}^{C} \sum_{h=1}^{F} \sum_{w=1}^{F} K_{i,o,h,w} I_{o,xS+h,yS+w} \tag{3.1}$$

Equation 3.2 provides the computation of a max-pooling layer with window size W and stride S.

$$O_{i,x,y} = max(\{I_{i,xS+h,yS+w} | h \in [0, F), w \in [0, F)\}) \tag{3.2}$$

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 3.3. The equation works for all the layers with 2D window-structured on feature-map dimensions. Considering the factor of padding, we improve Equation 3.3 to Equation 3.4 with Z as the padding size.

$$\begin{aligned} Dep(\langle x, y\rangle) &= \\ \{\langle i, j\rangle | xS < i \leq xS + F, yS < j &\leq yS + F\} \end{aligned} \tag{3.3}$$

$$\begin{aligned} Dep(\langle x, y\rangle) &= \\ \{\langle i, j\rangle | xS - Z \leq i < xS + F - Z, 0 \leq i &< H, \\ yS - Z \leq j < yS + F - Z, 0 \leq j &< W\} \end{aligned} \tag{3.4}$$

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 3.1 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 Fig. 3.1.

The algorithm is described in Algorithm 1. 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

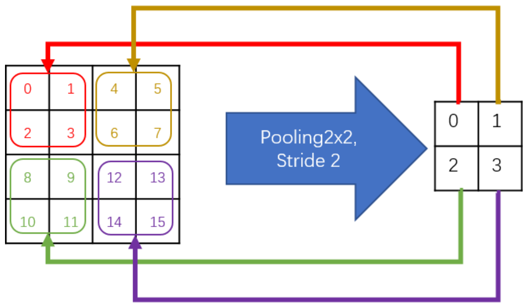

Figure 3.1: Data Request List Generation

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 1 Algorithm for data request list generation

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

The algorithm 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 backwardly 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 3.2 provides the comparison between the non-pipelined design and the design with our algorithm. The example in the figure includes one 3x3 stride 1 convolution layer and one 2x2 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.

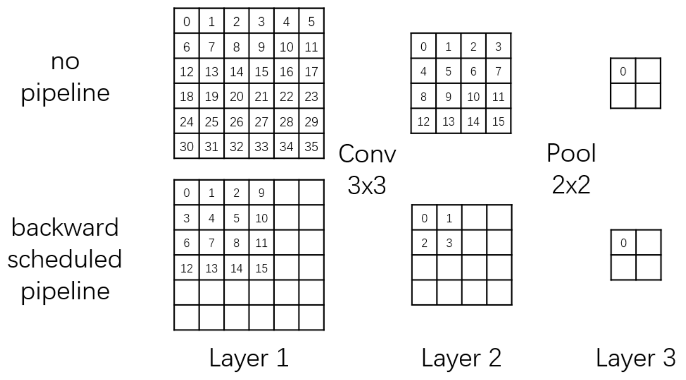

Figure 3.2: Backward Pipeline Scheduling Flow

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 2. In the algorithm, the function *window_operation* denotes the compute process of the 2D-window operation of the current layer such as Equation 3.1 or Equation 3.2. We denote the enqueue operation as symbol $\ll$ and the dequeue operation as symbol $\gg$.

Algorithm 2 Algorithm for layer behaviour

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

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 3.5 where F and L represent the total latency to complete lines 8-10 and lines 11-13 in Algorithm 2 respectively. With the same notation, the average data production rate (P) is shown in Equation 3.6.

$$R = \frac{curList.length}{curList.length \cdot F + curCompList.length \cdot L} \tag{3.5}$$

$$P = \frac{nextList.length}{curList.length \cdot F + curCompList.length \cdot L} \tag{3.6}$$

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 3.7 to constrain the latency F and latency L and eliminate the bottleneck effect in the pipeline.

$$\begin{aligned} Constant &\approx HI_A \cdot WI_A \cdot F_A + HO_A \cdot WO_A. \cdot L_A \\ &\approx HI_B \cdot WI_B \cdot F_B + HO_B \cdot WO_B \cdot L_B \end{aligned} \tag{3.7}$$

CHAPTER 4

HARDWARE IMPLEMENTATION

This chapter 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.

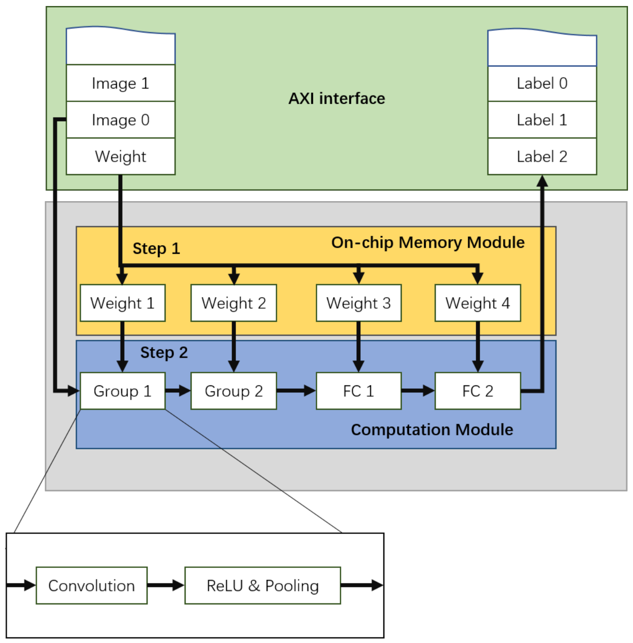

Figure 4.1: Block Structure of the Design

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 Fig. 4.1. These groups are instantiated as 2D-window modules which will be further discussed in Section 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 Fig. 4.1. The AXI4 interface first streams in the weight data as shown in step 1 in Fig. 4.1. 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 [13], 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.

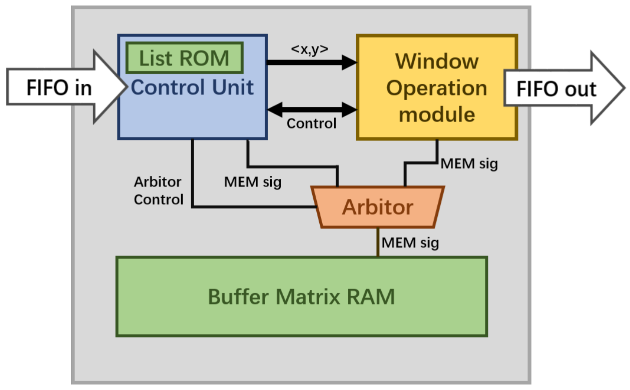

Figure 4.2: General Block Structure of 2D-Window Operation Modules

4.2 2D-Window Modules

The convolution and pooling layer are implemented as 2D-window operation modules. The 2D-Window operation modules are designed to behave as described in Algorithm 2. Figure 4.2 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 3.1 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.

Listing 4.1: Tiled Convolutional Layer Pseudo Code

```
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);
}
```

4.3 Batch Processing

In [13], 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 [13] does not give much performance increase for smaller CNN due to the smaller number of filters in those CNNs. Therefore, even

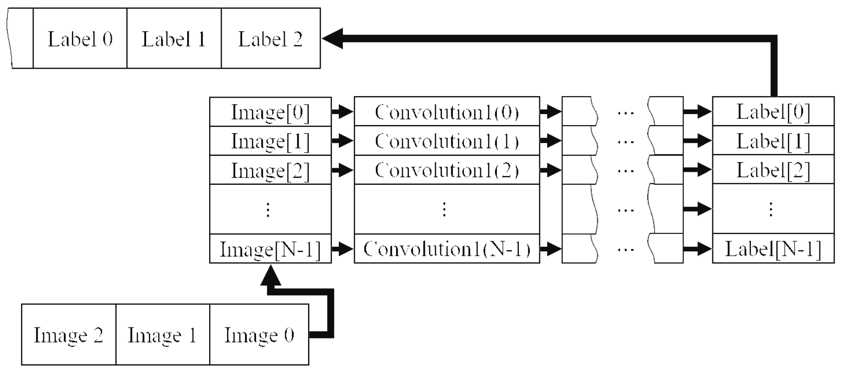

Figure 4.3: Batch Processing Computation Module

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 [21]. 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 Fig. 4.3 with N as the batch size.

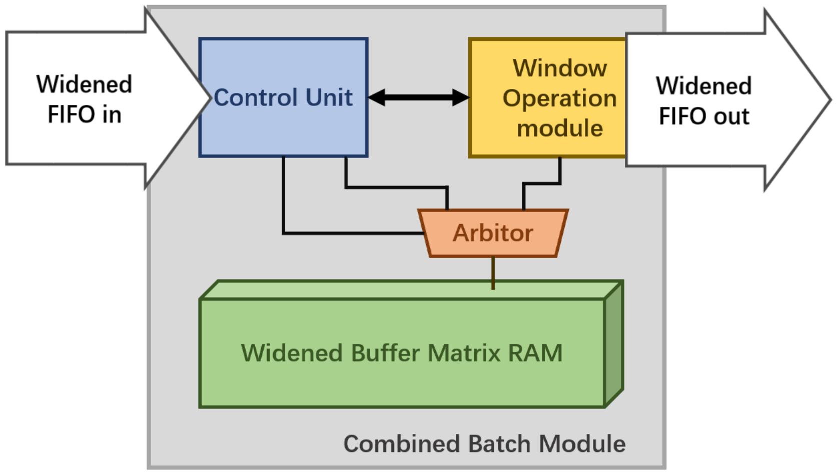

Figure 4.4: Structure for Batch Mode 2D Window Array

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 Fig. 4.4. 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.

Listing 4.2: Batched Convolutional Layer Pseudo Code Batch

```
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++){
      #pragma HLS unroll
                Oarray[cb][co]+=
   weight[co][ci][h][w]*Iarray[cb][ci];
              }}}}}
   return pack(Oarray);
}
```

CHAPTER 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 5.1 and 5.2. All the computations are fully parallelized to effectively and quickly generate output.

Table 5.1: Xilinx ZC706 Device Spec

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

Table 5.2: 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)

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 5.3 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 Fig. 5.1. 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 5.4. We achieve single image latency of 653.4 µs and throughput of 1530.3 image/s in our implementation of the CifarNet.

Table 5.3: Resource utilization and Performance Statistic of the 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 5.4: Area and Performance for the 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	NA	NA

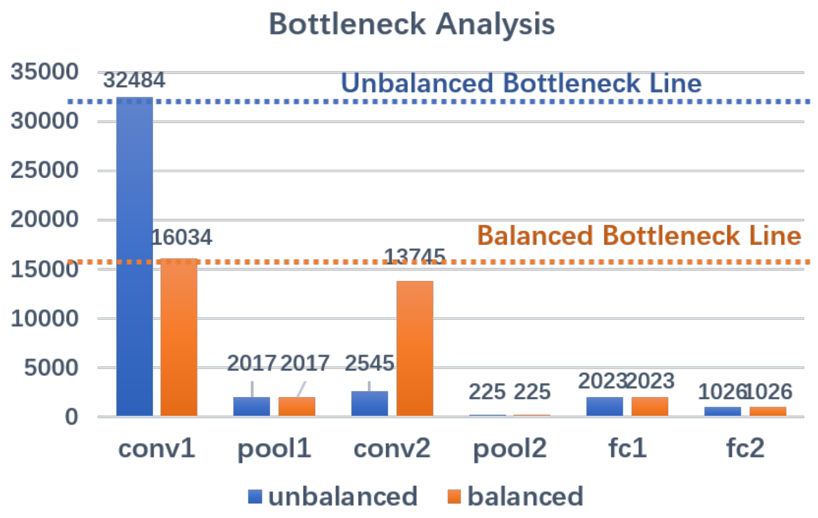

Figure 5.1: Latency Comparison for Latency Balancing

5.2 Performance Comparison

The tables comparing the performance of NVIDIA Jetson TX1 and our design are shown in Tables 5.5 and 5.6. For an image from the MNIST data set, TX1 takes 0.91 ms to classify. The throughput of GPU for the MNIST data set is 26455 image/s if we set the batch size to be 25. For an image from the Cifar-10 data set, 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 5.2x 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

Table 5.5: Performance Comparison for the 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%

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 [22]. We also compare our LeNet single image latency with that of [23] and [24]. The result is listed in Table 5.7. We achieve 11x and 7.5x speedup by enabling proper pipeline among layers and further optimization in parallel computation architecture.

Table 5.6: Performance Comparison for the 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 5.7: Performance Comparison with Previous Work for the LeNet

	[23]	[24]	Our work
CNN model	LeNet-5	LeNet-5	LeNet-5
platform	ZC706	VC709	ZC706
Precision	fixed(25)	fixed(8-16)	fixed(16)
Latency	2ms	1.318 ms	175.7 µs

CHAPTER 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 work, 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 5.2x 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.

REFERENCES

[1] Xilinx, "Vivado High Level Synthesis," http://www.xilinx.com/products/design-tools/vivado.html.

[2] CALYPTO, "Catapult C Synthesis," http://www.calypto.com/catapult-c-synthesis.php.

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

[4] 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. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423 pp. 33–36.

[5] 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.

[6] 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. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554775 pp. 1–10.

[7] 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 *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 819–826.

[8] 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.

[9] 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. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265 pp. 26–35.

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

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

[12] 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.

[13] 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. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060 pp. 161–170.

[14] 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 *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–4.

[15] 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 *2011 International Conference on Field-Programmable Technology*, Dec 2011, pp. 1–8.

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

[17] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 629–634.

[18] 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 *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021741 pp. 15–24.

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

[20] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[21] 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.

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

[23] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," in *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, Dec 2016, pp. 1–6.

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