UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Xinyu Zhang

Committee in charge:

Professor Ken Kreutz-Delgado, Chair
Professor Ryan Kastner
Professor Larry Smarr

2017

The Thesis of Xinyu Zhang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Professor Ken Kreutz-Delgado, for his continuous help and support. I would like to thank my committee members Professor Larry Smarr and Professor Ryan Kastner, as well as my research collaborators Srinjoy Das and Ojash Neopane. I am also grateful to the Pacific Research Platform and Xilinx for their support of our research.

All of the chapters are currently being prepared for submission for publication of the material in "A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA", Xinyu Zhang, Srinjoy Das, Ojash Neopane, and Ken Kreutz-Delgado. The thesis author was the primary investigator and author of this material.

ABSTRACT OF THE THESIS

A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA

by

Xinyu Zhang

Master of Science in Computer Science

University of California, San Diego, 2017

Professor Ken Kreutz-Delgado, Chair

In recent years deep learning algorithms have shown extremely high performance on machine learning tasks such as image classification and speech recognition. In support of such applications, various FPGA accelerator architectures have been proposed for convolutional neural networks (CNNs) that enable high performance for classification tasks at lower power than CPU and GPU processors. However, to date, there has been little research on the use of FPGA implementations of deconvolutional neural networks (DCNNs). DCNNs, also known as generative CNNs, encode high-dimensional probability distributions and have been widely used for computer vision applications such as scene completion, scene segmentation, image creation, image denoising, and super-resolution imaging. We propose

an FPGA architecture for deconvolutional networks built around an accelerator which effectively handles the complex memory access patterns needed to perform strided deconvolutions, and that supports convolution as well. We also develop a three-step design optimization method that systematically exploits statistical analysis, design space exploration and VLSI optimization. To verify our FPGA deconvolutional accelerator design methodology we train DCNNs offline on two representative datasets using the generative adversarial network method (GAN) run on Tensorflow, and then map these DCNNs to an FPGA DCNN-plus-accelerator implementation to perform generative inference on a Xilinx Zynq-7000 FPGA. Our DCNN implementation achieves a peak performance density of 0.012 GOPs/DSP.

Chapter 1

Need for Speed: Neural Networks

The wave of Artificial Intelligent (AI) has been overwhelming since the beginning of the 21st century. Brain-inspired neural network algorithms have become the state-of-the-art in numerous applications, including computer vision and audio recognition [LBH15][Sch15]. To meet the growing demands of computation and needs of real-time and low-power applications, researchers have begun to design accelerators for neural networks.

1.1 The AI Strikes Back

The AlphaGo [SHM+16] win over Go grandmaster Sedol Lee [Wik17a] was one of the most exciting news in the AI field in 2016. Because there are 2×10^{170} possible layouts of checkerboard in Go [Wik17b], a number greater than the total number of atoms in the universe, the game of Go is one of the hardest tasks for humans to excel in. If we can design an AI to defeat the top human players in Go, it seems plausible that AI may be able to exhibit superhuman performance in almost any arena, as long as we can collect enough training data, a prospect that is both exciting and scary. So are we on the eve of massive AI revolution? What could stop AI from automating vast areas of human labor and increasing the productivity of the whole society? The only real impediment, it would appear, is if we are unable to break the computational bottleneck associated with the increasing complexity of AI algorithms and the looming end of Moore's law.

1.1.1 Behind the Go Master

The algorithm behind AlphaGo is a sophisticated combination of reinforcement learning and deep learning. Reinforcement learning [SB98] [BPW+12] provides the strategic ability to think ahead and make the best current decision based on the potential future rewards. Deep learning [Sch15] [LBH15] provides the awareness of the gaming environment and the possible consequences of local moves by modeling their distributions, which makes up most of the algorithm's computational load.

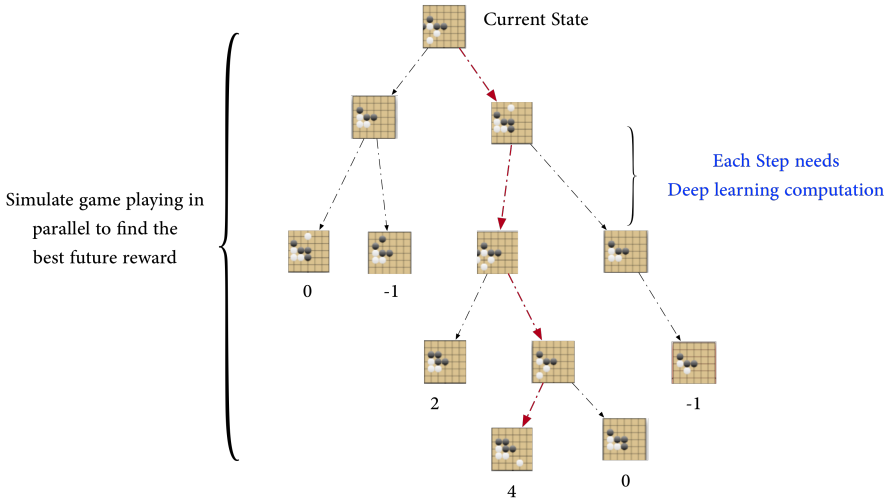


Figure 1.1: Visualization of possible future game moves

Deep learning algorithms have shown extremely high performance on machine learning tasks. However, such strong performance does not come for free. The AlphaGo algorithm needs 1202 CPUs and 176 GPUs working in parallel [SHM+16] to play as good as a professional player, and these numbers get nearly doubled when it plays at the level of an international grandmaster such as Sedol Lee. Furthermore, there is also an enormous amount of engineering work required to assemble and maintain the huge computing cluster of required CPUs and GPUs. In sum, a currently prohibitive computational burden prevents the advanced algorithms underlying a state-of-the-art system like AlphaGo from becoming commercially available to the general public.

1.1.2 End of Moore's Law

If Moore's Law [Sch97] remains true into the indefinite future, then the computational burden will never be a headache. Thanks to Moore's Law, the computational power of an 2010 iPhone is stronger than a 1980s supercomputing center. However, we have already begun diverging from Moore's Law, and if we can not push processor speeds farther, we will have to figure out other ways to meet the computational burden needed to implement ever-more-powerful AI algorithms. To accelerate computation, engineers are now designing specialized hardware for deep learning algorithms, particularly for implementation of the convolutional neural networks (CNNs), which have become the state-of-the-art algorithm for applications like computer vision and audio recognition [Sch15] [LBH15] [GBC16].

1.2 Possible Hardware Accelerator Choices

Deep learning algorithms have specific execution patterns and are very computationally intensive, which makes general purpose processor implementations uneconomical in chip area and power consumption. A popular solution is to use dedicated hardware that executes only deep learning algorithms but at very high speeds. The pros and cons of different types of deep learning accelerators are summarized in the following.

1.2.1 ASICs

Realizing deep learning algorithms in circuit logic by building a dedicated application-specific integrated circuit (ASIC) can provide the highest computational throughput and many other benefits in the same time. However, it's well known that designing and implementing ASICs are very difficult and expensive.

Pros

- The highest computational throughput
- The lowest power consumption

- The smallest chip area
- Inexpensive for users when it is mass-manufactured

Cons

- Difficult and expensive to design and manufacture (only affordable by a few companies)
- Not reconfigurable; functionality is fixed once made

Some pioneering research is Wave Computing's DPU [WXT+16], Google's TPU and China's DianNao [CLL+14]. However, they all have different architectures and specifications of usage, and none of these are ready to be applied in the massive production settings.

1.2.2 GPUs

A graphics processing unit (GPU) is a much more parallel processor than CPUs. GPUs execute executing with thousands of threads, while each thread can be programed to realize arithmetic operations with a compiler. Its parallelism and ease of programming make the GPU perhaps the most popular solution for deep learning applications and research.

Pros

- Much higher throughput than CPU
- Easy to program and many well-tested tools available

Cons

- Expensive for users
- High power consumption; inappropriate for embedded application
- Large chip size

1.2.3 FPGAs

The field-programmable gate array (FPGA) is an integrated circuit which is software definable. Programmers can implement customized algorithms in FPGA circuits and achieve ASIC level speedup. Moreover, unlike ASIC, the circuits logic in FPGA can be reprogramed.

Pros

- Faster than GPUs if carefully designed
- Easier to design and implement than ASICs
- Lower power consumption than CPUs and GPUs

Cons

- Slower than ASICs because of the switch delay introduced by reconfigurable inter-connections
- Higher power consumption than ASICs
- Harder to program than GPUs

1.2.4 Non-Volatile Memory

There is recent research exploiting the analog feature of non-volatile memory, like resistive random-access memory (RRAM), for approximate computing [LGS+15]. For example, a RRAM crossbar array can naturally transfer the weighted combination of input voltages to output voltages and realize matrix–vector multiplication, as shown in Figure 1.2.

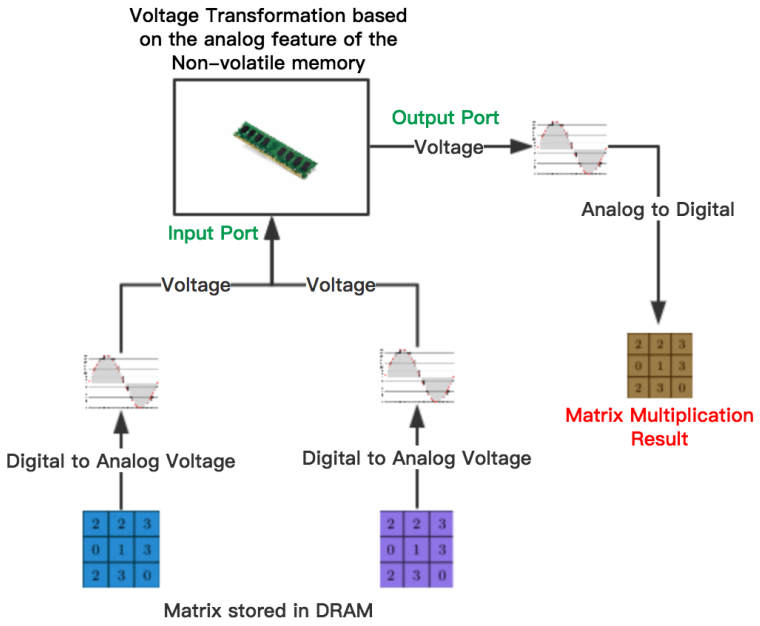


Figure 1.2: Matrix Multiplication using analog properties of silicon Material

This analog arithmetic feature of RRAM could be used to implement deep learning operation in analog circuits to save a significant amount of power at high speeds. However, these prototypes are far from production and only presented in few academic papers and laboratories [WXT+16].

1.3 Our choice - the FPGA

We choose an FPGA as the platform to develop our deep learning accelerator, because this allows us to design custom algorithmic circuits and realize high performance, while having the option to experiment with different hardware architectures. Moreover, the circuit logic of an FPGA can be used for ASIC prototyping.

1.4 Our Target Algorithm

While current FPGA-based accelerators only focus on enhancing the performance of convolutional neural networks (CNNs) in discriminative tasks [ZLS+15] [PSMC13], we design and implement an accelerator for deconvolutional neural

networks (DCNNs), which are also known as generative CNNs [GBC16]. Unlike discriminative CNNs that effectively "downsample" the input to produce classification [Sch15], DCNNs are generative models capable of generating data by "upsampling" the input using deconvolution layers [ZKTF10]. There are many applications of DCNNs, including multi-modal data modeling [WZX+16], super resolution [SCH+16] and image-to-image translation [IZZE16] [BKC15] (see Figure 1.3).

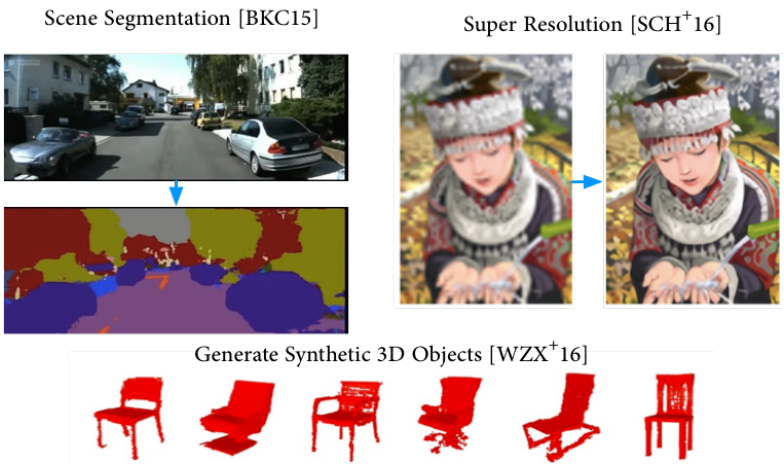


Figure 1.3: DCNNs work for pattern completion/generation (Images from [WZX+16] [SCH+16] [BKC15]).

These, and other exciting applications motivate us to design an FPGA-based accelerator with the ability to execute deconvolution operations with high throughput and low cost.

Chapter 2

Background Knowledge

In the previous chapter, we explain that the importance of designing specialized hardware to accelerate deep learning computation as a motivation to accelerate the computation of deconvolutional neural networks (DCNNs) using an FPGA. In this chapter, we provide background knowledge for DCNNs, the deconvolution layers, and how we train our DCNNs.

2.1 Neural Network

The use of (artificial) neural networks provides a computational approach that purports to mimic the way a brain performs distributed computation, where the behaviors of neural units ("neurons") are realized in a series of differentiable math operations (denoted by the symbols h, g, f) that are nestedly applied over high-dimensional input data $\boldsymbol{x}$ [i.e. $h(g(f(\boldsymbol{x})))$]. A layer, L, of a neural network is comprised of several well-defined math operations. The form and nature of these operators depends on the type of neural network that is implemented. We are concerned with the structure and operators that define convolutional and deconvolutional neural networks, as defined in references [Sch15] [DV16] [LBH15] and the notation we use is also described in those references.

2.2 Deconvolutional Neural Network

A deconvolutional neural network (DCNN) converts latent space representations to high-dimensional data similar to the training set by applying successive deconvolution operations in multiple layers [NHH15]. The latent space contains low-dimensional latent variables that provide a succinct ("conceptual") representations of the possible outputs (e.g. an image). Thus a latent variable may correspond to "chair" with the associated output being the image of a chair "generated" by the DCNN (see Figure 1.3). Figure 2.1 shows a 5-layer DCNN developed in [RMC15] that consists of 4 deconvolutional layers. The first layer is fully-connected and transforms an input size of 1x100 to an output size of 1024x4x4; layers 2 to 5 are deconvolution layers that project low-dimensional feature maps into corresponding high-dimensional ones through successive layers.

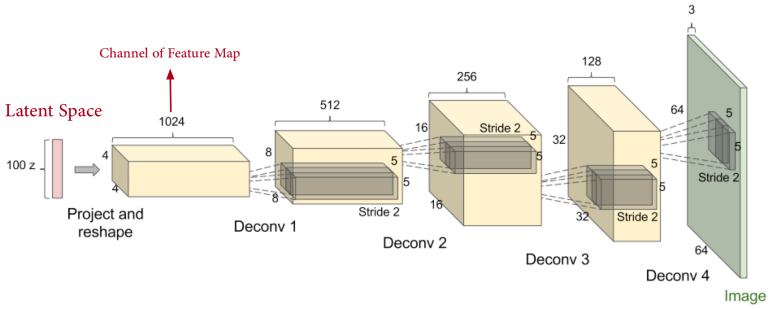


Figure 2.1: A DCNN that generates realistic 64x64 indoor scenes based on the use of four deconvolution layers that was trained on the Large-scale Scene Understanding (LSUN) Dataset [RMC15] [YSZ+15] (Image is taken and adapted from reference [RMC15]).

2.2.1 Deconvolution Layer

Figure 2.2 shows how a typical deconvolution layer works, where S and P denote the chosen values of stride and padding respectively for a given layer. The four steps required to implement the deconvolutional layer are: (1) multiply a single input pixel i_h, i_w by a $K \times K$ kernel; (2) add the result of step 1 to a local area in the output feature map that starts at $i_h \times S, i_w \times S$; (3) repeat 1 and 2 for all input pixels; (4) remove elements from output feature maps in the border by zero padding of size P. The pseudo code of a deconvolution layer as implemented in

CPU is shown in Algorithm. 1 which uses the loop variables defined in Figure 2.3.

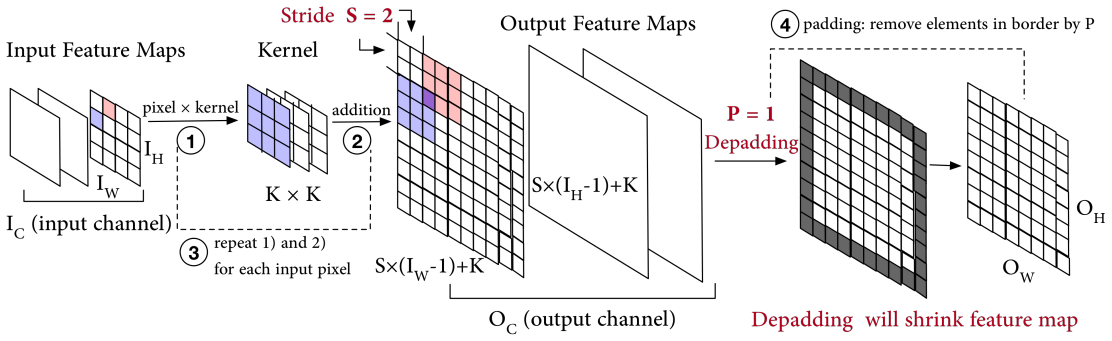


Figure 2.2: Visualization of a Single Deconvolution Layer

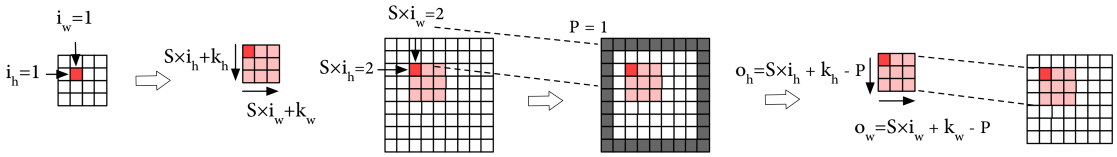


Figure 2.3: Visualization of Algorithm 1 with loop variables.

Algorithm 1 Deconvolution in CPU

```
1:  procedure DECONVOLUTION
2:      for i_c = 0 to I_C − 1 do
3:          for i_h = 0 to I_H − 1 do
4:              for i_w = 0 to I_W − 1 do
5:                  for o_c = 0 to O_C − 1 do
6:                      for k_h = 0 to K − 1 do
7:                          for k_w = 0 to K − 1 do
8:                              o_h ← S × i_h + k_h − P
9:                              o_w ← S × i_w + k_w − P
10:                             out[o_c][o_h][o_w] ← (in[i_c][i_h][i_w]
                                × kernel[o_c][i_c][k_h][k_w])
```

The relation of the input size $I_H \times I_W$ to output size $O_H \times O_W$ after applying

By convention we use capital letters e.g. O_H to denote specific paraters of the deconvolution layer whereas small letters e.g. o_h to denote its corresponding loop variable.

stride and padding are given in the following equations [DV16]:

$$\begin{aligned} O_H &= S \times (I_H - 1) + K - 2P \\ O_W &= S \times (I_W - 1) + K - 2P \end{aligned} \tag{2.1}$$

2.2.2 Batch Normalization and ReLu Activation Layers

We include batch normalization layers, comprised of element-wise math operations that normalize the output from deconvolution layer (Eq. 2.2), each followed by a ReLu activation function that sets all negative values in the feature map to zero (Eq. 2.3). We tacitly take these layers to be the post-processing part of the deconvolution layer and therefore they are not shown in Figure 2.1. Neither of stages change the size of feature maps.

Batch Normalization

The batch normalization layer contains four parameters γ, β, μ, σ, which are shared within a channel. γ and β are trained parameters, μ and σ are the statistical mean and standard variance of the output feature maps from its corresponding deconvolution layer, and they are calculated over training data. Letting "in" and "out" denote input and output feature maps, the equation of batch normalization layer is shown in Eq. 2.2:

$$\text{out} = \frac{\text{in} - \mu}{\sigma} \times \gamma + \beta \tag{2.2}$$

Based on previous research [Shi00], the values of hidden layers inside a neural network tend to gradually deviate from zero mean and unit variance; this process is called internal co-variate shift problem. This problem makes the network harder to converge, especially for DCNNs. Batch normalization layer was found to be a very effective method to remove this shift effect [IS15].

ReLu Activation Function

The layer of rectified linear units (ReLu activation functions) does not have parameters and its equation is simply shown in Eq. 2.3:

$$\text{out} = \max(\text{in}, 0) \tag{2.3}$$

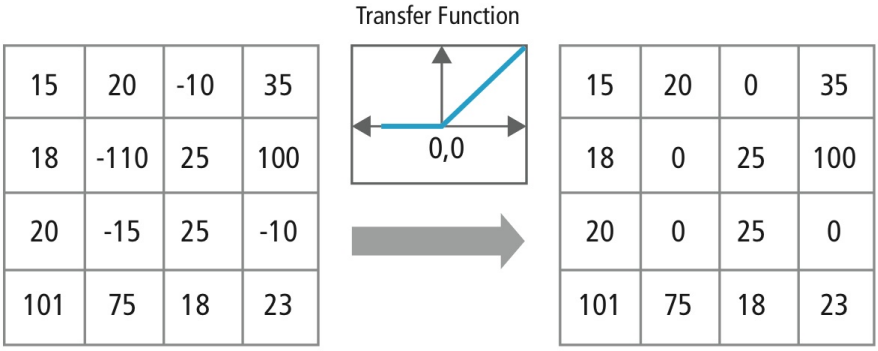


Figure 2.4: ReLu layer sets zeros for negative values in feature map

2.3 Generative Adversarial Network Training

Generative adversarial network training (GAN) is a state-of-the-art method to train generative neural networks such as DCNNs [Goo16]. The idea of GAN is to train a generator the G network and discriminator D network simultaneously (Figure 2.5). The four steps required to train G network are: (1) generate samples from G; (2) label generated samples with 0 (fake), and train D to minimize the loss $\mathcal{L}$ defined below; (3) label generated samples with 1 (real), and train G to maximize loss $\mathcal{L}$; (4) repeat steps 1, 2 and 3 until both G and D converge [AB17].

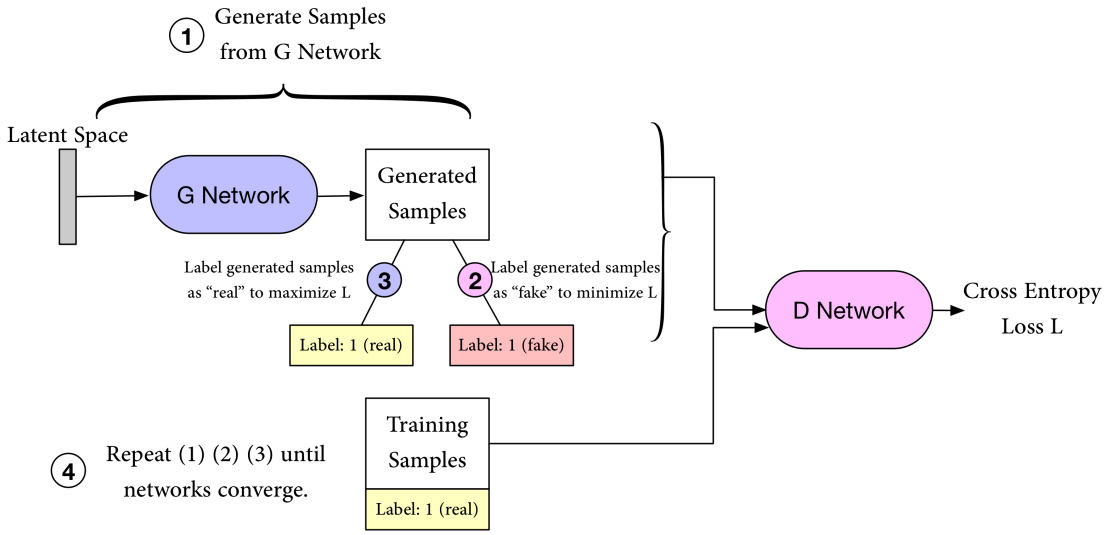


Figure 2.5: Visualization of a GAN training process [Goo16].

The G network is usually a DCNN with an input $\boldsymbol{z}$, where $\boldsymbol{z}$ is a sample from a simple low-dimensional prior distribution (latent space), such as the $[-1, 1]$ uniform distribution in the simplest case. The output of the network $G(\boldsymbol{z})$ is a generated sample that is intended to come from the training data distribution. The D network is a binary classifier with an input that consists of both training samples $\boldsymbol{x}$ and the generated samples $G(\boldsymbol{z})$. The loss function of the discriminator is the binary cross-entropy loss [GBC16]:

$$\mathcal{L} = -\frac{1}{2}E_{\boldsymbol{x}}(\log D(\boldsymbol{x})) - \frac{1}{2}E_{\boldsymbol{z}}(\log(1 - D(G(\boldsymbol{z}))))$$

where $D(x) \in \{0, 1\}$ is the classification result for input $\boldsymbol{x}$, E_x denotes expectation of training data distribution, and E_z denotes expectation of the prior distribution of sample $\boldsymbol{z}$. The training process set up a competition between the G network and the D network. G is trained to maximize the loss $\mathcal{L}$ by generating samples indistinguishable from real samples to fool D, while D is trained to minimize the loss $\mathcal{L}$ by improving the discriminative criterion to raise the requirements for G. When the training is finished, the G network is intended to generate fake samples that are ideally statistically indistinguishable from the real data samples.

Chapter 3

Introduction

Chapters 1 and 2 described the motivations behind our intent to design an FPGA-based accelerator for deconvolutional neural networks (DCNNs), and background knowledge about DCNNs. In this chapter, we introduce the design challenges, and describe our intended contributions.

3.1 Design Challenges

We follow the design paradigm of the discriminative CNN convolution accelerator proposed in [ZLS+15], extending this design to implement customized circuit logic for a single deconvolution layer in FPGA. This layer can be reused for each layer of a DCNN to increase the overall throughput (see Figure 3.1).

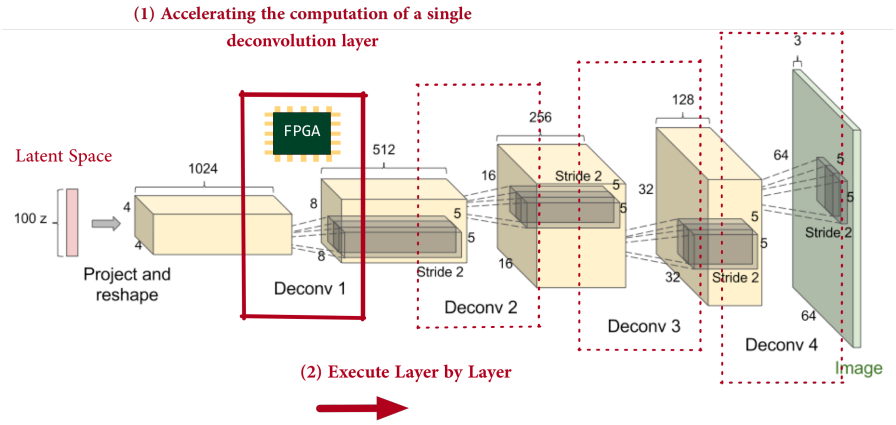


Figure 3.1: Execution Paradigm of our accelerator (image is adopted from [RMC15]).

It is the case that a direct translation of CPU-optimized deconvolution algorithms to an FPGA will generally lead to inefficient implementations. A suitable adaptation of the deconvolution operation to a hardware substrate such as an FPGA is therefore necessary in order to achieve high performance with low implementation complexity. In addition, although recent research shows that discriminative CNNs are generally robust to low bitwidth quantization [WLW+16] [DR95], it is important to be able to systematically study the effects of such bitwidth reductions on the quality of inference from a generative model such as DCNN implemented with finite precision on FPGA. Thus we proposed the use of metrics which quantify the effects of such approximations in DCNNs in order to achieve an efficient design optimized for performance and power.

3.2 Contributions

To address the issues described above, we make the following contributions in this thesis.

1. We create a deconvolution accelerator with reverse looping and stride hole skipping to efficiently implement deconvolution on an FPGA, where our proposed solution, in a nontrivial way, reuses the same computational architecture proposed for implementing a convolution accelerator in [ZLS+15].

2. We propose a three-step procedure to design the deconvolution accelerator as follows.

   (a) At the highest design level, we train DCNNs using the generative adversarial network method (GAN) [GPAM+14] and use statistical tests to quantitatively analyze the generative quality under different bitwidth precisions to select the most cost-efficient bitwidth.

   (b) We use the roofline model proposed in [ZLS+15] to explore the design space in order to find the set of high-level constraints that achieves the best tradeoff between memory bandwidth and accelerator throughput.

   (c) We use loop unrolling and pipelining, memory partitioning, and register insertion to further optimize performance.

3. We validate our procedure via two implementations on a Xilinx Zynq-7000 FPGA.

3.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 4 presents our methodology for efficiently implementing an FPGA-based deconvolution accelerator. Chapter 5 explains our three-step design methodology. Chapter 6 shows our experimental results. Section 7 concludes the thesis.

Chapter 4

Deconvolution Hardware Design

In this chapter, we explain our methodology for efficiently implementing an FPGA-based deconvolution accelerator.

4.1 Efficiency Problem of FPGA Implementation

An FPGA accelerator usually consists of processing elements (PEs), registers, and local memory elements referred to as block RAMs (BRAMs). Processing elements operate on data provided by the local memory, which communicates with external dual data rate (DDR) memory using direct memory access (DMA). Figure 4.1 shows a traditional implementation of deconvolution, where T_{I_H}, T_{I_W}, T_{I_C}, T_{O_H}, T_{O_W}, and T_{O_C} are the dimensions of the input and output block. Replacing I_H, O_H with T_{I_H}, T_{O_H} in Eq. 2.1, we have:

$$T_{O_H} = S \times (T_{I_H} - 1) + K - 2P \tag{4.1}$$

Here the zero padding $P = 0$ because blocks are inside input feature maps. However, Eq. 4.2 shows that the deconvolution results of input blocks overlap with each other:

$$\left\lceil \frac{I_H}{T_{I_H}} \right\rceil \times T_{O_H} > O_H \tag{4.2}$$

Deconvolution arithmetic requires the overlapping regions between output blocks to be summed together [DV16] (shown in Figure 4.1), which can be realized in

processor-based implementations. However handling such operations in FPGAs requires either the design of additional hardware blocks which creates overhead or communicating with a host processor which can increase system latencies thereby precluding real-time applications.

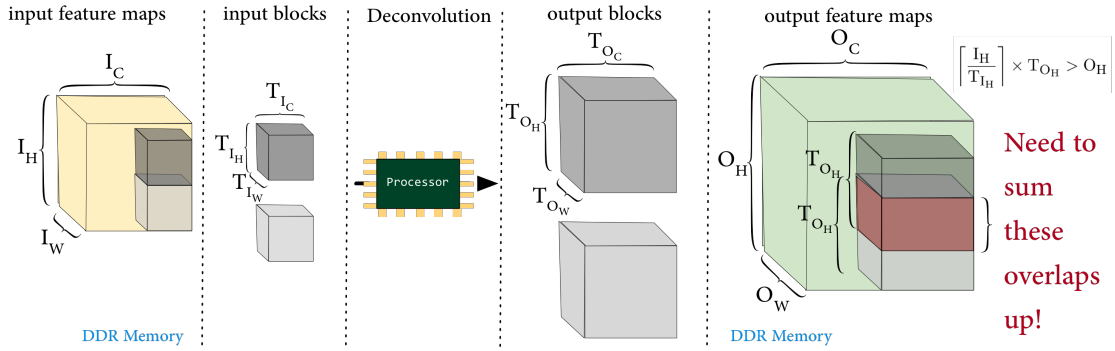


Figure 4.1: Traditional implementation of deconvolution. The input feature map is first divided into separate blocks and PEs read each block from DDR and process the deconvolution operations on this block. Finally the results are stored back to the DDR.

4.2 Reverse Looping

To avoid the overlapping sum problem, we propose a technique called reverse looping, where instead of directly deconvolving the input space, we use the output space to determine which input blocks to deconvolve and thus eliminating the need for the additional summation operations described above. This procedure is indicated in Figure 4.2. We first take a block in the output space and determine which inputs are needed to calculate the values in the block. Then, for each block, the input is deconvolved and the appropriate output is extracted. This is done sequentially until values have been computed for the entire output space.

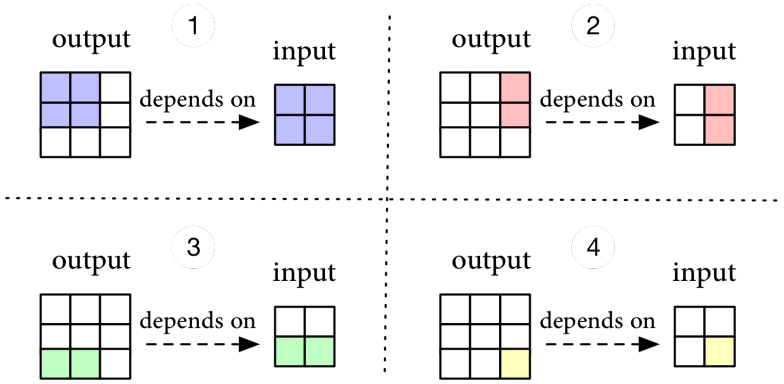


Figure 4.2: An efficient way to deconvolve

The loop iterations over i_h and i_w in the CPU implementation shown in Algorithm 1 need to be recast over o_h and o_w. Referring to Algorithm 1 and Fig. 2.3, we have:

$$o_h = i_h \times S + k_h - P \tag{4.3}$$

Rearranging terms, we get:

$$i_h = \frac{o_h + P - k_h}{S} \tag{4.4}$$

Algorithm 2 shows an deconvolution implementation with reverse looping.

Algorithm 2 Deconvolution with Reverse Looping

```
1: procedure ReverseDeconvolution
2:     for k_h = 0 to K − 1 do
3:         for k_w = 0 to K − 1 do
4:             for o_h = 0 to T_{O_H} − 1 do          ▷ Loop over O_H instead of I_H
5:                 for o_w = 0 to T_{O_W} − 1 do      ▷ Loop over O_W instead of I_W
6:                     for o_c = 0 to T_{O_C} − 1 do
7:                         for i_c = 0 to T_{I_C} − 1 do
8:                             COMPUTE(k_h, k_w, o_h, o_w, o_c, i_c)
9: procedure Compute(k_h, k_w, o_h, o_w, o_c, i_c)
10:    i_h ← (o_h + P − k_h)/S                        ▷ Correspond to Eq. 4.4
11:    i_w ← (o_w + P − k_w)/S                        ▷ Generalize to i_w and o_w
12:    if i_h, i_w ∈ ℤ then
13:        out[o_c][o_h][o_w] ← in[i_c][i_h][i_w] × kernel[o_c][i_c][k_h][k_w]
```

Unfortunately Eq. 4.4 generally results in a non-integer value for the loop variable i_h, which is invalid [DV16]. One way to address this problem would be to

monitor i_h so that fractional values can be discarded. However this would consume additional hardware resources and create unnecessary latencies in the system.

4.3 Stride Hole Skipping

In this section, we propose a technique called stride hole skipping to ensure i_h of Eq. 4.4 is an integer. Toward this end, we recast o_h in terms of two new variables, o'_h and f_h and show that this leads to an effective way of solving the aforementioned problem. First note that a sufficient condition for i_h to be an integer in Eq. 4.4 is:

$$(o_h + P - k_h) \bmod S = 0 \tag{4.5}$$

Assuming $\frac{O_H}{S}$ is an integer (O_H is defined in Eq. 2.1), we can recast o_h as follows:

$$\begin{aligned} o_h = S \times o'_h + f_h, \quad f_h \in \{0, 1, ..., S-1\} \\ o'_h \in \{0, 1, ..., \frac{O_H}{S} - 1\} \end{aligned} \tag{4.6}$$

Using the definition of o_h in Eq. 4.5, we can recast the sufficient condition Eq. 4.5 in terms of f_h as below:

$$(f_h + P - k_h) \bmod S = 0 \tag{4.7}$$

Eq. 4.6 implies that we can rewrite f_h as:

$$f_h = S - ((P - k_h) \bmod S) \tag{4.8}$$

This can be verified by plugging in Eq. 4.8 into Eq. 4.7 which yields the following identity:

$$(P - k_h - (P - k_h) \bmod S) \bmod S = 0 \tag{4.9}$$

To prevent f_h from taking a value equal to S, we enforce the additional condition:

$$f_h = (S - ((P - k_h) \bmod S)) \bmod S \tag{4.10}$$

By using Eq. 4.10 to choose values for f_h, we can ensure that o_h computed from Eq. 4.6 meets the condition in Eq. 4.5. Therefore we can avoid the previously mentioned issue of discarding fractional values of i_h that we would otherwise encounter from a direct application of Eq. 4.4. The pseudo code for deconvolution on FPGA is shown in Algorithm 3.

Algorithm 3 Our FPGA Implementation of Deconvolution with reverse looping and stride hole skipping

```
1: procedure ReverseSkippingDeconvolution
2:     for k_h = 0 to K − 1 do
3:         for k_w = 0 to K − 1 do
4:             for o'_h = 0 to T_OH/S − 1 do
5:                 for o'_w = 0 to T_OW/S − 1 do                ▷ loop T_OW
6:                     for o_c = 0 to T_OC − 1 do               ▷ loop T_OC
7:                         for i_c = 0 to T_IC − 1 do           ▷ loop T_IC
8:                             COMPUTE(k_h, k_w, o'_h, o'_w, o_c, i_c)
9: procedure Compute(k_h, k_w, o'_h, o'_w, o_c, i_c)
10:    f_h ← (S − ((P − k_h) mod S)) mod S
11:    f_w ← (S − ((P − k_w) mod S)) mod S
12:    o_h = o'_h × S + P + f_h
13:    o_w = o'_w × S + P + f_w
14:    i_h ← (o_h − k_h)/S
15:    i_w ← (o_w − k_w)/S
16:    out[o_c][o_h][o_w] ← in[i_c][i_h][i_w] × kernel[o_c][i_c][k_h][k_w]
```

The difference in architecture between the deconvolution accelerator in Algorithm 3 and the convolution accelerator proposed in [ZLS+15] is the local memory addressing. The close similarity means that we can share the basic computation architecture for both convolution and deconvolution layers.

4.4 ReLu and Batch Normalization

Notably, our deconvolution accelerator will optionally apply ReLu and batch normalization computation to output feature maps of deconvolution layer. They are comprised of element-wise math operations which are easy to implement.

Chapter 5

Three-Step Design Optimization

In the previous chapter, we presented our methodology for efficiently implementing an FPGA-based deconvolution accelerator. In this chapter, we explain the three-step design optimization methodology which goes as follows:

1. Use statistical analysis to find out the most cost-efficient bitwidth for the hardware system.

2. Explore the design space using the roofline model proposed in [ZLS+15] to optimize T_{O_H}, T_{O_W}, T_{O_C}, and T_{I_C}.

3. Apply very-large-scale circuits integration (VLSI) optimizations to further improve performance.

5.1 Statistical Analysis

It is important to study the effect of bitwidth reduction on the quality of inferential samples drawn from the generative model. To find out the most cost-efficient bitwidth for DCNNs using an appropriate metric, we fix $T_{O_H}, T_{O_W}, T_{O_C}, T_{I_C}$, and study the trade-off between generative quality and implementation complexity over a range of bitwidths using statistical analysis. In order to study the trade-off, we need to quantify the visual quality of the output of DCNNs. Traditional measures of performance, such as Kullback-Leibler divergence and log-likelihood

are not feasible in the high-dimensional settings that the typical deconvolutional neural networks are used in.

5.1.1 Null Hypothesis Test

To overcome this drawback, we turn to statistical methods that allow us to determine the closeness of statistics from training data to statistics created from the full-precision DCNN and a low-bitwidth (10bit) DCNN (i.e. 10bit) as a way to measure the generative quality of low-bitwidth DCNN.

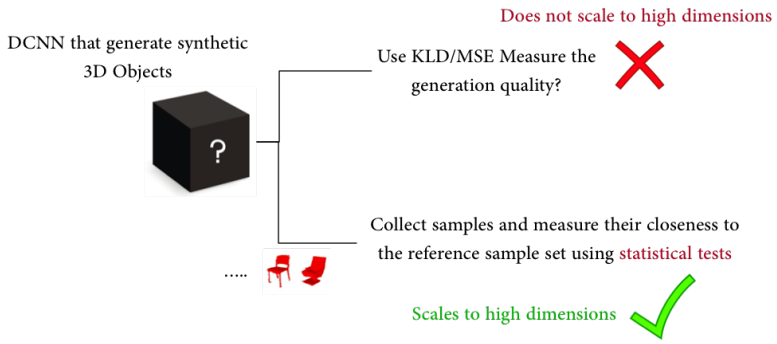


Figure 5.1: Measure generation quality quantitatively using statistical tests

Specifically, we design a null hypothesis test and use its p-value to represent generative quality. Given samples $\{X_i\}_{i=1}^{m}$, $\{Y_i\}_{i=1}^{n}$, and $\{Z_i\}_{i=1}^{r}$ respectively from the training data, low-bitwidth DCNN, and full-precision DCNN, the null hypothesis is that the samples Y are closer to samples X than samples Z. Figure 5.2 shows the process of judging our null hypothesis using test statistics.

- a p-value > 0.5 indicates the low bitwidth DCNN is more similar to the training data
- a p-value < 0.5 indicates the full precision DCNN is more similar to the training data

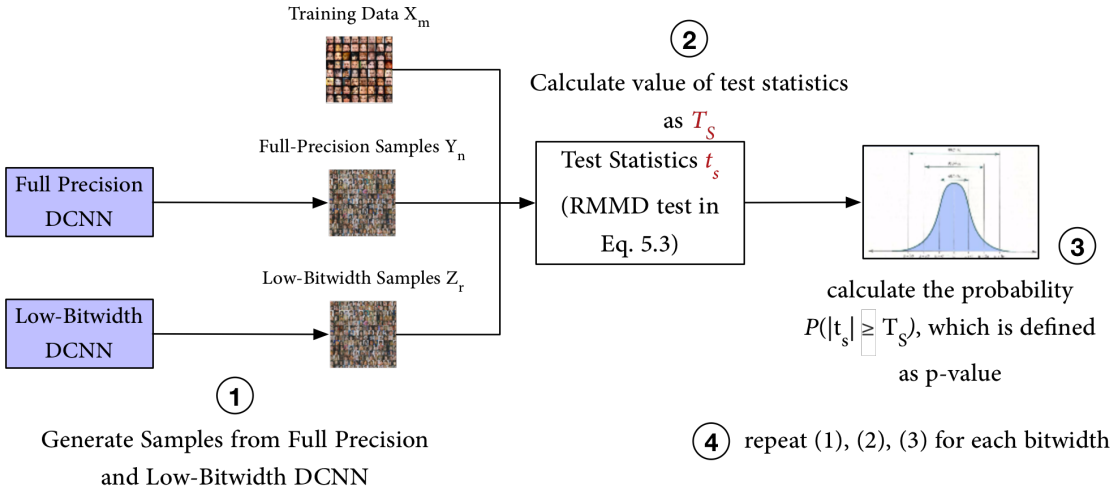


Figure 5.2: Process of judging a null hypothesis

1. Generate samples from full-precision and low-bitwidth DCNNs

2. Calculate the value T_S of test statistics t_s from samples $\{X_i\}_{i=1}^m$, $\{Y_i\}_{i=1}^n$, and $\{Z_i\}_{i=1}^r$.

3. Calculate $P(|t_s| \geq T_S)$, the value of which is defined the as p-value. If the p-value is too small, then the test statistics value exists in the two tails of the distribution under the null hypothesis. Therefore, the null hypothesis is unlikely to be true and these low-bitwidth generated samples are significantly different than full precision and real samples in terms of visual quality and, vice versa.

4. Repeat 1, 2, and 3 for each low-bitwidth DCNN

5.1.2 Test Statistics

We use the Relative Maximum Mean Discrepancy (RMMD) Test proposed by [BBB+15] as our test statistic. The RMMD is an extension of the Maximum Mean Discrepancy (MMD), two sample test proposed by [GBR+12]. Given samples $\{X_i\}_{i=1}^m$ and $\{Y_i\}_{i=1}^n$ from distributions P_x and P_y the MMD test statistic is given

by:

$$\begin{aligned} \mathrm{MMD}^2(X,Y) = \frac{1}{m(m-1)} \sum_{i=1}^{m} \sum_{j \neq i}^{m} k(x_i, x_j) \\ + \frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j \neq i}^{n} k(y_i, y_j) - \frac{2}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} k(x_i, y_j) \end{aligned} \tag{5.1}$$

the null hypothesis $H_0 : P_x = P_y$ is tested versus alternative $H_1 : P_x \neq P_y$. In the above equation, k is the Radial Basis Function given by [GBR+12]

$$k(x, y) = \exp ||x - y|| \tag{5.2}$$

The RMMD test builds upon the standard MMD framework by computing the MMD test statistic between two pairs of distributions. Given samples $\{X_i\}_{i=1}^{m}$, $\{Y_i\}_{i=1}^{n}$, and $\{Z_i\}_{i=1}^{r}$ respectively from the training data, low-bitwidth DCNN, and full-precision DCNN, RMMD tests the null hypothesis $H_0 : MMD^2(X,Y) < MMD^2(X,Z)$ against the alternative $H_1 : MMD^2(X,Z) < MMD^2(X,Y)$.
The equation of RMMD test, distribution of test statistics, and p-value for testing H_0 against H_1 are given by [BBB+15]:

$$ts = -\frac{\mathrm{MMD}_u^2(X_m, Y_n) - \mathrm{MMD}_u^2(X_m, Z_r)}{\sqrt{\sigma_{XY}^2 + \sigma_{XZ}^2 - 2\sigma_{XYXZ}}} \tag{5.3}$$

$$t_s \sim \mathcal{N}\left[0\,, \quad \begin{pmatrix} \sigma_{XY}^2 & \sigma_{XYXZ} \\ \sigma_{XYXZ} & \sigma_{XZ}^2 \end{pmatrix}\right] \tag{5.4}$$

$$p \leq \Phi(t_s) \tag{5.5}$$

where Φ is the Normal Cumulative Distribution Function, σ_{XY} denotes the covariance matrix of $\mathrm{MMD}_u^2(X_m, Y_n)$, σ_{YZ} denotes the covariance matrix of $\mathrm{MMD}_u^2(Y_n, Z_r)$, σ_{XYXZ} denotes the covariance matrix of $\mathrm{MMD}_u^2(X_m \cup Y_n, X_m \cup Z_r)$. The p-value in the above equation indicates the probability that, based on the observed samples, the distribution based on the low bitwidth DCNN is closer to the training data

than the distribution based on the full precision DCNN is to the training data.

5.2 Roofline Analysis

The generative quality is determined by choosing the optimal bitwidth using the procedure described in Section 5.1. We now turn to further increasing the throughput by optimizing with respect to T_{O_H}, T_{O_W}, T_{O_C}, and T_{I_C}, which are the height, width, channel size of the output block, and the channel size of input block respectively (see Figure 4.1). This is done using the roofline analysis proposed in [ZLS+15]. Figure 5.3 shows an example roofline plot where the X axis denotes the number of operations per memory access and Y axis denotes the number of operations per cycle.

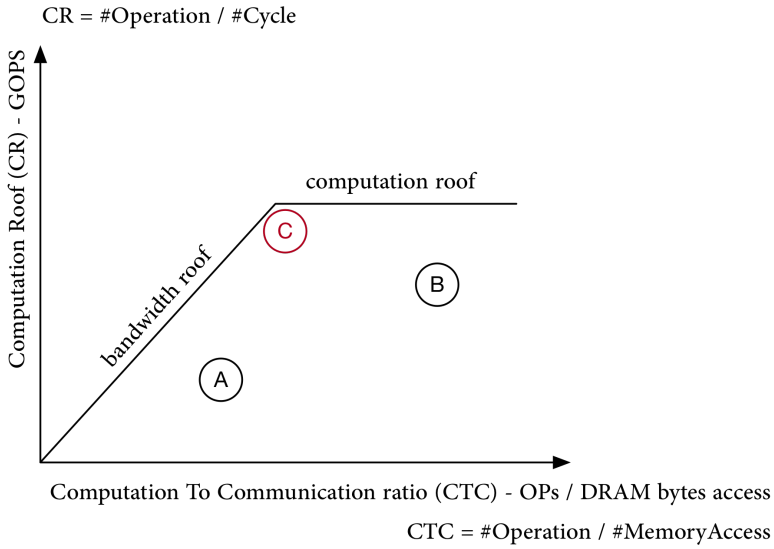


Figure 5.3: Roofline Model, adopted from [ZLS+15]

In this drawing, A, B and C correspond to designs of accelerator with different values of $T_{O_H}, T_{O_W}, T_{O_C}, T_{I_C}$. Design A transfers too much data, so computation speed is low, and therefore falls well beneath the computation roof. Design B lies well beneath the bandwidth roof, which means the system performance is dominated by memory transfers. Design C is more efficient than A and B and has a balance between computation speed and memory bandwidth. This technique is

described in [ZLS+15] where it was used for the design of convolution accelerator. We apply roofline analysis to design a deconvolution accelerator and estimate the computation to communication ratio (CTC) and computational roof (CR) for a given deconvolution layer.

Computation to Communication Ratio

Let α_{in}, α_w, α_{out} and B_{in}, B_w, B_{out} denote the trip counts and buffer sizes of memory accesses to input/output feature maps, weights, respectively. The CTC is given by [ZLS+15]:

$$\begin{aligned} \text{CTC} &= \frac{\text{total number of operations}}{\text{total amount of external memory access}} \\ &= \frac{2 \times I_C \times O_C \times I_H \times I_W \times K^2}{\alpha_{in} B_{in} + \alpha_w B_w + \alpha_{out} B_{out}} \end{aligned} \tag{5.6}$$

$$\alpha_{out} = \frac{O_C}{T_{O_C}} \frac{O_H}{T_{O_H}}, \alpha_{in} = \alpha_w = \frac{I_C}{T_{I_C}} \alpha_{out} \tag{5.7}$$

$$B_{in} = T_{I_C} \left(\frac{T_{O_H} + K}{S} \right) \left(\frac{T_{O_W} + K}{S} \right) \tag{5.8}$$

$$B_{out} = T_{O_C} T_{O_H} T_{O_W}, \qquad B_{weight} = T_{O_C} T_{I_C} K^2 \tag{5.9}$$

$$0 \leq B_{in} + B_w + B_{out} \leq \text{BRAM}_{\text{capacity}} \tag{5.10}$$

Computation Roof

Let PD denotes the pipeline depth and II is the number of cycles between the start of each loop iteration T_{O_W}, the CR is given by [ZLS+15]:

$$\begin{aligned} \text{CR} &= \frac{\text{total number of operations}}{\text{number of execution cycles}} \\ &= \frac{2 \times I_C \times O_C \times I_H \times I_W \times K^2}{\alpha_{in} K^2 T_{O_H} (\text{PD} + \text{II}(T_{O_W} - 1))} \end{aligned} \tag{5.11}$$

where

$$\begin{cases} 0 \leq T_{O_C} T_{I_C} \leq (\#\,\text{of DSPs}) \\ 0 < T_{I_C} \leq I_C \\ 0 < T_{O_C} \leq O_C \\ 0 < T_{O_H} \leq O_H \\ 0 < T_{O_W} \leq O_W \end{cases}$$

Note that $0 \leq T_{O_C} T_{I_C} \leq (\#\,\text{of DSPs})$ will not hold true when the bitwidth is greater than 18, because the maximum bitwidth of the multipliers used in our implementation is 18-bit [XIL16]. Since we use a bitwidth of 12 in all our experiments this constraint is therefore valid.

5.3 VLSI Level Optimization

5.3.1 Loop Unrolling

Loop unrolling is a key technique of high level synthesis [CM08]. It works by generating parallel hardware to accelerate FPGA program execution. Figure 5.4 illustrates how it works.

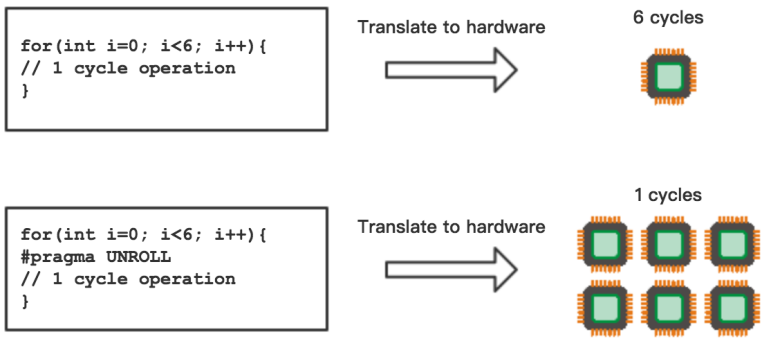


Figure 5.4: Loop Unrolling

The innermost loop T_{O_C} and T_{I_C} in Algorithm 3 are unrolled and can be executed in a constant amount of cycles P, which forms the processing engine as shown in Figure 5.5.

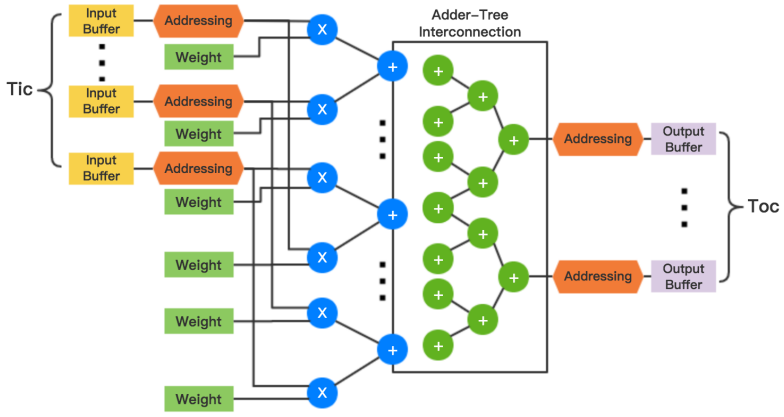


Figure 5.5: Processing Engine

5.3.2 Loop Pipelining

Loop pipelining is a fundamental technique to improve throughput. We pipeline the loop T_{O_W} in Algorithm 3 with an interval of 2. The pipeline execution paradigm shows in Figure 5.6.

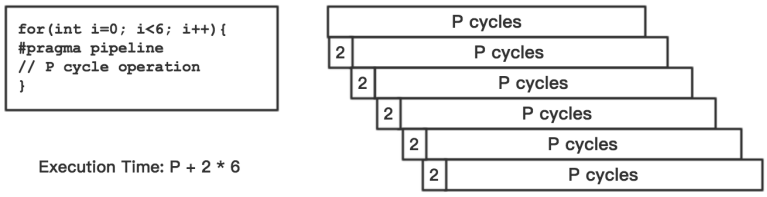


Figure 5.6: Pipelining Execution

5.3.3 Memory Partitioning

Memory partitioning can increase the available bandwidth of on-chip local memory by creating more read and write interfaces (shown in Figure 5.7).

Figure 5.7: Visualization of Memory Partitioning

5.3.4 Register Insertion

The critical path length and pipeline interval are constrained by the on-chip local memory bandwidth, especially when the size of the processing engine is large. To further improve performance, we insert registers to economize local memory bandwidth, which is illustrated in Figure 5.8.

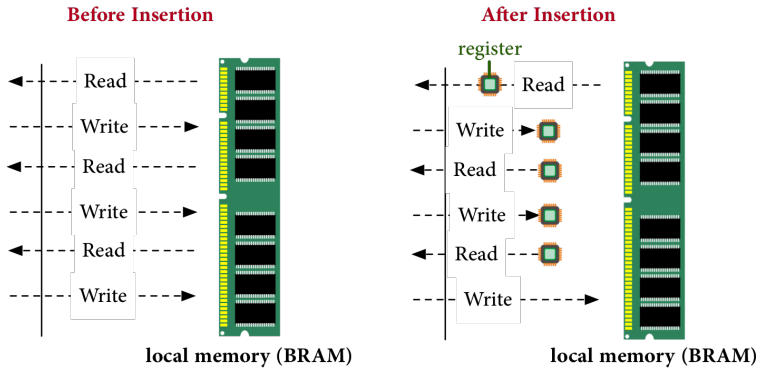


Figure 5.8: Insert register to reduce local memory (BRAM) writes

Chapter 6

Evaluation

In the previous chapter, we explained the three-step design methodology. In this chapter, we show the experimental setup and results.

6.1 DCNN implementation in Tensorflow

We train two deconvolutional neural networks (DCNNs) using the generative adversarial network (GAN) method (described in chapter 2) on the MNIST and CelebA Human Face datasets [LLWT15]. The training work flow is implemented using the Python Programming Language [Lan17] and Tensorflow [AAB+16] on a K-80 GPU.

6.1.1 Dataset Overview

MNIST is a database of handwritten digits with a training set of 60,000 examples and a test set of 10,000 examples [LBBH98]. Figure 6.1 shows some sample images from the database. The size of each image is 28×28.

Figure 6.1: Samples from MNIST database [LBBH98]

CelebA is a database of human faces with a set of 202599 examples in total [LLWT15]. Figure 6.2 shows some sample images from the database. We scale each image to 3×64×64, where "3" denotes RGB channels.

Figure 6.2: Samples from CelebA Face database [LLWT15]

6.1.2 Network Configuration

Figure 6.3 and 6.4 shows the configurations of the MNIST and Face DCNN.

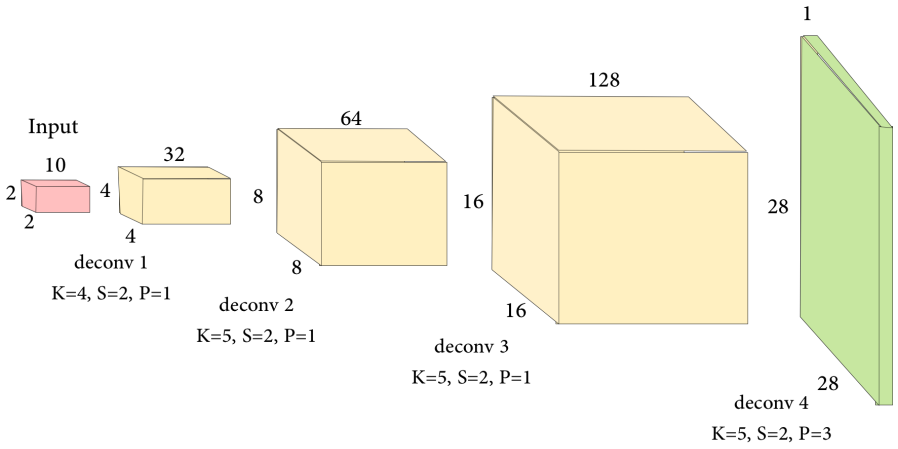


Figure 6.3: MNIST Deconvolutional Neural Network Configuration.

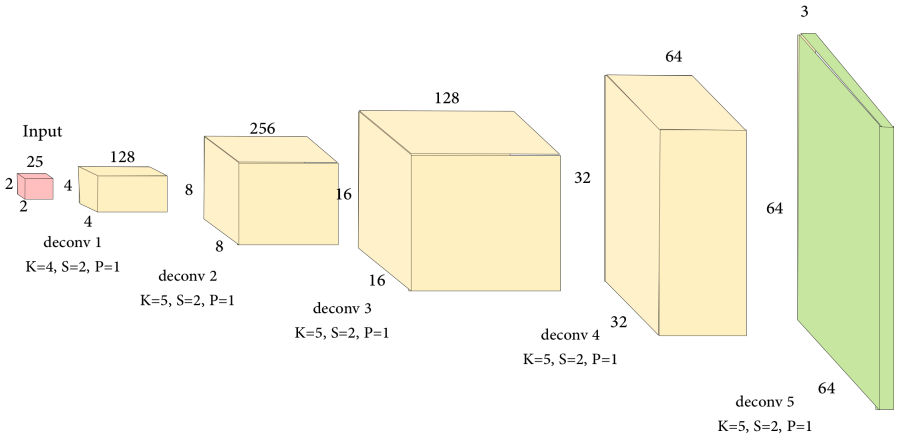
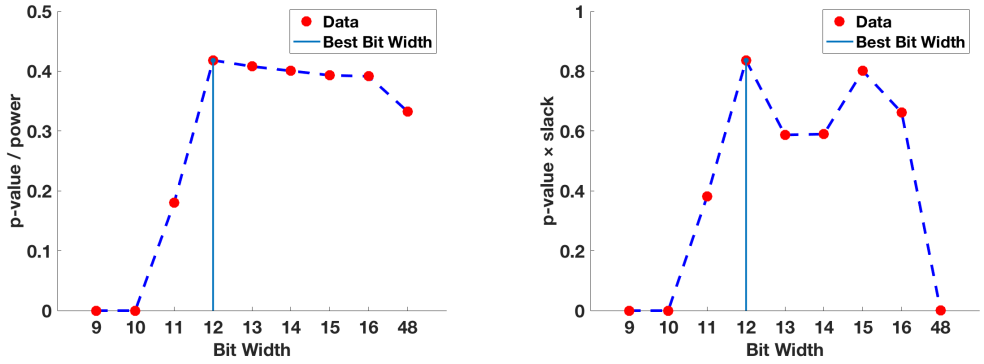


Figure 6.4: Face Deconvolutional Neural Network Configuration.

6.2 Statistical Analysis

We export the parameters of the DCNNs described above to a hardware simulator to generate samples in different bitwidth settings, and study the trade-off between generative quality and system complexity over a range of bitwidths by determining p-value × minimum slack and p-value/power as a function of bitwidths. The two curves are shown in Figure 6.5. Both curves peak at bitwidth 12, which we take to be a good choice because it represents a high p-value (generative quality) with a low power consumption and high minimum slack.



(a) p-value/power vs bitwidth

(b) p-value×slack vs bitwidth

Figure 6.5: Approximate concave curves based on trade-off between generative quality and implementation complexity.

6.3 Hardware System

We implemented the deconvolution accelerator IP with Vivado HLS (v2016.2). We use *ap_fixed.h* from Vivado Math Library to implement fixed point arithmetic operations with arbitrary bitwidth precision, and use *hls_stream.h & ap_axi_sdata.h* to model streaming data structure. The hardware system is built on a Xilinx Zynq-7000 FPGA XZ7020 with Vivado Design Suite and Xilinx SDK. The FPGA 7Z020 is programed with our accelerator IP and the ARM processor is used to initialize the accelerator, set parameters, and transfer data for each layer. An overview of the implementation block diagram is in Figure 6.6 and the Vivado Design block diagram is shown in the Figure 6.7.

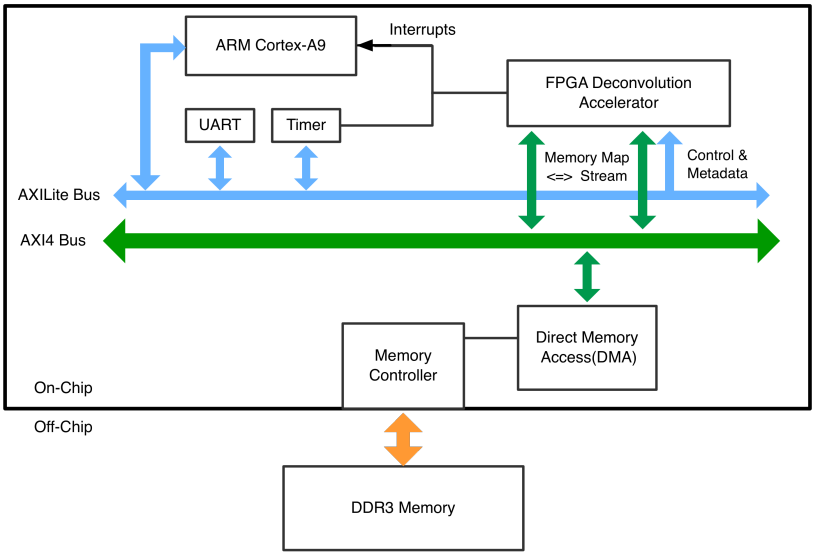


Figure 6.6: Overview of Implementation Block Diagram.

The FPGA 7Z020 is similar in size to a quarter coin and has 220 DSP slices, 85K logic ceils, 4.9MB block rams and 200 I/O pins. Our accelerator can fit into this small chip and be shipped into much smaller embedded devices in production.

6.4 Generation Result

Figure 6.8 shows some generated faces and digits from our trained DCNNs. Figure 6.9 shows the output of DCNNs under different bitwidths for the same input. Visually evaluating the degradation of image quality is only feasible in the cases of extremely low bitwidth such as 8 bits. Our proposed methodology provides an analytical framework for quantifying the trade-off between image quality and implementation complexity over a range of bitwidths.

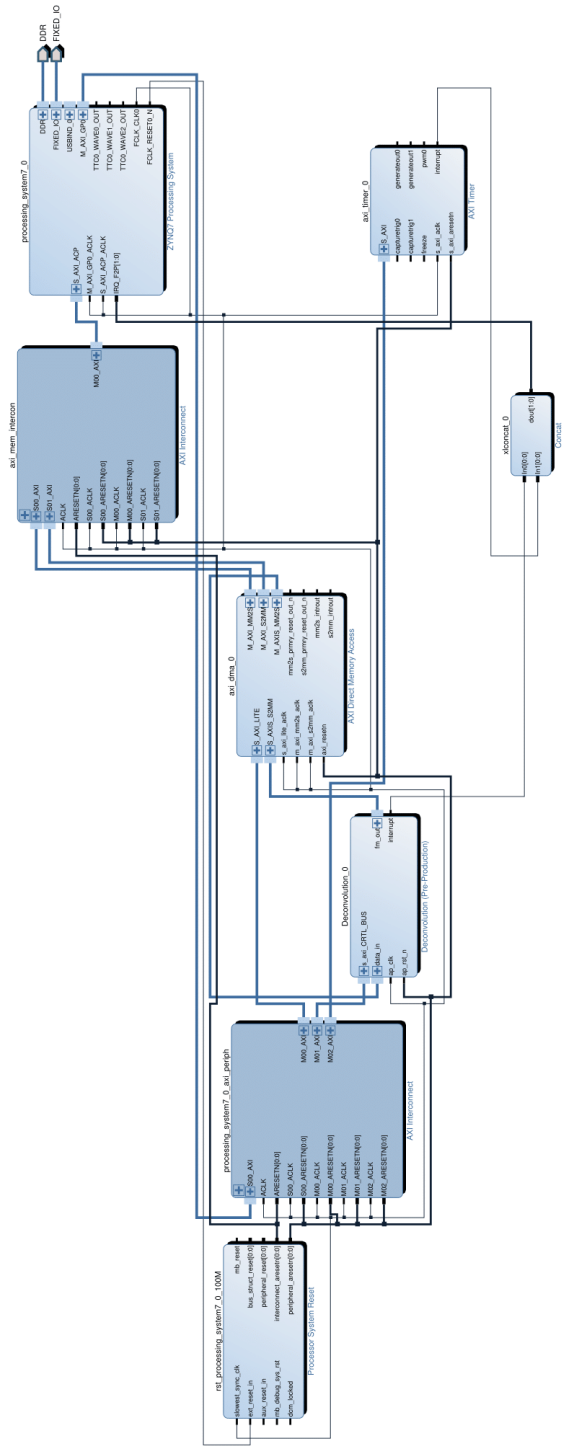


Figure 6.7: Detailed Hardware Block Design, where the deconvolution block denotes the accelerator, and the zynq7 processing system represents the ARM processor.

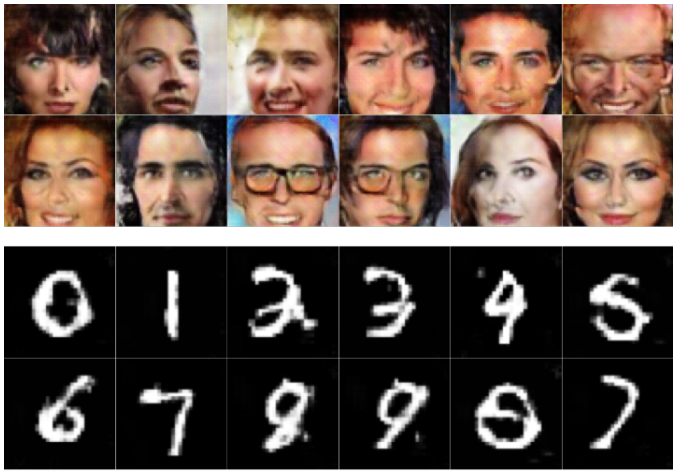


Figure 6.8: Sample MNIST and CelebA images generated by the full precision DCNN.

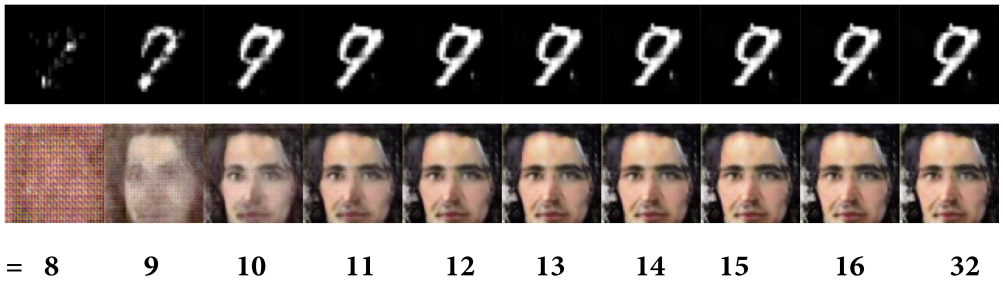


Figure 6.9: Images generated by different bitwidth DCNNs.

6.5 Roofline Analysis

Figure 6.10 shows all constraint-admissible design solutions for the first layer of our CelebA DCNN, where the best design is shown as located at the left corner of the roof.

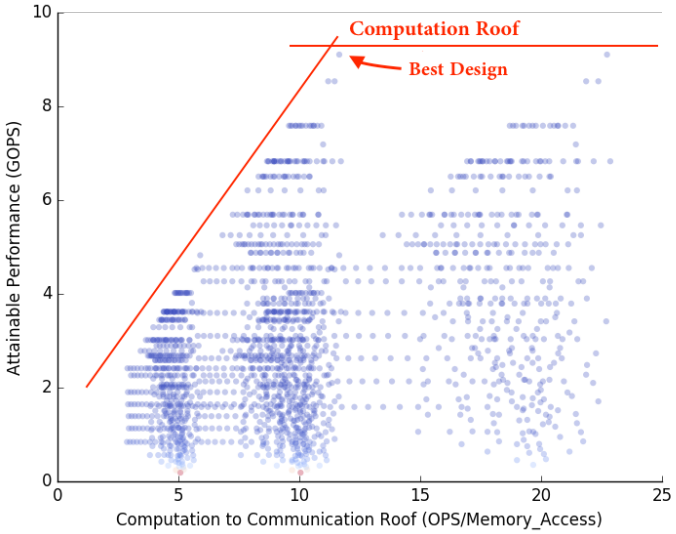


Figure 6.10: Design space Exploration for a layer with input 10x2x2 and output 64x4x4.

6.6 Performance

Table 6.1 shows the utilization rate after place and route, and we compare our DCNN performance with some existing CNN accelerators for reference in table 6.2. The performance can be further improved by implementing a ping-pong buffer in our system.

Table 6.1: FPGA Resource Utilization

DSP	LUT	FF	BRAM
95%	48%	29%	48%

Table 6.2: Comparison to previous implementations

	Chip	Precision	#DSP	Freq	GOPS	GOPS/DSP
[PSMC13]	VLX240T	Fixed	768	150M	17	0.022
[ZLS+15]	VX485T	Float	2800	100M	61.62	0.022
Ours	7Z020	16Fixed	220	100M	2.6	0.012

Chapter 7

Conclusion

7.1 Summary

In this thesis, we develop an FPGA-based deconvolution accelerator for deconvolutional neural networks and propose a three-step design methodology which first uses statistical analysis to find the most cost-efficient bitwidth, then explore the design space with roofline model [ZLS+15], and use VLSI optimization methods to produce the final design. Finally, we implement our method on a Zynq-7000 FPGA and realize a performance density of 0.012 GOPs/DSP.

7.2 Future Work

7.2.1 ARM Code Optimization

The opportunity remains to significantly optimize the C code we wrote for the embedded arm processor, which will increase the overall throughput.

7.2.2 Use a Larger FPGA

The Xilinx FPGA 7Z020 is a small board with limited resources and designed for embedded applications. Given the increasing popularity of deploying large FPGAs (i.e. Xilinx Virtex 7 Series) in data centers, it is worth testing our accelerator

on large FPGA boards.

7.2.3 Ping-Pong Buffer

By using a ping-pong buffer, the throughput of an FPGA can be significantly increased because memory transfer and computation can be fully parallelized under that setting. Since our implementation is the first attempt to design an FPGA-based accelerator for deconvolution, and [SFM16] shows that high-level synthesis may not be able to generate correct verilog implementation when ping-pong buffer is used, we did not include it in the current version of our accelerator.

7.2.4 Convolution and Deconvolution

In Chapter 4, we showed that our deconvolution accelerator reuses the same computational architecture with the one proposed in [ZLS+15]. It will be interesting to investigate the use of the accelerator in applications involving sequential alternating convolutions and deconvolutions.

7.2.5 From Prototype to Applications

Although we successfully implemented our accelerator on an FPGA, our implementation does not have proper input and output devices. To run a DCNN on our current implementation, we need to compile network parameters into a C program executable for ARM, which is not automatic. For this prototype to be useful, we need to support camera, monitor, Linux OS, file system and IP network functions in the C source code for ARM.

7.2.6 Low Bitwidth Training

In this work, we train our DCNNs using tensorflow in double precision, and scale them to 12bit for FPGA with tolerable generative quality degradation. However, [ZWN+16] shows that the quantization step and its performance penalty can be avoided for discriminative CNNs by training neural networks under low bitwidth settings. For achieving better generative quality and higher throughput in

hardware, it is worth investigating low bitwidth training problem for generative neural networks like DCNNs.

ACKNOWLEDGEMENTS

All of the chapters are currently being prepared for submission for publication of the material in "A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA", Xinyu Zhang, Srinjoy Das, Ojash Neopane, and Ken Kreutz-Delgado. The thesis author was the primary investigator and author of this material.

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[AB17] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. In *NIPS 2016 Workshop on Adversarial Training. In review for ICLR*, volume 2016, 2017.

[BBB+15] Wacha Bounliphone, Eugene Belilovsky, Matthew B Blaschko, Ioannis Antonoglou, and Arthur Gretton. A test of relative similarity for model selection in generative models. *arXiv preprint arXiv:1511.04581*, 2015.

[BKC15] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[BPW+12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[CLL+14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[CM08] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[DR95] Gunhan Dundar and Kenneth Rose. The effects of quantization on multilayer neural networks. *IEEE Transactions on Neural Networks*, 6(6):1446–1451, 1995.

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

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GBR+12] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[Goo16] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks, 2016.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

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

[IZZE16] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arXiv preprint arXiv:1611.07004*, 2016.

[Lan17] Python Programming Language. Python software foundation, 2017.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LGS+15] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang. Rram-based analog approximate computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(12):1905–1917, Dec 2015.

[LLWT15] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[NHH15] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1520–1528, 2015.

[PSMC13] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19. IEEE, 2013.

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

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[Sch97] Robert R. Schaller. Moore's law: Past, present, and future. *IEEE Spectr.*, 34(6):52–59, June 1997.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[SCH+16] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1874–1883, 2016.

[SFM16] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. *arXiv preprint arXiv:1607.00064*, 2016.

[Shi00] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 90(2):227–244, 2000.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Wik17a] Wikipedia. Alphago versus lee sedol — wikipedia, the free encyclopedia, 2017. [Online; accessed 24-January-2017].

[Wik17b] Wikipedia. Go and mathematics — wikipedia, the free encyclopedia, 2017. [Online; accessed 15-March-2017].

[WLW+16] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[WXT+16] Yu Wang, Lixue Xia, Tianqi Tang, Boxun Li, Song Yao, Ming Cheng, and Huazhong Yang. Low power convolutional neural networks on a chip. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pages 129–132. IEEE, 2016.

[WZX+16] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T Freeman, and Joshua B Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems*, pages 82–90, 2016.

[XIL16] XILINX INC. *7 Series DSP48E1 Slice*, 9 2016. Rev. 1.9.

[YSZ+15] Fisher Yu, Ari Seff, Yinda Zhang, Shuran Song, Thomas Funkhouser, and Jianxiong Xiao. Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*, 2015.

[ZKTF10] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2528–2535. IEEE, 2010.

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

[ZWN+16] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.