Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Fused Convolutional Neural Network Accelerators

A Thesis presented

by

Manoj Alwani

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

December 2015

Stony Brook University

The Graduate School

Manoj Alwani

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Michael Ferdman - Thesis Advisor
Assistant Professor, Computer Science

Dr. Nima Honarmand
Assistant Professor, Computer Science

Dr. Dimitris Samaras - Thesis Committee Chair
Associate Professor, Computer Science

Dr. Alex Berg
Assistant Professor, Computer Science, University of North Carolina at Chapel Hill

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Thesis

Fused Convolutional Neural Network Accelerators

by

Manoj Alwani

Master of Science

in

Computer Science

Stony Brook University

2015

Deep convolutional neural networks (CNNs) are rapidly becoming the dominant approach to computer vision and a major component of many other pervasive machine learning tasks, such as speech recognition, natural language processing, and fraud detection. As research and development of CNNs progresses, the size of the networks grows, leading to large increases in the computation and bandwidth required to evaluate these networks. Typical CNNs in use today already exceed the capabilities of general-purpose CPUs, resulting in rapid adoption and active research of CNN hardware accelerators such as GPUs, FPGAs, and ASICs. In this work, we develop a novel CNN accelerator architecture and design methodology that breaks away from the commonly accepted practice of processing the networks layer by layer. By modifying the order in which the original input data are brought on chip, changing it to a pyramid-shaped multi-layer sliding window, our architecture enables effective on-chip caching during CNN evaluation. The caching in turn reduces the off-chip memory bandwidth requirements, which is a primary bottleneck in many CNN environments.

Contents

1 Introduction **1**

2 Background on CNNs **4**
2.1 Applications to Deep Learning 6
2.2 FPGA Implementation 7

3 Cross-Layer Optimization **9**
3.1 The Cost of Data Transfer 9
3.2 Cross-Layer Design Optimization 13
3.2.1 Recomputing vs. Storing 16
3.3 Methodology 18
3.3.1 CNN Accelerator Architecture 18
3.3.2 HLS-Based Accelerator Specification 19
3.4 Results 25
3.4.1 FPGA Evaluation 25
3.4.2 FPGA Implementation 25
3.4.3 Resource Utilization 26
3.4.4 Comparison to Related Work 26
3.4.5 Model for computation and communication 28

4 Opportunities for Cross-Layer Optimization **29**
4.1 Exploration Tool 29
4.2 Tradeoff Evaluation 31
4.3 Evaluation of pyramid pipelining 33

5 Related Work **37**
5.1 Convolutional Neural Networks on GPUs 38
5.2 CNNs on FPGAs 38

5.3 CNNs on ASICs 40

6 Conclusions **42**

List of Figures

2.1 Example of convolution layer operations. There are N input feature maps, M output feature maps and $M \times N$ filters each of size $K \times K$ 5
2.2 AlexNet Network [20] which won ImageNet 2012 competition . . 5

3.1 Input, output and weight size for different layers of VGG. This data assumes that the pooling layers are merged into their prior convolution layer, e.g. "Conv 4" contains one convolutional and one pooling layer. 11
3.2 Example of back-mapping and overlapped computation 12
3.3 Example of single pyramid and multi-pyramid approach 17
3.4 Accelerator Design with fused N layers. 18
3.5 convolution Engine Design with unroll factor TM = 2 and TN = 3 23
3.6 Pipelining applied to fused CNN accelerator 25

4.1 Tradeoffs of layer fusion of VGG network 32
4.2 Tradeoffs for layer fusion of AlexNet network 33
4.3 Different combinations for VGG 35
4.4 Different combinations for AlexNet 36

List of Tables

2.1 Convolution layers parameters 6

3.1 FPGA Resource Utilization 26
3.2 FPGA Occupation Comparison 26
3.3 Resource usage of convolution layer 1 and 2 27
3.4 Resource usage of fused accelerator on VCX707 28

Acknowledgements

I would like to express my gratitude to my advisor Dr. Michael Ferdman for his constant encouragement and guidance I received during the tenure of my Master thesis. From his serious attitude towards everything, I have not only learnt to pursue perfection but also truly learnt how to be a work in the multiple fields. Furthermore, I would like to thank my committee members, Dr. Nima Honarmand, Dr. Dimitris Samaras, and Dr. Alex Berg, for their precious time.

I express my deepest gratitude to Dr. Peter Milder for his invaluable help and discussion through out my master program. His constructive suggestions have always been helpful for the improvement of my work.

I would like to thank my family and friends, who have always supported me throughout entire process. Finally, I would like to specially thank the members of the Computer Architecture Stony Brook (COMPAS) Lab for their support and friendship during the past one and half years.

Chapter 1

Introduction

Deep learning is a branch of machine learning which process the data in multiple layers to model high level abstraction of data and increase prediction accuracy. There are various deep learning architectures such as Convolution neural network, deep belief networks and recurrent neural networks which are applied to fields like Computer Vision, Natural Language Processing and bioinformatics where they have achieved state-of-the-art results on various task. In these architectures, Convolution Neural Networks have got lot of attention because they are inspired by living creatures vision system and have been applied in the various fields using the same architecture.

Deep convolutional neural networks (CNNs) have revolutionized the accuracy of recognition in computer vision. More broadly, this is part of a trend—using CNNs with many layers—that has been instrumental to rapid progress on accuracy in natural language processing, information retrieval, and speech recognition.

Underlying the accuracy improvements of CNNs are massive increases in computation. With each newly developed network, as the accuracy of recognition increases, the number of computations required to evaluate the network also grows. Already, general-purpose CPUs have become a limiter for modern CNNs because of the lack of computational parallelism. As a result, there has been significant interest in developing and adapting hardware accelerators for CNNs [17] such as GPUs [10], FPGAs [26, 33, 23, 7], and ASICs [8].

Although the CNN computation is mathematically simple, the sheer volume of operations precludes a dataflow implementation even for a single layer. Each convolution layer requires iterative use of the available compute units. Research into the design of CNN accelerators has therefore concentrated on developing a CNN "building block" that iteratively evaluates the network. A number of methodolo-

gies have been developed for optimizing the architecture of such CNN accelerator building blocks, concentrating either on specific constraints [23] or evaluating the design space of compute units and memory bandwidth [33].

Traditional implementations of CNNs (both hardware and software) evaluate the network by following its structure, one layer at a time. This approach produces a large amount of hidden layer data (the layer between input and output are called hidden layers) that are gradually streamed out to memory as the computation progresses. Upon completing a layer, the hidden layer data are streamed back as an input to the same compute units, repeating the process until all layers have been evaluated. As the size of CNNs grows, the amount of hidden layer data that must be shuttled between the compute units and memory increases and the system becomes memory-bandwidth bound, limiting further performance gains even if more compute resources are available.

We observe that an additional dimension exists for CNN accelerator architectures that focuses on the dataflow *across* convolutional layers. Rather than processing each CNN layer to completion before proceeding to the next layer, it is possible to restructure the computation such that multiple convolutional layers are computed together as the input is brought on chip, obviating the need to store or retrieve the intermediate data from off-chip memory. This accelerator organization is made possible by the nature of CNNs. That is, each point in a hidden layer output of the network depends on a well-defined region of the initial input to the network.

In this work, we develop a novel CNN accelerator architecture and design methodology that breaks away from the commonly accepted practice. By modifying the order in which the original input data are brought on chip, changing it to a pyramid-shaped multi-layer sliding window, our architecture enables effective on-chip caching during CNN evaluation. The caching in turn reduces the off-chip memory bandwidth requirements which is a primary bottleneck in many CNN environments. Our architecture promotes energy-efficiency by minimizing data movement and improves performance in bandwidth-limited scenarios.

We validate our approach by implementing the proposed CNN architecture on a Xilinx Virtex-7 FPGA. Using analytic modeling and our FPGA prototype, we demonstrate:

- Recompute and reuse model to fuse multiple layers of network for computation which saves external memory access of fused layers.

- Benefits of reuse model over recompute model.

- Applicability of our approach/mathematical model.
- We have shown that our approach gets 1.62x increase in computation to communication ratio over the state-of-the-art design and saves 2.17MB of off-chip data transfer by caching the computations using 55.86KB of on-chip buffer.

The rest of this thesis is organized as follows. In Chapter 2 we provide the relevant background on CNNs and their implementation on FPGA. In Chapter 3 we have explained our method in detail and compared our method with state of the art approach. In Chapter 4 we have explained our evaluation tool which can take any network and explore all the possible solutions of layer fusion. In Chapter 5 we summarize the related work and in Chapter 6 we conclude."

Chapter 2

Background on CNNs

A convolutional neural network (CNN) performs feature extraction using a series of *convolutional layers*, typically followed by one or more dense ("fully connected") neural network layers and finally a classification layer. Figure 2.1 shows an example of one convolutional layer. Each layer takes as input a *feature map* (consisting of N channels of $R \times C$ values), and convolves it with $M \times N$ different $K \times K$ filters (whose weights were previously determined through a learning algorithm such as back propagation). The convolution is performed by sliding filters across the input feature map with a stride of S. (Where a filter moves S locations at each step.) At each location, a filter's values are multiplied with the overlapping values of the feature map. The resulting products are summed together, contributing to one value in the output feature map (which consists of M channels of two-dimensional values of dimensions $(\frac{R-K}{S}+1) \times (\frac{C-K}{S}+1)$).

Typically, the output feature map then undergoes a non-linear operation such as rectification (e.g., ReLU [20]), optionally followed by a subsampling operation (e.g., pooling).The nonlinear and subsampling operations are small, operating locally on a single channel of the feature map; these typically require a very small percentage of the overall computation. The filter weights of the convolution layers are trained using back propagation algorithm. Figure 2.2 [20] shows a CNN which won the ImageNet 2012 contest. This network contains 5 convolution layers and 3 pooling layers, which are followed by 3 dense fully connected layers and classification layer. The classification layer has 1000 elements which shows likelihood of 1000 categories. Each convolution layer in this network is followed by ReLU non-linearity. As shown in the Figure 2.2, layer 1 in the network receives 3 input features of size 224×224 and generates 96 feature maps as output of 55×55 size. Layer 1 is a convolution layer with kernel size 11 $\times$11 and stride 4. The out-

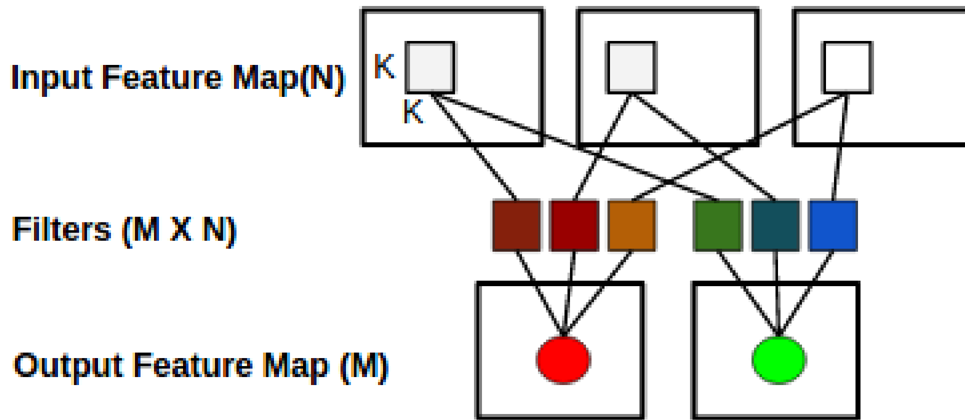

Figure 2.1: Example of convolution layer operations. There are N input feature maps, M output feature maps and $M \times N$ filters each of size $K \times K$

put of layer 1 is partitioned in two sets each of size 48 feature maps. These 2 sets are then trained parallel in two different GPUs which makes training of these networks faster. The network again merge the inputs after second pooling and then again partition the output in two sets. Table 2.1 which shows parameters of all the convolution layers of this network. The training of this network take **six days** on GPUs and time is higher for networks with deeper layers. This network got benchmark results and got more than 10% higher accuracy than traditional vision based approach which sparked the interest in the implementation of deeper CNN. Current networks use more than 30 convolution layers in the CNN to increase the recognition accuracy.

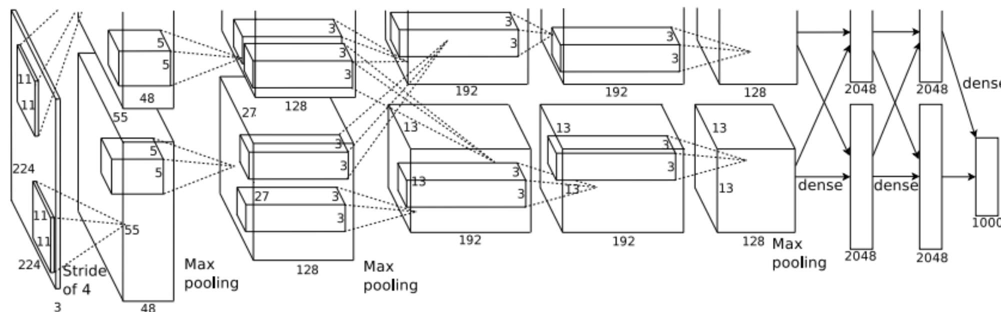

Figure 2.2: AlexNet Network [20] which won ImageNet 2012 competition

Table 2.1: Convolution layers parameters

Layer	1	2	3	4	5
input channels	3	48	256	192	192
output channels	48	128	192	192	128
output rows	55	27	13	13	13
output cols	55	27	13	13	13
Kernel size	11	5	3	3	3
Stride	4	1	1	1	1
Sets	2	2	2	2	2

The training of CNNs are usually done on GPU because GPUs provides large amount of parallelism and huge data transfer rates but the power requirement of GPUs is high which make them unsuitable for embedded platforms.

2.1 Applications to Deep Learning

Deep learning is a branch of machine learning which process the data in multiple layers to model high level abstraction of data. There are various deep learning architectures such as Convolution neural network, deep belief networks and recurrent neural networks which are applied to fields like Computer Vision, Natural Language Processing and bioinformatics where they have achieved state-of-the-art results on various task. In these architectures, Convolution Neural Networks have got lot of attention because they are inspired by living creatures vision system. Some the applications of CNN are mentioned below:

Image Recognition: CNNs are often used for image recognition. They have achieved an error 0.23 % on digit recognition task [11] which is the lowest error rate achieved on that dataset. They are also used for object classification and detection. For example, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is a benchmark in object classification and detection, with millions of images and hundreds of object classes. In the ILSVRC 2014, almost every highly ranked team used CNN as their basic framework. The winner GoogLeNet [30] applied CNN with more than 30 layers and achieved the performance which are close to human accuracy. The CNN are also being used for video analysis [19] and face detection [16].

Natural Language processing: CNNs are also being applied to natural lan-

guage processing [13] to detect part-of-speech tags, semantic role labels and language modeling. In [13], resercher have shown how a single architecture can be used to do different tasks in Natural Language Processing.

Gaming: CNNs are also being used in games in which CNNs are trained by supervised learning from database of human professional games [12] and also been used with re-inforcement learning to learn the games itself [22].

Drug Discovery: CNNs also becomes famous recently as they have used in finding the treatment of **ebola virus** [32]. Not only ebola they are also used for predicting the treatment of multiple sclerosis and identifying the type the brain and breast cancer.

2.2 FPGA Implementation

As computation requirement of CNNs are large, training these networks take lot of time even on GPUs. The massive amount of parallelism provided by GPUs and huge data transfer rates make them ideal candidate for training and testing of these networks but GPUs consume lot of power in training and testing which make them far from ideal for embedded systems. As a result, a number of researcher have proposed implementation of CNNs on FPGA and ASIC platforms [33, 8, 7]. The FPGA based accelerator have got of attention in the recent years because they are highly energy efficient, provide good performance and give reconfiguration options. All of these techniques focus their attention of optimizing convolution layer as it is the most computational and bandwidth demanding layer of CNN. They consider convolution layer of CNN as a deep nested loop and apply loop transformation and tiling to increase performance of convolution. Code 2.1 shows psuedo code of convolution layer. For simplification this code is not taking bias addition in account.

Listing 2.1: Pseudocode for Convolution Layer

```
for(m=0; m < M; m++)                // output feature map
 for(n=0; n< N; n++)                // input feature map
  for(row=0; row < R; row++)         // Rows
   for(col=0; col < C; col++)        // Cols
    for(kx=0; kx < K; kx++)          // Kernel loop X
     for(ky=0; ky < K; ky++)         // Kernel loop Y
```

```
output[m][row][col] += filter[m][n][kx][ky]
                       * input[n][S*row+kx][S*col+ky];
```

All the FPGA based method apply different transformation on code 2.1 to optimize convolution engine. In [15, 14] researchers have optimized their filtering module and unroll the kx and ky loop in code 2.1. They have implemented this design for automotive robotics. This kind of design uses very less resources and very efficient in filtering but this design uses fixed hardware for all the kernels which make them underutilized for small kernel sizes.

In [28] have optimized their kernel loops (kx, ky) as mentioned in [15] and also use parallelism with in feature map (m, n).

In [8] researcher have proposed ASIC based accelerator for ubiquitous machine learning and optimized different modules of neural networks (pooling, convolution etc.) in way so that they minimize the external data access. They have also applied loop tiling techniques in convolution and pooling layers to increase data locality and reduces external memory accesses.

In [23] they target communication of convolution engine with external memory. In this paper they use onchip buffers to maximize the data reuse and reduce the bandwidth requirement. But they don't target to maximize their computational performance.

In [33] researcher have proposed roofline model to maximize the computation and increase the bandwidth. They apply parallelism with in feature maps(M, N) and also across layer size(R, C). We have compared our proposed method as [33] address both computation and bandwidth issue of CNN. In our approach, we have not only optimized our convolution engine but also we fuse different layers of CNN which reduce the bandwidth requirement by huge amount.

Chapter 3

Cross-Layer Optimization

Convolution layer of CNN is most computation and bandwidth demanding layer. The sheer volume of operations precludes a dataflow implementation even for a single layer. Because of large amount of operations convolution layer requires iterative use of available compute units in the system. Research into the design of CNN accelerators has therefore concentrated on optimizing convolution layer that efficiently use the available compute units iteratively with minimum amount of external memory access. These techniques process the network using traditional layer-by-layer approach and don't focus on minimizing the external memory access of hidden layer data which is loaded to and fro from external memory when we move from one layer to another layer. In this chapter, we show that for deeper networks hidden layer data is too large for earlier layer of networks and efficient technique is needed to minimize external data access for these layers. We then present a new approach that fuses multiple layers which can minimize external data access for these layers. We implement a proof-of-concet accelerator using our approach using a high level synthesis tool (Vivado) and compare and contrast it with the state-of-the-art accelerator.

3.1 The Cost of Data Transfer

Existing techniques construct accelerators consisting of the multipliers and adders needed to perform the convolution, as well as the on-chip memory buffers to hold data and filter weights. The accelerators are used iteratively, performing one layer of computation at a time. For each layer, input feature maps and filter weights are brought from off-chip DRAM into local buffers, the convolutions are performed,

and data is written back into DRAM. The large volume of data comprising the feature maps can stress an accelerator's memory system and become the bottleneck. This has inspired efforts to optimize the memory accesses patterns for *this layer-by-layer approach*. For example [23] and [33] use loop transformations with the goal of balancing resources between arithmetic structures, on-chip memories, and memory bandwidth.

Importantly, although a number of approaches have focused on effectively managing on-chip memory and off-chip bandwidth while evaluating a convolutional layer, existing approaches forego the possibility of restructuring the computation *across* layers to minimize bandwidth usage. Because prior approaches consider each convolutional layer separately, they start with the assumption that every layer must bring its entire input feature map from off-chip DRAM and must write the output feature map back when the computation finishes. This transfer of feature map data to and from external memory is costly in terms of memory bandwidth and energy [23].

As deep learning algorithms continue to advance, the amount of feature map data moving between layers grows and represents an increasingly large amount of the data movement associated with the whole algorithm. For example, 25% of the overall data used in convolutional layers of AlexNet [20] (2012) were feature map data (with the rest being the filter weights); in VGG [29] (2014) and GoogLeNet [30] (2014), this value has increased to over 50%.

To illustrate this, Figure 3.1 shows the size (in MB) of the feature maps (input and output) and the filter weights of each of the convolutional layers of the VGG-E network [29]. The height of each bar represents the amount of data that must be transferred to and from DRAM (feature maps and weights) when an accelerator iteratively evaluates the layers. In the early layers, the size of the input and output feature maps dominates. For example, the first convolutional layer requires 0.6MB of input and 7KB of weights; it produces a 12.8MB output feature map. This 12.8MB feature map is then used as the input of the following layer (along with 147KB of weights). Performing the evaluation of network one layer at a time requires storing the entire 12.8MB feature map to DRAM only to immediately read it back as the input of the following layer, and repeating this back-and-forth data shuffling for every subsequent layer.

This work identifies a key opportunity in restructuring the CNN evaluation by considering the dimension of fusing the computation of adjacent layers, largely eliminating the off-chip feature map data transfer. We develop an accelerator architecture that maximizes data reuse by foregoing the prevalent assumption that

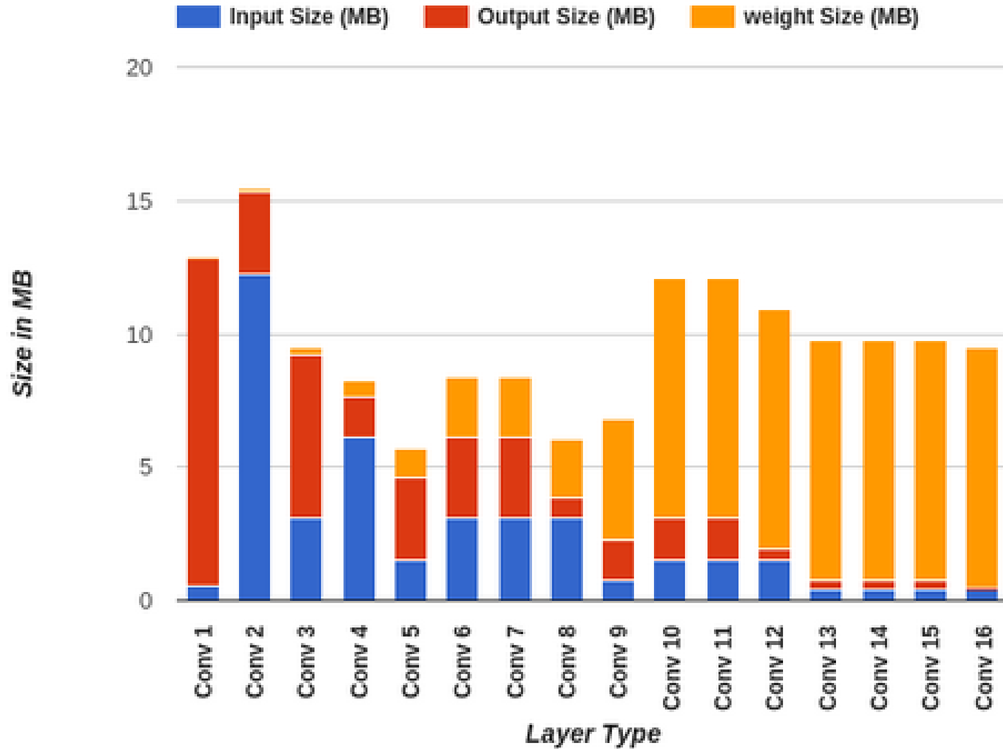

Figure 3.1: Input, output and weight size for different layers of VGG. This data assumes that the pooling layers are merged into their prior convolution layer, e.g. "Conv 4" contains one convolutional and one pooling layer.

all intermediate feature maps must be stored in off-chip memory. Our design primarily targets the early layers of the networks, whose data transfers consist predominantly of the feature map data. In our approach, only the input feature map of the first of the fused layers is brought on chip. As this initial feature map is read from memory, we compute the intermediate values of all of the fused layers that depend on the data region being read and we do not write any intermediate values out to off-chip memory. Only the output feature map of the last of the fused layers need be retained in its entirety. This last output feature map is either written to off-chip memory or simply retained in on on-chip cache (in case the last of the fused layers is one of the latter layers whose output feature maps are small enough to fit entirely on chip).

It is important to note that the dataflow of the convolution operation requires

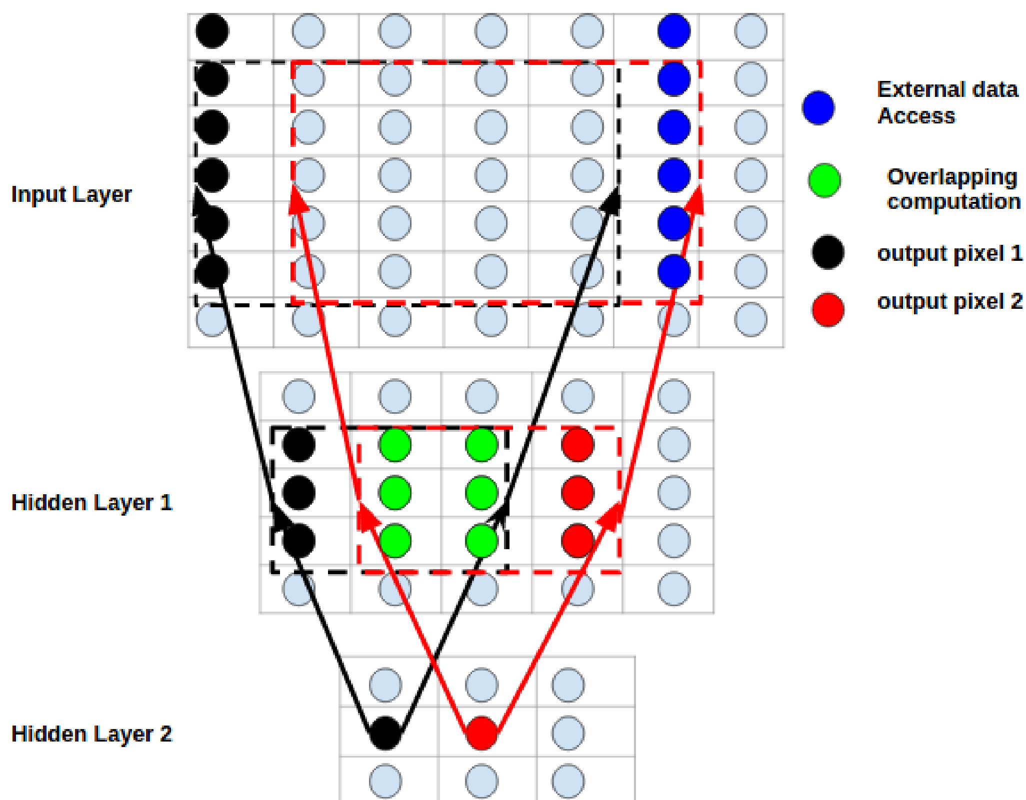

Figure 3.2: Example of back-mapping and overlapped computation

that each value in the output feature map of a layer depends on a larger region of the previous level. This fan-in pattern implies that, as the number of fused convolutional layers increases, the number of output values that can be computed by using the input region decreases. Figure 3.2 demonstrates the layer fusion process with an example. Here, we describe the operation on a 5×5 tile of the input layer (indicated with the black dashed outline). The first layer convolves this tile with a 3×3 filter, producing the 3×3 square shown in Hidden Layer 1 (also indicated with the black dashed outline). The second layer convolves this 3×3 square with another 3×3 filter, producing a single point (black circle) in Hidden Layer 2.

To reason about this process, we can use a "back-mapping" technique in which we start from a single value of an output layer and trace its dependencies to find the region of the input feature map that it depends on. If the layers are visualized

spatially, this process creates a computation **pyramid** across multiple layers of feature maps. Once the black input tile is loaded on chip, we compute the entire pyramid of intermediate values without transferring any additional feature map data to or from off-chip memory. When we reach the tip of the pyramid, only the tip of the pyramid needs to be retained as the output of the computation.

After finishing the computation of a pyramid, it is not necessary to load an entire new input tile (pyramid base) to continue. Instead, it is possible to load only one new column of the input tile (while discarding the left-most column of the old tile). We show the new input feature map data that must be loaded with the column of blue circles. The new input tile is indicated with red dashed outlines, forming a new red pyramid. At this time, the red pyramid can be evaluated in the same way, yielding one value in the Hidden Layer 2 output feature map.

Critically, some intermediate values (represented by green dots) in Hidden Layer 1 are needed for computing both the blue and red outputs of Hidden Layer 2. In other words, because their pyramids overlap, a number of intermediate values are used to compute each output value. There are two possible approaches to handle this situation. The CNN accelerator can either **recompute** the values each time they are needed or cache and **reuse** the intermediate results while computing the next pyramid. Recomputing the values obviously adds extra arithmetic operations, but has the advantage of simplicity; each pyramid's internal dataflow is the same. Caching the intermediate results saves this extra computation, but requires extra on-chip buffering and makes the computation for each pyramid irregular because some pyramids must perform more computation than others. Section 3.2 presents a framework for exploring the degrees of freedom available to fused-layer CNN accelerators, including whether it is better to cache the intermediate results or to recompute the values for each pyramid.

3.2 Cross-Layer Design Optimization

This section presents a technique for evaluating the costs and benefits of the fused-layer architecture described in Section 3.1. Given a set of layers to fuse, we start from the final layer and work backwards to find the dimensions of the pyramid. Based on the pyramid dimensions, we evaluate the costs in terms of needed storage and arithmetic operations, as well as the benefit in the amount of off-chip data transfer avoided.

The following equations allow us to calculate the dimensions of the pyramid, based on the structure of each stage. Let the size of the pyramid at the output of a

given layer be $N \times R \times C$ (by construction, if this is the final layer of a pyramid, then $N \times R \times C = N \times 1 \times 1$, a single value for all N channels.) Here N is number of channels in that layer, R is number of rows for feature map and C is number of columns for feature map. If this layer is a convolutional layer with M channels, we compute the pyramid size at this layer's *input* as $M \times R' \times C'$, according to:

$$\begin{aligned} R' &= S \cdot R + K - S \\ C' &= S \cdot C + K - S \end{aligned} \tag{3.1}$$

where the convolutional kernel is of size $K \times K$, and it is applied with stride S.

If instead the layer performs pooling and not convolution, we use (3.1), but where $K \times K$ is the size of the pooling window and S is its stride. (Because the pooling operation is performed localized over small tiles, we always fuse the pooling layer into the previous convolutional layer, as it saves bandwidth at virtually no cost.)

Following this procedure, we can start from the output of a pyramid, and calculate the dimensions of that pyramid at each level (i.e., the values at each level upon which this output depends). Based on this knowledge, we can model how the **recompute** and **reuse** models behave and compare them. First, the recompute model simply treats each successive pyramid as an independent computation, and redundantly calculates any intermediate values that the pyramid may share with its neighbors. (For example, when computing the two pyramids shown in Figure 3.2, the green circles in Hidden Layer 1 will need to be recomputed under this model.) To evaluate the costs of recomputation, we can simply find the dimensions of a pyramid for one value in the pyramid's output layer. Then, we move the output value one location over and find the dimension and locations of its pyramid. By calculating all the locations where these pyramids overlap (e.g. the green circles in Figure 3.2), we can calculate the number of computations that must be repeated under the recompute model.

The recompute model treats every pyramid's computation as an independent operation. However, we can see that as a pyramid shifts from one location to the next, a portion of the computation needed for the new location's output was also performed for the previous output location. The reuse model aims to exploit this fact by storing the relevant portions of a pyramid's computation to be used in the following pyramid. For example, in Figure 3.2 we see six values in hidden layer 1 that are first computed by the black pyramid, and reused for red pyramid. By storing these values in on-chip memory, the red pyramid can simply reuse them while computing its own values. In Figure 3.2 for simplifications we assume that

input and output for each layer has only one channel and in real network each layer can have different number of channels.

At first glance, it may appear that this operation is roughly equivalent to what is done in the recomputation model: these six points are either recomputed or stored. However, it's important to remember that each of those values required a relatively large amount of operations to compute: each green values was the result of one step of the convolution of the previous layer and a 3×3 kernel. Therefore each point required 9 multiplications and 8 additions to compute. We also to have to recompute the same values again when we go to the next row of the output feature map. So, under the reuse model, we store six values for this layer and can use for next pyramid in the same row or the pyramids in the next row, while the recompute model would need to repeat $6 \cdot (9 + 8) = 102$ arithmetic operations for each new pyramid (in the same row and in the next row). This example suggests that the recompute method may be much less efficient than the reuse model. In Section 3.2.1 we will evaluate this question and show that this is indeed true.

Based on the structure of a convolutional layer, and the sizes and locations of the pyramids at that layer's output, we can compute the amount of storage that must be added at that layer under the reuse model. If the size of the pyramid at the output of a given layer is $N \times R \times C$, and this layer convolves a filter of size $K \times K$ with stride S, then the reuse model will require storage of $N \times R \times (K - S)$ elements on the right side of the tile (to be reused by the pyramids on the right) and $N \times (K - S) \times C$ elements at the bottom (to be reused by pyramids in the next rows). Once these computations are stored these are used by all the pyramids which overlap with this region.

As the number of layers fused using this approach increases, the additional costs (the amount of extra on-chip memory required or the amount of redundant computation performed) increases. Thus, there is a trade-off between the costs incurred and the benefits: the amount of DRAM transfer saved. We can consider the approach where all layers are fused into one single pyramid as an extreme: increasing costs by the largest amount to save the most bandwidth. However, we can also choose other trade-off points, where we decompose the layers using more than one pyramid. For example, Figure 3.3 shows two examples: on the left, all layers are fused into a single pyramid, so there will be the lowest possible memory transfer requirement (where only the input data from layer 1 and the final output values need to be loaded/stored from DRAM), but the input tile and intermediate values will need to be large enough to reach all the way to layer 4. On the right, we see another possibility, where we decompose into two pyramids. This system will have higher off-chip memory transfer requirements, because layer 1's output will

be stored, and then read-back to compute the pyramid for layers 2–4. However the benefit of this multi-pyramid example is that the on-chip memory requirement will be reduced, as the input tile and intermediate results will shrink in size. In this way, we obtain a space of tradeoffs: at one extreme, all layers are placed in a single pyramid. In the other extreme, if each layer gets its own pyramid, the system is simply processing in the typical layer-by-layer technique that doesn't reuse data between layers. When we explore over ways to partition a network, we consider all possible locations to start and stop pyramids, by evaluating the above model for each set of possibilities.

As we are fusing multiple layers using pyramids we store weights of the all layers on chip before starting the computation so that we don't have to load these weights again and again from external memory for each pyramid. This makes fusion based approach applicable for only earlier layers of networks because for later layers of network weight size becomes too large and it can't be fit on chip. As the main contribution of our approach is to minimize intermediate data transfer, it make sense to use this approach only for earlier layer of networks where feature map size is too large and requires huge bandwidth to transfer data to and fro from external memory. For the later layers, when intermediate layer data is small it can be fit on chip and there is no requirement to use fusion based approach. In the rest of this thesis, we assume that weights for all the layer we are merging in the pyramid are available on chip for pyramid computations.

3.2.1 Recomputing vs. Storing

Based on the model above, the first subject to address is the relative merit of the reuse and recompute models: is it better to store intermediate results or to recompute them? For example, in Figure 3.2 the six green values in the first hidden layer can be either reused or recomputed. As mentioned above, each of these values requires 17 arithmetic operations to compute; furthermore, if recomputation is performed, the recomputation must take place when the pyramid moves from left to right, and also again when the pyramid moves down to the next row. In the reuse model, once the value is stored we can use it again in all subsequent times it is needed, including in the following row.

To evaluate the relative efficiency of the two models, we compared them on real-world networks. The results showed that a relatively small amount of storage can allow reuse to replace a relatively large amount of recomputation. For example, when fusing the first two layers of AlexNet, the recompute model would need to perform an extra 678 million multiplications and 668 million extra addi-

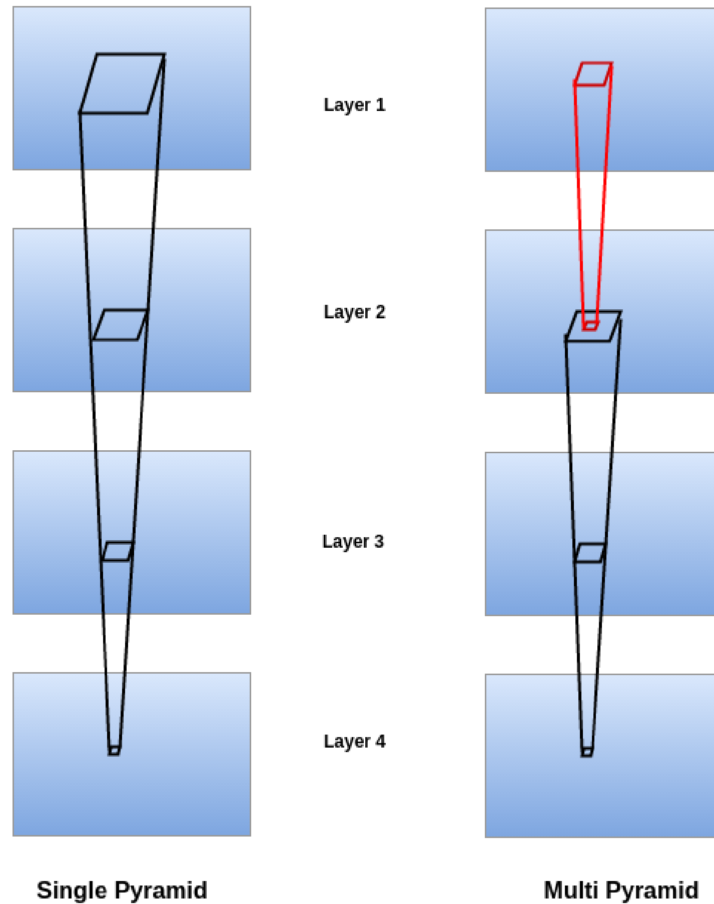

Figure 3.3: Example of single pyramid and multi-pyramid approach

tions, a 80.46% increase in the overall number of arithmetic operations. On the other hand, the results show that the reuse model only requires 55.86 KB of added on-chip storage.

Further experimentation shows that as the network depth increases (that is, we consider fusing more layers), the difference between the two methods grows more extreme. For example, if we fuse all 19 convolution and pooling layers of VGG-E [29], this would require 470 million extra multiplications and 418 million extra additions in convolution layers, a 95% increase in the overall arithmetic operation count in the recompute model; meanwhile, we can use 1.4 MB of intermediate storage, and avoid needing to transfer any intermediate feature map data. For these reasons, in the remainder of this work, we focus solely on the reuse model.

3.3 Methodology

This section describes the architecture of our fused-layer CNN accelerator. First, we give a high-level view of our accelerator. Then, we describe how we represent this architecture in a flexible parameterized way using high-level synthesis and then compare our approach with the state of the art technique.

3.3.1 CNN Accelerator Architecture

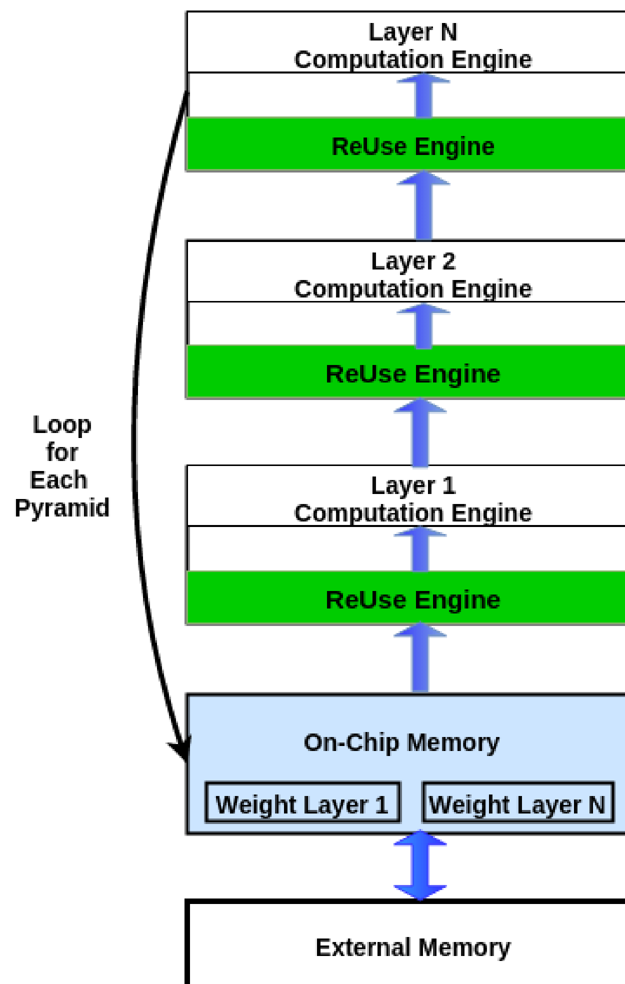

Figure 3.4: Accelerator Design with fused N layers.

Figure 3.4 shows our accelerator design on FPGA. Data are first loaded from external memory and stored in on-chip buffers. These on-chip buffers are then used to feed each layer. Each layer of this network represents a layer inside a pyramid and it is composed of two parts as shown by dotted black box in the figure. In the first part, it reuses the computations done by the previous pyramids and store these computations in the input buffer along with new computations of the this pyramid. In the second part layer do the computation on this new input buffer using that layers computation engine. Each layers computation engine is optimized to do computations in parallel. For example, Figure 3.5 shows optimized engine for convolution layer which is doing computations using it's parallel computation units. The computation engine of convolution layer do parallel multiply and accumulate units which are fed by on-chip memories of weights and input buffer of the pyramid. It also generates output in parallel which is stored in the on-chip buffer which acts a input for next layer of pyramid. As shown in the Figure 3.4 our accelerator runs a loop across all the layers to do computation for all the pyramids in the network.

3.3.2 HLS-Based Accelerator Specification

We have implemented our accelerator design in Vivado HLS (v2015.3). Vivado HLS allows implementing accelerator with C/C++ language and provides HLS defined pragmas which can be used to refine accelerator designs. It also allows to export the RTL as a Vivado's IP core.

We started our implementation using the layer-by-layer approach as mentioned in [33] and got the same results. They have implemented their approach only for convolution layers and they don't do non-linear and pooling operations. We implemented optimized pooling and non-linear modules so that layer fusion can be applied. Listing 3.1 describes a psuedo code for fused accelerator. We first perform back-mapping to get parameters of all the layers of the pyramid. After getting the parameters it applies each layer's operations in two components. In the first component it calls **Reuse Engine** (**ReUseComputation function**) for each layer to reuse computations of previous pyramids and in the second component it applies the operations of each layer. These components are shown in Figure 3.4 inside black box.

Our accelerator design is divided into three components. The first component is **Reuse Engine** which reuse the computations from previous pyramids and store these computations from Reuse buffers to the input buffer of current pyramid. It also store the new computations from the current pyramids in the Reuse

buffers which are used by next pyramids. The second component of our accelerator design is **Computation Engine**. This computation engine is convolution engine for convolution layer and pooling engine for pooling layer. We have explained our convolution engine below as this the most computation demanding layer in CNN. The third component of our accelerator design is **Pipelined Pyramids** which pipeline the computations of pyramids so that each pyramid can run as soon as the resources are available. These components are explained in the following sections.

Listing 3.1: Pseudocode for Fused Layer CNN Accelerator

```
ComputePyramid(InputP, outputP, rowP, colP) {
 BackMapping(rowP, colP);
 ReUseComputation(input1, InputP, ReuseMemoryLeft1, ReuseMemoryTop1, otherArguments);
 Layer1Computation(inputP, output1, otherArguments);
 ReUseComputation(input2, output1, ReuseMemoryLeft2, ReuseMemoryTop2, otherArguments);
 Layer2Computation(input, output2, otherArguments);
 ReUseComputation(inputN, outputN-1, ReuseMemoryLeftN, ReuseMemoryTopN, otherArguments);
 LayerNComputation(inputN, output, otherArguments);
 WriteOutput(outputP, otherArguments);
}
```

Reuse Engine

Reuse engine is our main module which is used to read previous pyramids computation from reuse buffer and write new computations. Listing 3.3 shows pseudo code of our Reuse engine. The code first uses row and col values to check position of pyramid in network. If the row and col values are zero then this is the first pyramid of the network and it just uses **NewData**. This is like the black pyramid in Figure 3.2. If the pyramid is in the first row (row = 0) then it reuses computations from **ReuseMemoryLeft** buffer and new data is used from **NewData**. This corresponds to the red pyramid in Figure 3.2, with green pixels stored in **ReuseMemoryLeft** buffer and blue pixels are new data. If the pyramid is in the first column (col = 0) then it reuses computations from **ReuseMemoryTop**. If the pyramid is in the middle then this pyramid reuse the computations from both **ReuseMemoryLeft** and **ReuseMemoryTop**. In all three reuse cases, the layer uses already done computations form reuse buffers and new computations are loaded from **NewData**. Reuses buffers are accessed through load and store modules, which are described in Listing 3.2.

Listing 3.2: Pseudocode for load and store recomputations

```
LoadData(output, ReuseBuffer, R, C, offsetY, offsetX) {
 Load data of tile size RXC from ReuseBuffer at offset (offsetY, offsetX)
}

StoreData(ReuseBuffer, input, R, C, offsetY, offsetX) {
 Store data of tile size RXC in ReuseBuffer from input at offset (offsetY, offsetX)
}
```

Listing 3.3: Pseudocode for reading and writing Reuse computations

```
ReadWriteComputation(NewData, output, ReuseMemoryLeft, ReuseMemoryTop, row, col, K, S, TileX, TileY) {

//Read from Reuse memory
if(row == 0 && col == 0)
  Load(output, NewData, TileX, TileY, 0, 0);

else if(row == 0)
{
 Load(output, ReuseMemoryLeft, TileY, K-S, 0, 0);
 Load(output, NewData, TileY, TileX-(K-S), 0, K-S);
}
else if(col == 0)
{
 Load(output, ReuseMemoryTop, K-S, TileX, 0, 0);
 Load(output, NewData, TileY, TileX-(K-S), K-S, 0);
}
else
{
 Load(output, ReuseMemoryLeft, TileY, K-S, 0, 0);
 Load(output, ReuseMemoryTop, K-S, TileX, 0, 0);
 Load(output, NewData, TileY-(K-S), TileY-(K-S), K-S, K-S);
}

//Write to Reuse Memory
store(ReuseMemoryLeft, output, TileY, K-S, 0, (TileX - (K-S));
store(ReuseMemoryTop, output, TileY, K-S, 0, (TileX - (K-S));
}
```

Convolution Engine

Listing 3.4 shows psuedo code for our convolution Engine. The convolutional layer uses filters of size $K \times K$. The output buffer and input buffer have M and N channels, respectively. Filters and biases are stored on-chip and are reused across pyramids. We have fused non-linear layer ReLU inside our convolutional layer so that we don't have to allocate separate on-chip buffer for these layers.

Listing 3.4: Pseudocode for Fused Layer Convolution Engine

```
for(m = 0; m < M; m += TM)
 for(n = 0; n < N; n += TN)
// input is input to this layer pyramid
// output is results of this layer's pyramid

//Convolution Layer
for(r = 0; r < R; r += 1 )
 for(c = 0; c < C; c += 1 )
  for(kx= 0; kx< K;kx += 1 )
   for(ky= 0; ky< K;ky += 1 )

for(u1 = 0; u1 < TM; u1 += 1) { // unrolled by HLS tool
   if(n==0) output[m+u1][r][c]=bias[m+u1];
   for(u2 = 0; u2 < TN; u2 += 1)  // unrolled by HLS tool
     output[m+u1][r][c] += filter[m+u1][n+u2][kx][ky] * input[n+u2][S*r+kx][S*c+ky];
   //ReLU Layer
   if(kx==K-1 && ky==K-1 && output[m+u1][r][c] < 0)
       output[m+u1][r][c] = 0;
}
```

To use FPGA resources in parallel, we instruct the HLS tool to unroll the loops in our C++ code. This directs the tool to create a kind-of SIMD design and create parallel copies for all the indices of the loop. Each parallel unit in this loop unrolled design performs input/output operations according to the loop

indices. We used nested unroll loops in our code to perform more operations in parallel. Code 3.4 shows our UNROLL directive which create TM * TN parallel copies of hardware to do multiply and accumulate operation. The outer unroll loop add bias to the output and also do ReLU operation. This parallel hardware is SIMD accelerator which takes different values of input and filters and do multiply and accumulation in parallel. It also add bias and do ReLU operation on output. Figure 3.5 shows our convolution engine design for $TM = 2$ and $TN = 3$. As shown in Figure 3.5 there are two parallel computations units (Red and Green), which takes three inputs in single cycle and generate two outputs. One of the limitations of the UNROLL directive is that it can only be applied when loop bound is fixed. As the computations in our layer fusion approach are irregular the tile size of pyramid changes from one pyramid to one another pyramid so unrolling (R, C) in code 3.4 is not possible. We can unroll the filter loops but the maximum number of filter values are limited. For example, in AlexNet biggest filter size is 11×11 unrolling them creates only 121 parallel units while in VGG the biggest filter size is only 3×3 so if we unroll these loops we under utilize the available resources. To properly utilize the hardware resources, we unroll the input and output loop. The input/output loop size (M and N) are fixed for any convolution layer but these values can be very big for some layer and allocation of $M \times N$ resources is not possible some times. To overcome this limitation, we traverse these loops using constant Tile factors TM and TN as shown in code 3.4. Constant tile factor creates $TM \times TN$ parallel copies of the hardware and we iterate over the available hardware again and again to complete computations for current layer.

The parallel compute units in convolution engine are feeded by input and filters values which are stored in on-chip buffers inside FPGA as shown in Figure 3.5 . These on-chip buffers are called block RAM **(BRAM)** in FPGA terminology. It is not possible to feed all the parallel units in single cycle because each BRAM has only two read and write ports. To feed all the computation units parallel, we partition the input BRAM in TN units which provides port to each of this partitioned BRAM and make it possible to access TN values of input buffer in single cycle. We also do TM partitions of our output BRAM and $TM \times TN$ partition of our filter BRAM for the same purpose. In Code 3.4 input, output and filter are our partitioned BRAM's. We have used the PIPELINE directive in our convolution engine code which feeds all the computation units in single cycle.

Our convolution engine looks same as mentioned in [33] but they apply their convolution engine for fixed input, filter and output size which is decided by their roofline model. They also load input and filter BRAM from external memory

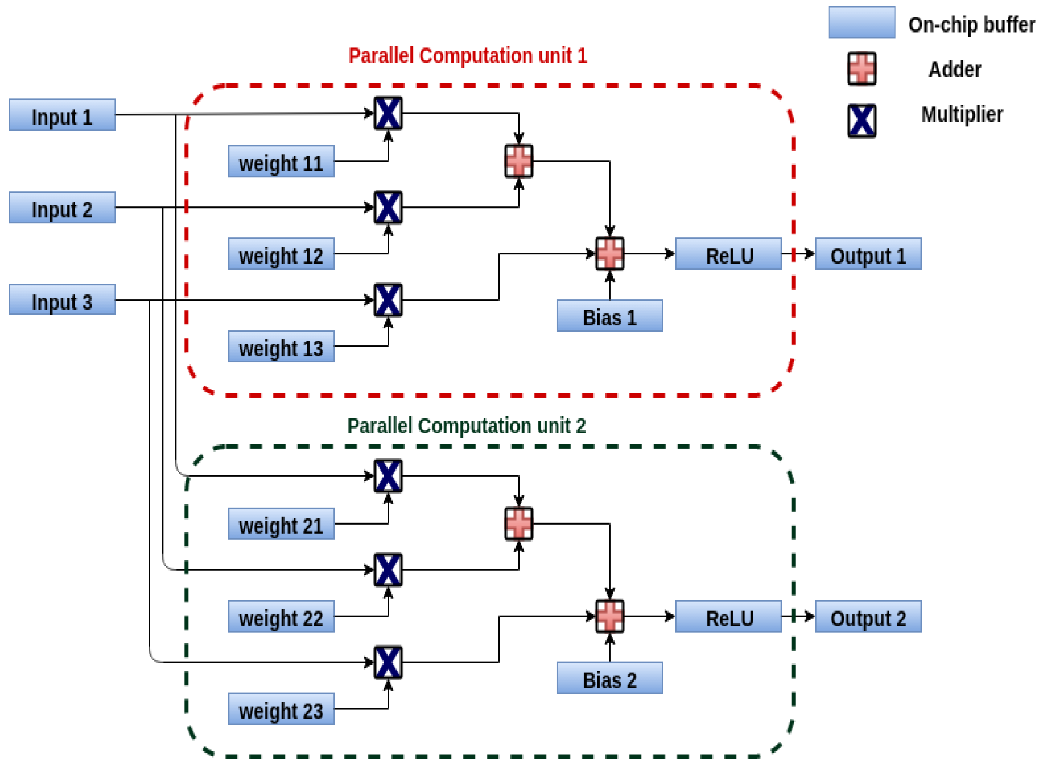

Figure 3.5: convolution Engine Design with unroll factor TM = 2 and TN = 3

before starting their convolution engine operations. In our case, our input size changes according to position of pyramid in network and we access external memory only once to store all the filter values in on-chip buffer which are used again and again for all the pyramids calculations. This reduce the external memory accesses by huge amount.

They also use fixed unroll factors $TM = 64$ and $TN = 7$ for their convolution engine and use the same convolution engine for all the layers which makes their convolution engine underutilized for layers which have outputs channels less then 64 and inputs channels less then 7. For example, in AlexNet network their convolution engine was underutilized for first convolution layer which has 48 output channels and 3 input channels. In our design, we give different resources (different unroll factors) to different convolution layers of the network according to their input and output channel size which minimize this under-utilization. The unroll factors for each of our convolution layer is decided by 3.2. The L in 3.2 refers to total number of layers in the network.

$$\sum_{i=1}^{L} TM_i \times TN_i <= available\ DSP\ Resources \tag{3.2}$$

We have also fused bias addition and ReLU layer computation in our convolution engine which is not proposed in their design. ReLU fusion saves a lot of on-chip storage because we don't have to allocate on-chip buffer for ReLU layer operations.

Pipelined Pyramids

Figure 3.4 shows our accelerator design in which **for loop** is applied to run different pyramids in the network one-by-one. For each pyramid accelerator goes through each layer one-by-one and utilize that layer resources. When the pyramid completes the computations of any layer L it get the output of that layer and moves to layer $L+1$ and frees the resources of layer L which are not utilized until this pyramids complete it's all the remaining layer and next pyramid comes up to layer L. For example, if we have N layers and any pyramid using the resources of layer 1 and moves to layer 2, then resources of layer 1 becomes idle until the pyramids completes it's next $N-1$ layer computations and next pyramid starts it's computation. This shows that our resources are idle for most of the time.

To overcome this limitation, we use dataflow optimization provided by HLS. Dataflow optimization pipeline the pyramid computations and run the next pyramid as soon as the resources are available to run it. For example, if any pyramid use the resources of layer 1 and moves to layer 2, then resources of layer 1 becomes available and next pyramid start using it. The dataflow optimization creates a ping-pong buffers between any two layers of our accelerator so that any two layers of accelerator don't read and write from the same buffer.

Figure 3.6 shows the pipelined pyramids for N layer Network. As shown in the Figure pyramid 2 starts using **Layer 1** as soon as pyramid 1 leaves resources allocated for **Layer 1** and same applies to other layers. This pipeline make sure that our resources become busy all the time.

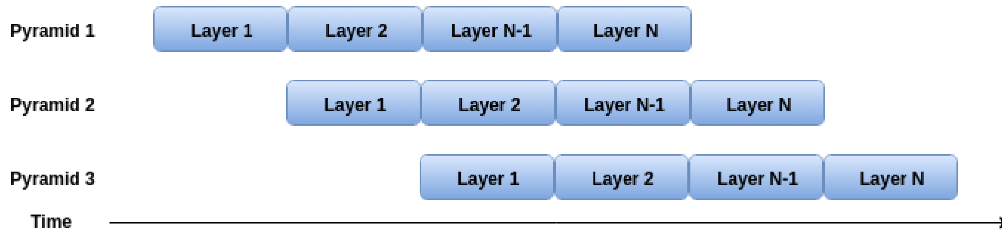

Figure 3.6: Pipelining applied to fused CNN accelerator

3.4 Results

3.4.1 FPGA Evaluation

In order to demonstrate our fused-layer CNN accelerator and compare it to an existing carefully-optimized FPGA implementation, we used the HLS methodology from Section 3.3.2 to implement a fused accelerator for initial layers of AlexNet [20], which is the network studied in [33].

3.4.2 FPGA Implementation

Based on the exploration tool described in Section 4.1, we choose to fuse the first two convolutional layers of the network; these are the layers with the largest feature map size and consume most of the feature map bandwidth. We configure our on-chip resources to roughly match the resources used in [33]. (The acutal costs are later evaluated in Section 3.4.4.) Additionally, we include nonlinearities after each convolutional layer that [33] omit but are critical to real-world CNN algorithms: a rectified linear operation (which performs function $f(x) = \max(x, 0)$ on each output value), and a 3×3 pooling layer (with stride 2) that keeps the largest value in each 3×3 region of the output.

We implement the fused-layer CNN acclerator using a NetFPGA-SUME field-programmable gate array (FPGA) development board. The NetFPGA-SUME contains a Xilinx Virtex-7 XC7V690T FFG1761-3 FPGA, and two 4GB DDR3 SODIMMs (1700 mbps) with 64 bit buses.

After creating the accelerator IP using Vivado HLS 2015.2, we integrated the accelerator into a basic system using Vivado IP integrator 2015.3. The system includes a 32-bit MicroBlaze processor with on-chip memory (used for control), a timer (for run-time measurement), a UART, and a DDR3 controller, connected to off-chip DRAM. All the components in the system are interconnected using an

Resource	Used	Available	Utilization %
LUT	327,996	433,200	75.71
FF	43,000	174,200	24.68
BRAM	967.5	1,470	65.82
DSP	2,336	3,600	64.78

Table 3.1: FPGA Resource Utilization

32 bit	DSP	LUT	FF
Fixed Point (adder)	2	0	0
Fixed Point (multiplier)	2	0	0
Floating point (adder)	2	214	227
Floating Point (multiplier)	3	135	128

Table 3.2: FPGA Occupation Comparison

AXI4 interconnect. To measure the accelerator's runtime, the MicroBlaze reads the current timestamp from the timer, starts the accelerator's computation, waits for the accelerator to finish processing, and then records the timestamp. The MicroBlaze, memory controller, UART, and timer run at 200 MHz, while the HLS-generated logic runs at 100 MHz.

3.4.3 Resource Utilization

Table 3.1 shows the resource utilization of our accelerator. The placement and routing is completed with vivado tool set. As shown in the table, our resources are not fully utilized for XC7V690T board and it shows that we can fuse more layers in the networks. We have only utilized 65.82% of DSP units and 65.83% of BRAM which shows that we can fuse some more layers on this board.

Table 3.2 shows occupation of DSP resources for each multiplier and adder in the network.

3.4.4 Comparison to Related Work

To compare our convolution engine performance with [33] we run our simulation on VCX707 board after limiting our resources to match their design. We have limited our unroll factors of first layer to $TM1 = 48$ and $TN1 = 3$ and second layer to $TM2 = 64$ and $TN2 = 5$ so that we can meet the DSP resources used by [33].

Table 3.3: Resource usage of convolution layer 1 and 2

Resource	DSP	LUT	FF
Convolution 1	726	58311	70332
Convolution 2	1610	136514	157847
Total	2336	194825	228179
Method [33]	2240	186251	205704

We derived unroll factors for each layer using equation 3.2. For any convolution layer with unroll factor TM and TN, total number of multiplier and adder used by layer are $TM \times TN$. As we know from Table 3.2 that each floating point multiplieruses 3 DSP units and each floating point adder uses 2 DSP units so total DSP units used for multiply and accumulate are $5 \times TM \times TN$. As we are adding bias in our convolution engine which requires TN adder and 2 $timesTN$ DSP units. Equation 3.3 shows total DSP units for convolution layers. The number of flip flop and LUT used by any convolution layer can also be calculated using the similar way.

$$Total\ DSP\ Units = 5 \times TM \times TN + 2 \times TN \tag{3.3}$$

Table 3.3 shows our resource utilization for convolution layers. As shown in the table, we have used total 2332 DSP units for our two fused convolution layers while [33] have used 2240 units. We have used slightly higher number of DSP resources because our fused accelerator is doing two convolutions layers in one pyramid and our convolution engine is fused with bias addition and ReLU operations. Method [33] is doing only one convolution layer at one point of time because of layer-by-layer approach and they use same resources when they move from one convolution layer to next. The number in this table are generated by vivado HLS after synthesis step and it matches with our estimation in equation 3.3.

Table 3.4 shows resource utilization for our complete fused accelerator (with two convolution and two pooling layers) on VCX707 board and compare it method [33]. As shown our fused accelerator completely utilize all the resources of this board and fuse 2 convolution and 2 pooling layers of AlexNet. Our numbers for FF and LUT is slightly higher because we are also doing pooling in our accelerator while method [33] only do convolution with these resources. Our BRAM utilization is 14% higher then [33] because we are using BRAM's for pooling layer and to store computations.

Table 3.4: Resource usage of fused accelerator on VCX707

Resource	Available	Method [33] utilization	Proposed Method utilization
DSP	2800	2240	2336
BRAM	2060	1024	1320
FF	303600	205704	277450
LUT	607200	186251	331536

3.4.5 Model for computation and communication

As we are using almost same resources as mentioned by [33]. To show that our method method saves more bandwidth as compared to their approach we have calculated Computation to Communication (CTC) Ratio using equation 3.4. CTC ratio gives estimate of how much computation is done for each external memory byte access. As we are doing our computations in pyramids we calculate CTC ratio for pyramid using equation 3.5.

$$CTC = \frac{total\ number\ of\ operation\ for\ all\ layers}{total\ bytes\ of\ external\ data\ access} \tag{3.4}$$

$$CTC\ pyramid = \frac{total\ number\ of\ operations\ inside\ pyramid}{total\ bytes\ of\ external\ data\ access\ by\ pyramid} \tag{3.5}$$

We found that [33] can achieve maximum of 83.08 Flop/byte CTC ratio for first convolution layer and maximum 162.61 Flop/byte CTC ratio for second convolution layer. As our tile size changes from one pyramid to another pyramid we get maximum of 261.19 Flop/byte CTC. It shows that our technique achieves 1.61x higher CTC ratio and able to do 1.61 times more computation for each external data access.

As our CTC ratio is high from their methods it shows that we do more computation for each external byte access. As the number of computations done by both the models are fixed we do very less external data access which saves bandwidth. Using our technique we save 2.08 MB of external memory access as compared to layer-by-layer approach with the same number of computations done by both the networks. We only take 55.88 KB of extra on-chip storage to store redundant computations.

Chapter 4

Opportunities for Cross-Layer Optimization

In order to evaluate how fusing layers with the reuse model affects the on-chip storage and off-chip bandwidth requirements of real-world CNNs, this section describes a tool to evaluate the technique on CNN structures, and presents a tradeoff evaluation study using it. Using this tool, we can evaluate how we can use mutiple pyramids to fuse different layers of the pyramids.

4.1 Exploration Tool

Based on the back-mapping technique described in last chapter, we have constructed a tool for exploring the tradeoffs in fused-layer CNN accelerator design. In order to enable the tool to easily evaluate different CNN networks (e.g., AlexNet or VGG), we built it using the Lua scripting language as part of the Torch machine learning framework [3], a popular system among those working with deep learning algorithms. Our tool is able to read in a Torch description of a convolutional network and analyze the costs (in terms of added on-chip memory capacity or added arithmetic operations) and benefits (off-chip data accesses saved) for all possible pyramids and combinations of pyramids within the system. The system is able to quickly enumerate and evaluate all possible options; even for a large network like VGG, it is able to search through the design space in just a few minutes on a single CPU.

We have implemented our back-mapping technique in this tool to evaluate recompute and reuse models. Given a set of layers to fuse, we start from the

final layer and work backwards to find the dimensions of the pyramid. Based on the pyramid dimensions, we evaluate the costs in terms of needed storage and arithmetic operations, as well as the benefit in the amount of off-chip data transfer avoided.

The following equations allow us to calculate the dimensions of the pyramid, based on the structure of each stage. Let the size of the pyramid at the output of a given layer be $N \times R \times C$ (by construction, if this is the final layer of a pyramid, then $N \times R \times C = N \times 1 \times 1$, a single value for all N channels.) Here N is number of channels in that layer, R is number of rows for feature map and C is number of columns for feature map. If this layer is a convolutional layer with M channels, we compute the pyramid size at this layer's *input* as $M \times R' \times C'$, according to:

$$\begin{aligned} R' &= S \cdot R + K - S \\ C' &= S \cdot C + K - S \end{aligned} \tag{4.1}$$

where the convolutional kernel is of size $K \times K$, and it is applied with stride S.

If instead the layer performs pooling and not convolution, we use (4.1), but where $K \times K$ is the size of the pooling window and S is its stride.

Following this procedure, we can start from the output of a pyramid, and calculate the dimensions of that pyramid at each level (i.e., the values at each level upon which this output depends). Based on this knowledge, we can model how the **recompute** and **reuse** models behave and compare them. By calculating all the locations where these pyramids overlap, we can calculate the number of computations that must be repeated under the recompute model. The reuse model aims to exploit this fact by storing the relevant portions of a pyramid's computation at each layer to be used in the following pyramid at that layer. The reuse model will require storage of $N \times R \times (K - S)$ elements on the right side of the tile (to be reused by the pyramids on the right) and $N \times (K - S) \times C$ elements at the bottom (to be reused by pyramids in the next rows). Once these computations are stored these are used by all the pyramids which overlap with this region. Using this tool we analyzed that recompute model is not a ideal model for deeper networks as mentioned in last chapter. We have also used this tool to explore all the possible ways to apply layer fusion using reuse model as mentioned in section 4.2.

In Section 4.2, we will provide examples of using this tool to analyze the opportunities for fused-layer acceleration in two real-world CNNs.

4.2 Tradeoff Evaluation

This section presents an exploration of the effects of applying layer fusion to real-world CNN algorithms. Using the evaluation tool described in Section 4.1, we determine the costs (in terms of additional on-chip buffers needed) and benefits (in terms of off-chip communication saved) for all possible groupings of fused layers.

For any given network, there are a number of ways which one may choose to fuse layers into distinct groups. If we have a network with ℓ layers, we have $2^{\ell-1}$ possible ways to fuse these layers (including the extreme cases where all layers are fused into one layer, and where no layers are fused). For example, if a network has three layers, we can choose to group the layers in groups of $(1, 1, 1), (2, 1), (1, 2)$, or (3). Although we typically expect that the best solutions involve fusing pooling layers with the convolutional layers preceding them, for the purposes of this analysis, we treat them as independent layers, which may or may not be merged; this allows the optimization steps to consider their effects for us.

For each network we enumerate all possibilities, and for each we compute how much data must be transferred to and from DRAM. Figure 4.1 and 4.2 show these results for VGG and AlexNet, respectively. For VGG, we consider fusing the first 11 layers (8 convolutional and 3 pooling), giving 1024 possible combinations. Each point on the graphs represents one possible configuration; the x-axis value indicates the on-chip storage required for that configuration, and the y-axis indicates its data transfer requirement. The best points are those closest to the origin, representing an ideal of very low bandwidth and storage costs. One can observe that most of the configurations are sub-optimal—they are dominated by designs that are better on both axis while being equivalent or better on the other. On each graph, a solid black line connects the Pareto optimal designs (those which are not dominated by any other).

These Pareto optimal points represent the set of tradeoffs that a designer may want to consider, although the extremes may still offer unattractive results. For example, the point labeled A in Figure 4.1 has the lowest on-chip storage cost; it represents a layer-by-layer design (with some pooling layers fused, which do not increase the on-chip cost). The point labeled C represents another extreme, where all 11 layers considered have been fused into one. This point has minimum bandwidth cost: the only feature maps it must transfer are the input and the final output. This requires transferring only 1.37 MB of data, but it requires 701 KB of on-chip memory to buffer intermediate results. This is 6.2x more on-chip memory

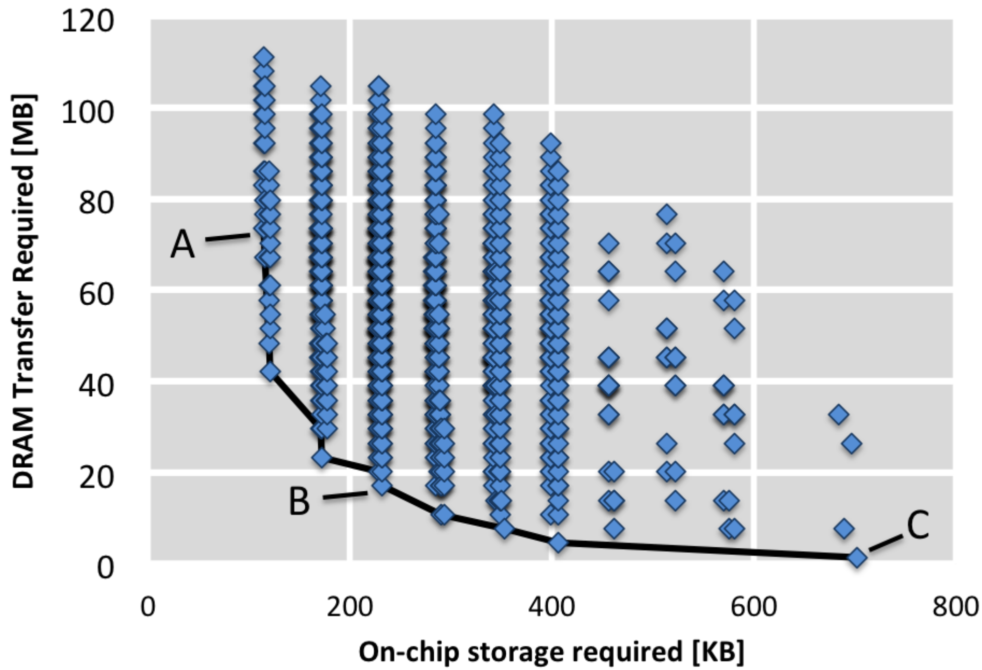

Figure 4.1: Tradeoffs of layer fusion of VGG network

than the layer-by-layer approach requires, but it yields a 53.6x reduction in DRAM traffic required for the feature maps.

In reality, the middle points may offer the most attractive choices for designers to choose from. For example, the point labeled B represents a system that creates four pyramids, each of which fuses two convolutional layers (with pooling layers where appropriate). This configuration requires 17.1MB of data to be transferred to/from DRAM, but it requires only 232 KB of on-chip storage. Relative to the layer-by-layer approach (A), design B requires only 2.04x more on-chip memory than the minimum (A), but it reduces the required off-chip memory transfer by 4.31x.

The AlexNet CNN has five convolutional layers and three pooling layers; thus there are 128 possible combinations of different ways to fuse layers. Similar to VGG, point labeled A in Figure 4.2 has a lowest on-chip storage cost; it represents layer-by-layer design. Point labeled B has highest on-chip storage cost and it fuse all the layers in one pyramid. The point labeled B represents a system that creates two pyramids, the first pyramid fuses first two convolution and pooling layers and

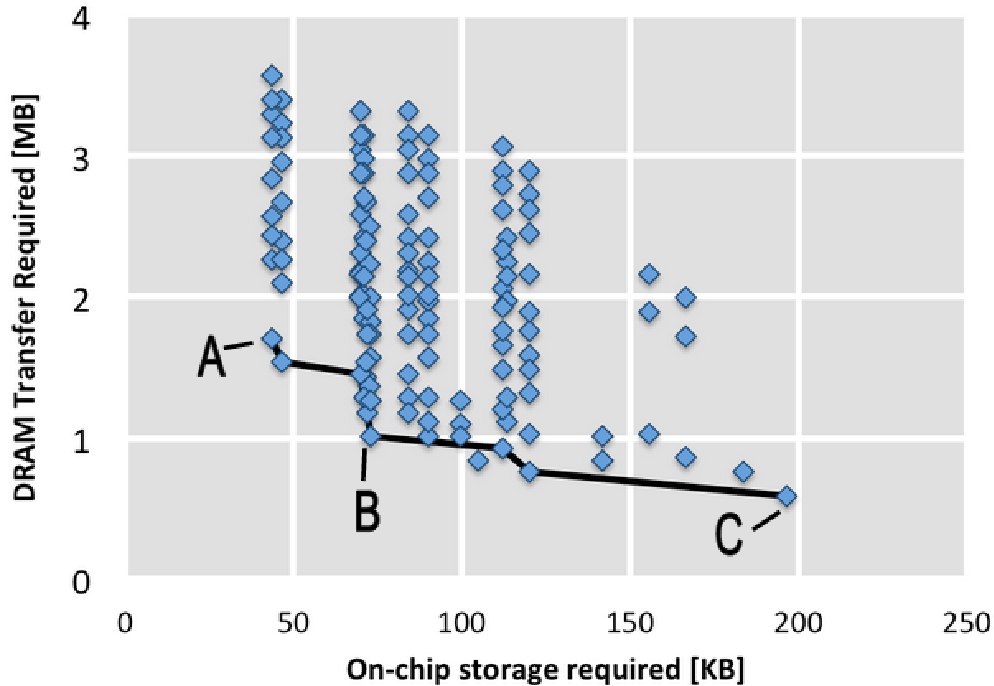

Figure 4.2: Tradeoffs for layer fusion of AlexNet network

next pyramid fuses remaining layers. We have used point B for our accelerator design in the previous chapter.

These results show that layer fusion is able to save a large amount of memory bandwidth at a modest increase in on-chip memory cost.

4.3 Evaluation of pyramid pipelining

As mentioned in the section 3.3.2, we pipeline the pyramids in the network so that we can utilize the available resources all the time. If any stage of the pipeline takes more execution cycles than the other stages, then the slow stage becomes the bottleneck for the whole pipeline. To overcome this problem, we allocate the FPGA resources to different convolution layers in a way such that all the layers in the pipeline take almost the same number of execution cycles.

As shown in pseudo code 2.1, each convolution layer computations depends on the number of output feature map channels (M), number of input feature map

channels (N), output feature map size (R, C), and kernel size (K). As we are using pyramids for computation, our feature map size depends on the base of the pyramid at each layer. We unroll the convolution layer loops for the number of output/input feature map channels (M, N) and output map size (R, C) with different unroll factors (*TM , TN, TR, TC*) for all the convolution layers, finding the combination of unroll factors at each layer that takes the same execution cycles for all the layers. This approach produces a balanced pipeline for the network. We do not unroll the kernel loop because networks such as GoogLeNet [30] use kernel size 1×1 (to increase the depth of network), which doesn't provide any degree of freedom for unrolling the loops.

In our approach, we allocate separate resources to each convolution layers because each convolution layer has different parameters (*M, N, R, C*) and allocating separate resources makes them more optimized. It is also possible to design only one convolution module with maximum unroll factors to utilize all the available DSP resources and use this module for all the convolution layers as mentioned in [33]. But using fixed unroll factors for all layers underutilize the resources for layers where unroll factors are higher than loop limits. For example, [33] use fix unroll factors $TM = 64$ and $TN = 7$ to design their convolution module and use this module iteratively for all the layers of AlexNet network. This module is underutilize for first layer of network where number of output channels (M) are 48 and number of input channels (N) are 3.

Our exploration tool exhaustively searches all possible unroll factors for all the convolution layers in the pipeline by which we can get same executation cycles for all the layers. For each valid combination of layer fusion described in section 4.2, the tool explores the optimized solution to determine a balanced pipeline. Figure 4.3 shows the analysis for the VGG network (fusing the first eight layers). It uses three sets of pyramids to fuse eight convolution layers. The *X* axis in the plot shows the total DSP resources used by all fused layers of pyramids and the *Y* axis shows the execution cycles for all the layers. Each point in the plot shows one combination in which all layers execute in approximately the same number of execution cycles. The optimal solution for an FPGA is the one that takes the minimum execution cycles and utilizes all of the available DSP resources. This figure shows the scenario where we have unlimited DSP units and try to find out the unroll factor combinations by which we can get same execution cycles. The right most point in the figure shows the point where all the layers are fully unrolled and execute in single cycle.

To find the optimal solutions for any FPGA, we take the point on the *X* axis that matches the available DSP resources; all solutions left of that point are valid

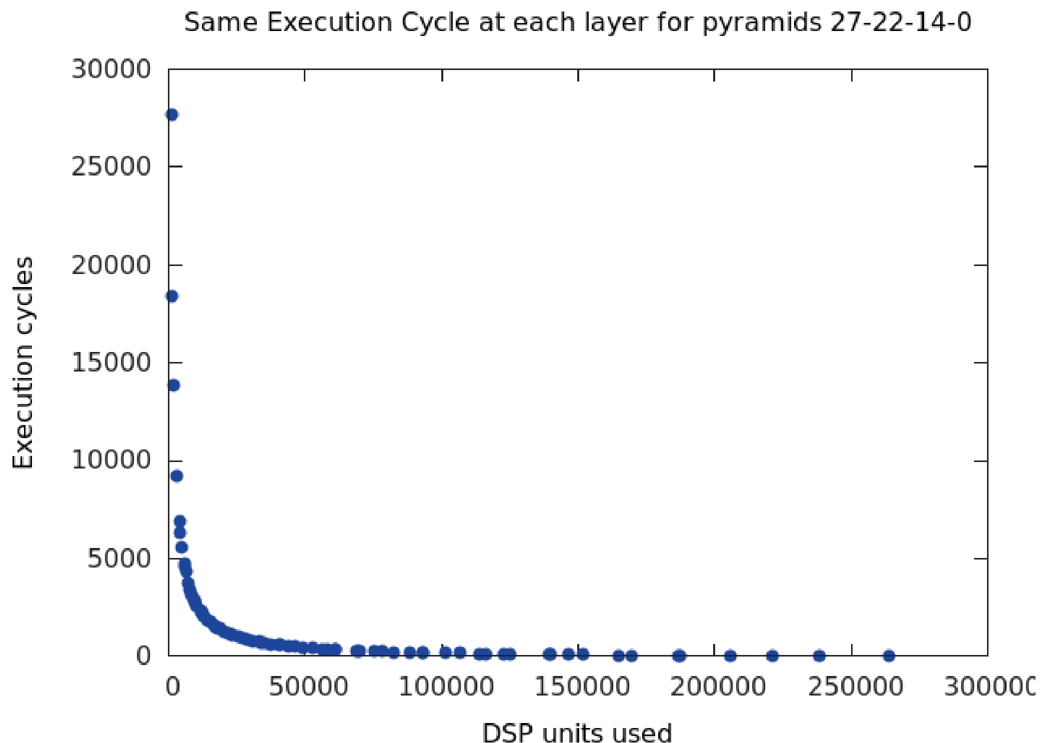

Figure 4.3: Different combinations for VGG

combinations for that FPGA. We can also put a constraint in our tool to find a solution for any particular FPGA. For example, Figure 4.4 shows the output of the tool for first two convolution layers of AlexNet on XC7V690T board which has 3600 DSP units available. The optimal solution for this FPGA is the one that takes the minimum execution cycles and utilizes all of the available DSP resources. This solution is the rightmost point in the plot, utilizing around 3600 DSP units and taking approximately 1200 cycles for each of the two layers.

In our approach, we unroll all the convolution layers such that all layers get the same execution cycles. It is possible that these convolution layers are in different set of pyramids. For example, if we have five layers *1-2-3-4-5*, and first set of pyramids are fusing the first three layers (*1-2-3*) and the second set of pyramids is fusing the next two layers (*4-5*). In this case, the second set of pyramids (*4-5*) can start their computation when the first set of pyramids (*1-2-3*) are done with their computation and write their output in the external memory (which is loaded

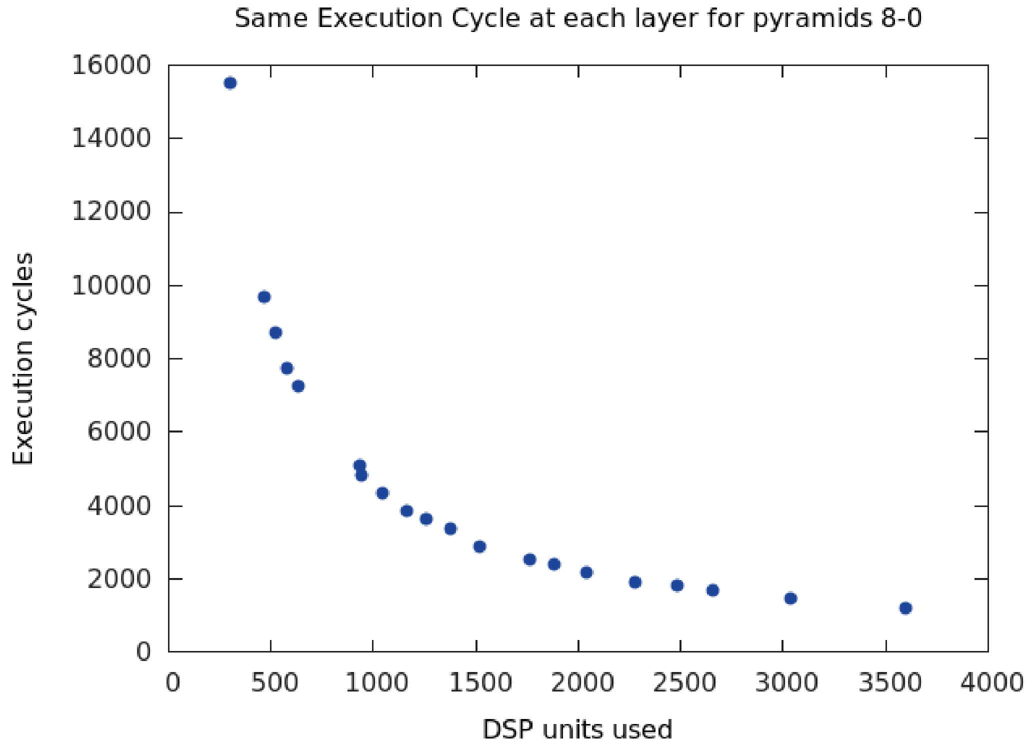

Figure 4.4: Different combinations for AlexNet

by next set of pyramids as input, albeit the order in which data are stored differs from the order in which they are loaded, requiring the round trip through external memory). It is possible that while unrolling the loops, 60 % of DSP resource are allocated to the first set of pyramids and 40 % to next set. In this case, when we are doing computation for the first set (*1-2-3*), we are only utilizing 60 % of the available resources, under-utilizing the available resources of the FPGA. But this is ideal solution for scenario when we are doing computations for more than one image. In this case, the first set of pyramids start computation on the second image as soon as they finish their computation on the first image, and this scenario utilizes 100 % of available resources of FPGA at all times.

These results show that we can explore all the possible combinations for layer fusion in any network. We explore all the combinations in which layer fusion can be applied and consider all the resource partitioning among layers to yield a balanced pipelined design.

Chapter 5

Related Work

CNNs have got a lot of success in recent years and researcher have mapped this model in different programming languages and different platforms. There are lot of software implementation available today for CNNs and these are now used even used by many companies for their artificial intelligence applications. To advance the progress in the direction of deep learning researchers have provided their open source platform for everyone which has sparked the growth in this direction. For example following are famous framework/libraries available today for CNNs implementation.

TensorFlow [2] — Google recently provided their open source machine learning library in C++ and Python with APIs for both. It provides parallelization with CPUs and GPUs.

Caffe [18] — Caffe has been the most popular library for convolutional neural networks. It is created by the Berkeley Vision and Learning Center (BVLC). It is developed in C++, and has Python and MATLAB wrappers. It supports both CPU and GPU and easily switching between them.

Torch [3] — An open source software library written in C and lua. It is currently used and developed by Facebook AI research, Twitter, Google Deep Mind and Element Inc.

Theano [5, 6] — Theano is written in Python and compatible with popular numpy library. Theano allows user to write neural networks using symbolic mathematical expressions which are automatically compiled to CUDA code. Theano is primarily developed by a machine learning group at the Université de Montréal.

MatConvnet [31] — Matconvnet is matlab implementations of CNNs and it

uses GPU routines for fast implementation of CNN layers.

OpenNN [1] — An open source C++ library which provides parallelization with CPUs.

Veles [4] — Neural network training management software from Samsung which is written in python and use CUDA and OpenCL for fast implementation of Neural Networks.

5.1 Convolutional Neural Networks on GPUs

General-purpose CPUs have become a limiter for modern CNNs because of the lack of computational parallelism. Most of the implementations mentioned above use GPU's to train neural networks because GPU's provide massive parallelism and high data transfer rates. We have used **torch** for our research, which is written in lua scripting language. Constructing new network and training them is easier in torch and it also provides tools to visualize the trained networks. We have used torch for our exploration framework as mentioned in chapter 4. Similar to our accelerator, torch provides option to save on-chip storage of non-linear and apply non-linear operation along with convolution layers. Torch also uses GPU routines to train their networks fast. Torch, Caffe, Theano and other libraries which implement CNN's on GPU use the routine provided by **NVIDIA**. NVIDIA provides a new library called cuDNN [10] which provides faster routines dedicated for deep learning and currently used by almost all the libraries. Even after using GPU's massive parallelism training deeper networks takes several days and it consumes lot of power and energy. It is not possible to use GPU's for embedded systems because of their energy requirement. There are some efforts has been made in direction of fusing pooling layer with convolution layer to save external memory access by hidden layers. In [24] author has proposed to fuse the pooling with convolution and trained the convolution neural network. But this approach is only applicable when convolution layer is followed by pooling layer. In our approach, we fuse multiple layers and it can be any layer (pooling, convolution or padding).

5.2 CNNs on FPGAs

Because of GPU's high power requirement a lot of research has been done to develop dedicated accelerator on FPGA for embedded systems. As convolution

layer of CNN's is most computation and bandwidth demanding, many researcher have focused on optimizing convolution layer of CNN's. It has been admitted by recent works [23, 33] that data transfer should be primary concern to achieve efficient processing and they have optimized their convolution layer to minimize external memory accesses. Work [23, 25] target to reduce the communication of convolution engine with external memory. In this approach they use onchip buffers to maximize the data reuse and reduce the bandwidth requirement. But they don't target to maximize their computational performance and also don't focus on reducing the communication of hidden layer data with external memory. In our approach, we not only maximize our computational performance but also reduce the external memory accesses by both convolution layers and hidden layers.

In [33] researchers have proposed roofline model to maximize computation and reduce the bandwidth requirement. They apply parallelism with in feature maps and also across layer size to fully utilize the available resources on FPGA board and use on-chip buffers to reuse the loaded data as much as possible and minimize the communication of convolution layer with external memory. This methods also similar to previous methods focus on minimizing the external memory access of convolution layer and don't focus on minimize communication of hidden layer data with external memory. They have also designed a single convolution engine which utilize all the available resources on FPGA for doing computation and use this engine for all the convolution layers. As this convolution engine uses fixed hardware it is underutilized for layers which require fewer resources for their computation. In our method, we allocate the resources to each convolution layers according to their requirement which makes our approach more computationally optimized and we also apply layer fusion to reduce external memory access of hidden layers which makes our approach more bandwidth optimized.

In [15, 14] researchers have proposed systolic implementation and optimized their filtering module. They have used their design for automotive robotics.This kind of design uses very less resources and very efficient in filtering but this design uses fixed hardware for all the kernels which make them underutilized for small kernel sizes. Our method is different from their approach as our optimization does not depends on kernel size.

Implementation in work [28] have also used systolic implementation to optimized their filtering module and they have also used parallelism within feature maps to maximize resource utilization. In their implementation they don't use onchip buffers to increase data utilization. As a result, their approach requires very high external memory bandwidth to feed their parallel compute units. Similar to

their approach, we also use parallelism within our feature maps (unroll factors) to increase resource utilization but we use on-chip buffers to reuse data and minimize the external memory access which reduce the bandwidth requirement.

5.3 CNNs on ASICs

Researcher's have also focused on dedicated hardware implementation on ASIC platform. Implementations [8, 9, 21] are three ASIC representations from the same group. The earliest approach in [8] researcher have proposed a method for ubiquitous machine learning and optimized different modules of neural networks (pooling, convolution etc.) in way so that they minimize the external data access. They have used loop tiling techniques to increase data locality which reduces external memory accesses. In [21] they have proposed an generalized machine learning accelerator which can be used for several machine learning techniques. In this approach they have optimized different computational primitives used by different algorithms and focused on data reuse and data locality to reduce external memory access. Both of these methods don't focus on reducing the external memory accesses for hidden layers and transfer data to and fro from external memory when they move from one layer to another layer. In our approach, we have also designed dedicated modules (computation engines) for different layers of CNN but we also use layer merging to reduce the external memory accesses.

In [9] they have proposed a method for optimizing layers which have less hidden layers data and high filter weights data like fully connected layers and convolution layers with private kernels. They have shown that for these layers filter weights data size is too large and moving it to and fro in external memory consumes lot of energy so they proposed a solution to store the whole weights data in the on-chip memory and move hidden layer data to and fro from external memory. The drawback of this method is that it requires a lot of on-chip storage to store this huge amount of filter weights. If we have large amount of on-chip storage then we can also store hidden layer data in on-chip memory because this data is only fraction of filter weight size. As we have shown in our analysis in section 3.1 that for deeper layers of networks filter weight data is large as compared to hidden layer data and this problem aggravates for networks which are very deep like VGG-E [29] and GoogLeNet [30]. For these networks filters weight can range from few MB to GB and we can't store this filter weight data on-chip. Our approach is similar to their approach as we also store filter weights in on-chip memory but we use this approach for earlier layers of networks where

filter weights are very small. For later layers as shown in section 3.1 hidden layer data can stay on-chip as it is very small as compared to filter weights. The major difference in our approach and their is that we focus on reducing external memory access for earlier layers of network where hidden layer data is too large and they focus on reducing external memory access for later layers where filter weight size is large but compared to their approach we use only small amount of cache to minimize external memory access.

In [27], authors have proposed a generalized convolution engine which can be used for various computer vision and computational photography algorithms which have convolution like data-flow patterns. Similar to our convolution module which fuse bias addition and ReLU operation they also fuse multiple arithmetic operation with-in their convolution engine to reduce memory storage requirements but they don't fuse multiple layers together like our approach. Moreover, their accelerator can only be used for algorithms which do 1D or 2D convolution and they haven't shown their applicability for algorithms which use 3D convolution like CNN.

Chapter 6

Conclusions

In this thesis, we develop a novel CNN accelerator architecture and design methodology that breaks away from the commonly accepted practice of processing network layer by layer. By modifying the order in which the original input data are brought on chip, changing it to a pyramid-shaped multi-layer sliding window, our architecture enables to fuse multiple layers of CNN and save the external memory accesses of the hidden layers. We proposed the recompute model and the reuse model to fuse multiple layers of computation and showed the benefits of the reuse model over the recompute model. The reuse model uses on-chip buffers to reduce the off-chip memory bandwidth requirements, whereas the recompute model recomputes the needed intermediate values from the input data. We showed that caching a limited number of values in the reuse model reduces the off-chip memory bandwidth requirements, which is a primary bottleneck in many CNN environments. Furthermore, we implemented a tool which evaluate any CNN and propose how layer fusion can be applied to reduce external memory accesses.

Bibliography

[1] Opennn. http://opennn.cimne.com/.

[2] Tensorflow. http://www.tensorflow.org/.

[3] Torch. http://torch.ch/.

[4] Veles. https://velesnet.ml/.

[5] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[6] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[7] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *SIGARCH Comput. Archit. News*, 38(3):247–257, June 2010.

[8] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

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

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

[11] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012.

[12] Christopher Clark and Amos J. Storkey. Teaching deep convolutional neural networks to play go. *CoRR*, abs/1412.3409, 2014.

[13] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *International Conference on Machine Learning, ICML*, 2008.

[14] C Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Proc. Embedded Computer Vision Workshop*, 2011.

[15] C. Farabet, C. Poulet, Y. Han, J., and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *In Field Programmable Logic and Applications, 2009*, pages 32–37. IEEE.

[16] Sachin Sudhakar Farfade, Mohammad J. Saberian, and Li-Jia Li. Multi-view face detection using deep convolutional neural networks. *CoRR*, abs/1502.02766, 2015.

[17] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 27–40, New York, NY, USA, 2015. ACM.

[18] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe:

Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[19] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *CVPR*, 2014.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Neural Information Processing Systems (NIPS)*, 2012.

[21] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. *SIGPLAN Not.*, 50(4):369–381, March 2015.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis.

[23] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, Oct 2013.

[24] Maurice Peemen, Bart Mesman, and Henk Corporaal. Efficiency optimization of trainable feature extractors for a consumer platform. In *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*, ACIVS'11, pages 293–304, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Maurice Peemen, Bart Mesman, and Henk Corporaal. Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 169–174, San Jose, CA, USA, 2015. EDA Consortium.

[26] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers,

Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[27] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: Balancing efficiency & flexibility in specialized computing. *SIGARCH Comput. Archit. News*, 41(3):24–35, June 2013.

[28] Murugan Sankaradas, Venkata Jakkula, Chakradhar Srimat Cadambi, Srihari, I Durdanovic, E Cosatto, and P Grag, H. A massively parallel coprocessor for convolutional neural networks. In *20th IEEE International Conference on Applications-specific Systems, Architecture and Processors, 2009(ASAP)*, pages 53–60. ACM.

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[30] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference on*, pages 1–9, June 2015.

[31] A. Vedaldi and K. Lenc. Matconvnet – convolutional neural networks for matlab.

[32] Izhar Wallach, Michael Dzamba, and Abraham Heifets. Atomnet: A deep convolutional neural network for bioactivity prediction in structure-based drug discovery. *CoRR*, abs/1510.02855, 2015.

[33] 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*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.