Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

Chen Zhang[1]
chen.ceca@pku.edu.cn

Peng Li[2]
pengli@cs.ucla.edu

Guangyu Sun[1,3]
gsun@pku.edu.cn

Yijin Guan[1]
guanyijin@pku.edu.cn

Bingjun Xiao[2]
xiao@cs.ucla.edu

Jason Cong [2,3,1,*]
cong@cs.ucla.edu

[1]Center for Energy-Efficient Computing and Applications, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]PKU/UCLA Joint Research Institute in Science and Engineering

ABSTRACT

Convolutional neural network (CNN) has been widely employed for image recognition because it can achieve high accuracy by emulating behavior of optic nerves in living creatures. Recently, rapid growth of modern applications based on deep learning algorithms has further improved research and implementations. Especially, various accelerators for deep CNN have been proposed based on FPGA platform because it has advantages of high performance, reconfigurability, and fast development round, etc. Although current FPGA accelerators have demonstrated better performance over generic processors, the accelerator design space has not been well exploited. One critical problem is that the computation throughput may not well match the memory bandwidth provided an FPGA platform. Consequently, existing approaches cannot achieve best performance due to under-utilization of either logic resource or memory bandwidth. At the same time, the increasing complexity and scalability of deep learning applications aggravate this problem. In order to overcome this problem, we propose an analytical design scheme using the roofline model. For any solution of a CNN design, we quantitatively analyze its computing throughput and required memory bandwidth using various optimization techniques, such as loop tiling and transformation. Then, with the help of roofline model, we can identify the solution with best performance and lowest FPGA resource requirement. As a case study, we implement a CNN accelerator on a VC707 FPGA board and compare it to previous approaches. Our implementation achieves a peak performance of 61.62 GFLOPS under 100MHz working frequency, which outperform previous approaches significantly.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Microprocessor/microcomputer applications

Keywords

FPGA; Roofline Model; Convolutional Neural Network; Acceleration

1. INTRODUCTION

Convolutional neural network (CNN), a well-known deep learning architecture extended from artificial neural network, has been extensively adopted in various applications, which include video surveillance, mobile robot vision, image search engine in data centers, etc [6] [7] [8] [10] [14]. Inspired by the behavior of optic nerves in living creatures, a CNN design processes data with multiple layers of neuron connections to achieve high accuracy in image recognition. Recently, rapid growth of modern applications based on deep learning algorithms has further improved research on deep convolutional neural network.

Due to the specific computation pattern of CNN, general purpose processors are not efficient for CNN implementation and can hardly meet the performance requirement. Thus, various accelerators based on FPGA, GPU, and even ASIC design have been proposed recently to improve performance of CNN designs [3] [4] [9]. Among these approaches, FPGA based accelerators have attracted more and more attention of researchers because they have advantages of good performance, high energy efficiency, fast development round, and capability of reconfiguration [1] [2] [3] [6] [12] [14].

For any CNN algorithm implementation, there are a lot of potential solutions that result in a huge design space for exploration. In our experiments, we find that there could be as much as 90% performance difference between two different solutions with the same logic resource utilization of FPGA. It is not trivial to find out the optimal solution, especially when limitations on computation resource and memory bandwidth of an FPGA platform are considered. In fact, if an accelerator structure is not carefully designed, its computing throughput cannot match the memory bandwidth provided an FPGA platform. It means that the performance is degraded due to under-utilization of either logic resource or memory bandwidth.

[*] In addition to being a faculty member at UCLA, Jason Cong is also a co-director of the PKU/UCLA Joint Research Institute and a visiting chair professor of Peking University.

FPGA'15, February 22–24, 2015, Monterey, California, USA.

ACM 978-1-4503-3315-3/15/02 ...$15.00.
http://dx.doi.org/10.1145/2684746.2689060.

Unfortunately, both advances of FPGA technology and deep learning algorithm aggravate this problem at the same time. On one hand, the increasing logic resources and memory bandwidth provided by state-of-art FPGA platforms enlarge the design space. In addition, when various FPGA optimization techniques, such as loop tiling and transformation, are applied, the design space is further expanded. On the other hand, the scale and complexity of deep learning algorithms keep increasing to meet the requirement of modern applications. Consequently, it is more difficult to find out the optimal solution in the design space. Thus, an efficient method is urgently required for exploration of FPGA based CNN design space.

To efficiently explore the design space, we propose an analytical design scheme in this work. Our work outperforms previous approaches for two reasons. First, work [1] [2] [3] [6] [14] mainly focused on computation engine optimization. They either ignore external memory operation or connect their accelerator directly to external memory. Our work, however, takes buffer management and bandwidth optimization into consideration to make better utilization of FPGA resource and achieve higher performance. Second, previous study [12] accelerates CNN applications by reducing external data access with delicate data reuse. However, this method do not necessarily lead to best overall performance. Moreover, their method needs to reconfigure FPGA for different layers of computation. This is not feasible in some scenarios. Our accelerator is able to execute acceleration jobs across different layers without reprogramming FPGA.

The main contributions of this work are summarized as follows,

- We quantitatively analyze computing throughput and required memory bandwidth of any potential solution of a CNN design on an FPGA platform.
- Under the constraints of computation resource and memory bandwidth, we identify all possible solutions in the design space using a roofline model. In addition, we discuss how to find the optimal solution for each layer in the design space.
- We propose a CNN accelerator design with uniform loop unroll factors across different convolutional layers.
- As a case study, we implement a CNN accelerator that achieves a performance of 61.62 GFLOPS. To the best of our knowledge, this implementation has highest performance and the highest performance density among existing approaches.

The rest of this paper is organized as follows: Section 2 provides a background for CNN and roofline model. Section 3 presents our analytical approach for optimizing accelerator design. Section 4 describes implementation details. Section 5 shows our experiment result. Section 6 makes comparison between our implementation and existing work and Section 7 concludes the paper.

2. BACKGROUND

2.1 CNN Basics

Convolutional neural network (CNN) is first inspired by research in neuroscience. After over twenty years of evolution, CNN has been gaining more and more distinction in research fields, such as computer vision, AI (e.g. [11] [9]). As a classical supervised learning algorithm, CNN employs a feedforward process for recognition and a backward path for training. In industrial practice, many application designers train CNN off-line and use the off-line trained CNN to perform time-sensitive jobs. So the speed of feedforward computation is what matters. In this work, we focus on speeding up the feedforward computation with FPGA based accelerator design.

A typical CNN is composed of two components: a feature extractor and a classifier. The feature extractor is used to filter input images into "feature maps" that represent various features of the image. These features may include corners, lines, circular arch, etc., which are relatively invariant to position shifting or distortions. The output of the feature extractor is a low-dimensonal vector containing these features. This vector is then fed into the classifier, which is usually based on traditional artificial neural networks. The purpose of this classifier is to decide the likelihood of categories that the input (e.g. image) might belong to.

A typical CNN is composed of multiple computation layers. For example, the feature extractor may consist of several convolutional layers and optional sub-sampling layers. Figure 1 illustrates the computation of a convolutional layer. The convolutional layer receives N feature maps as input. Each input feature map is convolved by a shifting window with a $K \times K$ kernel to generate one pixel in one output feature map. The stride of the shifting window is S, which is normally smaller than K. A total of M output feature maps will form the set of input feature maps for the next convolutional layer. The pseudo code of a convolutional layer can be written as that in Code 1.

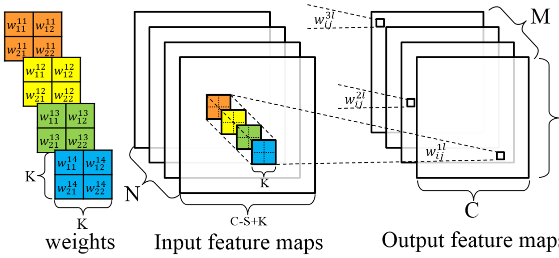


Figure 1: Graph of a convolutional layer

```
for(row=0; row<R; row++) {
 for(col=0; col<C; col++) {
  for(to=0; to<M; to++) {
   for(ti=0; ti<N; ti++) {
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
   L:  output_fm[to][row][col] +=
          weights[to][ti][i][j]*
          input_fm[ti][S*row+i][S*col+j];
} } } } } }
```

Code 1: Pseudo code of a convolutional layer

In the feedforward computation perspective, a previous study [5] proved that convolution operations will occupy over 90% of the computation time. So in this work, we will focus on accelerating convolutional layers. An integration with other optional layers, such as sub-sampling or max pooling layers, will be studied in future work.

2.2 A Real-Life CNN

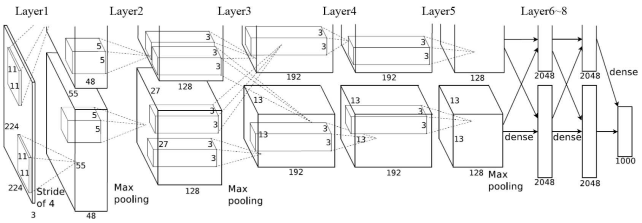


Figure 2: A real-life CNN that won the ImageNet 2012 contest [9]

Figure 2 shows a real-life CNN application, taken from [9]. This CNN is composed of 8 layers. The first 5 layers are convolutional layers and layers 6 ∼ 8 form a fully connected artificial neural network. The algorithm receives three 224x224 input images that are from an original 256x256 three-channel RGB image. The output vector of 1000 elements represents the likelihoods of 1000 categories. As is shown in Figure 2, Layer1 recieves 3 input feature maps in 224x224 resolution and 96 output feature maps in 55x55 resolution. The output of layer1 is partitioned into two sets, each sized 48 feature maps. Layer1's kernel size is 11x11 and the sliding window shifts across feature maps in a stride of 4 pixels. The following layers also have a similar structure. The sliding strides of other layers' convolution window are 1 pixel. Table 1 shows this CNN's configuration.

Table 1: CNN configurations

Layer	1	2	3	4	5
input_fm (N)	3	48	256	192	192
output_fm (M)	48	128	192	192	128
fm row (R)	55	27	13	13	13
fm col. (C)	55	27	13	13	13
kernel (K)	11	5	3	3	3
stride (S)	4	1	1	1	1
set #	2	2	2	2	2

2.3 The Roofline Model

Computation and communication are two principal constraints in system throughput optimization. An implementation can be either computation-bounded or memory-bounded. In [15], a roofline performance model is developed to relate system performance to off-chip memory traffic and the peak performance provided by the hardware platform.

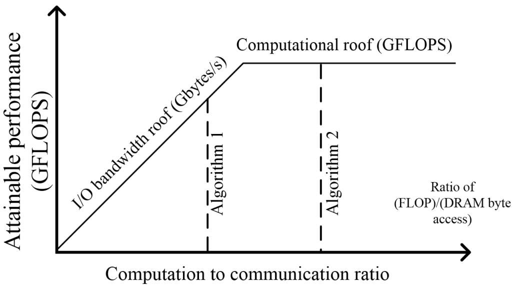


Figure 3: Basis of the roofline model

Eq. (1) formulates the attainable throughput of an application on a specific hardware platform. Floating-point performance (GFLOPS) is used as the metric of throughput. The actual floating-point performance of an application kernel can be no higher than the minimum value of two terms. The first term describes the peak floating-point throughput provided by all available computation resources in the system, or *computational roof*. Operations per DRAM traffic, or *computation to communication (CTC) ratio*, features the DRAM traffic needed by a kernel in a specific system implementation. The second term bounds the maximum floating-point performance that the memory system can support for a given computation to communication ratio.

$$Attainable\ Perf. = min \begin{cases} Computational\ Roof \\ CTC\ Ratio \times BW \end{cases} \tag{1}$$

Figure 3 visualizes the roofline model with computational roof and I/O bandwidth roof. Algorithm 2 in the figure has higher computation to communication ratio, or better data reuse, compared to Algorithm 1. From the figure, we can see that by fully-utilizing all hardware computation resources, Algorithm 2 can outperform Algorithm 1, in which computation resources are under-utilized because of the inefficient off-chip communication.

3. ACCELERATOR DESIGN EXPLORATION

In this section, we first present an overview of our accelerator structure and introduce several design challenges on an FPGA platform. Then, in order to overcome them, we propose corresponding optimization techniques to explore the design space.

3.1 Design Overview

As shown in Figure 4, a CNN accelerator design on FPGA is composed of several major components, which are processing elements (PEs), on-chip buffer, external memory, and on-/off-chip interconnect. A PE is the basic computation unit for convolution. All data for processing are stored in external memory. Due to on-chip resource limitation, data are first cached in on-chip buffers before being fed to PEs. Double buffers are used to cover computation time with data transfer time. The on-chip interconnect is dedicated for data communication between PEs and on-chip buffer banks.

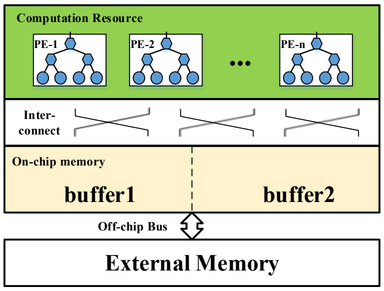


Figure 4: Overview of accelerator design

There are several design challenges that obstacle an efficient CNN accelerator design on an FPGA platform. First, loop tiling is mandatory to fit a small portion of data on-chip. An improper tiling may degrade the efficiency of data

External data transfer
To be discussed in Section 3.2

```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
    //load output feature maps
    //load weights
    //load input feature maps
```

On-chip data computation
To be discussed in Section 3.1

```
    for(trr=row;trr<min(row+Tr,R);trr++){
     for(tcc=col;tcc<min(col+Tc,C);tcc++){
      for(too=to;too<min(to+Tm,M);too++){
       for(tii=ti;tii<min(ti+Tn,N);tii++){
        for(i=0; i<K; i++) {
         for(j=0; j<K; j++) {
        L:  output_fm[too][trr][tcc] +=
              weights[too][tii][i][j]*
              input_fm[tii][S*trr+i][S*tcc+j];
    } } } } } }
    //store output feature maps
} } } }
```

Figure 5: Pseudo code of a tiled convolutional layer

reuse and parallelism of data processing. Second, the organization of PEs and buffer banks and interconnects between them should be carefully considered in order to process on-chip data efficiently. Third, the data processing throughput of PEs should match the off-chip bandwidth provided by the FPGA platform.

In this section, we start our optimization from Code 1 and present our methodology for exploring the design space to achieve an efficient design in successive steps. First, we apply loop tiling (Figure 5). Note that loop iterators i and j are not tiled because of the relatively small size (typically ranging from 3 to 11) of convolution window size K in CNN. Other loop iterators (row, col, to and ti) are tiled into tile loops and point loops (trr, tcc, too and tii in Figure 5). Second, we discuss the computation engine optimization and formulate the computational performance with the tilling factors (Section 3.2). Third, We use data reuse technique to reduce external memory traffic and formulate the computation to communication ratio with tiling factors (Section 3.3). Forth, with the two variables defined above, we define the design space, in which we present the computation-memory-access-matched-design under FPGA board specification (Section 3.4). Fifth, We discuss how to select a best uniform accelerator for the entire multi-layer CNN application (Section 3.5).

3.2 Computation Optimization

In this section, we use standard polyhedral-based data dependence analysis [13] to derive a series of legal design variants of equivalently CNN implementations through loop scheduling and loop tile size enumeration. The objective of computation optimization is to enable efficient loop unrolling/pipelining while fully utilizing of all computation resources provided by the FPGA hardware platform. In this section, we assume that all required data are on-chip. Off-chip memory bandwidth constraints will be discussed in Section 3.3.

Loop Unrolling. Loop unrolling can be used to increase the utilization of massive computation resources in FPGA devices. Unrolling along different loop dimensions will generate different implementation varients. Whether and to what extent two unrolled execution instances share data will affect the complexity of generated hardware, and eventually affect the number of unrolled copies and the hardware operation frequency. The data sharing relations between different loop iterations of a loop dimension on a given array can be classified into three categories:

```
//on-chip data computation
for(trr=row; trr<min(row+Tr,R); trr++){
 for(tcc=col; tcc<min(col+Tc,C); tcc++){
  for(too=to; too<min(to+Tm,M); too++){
   for(tii=ti; tii<min(ti+Tn,N); tii++){
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
    L:  output_fm[too][trr][tcc] +=
          weights[too][tii][i][j]*
          input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

Code 2: On-chip data computation

```
//on-chip data computation
for(i=0; i<K; i++) {
 for(j=0; j<K; j++) {
  for(trr=row; trr<min(row+Tr,R); trr++){
   for(tcc=col; tcc<min(col+Tc,C); tcc++){
    for(too=to; too<min(to+Tm,M); too++){
#pragma HLS UNROLL
     for(tii=ti; tii<min(ti+Tn,N); tii++){
#pragma HLS UNROLL
    L:  output_fm[too][trr][tcc] +=
          weights[too][tii][i][j]*
          input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

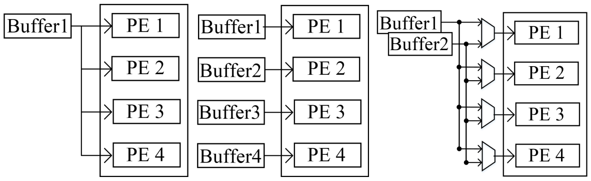
Code 3: Proposed accelerator structure

- *Irrelevant.* If a loop iterator i_k does not appear in any access functions of an array A, the corresponding loop dimension is *irrelevant* to array A.
- *Independent.* If the union of data space accessed on an array A is totally separable along a certain loop dimension i_k, or for any given two distinct parameters p_1 and p_2, the data accessed by $DS(A, i_k = p_1) = \bigcup Image\left(F_S^A, (\mathcal{D}_S \cap i_k = p_1)\right)$ is disjoint with $DS(A, i_k = p_2) = \bigcup Image\left(F_S^A, (\mathcal{D}_S \cap i_k = p_2)\right)$[1], the loop dimension i_k is *independent* of array A.
- *Dependent.* If the union of data space accessed on an array A is not separable along a certain loop dimension i_k, the loop dimension i_k is *dependent* of array A.

The hardware implementations generated by different data sharing relations are shown in Figure 6. An independent data sharing relation generates direct connections between buffers and computation engines. An irrelevant data sharing relation generates broadcast connections. A dependent data sharing relation generates interconnects with complex topology.

The data sharing relations of Figure 5 is shown in Table 2. Loop dimensions *too* and *tii* are selected to be unrolled to avoid complex connection topologies for all arrays. *Too* and *tii* are permuted to the innermost loop levels to simplify HLS code generation. The generated hardware implementation can be found in Figure 7.

[1] The polyhedral annotations used here can be found in [13]



(a) irrelevant (b) independent (c) dependent

Figure 6: Hardware implementations of different data sharing relations

Table 2: Data sharing relations of CNN code

	input_fm	*weights*	*output_fm*
trr	dependent	irrelevant	independent
tcc	dependent	irrelevant	independent
too	irrelevant	independent	independent
tii	independent	independent	irrelevant
i	dependent	independent	irrelevant
j	dependent	independent	irrelevant

Loop Pipelining. Loop pipelining is a key optimization technique in high-level synthesis to improve system throughput by overlapping the execution of operations from different loop iterations. The throughput achieved is limited both by resource constraints and data dependencies in the application. Loop-carried dependence will prevent loops to be fully pipelined. Polyhedral-based optimization framework [16] can be used to perform automatic loop transformation to permute the parallel loop levels to the innermost levels to avoid loop carried dependence. Code structure after optimization for loop unrolling and loop pipelining is shown in Code 3.

Tile Size Selection. Fixing the loop structure, design variants with different loop tile size will also have significantly different performance. The space of all legal tile sizes for Code 3 are illustrated by Equation (2).

$$\begin{cases} 0 < Tm \times Tn \leq (\#\ of\ PEs) \\ 0 < Tm \leq M \\ 0 < Tn \leq N \\ 0 < Tr \leq R \\ 0 < Tc \leq C \end{cases} \tag{2}$$

Given a specific tile size combination $\langle Tm, Tn, Tr, Tc \rangle$, the computational performance (or computational roof in the roofline model) can be calculated by Equation (3). From the equation, we can observe that the computational roof is a function of T_m and T_n.

$$\begin{aligned} & computational\ roof \\ = & \frac{total\ number\ of\ operations}{number\ of\ execution\ cycles} \\ = & \frac{2 \times R \times C \times M \times N \times K \times K}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil \times \frac{R}{T_r} \times \frac{C}{T_c} \times (T_r \times T_c \times K \times K + P)} \\ \approx & \frac{2 \times R \times C \times M \times N \times K \times K}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{N}{T_n} \rceil \times R \times C \times K \times K} \end{aligned} \tag{3}$$

where $P = pipeline\ depth - 1$.

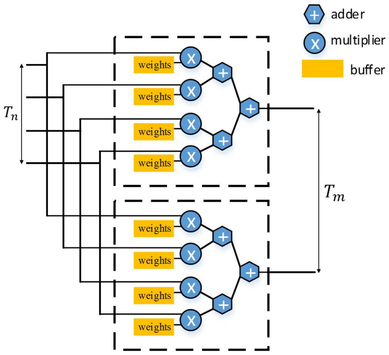
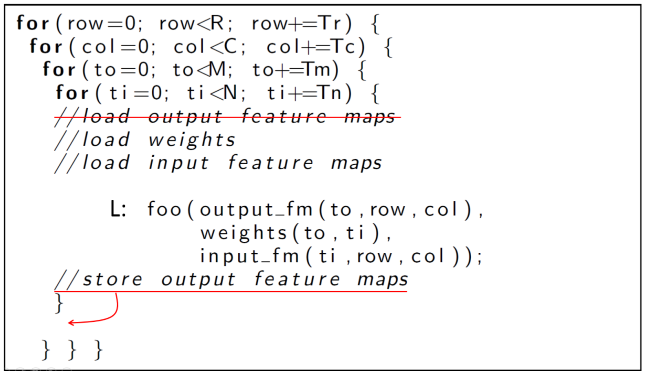


Figure 7: Computation engine

3.3 Memory Access Optimization

In Section 3.2, we discussed how to derive design variants with different computational roofs, assuming all data accessed by the computation engine are already buffered on-chip. However, design variants with the higher computational roof does not necessarily achieve the higher performance under memory bandwidth constraints. In this section, we will show how to reduce communication volume with efficient data reuse.

Figure 9 illustrates the memory transfer operations of a CNN layer. Input/output feature maps and weights are loaded before the computation engine starts and the generated output feature maps are written back to main memory.

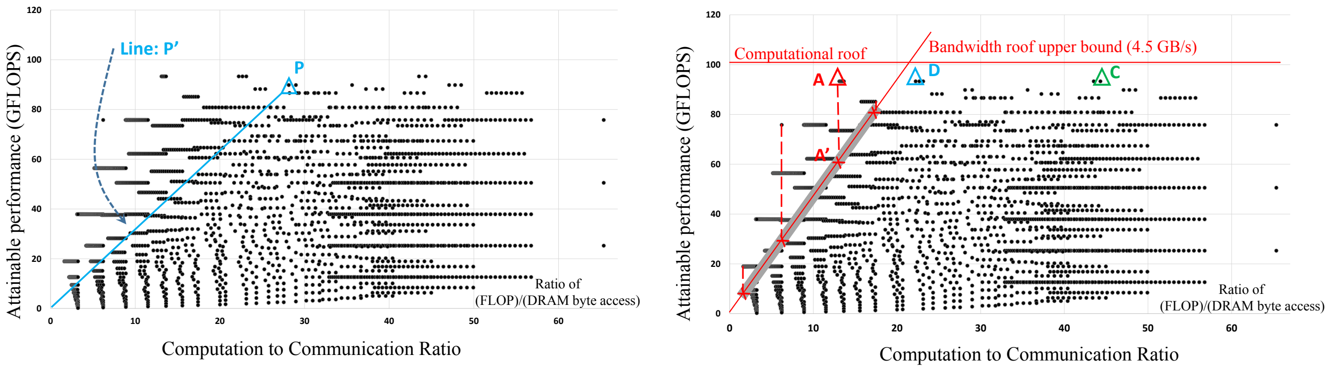


```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
   ~~//load output feature maps~~
   //load weights
   //load input feature maps

       L: foo(output_fm(to,row,col),
              weights(to,ti),
              input_fm(ti,row,col));
   //store output feature maps
   }

  } } }
```

Figure 9: Local memory promotion for CNN

Local Memory Promotion. If the innermost loop in the communication part (loop dimension *ti* in Figure 9) is *irrelevant* to an array, there will be redundant memory operations between different loop iterations. Local memory promotion [13] can be used to reduce the redundant operations. In Figure 9, the innermost loop dimension *ti* is irrelevant to array *output_fm*. Thus, the accesses to array *output_fm* can be promoted to outer loops. Note that the promotion process can be iteratively performed until the innermost loop surrounding the accesses is finally *relevant*. With local memory promotion, the trip count of memory access operations on array *output_fm* reduces from $2 \times \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}$ to $\frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c}$.

Loop Transformations for Data Reuse. To maximize the opportunities of data reuse through local memory promo-



(a) Design space of all possible designs

(b) Design space of platform-supported designs

Figure 8: Design space exploration

tions, we use polyhedral-based optimization framework to identify all legal loop transformations. Table 3 shows the data sharing relations between loop iterations and arrays. Local memory promotions are used in each legal loop schedule whenever applicable to reduce the total communication volume.

Table 3: Data sharing relations of communication part

	irrelevant dimension(s)
$input\_fm$	to
$weights$	row,col
$output\_fm$	ti

CTC Ratio. Computation to communication (CTC) ratio is used to describe the computation operations per memory access. Data reuse optimization will reduce the total number of memory accesses, thus increase the computation to communication ratio. The computation to communication ratio of the code shown in Figure 9 can be calculated by Equation (4), where $\alpha_{in}, \alpha_{out}, \alpha_{wght}$ and $B_{in}, B_{out}, B_{wght}$ denote the trip counts and buffer sizes of memory accesses to input/output feature maps and weights respectively.

$$\begin{aligned} & Computation\ to\ Communication\ Ratio \\ = & \frac{total\ number\ of\ operations}{total\ amount\ of\ external\ data\ access} \\ = & \frac{2 \times R \times C \times M \times N \times K \times K}{\alpha_{in} \times B_{in} + \alpha_{wght} \times B_{wght} + \alpha_{out} \times B_{out}} \end{aligned} \quad (4)$$

where

$$B_{in} = T_n(ST_r + K - S)(ST_c + K - S) \quad (5)$$

$$B_{wght} = T_m T_n K^2 \quad (6)$$

$$B_{out} = T_m T_r T_c \quad (7)$$

$$0 < B_{in} + B_{wght} + B_{out} \leq BRAM_{capacity} \quad (8)$$

$$\alpha_{in} = \alpha_{wght} = \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \quad (9)$$

Without $output\_fm$'s data reuse,

$$\alpha_{out} = 2 \times \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \quad (10)$$

With $output\_fm$'s data reuse,

$$\alpha_{out} = \frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c} \quad (11)$$

Given a specific loop schedule and a set of tile size tuple $\langle Tm, Tn, Tr, Tc \rangle$, computation to communication ratio can be calculated with above formula.

3.4 Design Space Exploration

As mentioned in Section 3.2 and Section 3.3, given a specific loop schedule and tile size tuple $\langle Tm, Tn, Tr, Tc \rangle$, the computational roof and computation to communication ratio of the design variant can be calculated. Enumerating all possible loop orders and tile sizes will generate a series of computational performance and computation to communication ratio pairs. Figure 8(a) depicts all legal solutions for layer 5 of the example CNN application in the rooline model coordinate system. The "x" axis denotes the computation to communication ratio, or the ratio of floating point operation per DRAM byte access. The "y" axis denotes the computational performance (GFLOPS). The slope of the line between any point and the origin point $(0, 0)$ denotes the minimal bandwidth requirement for this implementation. For example, design P's minimal bandwidth requirement is equal to the slope of the line P'.

In Figure 8(b), the line of bandwidth roof and computational roof are defined by the platform specification. Any point at the left side of bandwidth roofline requires a higher bandwidth than what the platform can provide. For example, although implementation A achieves the highest possible computational performance, the memory bandwidth required cannot be satisfied by the target platform. The actual performance achievable on the platform would be the ordinate value of A'. Thus the platform-supported designs are defined as a set including those located at the right side of the bandwidth roofline and those just located on the bandwidth roofline, which are projections of the left side designs.

We explore this platform-supported design space and a set of implementations with the highest performance can be collected. If this set only include one design, then this design will be our final result of design space exploration. However, a more common situation is that we could find several counterparts within this set, e.g. point C, D and some others in Figure 8(b). We pick the one with the highest CI value because this design requires the least bandwidth.

This selection criteria derives from the fact that we can use fewer I/O ports, fewer LUTs and hardwired connections etc. for data transfer engine in designs with lower bandwidth requirement. Thus, point C is our finally chosen design in this case for layer 5. Its bandwidth requirement is 2.2 GB/s.

3.5 Multi-Layer CNN Accelerator Design

Table 4: Layer specific optimal solution and cross-layer optimization

	Optimal Unroll Factor $\langle Tm, Tn\rangle$	Execution Cycles
Layer 1	$\langle 48, 3\rangle$	366025
Layer 2	$\langle 20, 24\rangle$	237185
Layer 3	$\langle 96, 5\rangle$	160264
Layer 4	$\langle 95, 5\rangle$	120198
Layer 5	$\langle 32, 15\rangle$	80132
Total	-	963804
Cross-Layer Optimization	$\langle 64, 7\rangle$	1008246

In previous sections, we discussed how to find optimal implementation parameters for each convolutional layer. In a CNN application, these parameters may vary between different layers. Table 4 shows the optimal unroll factors (Tm and Tn) for all layers of the example CNN application (see Figure 2).

Designing a hardware accelerator to support multiple convolutional layer with different unroll factors would be challenging. Complex hardware structures are required to reconfigure computation engines and interconnects.

An alternative approach is to design a hardware architecture with uniform unroll factors across different convolutional layers. We enumerate all legal solutions to select the optimal global design parameters. CNN accelerator with unified unroll factors is simple to design and implement, but may be sub-optimal for some layers. Table 4 shows that with unified unroll factors ($\langle 64, 7\rangle$), the degradation is within 5% compared to the total execution cycles of each optimized convolutional layer. With this analysis, CNN accelerator with unified unroll factors across convolutional layers are selected in our experiments. The upper bound of the enumeration space size is 98 thousand legal designs, which can finish in 10 minutes at a common laptop.

4. IMPLEMENTATION DETAILS

This section describes the detailed implementation of our solution.

4.1 System Overview

Figure 10 shows an overview of our implementation. The whole system fits in one single FPGA chip and uses a DDR3 DRAM for external storage. MicroBlaze, a RISC soft processor core developed for Xilinx FPGAs, is used to assist with CNN accelerator startup, communication with host CPU and time measurement etc. AXI4lite bus is for command transfer and AXI4 bus is for data transfer. The CNN accelerator works as an IP on the AXI4 bus. It receives commands and configuration parameters from MicroBlaze through AXI4lite bus and communicate with customized data transfer engines through FIFO interfaces. This data transfer engine can access external memory through AXI4 bus. Interruption mechanism is enabled between MicroBlaze and CNN accelerator to provide an accurate time measurement.

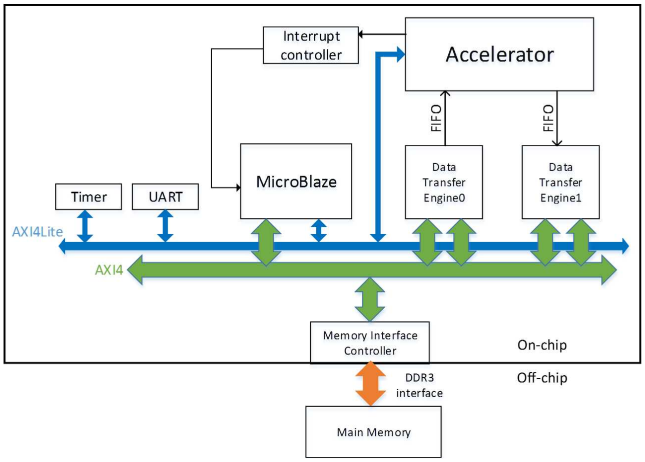


Figure 10: Implementation overview

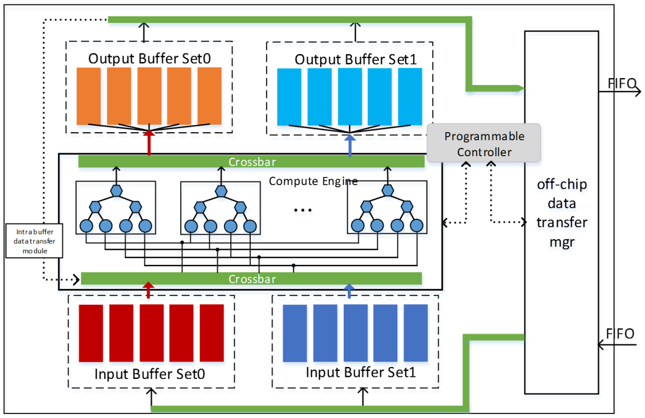


Figure 11: Block diagram of proposed accelerator

4.2 Computation Engines

The computation engine part in Figure 11 shows a block diagram of our implementation. They are designed according to our analysis in Section 3. The two-level unrolled loops (T_m, T_n in Figure 2) are implemented as concurrently executing computation engines. A tree-shaped poly structure like that in Figure 7 is used. For the best cross-layer design ($\langle T_m, T_n\rangle = \langle 64, 7\rangle$) case, the computation engine is implemented as a tree-shaped poly structure with 7 inputs from input feature maps and 7 inputs from weights and one input from bias, which is stored in the buffers of output feature maps. 64 poly structures are duplicated for unrolling loop T_m. An overview can be found in the compute engine part of Figure 11.

4.3 Memory Sub-System

On-chip buffers are built upon a basic idea of double-buffering, in which double buffers are operated in a ping-pong manner to overlap data transfer time with computation. Therefore, they are organized in four sets: two for input feature maps and weights and two for output feature

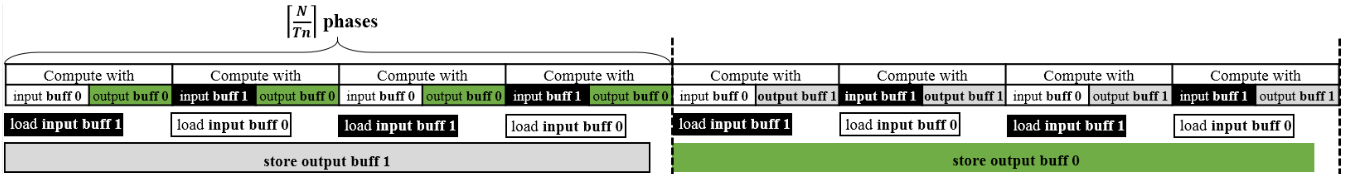


Figure 12: Timing graph

maps. We first introduce each buffer set's organization and followed by the ping-pong data transfer mechanism.

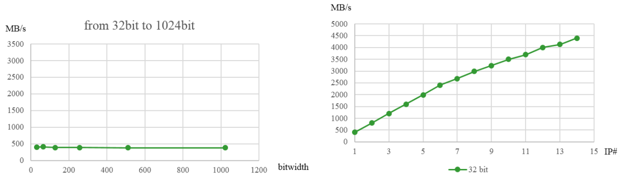
Every buffer set contains several independent buffer banks. The number of buffer banks in each input buffer set is equal to T_n (tile size of *input_fm*). The number of buffer banks in each output buffer set is equal to T_m (tile size of *output_fm*).

Double buffer sets are used to realize ping-pong operations. To simplify discussion, we use a concrete case in Figure 9 to illustrate the mechanism of ping-pong operation. See the code in Figure 9. The "off-load" operation will occur only once after $\lceil \frac{N}{T_n} \rceil$ times of "load" operation. But the amount of data in every *output_fm* transfer are larger than that of *input_fm* in a ratio of $\approx \frac{T_m}{T_n} = \frac{64}{7}$. To increase the bandwidth utilization, we implement two independent channels, one for load operation and the other for off-load operation.

Figure 12 shows the timing of several compute and data transfer phases. For the first phase, computation engine is processing with input buffer set 0 while copying the next phase data to input buffer set 1. The next phase will do the opposite operation. This is the ping-pong operation of input feature maps and weights. When $\lceil \frac{N}{T_n} \rceil$ phases of computation and data copying are done, the resulting output feature maps are written down to DRAM. The "off-load" operation would off-load results in the output buffer set 0 in the period of $\lceil \frac{N}{T_n} \rceil$ phases till the reused temporary data in the output buffer set 1 generates the new results. This is the ping-pong operation of output feature maps. Note that those two independent channel for load and store operation mechanism work for any other data reuse situation in this framework.

4.4 External Data Transfer Engines

The purposes of using external data transfer engines are in two folds: 1) It can provide data transfer between accelerator and external memory; 2) It can isolate our accelerator from various platform and tool specific bandwidth features. Figure 13 shows an experiment with AXI4 bus bandwidth in Vivado 2013.4. In these two figures, we set two parameters, bitwidth of AXI bus to DRAM controller and DRAM controller's external bandwidth, at their highest configurations while changing the number of IP-AXI interfaces and the bitwidth of each IP. In Figure 13(a), the increase in IP-AXI interface bitwidth has no effect on bandwidth (400MB/s under 100MHz frequency). In Figure 13(b), with more IP interfaces added to AXI bus, its bandwidth increases almost linearly and the highest bandwidth is about 4.5 GB/s. In our CNN accelerator design, a minimal bandwidth of 1.55 GB/s is required. Therefore, 4 IP interfaces are sufficient for this design according to Figure 13. We use two AXI-IP interfaces in data transfer engine 0 and two in data transfer engine 1, as is shown in Figure 10.



(a) Single IP's bandwidth-bitwidth relation

(b) bandwith-IP numbers relation

Figure 13: IP-DRAM bandwidth(Vivado 2013.4)

5. EVALUATION

In this section, we first introduce the environment setup of our experiments. Then, comprehensive experimental results are provided.

5.1 Experimental Setup

The accelerator design is implemented with Vivado HLS (v2013.4). This tool enables implementing the accelerator with C language and exporting the RTL as a Vivado's IP core. The C code of our CNN design is parallelized by adding HLS-defined pragma and the parallel version is validated with the timing analysis tool. Fast pre-synthesis simulation is completed with this tool's C simulation and C/RTL co-simulation. Pre-synthesis resource report are used for design space exploration and performance estimation. The exported RTL is synthesized and implemented in Vivado (v2013.4).

Our implementation is built on the VC707 board which has a Xilinx FPGA chip Virtex7 485t. Its working frequency is 100 MHz. Software implementation runs on an Intel Xeon CPU E5-2430 (@2.20GHz) with 15MB cache.

5.2 Experimental Results

In this subsection, we first report resource utilization. Then, we compare software implementation (on CPU) to our accelerator on FPGA. Finally, the comparison between our implementation and existing FPGA approaches is provided.

The placement and routing is completed with Vivado tool set. After that, the resource utilization of our implementation is reported out, as shown in Table 6. We can tell that our CNN accelerator has almost fully utilized FPGA's hardware resource.

Table 6: FPGA Resource utilization

Resource	DSP	BRAM	LUT	FF
Used	2240	1024	186251	205704
Available	2800	2060	303600	607200
Utilization	80%	50%	61.3%	33.87%

Table 5: Comparison to previous implementations

	ICCD2013 [12]	ASAP2009 [14]	FPL2009 [6]	FPL2009 [6]	PACT2010 [2]	ISCA2010 [3]	Our Impl.
Precision	fixed point	16bits fixed	48bits fixed	48bits fixed	fixed point	48bits fixed	32bits float
Frequency	150 MHz	115 MHz	125 MHz	125 MHz	125 MHz	200 MHz	100 MHz
FPGA chip	Virtex6 VLX240T	Virtex5 LX330T	Spartan-3A DSP3400	Virtex4 SX35	Virtex5 SX240T	Virtex5 SX240T	Virtex7 VX485T
FPGA capacity	37,680 slices 768 DSP	51,840 slices 192 DSP	23,872 slices 126 DSP	15,360 slices 192 DSP	37,440 slices 1056 DSP	37,440 slices 1056 DSP	75,900 slices 2800 DSP
LUT type	6-input LUT	6-input LUT	4-input LUT	4-input LUT	6-input LUT	6-input LUT	6-input LUT
CNN Size	2.74 GMAC	0.53 GMAC	0.26 GMAC	0.26 GMAC	0.53 GMAC	0.26 GMAC	1.33 GFLOP
Performance	8.5 GMACS	3.37 GMACS	2.6 GMACS	2.6 GMACS	3.5 GMACS	8 GMACS	61.62 GFLOPS
	17 GOPS	6.74 GOPS	5.25 GOPS	5.25 GOPS	7.0 GOPS	16 GOPS	61.62 GOPS
Performance Density	4.5E-04 GOPs/Slice	1.3E-04 GOPs/Slice	2.2E-04 GOPs/Slice	3.42E-04 GOPs/Slice	1.9E-04 GOPs/Slice	4.3E-04 GOPs/Slice	8.12E-04 GOPS/Slice

Table 7: Performance comparison to CPU

float	CPU 2.20GHz (ms)		FPGA	
32 bit	1thd -O3	16thd -O3	(ms)	GFLOPS
layer 1	98.18	19.36	7.67	27.50
layer 2	94.66	27.00	5.35	83.79
layer 3	77.38	24.30	3.79	78.81
layer 4	65.58	18.64	2.88	77.94
layer 5	40.70	14.18	1.93	77.61
Total	376.50	103.48	21.61	-
Overall GFLOPS	**3.54**	**12.87**	**61.62**	
Speedup	**1.00x**	**3.64x**	**17.42x**	

Table 8: Power consumption and energy

	Intel Xeon 2.20GHz		FPGA
	1 thread -O3	16 threads -O3	
Power (Watt)	95.00	95.00	18.61
Comparison	5.1x	5.1x	1x
Energy (J)	35.77	9.83	0.40
Comparison	89.4x	24.6x	1x

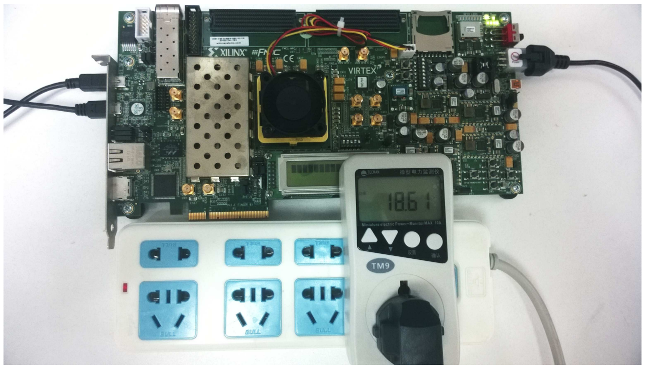


Figure 14: Power measurement of on-board execution

Table 9: Resource occupation comparison

32-bit	DSP	LUT	FF
Fixed point(adder)	2	0	0
Fixed point(mul.)	2	0	0
Floating point(adder)	2	214	227
Floating point(mul.)	3	135	128

The performance comparison between our accelerator and software based counterpart is shown in Table 7. We select our proposed cross-layer accelerator for comparison. The software implementations are realized in 1 thread and 16 threads using gcc with -O3 optimization options, respectively. **Overall**, our FPGA based implementation achieves up to a 17.42x speedup over software implementation of 1 thread. It also achieves a 4.8x speedup over software implementation of 16 threads. Our accelerator's overall performance achieves 61.62 GFLOPS.

Figure 14 shows a picture of our on-board implementation. A power meter is plugged in to measure its runtime power performance, which is 18.6 Watt. CPU's thermal design power is 95 Watt. Therefore, we can have a rough estimation of software and FPGA's implementations' power. Table 8 shows that the ratio of the consumed energy between software implementation and FPGA implementation is 24.6x at least. FPGA implementation uses much less energy than its software counterparts.

In Table 5, various existing FPGA based CNN accelerators are listed and compared to our implementation in this work. Since previous approaches use GMACS (giga multiplication and accumulation per second) and we use GFLOPS (giga floating point operations per second) as the performance metric, we first convert all result numbers to GOPS (giga operations per second) to employ the same metric. Note that each multiply-accumulate operation contains two integer operations. As shown in the 9th row of Table 5, our accelerator has a throughput of 61.62 GOPS, which outperforms other approaches by a speedup of at least 3.62x.

Since different work exploits different parallelism opportunities and use different FPGA platforms, it is hardly to have a straightforward comparison between them. In order to provide a fair comparison, we further present results of "performance density" in Table 5. It is defined as average GOPS per area unit (slice), which can represent the efficiency of a design independent of the FPGA platforms used for implementation. As shown in the last row of Table 5, our implementation achieves the highest performance density, which is 1.8x better than the second best. In addition, our method could achieve even better performance and performance density if using fixed point computation engines because fixed point processing elements uses much less resource (Table 9).

6. RELATED WORK

In this section, we discuss different design methods with reference to other previous work on FPGA-based CNN accelerator designs.

First, many CNN application accelerators are focused on optimizing computation engines. Implementations [6], [14] and [3] are three representatives. The earliest approach in work [6] was to build their CNN application mainly by software implementation while using one hardware systolic architecture accelerator to do the filtering convolution job. This design saves a lot of hardware resources and is used in embedded system for automotive robots. Implementations in work [14], [2] and [3] implement complete CNN applications on FPGA but exploits different parallelism opportunities. Work [14] and [2] mainly uses the parallelism within feature maps and convolution kernel. Work [3] uses "inter-output" and "intra-output" parallelism. Our parallelization method is similar to theirs, but they do not use on-chip buffers for data reuse; alternatively they use very high bandwidth and dynamical configurations to improve performance. Our implementation take advantage of data reuse, and balances limitations of bandwidth and FPGA computation power.

Second, work [12], considering CNN's communication issue, chooses to maximize date reuse and reduce bandwidth requirement to the minimum. But their approach do not consider maximizing computation performance. In addition, they need to take time (in order of 10 seconds) to program FPGA when shifting to next layers' computation while our solution only take no more than a microsecond to configure a few registers.

7. CONCLUSIONS

In this work, we propose a roofline-model-based method for convolutional neural network's FPGA acceleration. In this method we first optimize CNN's computation and memory access. We then model all possible designs in roofline model and find the best design for each layer. We also find the best cross-layer design by enumeration. Finally, we realize an implementation on Xilinx VC707 board which outperforms all previous work.

8. ACKNOWLEDGMENT

This work was supported in part by NSF China 61202072, RFDP 20110001110099, National High Technology Research and Development Program of China 2012AA010902 and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We thank the UCLA/PKU Joint Research Institute for their support of our research.

9. REFERENCES

[1] D. Aysegul, J. Jonghoon, G. Vinayak, K. Bharadwaj, C. Alfredo, M. Berin, and C. Eugenio. Accelerating deep neural networks on mobile processor with embedded programmable logic. In *NIPS 2013*. IEEE, 2013.

[2] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf. A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 273–284. ACM, 2010.

[3] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM, 2010.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGPLAN Not.*, 49(4):269–284, Feb. 2014.

[5] J. Cong and B. Xiao. Minimizing computation in convolutional neural networks. In *Artificial Neural Networks and Machine Learning–ICANN 2014*, pages 281–290. Springer, 2014.

[6] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37. IEEE, 2009.

[7] Google. Improving photo search: A step across the semantic gap. `http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html`.

[8] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(1):221–231, Jan. 2013.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[10] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 473–480, New York, NY, USA, 2007. ACM.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] M. Peemen, 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. IEEE, 2013.

[13] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[14] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. A massively parallel coprocessor for convolutional neural networks. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 53–60. IEEE, 2009.

[15] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[16] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.