Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

---

Chen Zhang et al

(FPGA '15)

Presenter : Tae-wan Kim (Louis)

February 16, 2020

Neural Acceleration Study

Contents

ABSTRACTION

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 underutilization 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.

- FPGA platform 장점
  1) High Performance
  2) Reconfigurability
  3) Fast Development round

- Memory Bandwidth와 Computation Throughput을 맞추는 것은 어렵다.

- Loop Tiling, Transformation 등의 방법을 통해 Optimize 했다.

- 100MHz, 61.62 GFLOPS, VC707 FPGA

1. INTRODUCTION

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.

▶ 기존 연구에서는 Computation Engine에 대해서만 고려했다.

▶ Buffer Management와 Memory Bandwidth도 고려해서 Optimize 했다.

▶ Reconfiguration을 하지 않았다.

▶ 본 연구가 기여하는 바를 요약하자면

1) Computing Throughput과 Bandwidth를 고려해 CNN을 최적화 하는 방법을 연구함

2) 제한된 Computation, Memory 조건에서 어떻게 optimal을 찾아야 하는지를 논함

3) 같은 Loop unrolling 을 CNN 모든 Layer에 똑같이 적용함

2. BACKGROUND

2.2 A Real-Life CNN

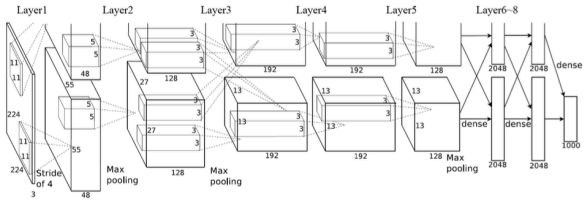


Figure 2: A real-life CNN that won the ImageNet 2012 contest [9]

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

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.

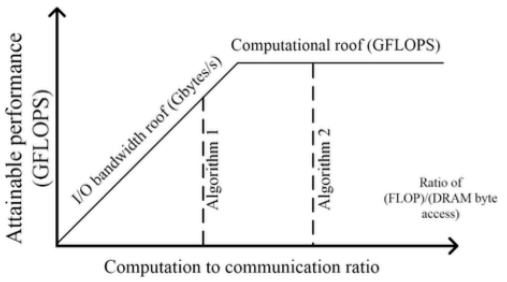


Figure 3: Basis of the roofline model

▶ 본 연구에서 제안하는 CNN Accelerator는 Real-Life CNN 모델을 ImageNet12 Dataset을 이용해 Accelerate한다.

▶ Roofline Model

▶ Algorithm 1 : Memory 에서 가져온 정보를 Computation을 많이 하지 않고 다시 Memory로 돌려보내는 경우

Bottleneck : Memory Operation

ex. Embedding, RNN

▶ Algorithm 2 : Memory 에서 가져온 정보를 Computation을 많이 하고 다시 Memory로 돌려보내는 경우

Bottleneck : Computation

ex. Convolution, CNN

▶ CTC : computation to communication ratio

▶ cf. Arithmetic Intensity

3. ACCELERATOR DESIGN EXPLORATION

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.

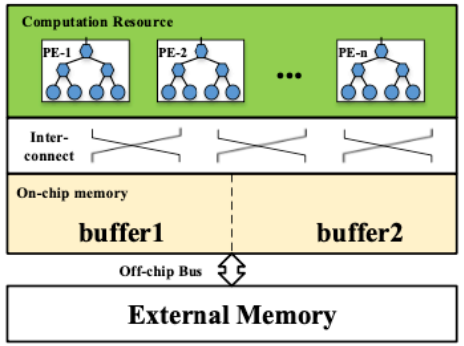


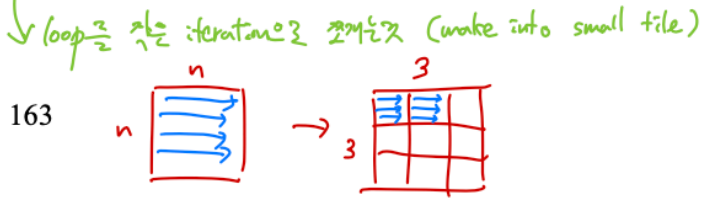
Figure 4: Overview of accelerator design

▶ Design Overview

1) PE : 연산기
2) On-chip Memory : Double buffered
3) External Memory : DRAM

3. ACCELERATOR DESIGN EXPLORATION

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



163

```
for(row=0; row<R; row+=Tr) {                    External data transfer
 for(col=0; col<C; col+=Tc) {                   To be discussed in Section 3.2
  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

▶ Loop Tiling : On-chip에 fit하게 for loop 내의 operation을 일정 단위를 자르는 것

1) On-chip Memory : Double buffered
2) External Memory : DRAM

3. ACCELERATOR DESIGN EXPLORATION

```
//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

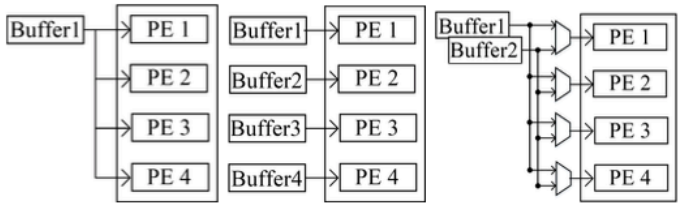
what extent two unrolled execution instances share data will affect the complexity of generated hardware, and eventually affect the number of unrolled copies and the hardware operation frequency. The data sharing relations between different loop iterations of a loop dimension on a given array can be classified into three categories:

- *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.

▶ Loop Unrolling

▶ Irrelevant, Independent, Dependent에 따라 H/W 구현이 다름



(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

3. ACCELERATOR DESIGN EXPLORATION

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 $\frac{M}{T_m} \times \frac{R}{T_r} \times \frac{C}{T_c}$.

▶ Design Overview

1) PE : 연산기
2) On-chip Memory : Double buffered
3) External Memory : DRAM

3. ACCELERATOR DESIGN EXPLORATION

3.4 Design Space Exploration

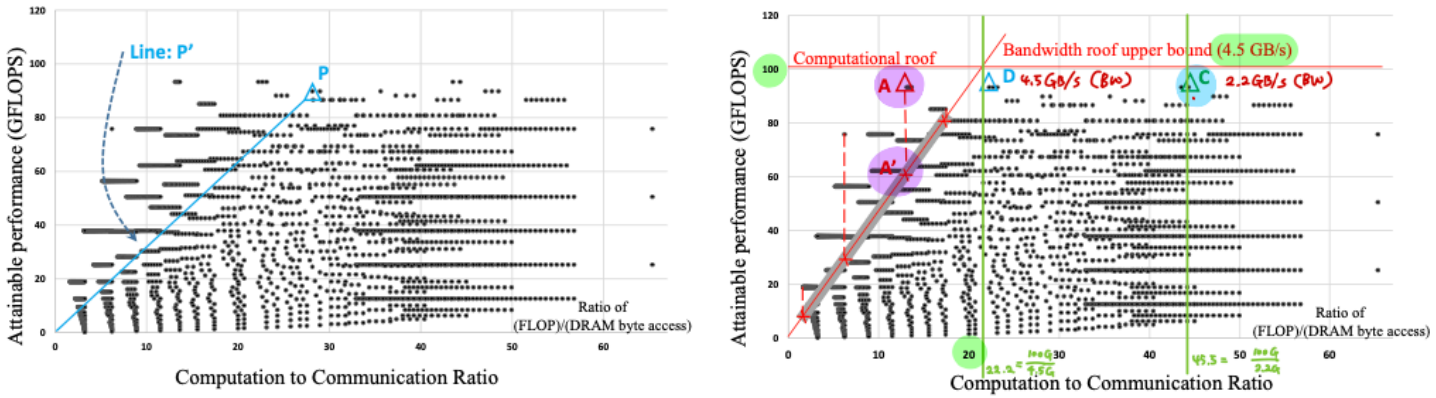
As mentioned in Section 3.2 and Section 3.3, given a specific loop schedule and tile size tuple $\langle Tm, Tn, Tr, Tc \rangle$, the computational roof and computation to communication ratio of the design variant can be calculated. Enumerating all possible loop orders and tile sizes will generate a 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.

▶Figure 8(a) : Real–Life CNN의 Conv5 Layer를 위한 All Legal Solution
▶P design을 위해서는 Bandwidth가 Slope P'보다 커야 한다.

▶Figure 8(b)의 Red Line이 H/W Constraint
▶A는 H/W Constraint에 의해 A'의 성능과 BW를 갖게 된다
▶C, D 구현이 모두 가능하다면 적은 BW를 요구하는 C를 선택한다



(a) Design space of all possible designs

(b) Design space of platform-supported designs

Figure 8: Design space exploration

3. ACCELERATOR DESIGN EXPLORATION

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.

▶ Layer마다 Loop unrolling factor의 optimum이 다르다.

▶ uniform 하게 Loop unrolling을 해도 전체 성능이 5%밖에 떨어지지 않는다.

5. Evaluation

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. 2012.11 출시

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 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**	

4.8x

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

- Proposed FPGA : Intel Xeon E5 CPU에 비해 4.8x
- Intel Xeon Silver 4210 : Intel Xeon E5에 비해 4x
- Nvidia GPU Titan XP (1.6GHz) : Intel Xeon Silver 4210 대비 30x

- Nvidia GPU Titan XP (1.6GHz) : Proposed FPGA 대비 30x

Thank you

Neural Acceleration Study