JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 12, No. 1, March 2018, pp. 24-35

Toward Optimal FPGA Implementation of Deep Convolutional Neural Networks for Handwritten Hangul Character Recognition

Hanwool Park, Yechan Yoo, Yoonjin Park, Changdae Lee, Hakkyung Lee, Injung Kim, and Kang Yi*

School of Computer Science and Electrical Engineering, Handong Global University, Pohang, Korea

{210000315, 21500429, 21400318, 21200594, 21200608, ijkim, yk}@handong.edu

Abstract

Deep convolutional neural network (DCNN) is an advanced technology in image recognition. Because of extreme computing resource requirements, DCNN implementation with software alone cannot achieve real-time requirement. Therefore, the need to implement DCNN accelerator hardware is increasing. In this paper, we present a field programmable gate array (FPGA)-based hardware accelerator design of DCNN targeting handwritten Hangul character recognition application. Also, we present design optimization techniques in SDAccel environments for searching the optimal FPGA design space. The techniques we used include memory access optimization and computing unit parallelism, and data conversion. We achieved about 11.19 ms recognition time per character with Xilinx FPGA accelerator. Our design optimization was performed with Xilinx HLS and SDAccel environment targeting Kintex XCKU115 FPGA from Xilinx. Our design outperforms CPU in terms of energy efficiency (the number of samples per unit energy) by 5.88 times, and GPGPU in terms of energy efficiency by 5 times. We expect the research results will be an alternative to GPGPU solution for real-time applications, especially in data centers or server farms where energy consumption is a critical problem.

Category: Artificial Intelligence

Keywords: Deep convolutional neural networks; Deep learning accelerator; FPGA optimal design; Hangul character recognition

I. INTRODUCTION

Deep convolutional neural network (DCNN) has been widely employed in image recognition applications such as face detection [1], image classification [2], and video classification [3]. Due to the high computational accuracy of DCNN, the development of a wide range of modern applications based on DCNN algorithms is actively underway.

However, the DCNN takes a lot of computation time to get results because of its complicated architecture and huge workload. To overcome this problem, hardware accelerators based on GPGPU with CUDA is commonly used [4] because GPGPU can provide parallel processing and high computation performance. However, the accelerators based on GPGPU consume much higher energy compared to the CPU-based software. Therefore, hardware accelerators based on field programmable gate array (FPGA) have recently emerged as a new alternative owing to advantages of high energy efficiency, good performance, and reconfigurability [5].

Previous research on FPGA-based DCNN accelerators mainly concentrated on optimizing the external memory transfers or computational processing resources [6, 7]. There was research on a design space exploration methodology that used roofline model, but it was only

Open Access http://dx.doi.org/10.5626/JCSE.2018.12.1.24 http://jcse.kiise.org

Received 14 September 2017; **Revised** 05 December 2017; **Accepted** 09 February 2018

*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

applied to convolution layers [8]. Suda et al. [9] presented a systematic methodology for maximizing throughput of an OpenCL-based FPGA accelerator to the entire DCNN architectures including convolution layers, normalization layers, pooling layers, and classification layers.

In this work, we present FPGA design optimization techniques for minimizing memory access time and maximizing computational performance for all the layers of DCNN network. We also demonstrate the effectiveness of the presented approach by targeting hand-written Hangul character recognizer implementation.

Our main contributions of this paper are summarized as follows. (1) We show FPGA design optimization techniques for DCNN including memory access optimization and computation optimization toward minimizing total execution time for any given DCNN model. (2) We propose a systematic approach to minimize the total execution time of any DCNN structure and to consume computational resource under the FPGA resource limitation. (3) We achieved the fastest and most accurate implementation for the hand-written Hangul character recognizer in FPGA. We applied our design techniques to the implementation of the DCNN for handwritten Hangul recognition (HHR) presented in [10] that has the best accuracy in the world. Our HHR with world's record implementation outperforms CPUs in terms of execution time and GPGPU in terms of energy efficiency. (4) We succeeded in implementing the first handwritten Hangul character recognizer into FPGA in the world. Therefore, we could not compare with other state-of-the-art works in this reference.

The rest of this paper is organized as follows. Section II describes a background of DCNN, HHR, and SDAccel development environment. Section III shows the design techniques to optimize DCNN architectures into an FPGA-based accelerator. Section IV presents the implementation details. Section V makes experimental results and comparison between our implementation and different implementation. Section VI concludes the paper.

II. BACKGROUND

A. DCNN Basics

DCNN has produced remarkable outcomes in computer vision fields for the past few years. Because of having many layers including hidden layers, DCNN represents much more efficient various nonlinear functions than shallow neural networks [11]. Fig. 1 shows the overall architecture of the DCNN. The DCNN is composed of three types of layers: convolution layer, max-pooling layer, and fully-connected layers. As you can see in Fig. 1, convolution layers and max-pooling layers are composed of 2D planes, called feature maps. Each plane is connected to one or more planes of the previous layer. The planes are composed of nodes. Each node is connected to a small region of connected input planes.

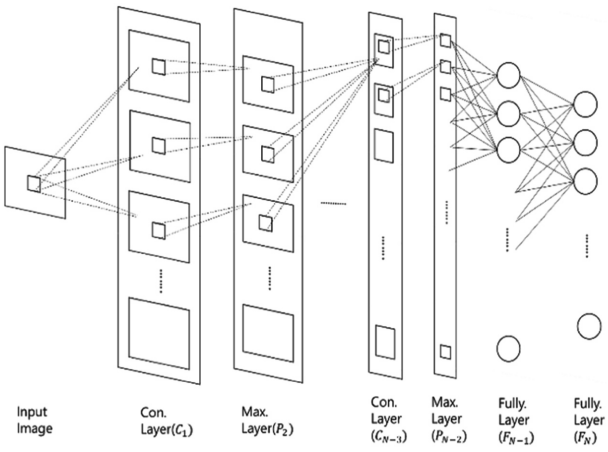

Fig. 1. The overall architecture of the DCNN.

Convolution layer is the most complex layer in DCNN. It is a process of extracting features from the input image or the feature maps of the previous layer. Eq. (1) [10] shows how each output node is calculated from the previous input image or the feature maps of the previous layer in the convolution layer.

$$X^n_{(p,i,j)} = f\left(\sum_{q \in C^n_p} \sum_{0 \le u,v \le M_n - 1} \left(w^n_{(q,p,u,v)} \times X^{n-1}_{(q,iS_n+u,jS_n+v)}\right) + \theta^n_p\right) \quad (1)$$

In Eq. (1), $X^n_{(p,i,j)}$ denotes the output of a node at coordinate (i, j) on the p^{th} plane of the n^{th} layer, C^n_p denotes the set of input planes is related to the p^{th} plane of the n^{th} layer, M_n denotes the width and height of the mask of layer n, $W^n_{(q,p,u,v)}$ denotes the weight of the connection from the input node to the activation node, $X^{n-1}_{(q,iS_n+u,jS_n+v)}$ denotes the node at coordinate (iS_n+u, jS_n+v) on the q^{th} plane of the $(n-1)^{\text{th}}$ layer, S_n denotes the stride size of the n^{th} layer, θ^n_p is a bias, and f is an activation function.

Max-pooling layer plays a role in sampling from the previous layer by choosing the maximum value among the input features. Eq. (2) [10] shows how output nodes of the max-pooling layer are calculated.

$$X^n_{(p,i,j)} = f\left(\max_{0 \le u,v \le M_n - 1} X^{n-1}_{(p,iS_n+u,jS_n+v)}\right) \quad (2)$$

where $X^n_{(p,i,j)}$, M_n, and $X^{n-1}_{(p,iS_n+u,jS_n+v)}$ denote the same meaning as the convolution layer.

Fully-connected layer combines the results from the previous layers and produced classification classes from the combined results. Eq. (3) [10] shows how each node is computed in the fully-connected layer.

$$X^n_p = f\left(\sum_q w^n_{(q,p)} X^{n-1}_q + \theta^n_p\right) \quad (3)$$

where X^n_p denotes the node on the p^{th} plane of the n^{th} layer, $w^n_{(q,p)}$ denotes the weight of the connection from the

input node to the activation node, and X_q^{n-1} denotes the node on the q^{th} plane of the $(n-1)^{th}$ layer.

B. Overview on the Target Application: Handwritten Hangul Recognizer

We applied our new design technique for DCNN to the DCNN-based HHR [10] for demonstration purposes. Our HHR [10] can recognize 2,350 different handwritten Hangul characters which achieved the world-best recognition rate on SERI95a and PE92. Our DCNN-based HHR is composed of 10 layers: 4 convolution layers, 4 max-pooling layers, and 2 fully-connected layers. Table 1 shows the details of the HHR structure and the complexity. Input feature maps are 64×64. Odd layers are convolution layers, and even layers are max-pooling layers except the last two layers which are fully-connected layers.

C. FPGA Development Environment

We use Xilinx SDAccel Development tool to implement the design expressed in C/C++ into a FPGA. SDAccel tool includes an optimized compiler that translates C codes into digital hardware system and makes efficient use of on-chip FPGA resources for hardware implementation. And, this tool also supports OpenCL and high-level synthesis (HLS) C, C++ libraries. Therefore, we can easily manipulate and optimize application source codes in C/C++ targeting HHR implementation. In this work, we use HLS C library for the optimal FPGA-based hardware implementation.

III. ACCELERATOR DESIGN OPTIMIZATION TECHNIQUES

A. Memory Access Optimization

There are several types of memory access overhead in FPGA-based accelerators. For example, there are data transmission and reception latency between the host processor and the FPGA-based accelerator. Also, off-chip global memory and on-chip local memory inside FPGA chip have different access delay time. The memory transfer time between off-chip and on-chip memory is significant. Therefore, optimizing the memory access pattern is critical for shortening the overall execution time. We suggest two approaches for minimizing the execution time of memory transfer: memory localization and embedding constant weights in local memory.

1) Memory Localization

Memory localization is to optimize data transmission for computation. Because all of the data to be used are in the off-chip global memory, the burst data transfer from the off-chip memory into the local on-chip memory before the computation can drastically reduce the overall time as shown in Fig. 2.

2) Embedding Constant Weights in Local Memory

Embedding constant weights in the local memory alleviates the communication overhead caused by accessing the global memory while fetching the operands. We use

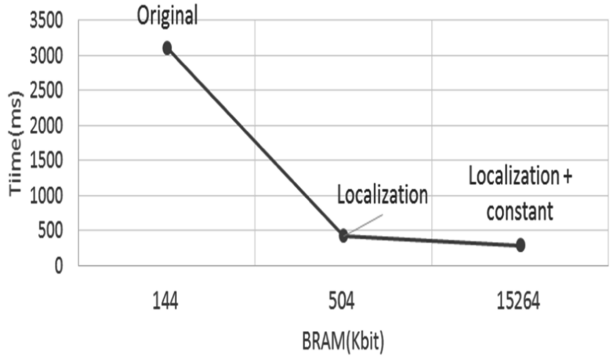

Fig. 2. The experimental result of memory access optimization.

Table 1. Our DCNN structure and complexity for HHR

Layer	Layer type	# of input feature maps	# of output feature maps	Input feature map size	Window size	Stride	# of weight	# of operations
C_1	Convolution	1	64	64 × 64	5 × 5	1	1,664	5,760,000
P_2	Max-pooling	64	64	60 × 60	2 × 2	2	0	230,400
C_3	Convolution	64	64	30 × 30	5 × 5	1	102,464	69,222,400
P_4	Max-pooling	64	64	26 × 26	2 × 2	2	0	43,264
C_5	Convolution	64	128	13 × 13	4 × 4	1	131,200	13,107,200
P_6	Max-pooling	128	128	10 × 10	2 × 2	2	0	12,800
C_7	Convolution	128	256	5 × 5	4 × 4	1	524,544	2,097,152
P_8	Max-pooling	256	256	2 × 2	2 × 2	2	0	1,024
F_9	Fully-connected	256	512	1 × 1	N/A	N/A	131,584	131,584
F_{10}	Fully-connected	512	2350	1 × 1	N/A	N/A	1,205,550	1,205,550

this constant embedding technique effectively because there are many constant weights in our CNN model.

3) The Experimental Result of Memory Access Optimization

Fig. 2 illustrates the experimental result of the memory access optimization techniques in case of the first convolution layer in our DCNN model. In Fig. 2, memory localization resulted in 86% reduction in run time while a 3.5× increase in the BRAM usage. As a result of applying both localization and embedding constant weights, 92% of the execution time is reduced, while BRAM usage is increased 106 times compared to the original implementation.

B. Computation Optimization

There are six types of optimization for computation time reduction. It is noteworthy to mention that the combination of these techniques can achieve further reduction in execution time.

1) Data Type Conversion to Fixed Point

In the original software code in C/C++, we used floating point data type for the HHR DCNN engine. However, the integer operation fits to FPGA compared to the floating point operation. Floating point operator needs more FPAG resources and consumes computing time. According to our experimental observation, data type conversion from floating point to fixed point may be beneficial in terms of resource saving and execution time reduction. However, to maintain the computing accuracy, we should preserve the fractional part of floating point data when converting data types as much as possible. Therefore, for higher accuracy, we need to multiply the input floating point values by a constant factor to move some fractional part digits to mantissa part. We describe this process of finding the best multiplication constant value for optimal bit length to obtain better accuracy and faster execution time.

***a) Conversion from floating point to fixed point*:** The conversion process from a floating-point value to a fixed-point value is performed by simple multiplication of a constant, 2^F, as shown in Eq. (4).

$$\begin{aligned} FixedPoint\ Value &= FloatingPoint\ Value \times 2^F \\ FixedPoint\ Value &= FloatingPoint\ Value << F \end{aligned} \quad (4)$$

where F is the fractional bit length in the floating-point data that needs to be preserved after the type conversion.

Depending on the F value, the accuracy of fixed-point value changes. Smaller F requires less resource overhead but it has less accuracy. However, if F becomes larger, the resource requirement increases and the accuracy gets higher. Therefore, we need to find the optimal F value, which maximizes accuracy while minimizing resource overhead.

Furthermore, we need to carefully consider the situation where two converted values are multiplied together to calculate the output nodes in convolution layers or fully-connected layers. Eq. (5) clearly shows this problem.

$$\begin{aligned} &output\_node = Fixed\_input\_node \times Fixed\_weights \\ &= Floating\_input\_node \times 2^F \times Fixed\_weights \times 2^F \\ &= Floating\_input\_node \times Floating\_weights \times 2^{2F} \end{aligned} \quad (5)$$

As you can see in Eq. (5), if we multiply the input node and weight will result into fixed-point values, we cannot get the proper output node value because the output node is multiplied by F twice. Therefore, in this case, we have to divide the output node value by 2^F in order to obtain the correct output as shown by Eq. (6).

$$\begin{aligned} outputNode &= \frac{FixedInputNode \times FixedWeights}{2^F} \\ outputNode &= FixedInputNode \times FixedWeights >> F \end{aligned} \quad (6)$$

b) Finding the optimal F value for type conversion: As described previously, there is a trade-off in accuracy and resource requirement with different F value choices. We performed experiments with two types of variables: 32 bits and 64 bits integer type variables. The experimental results are shown in Fig. 3.

With the 64 bits integer variables, the accuracy is saturated when F is 10, and higher F value does not much contribute to the accuracy improvement. On the other hand, with the 32 bits integer type variables, if F is 10, the recognition rate becomes higher and decreases drastically with larger F because the bit length 32 cannot cover the whole valid integer value ranges used for the operations in our HHR model. Based on the analysis on the experiments, we choose 10 for the F value and 32 bits integer type.

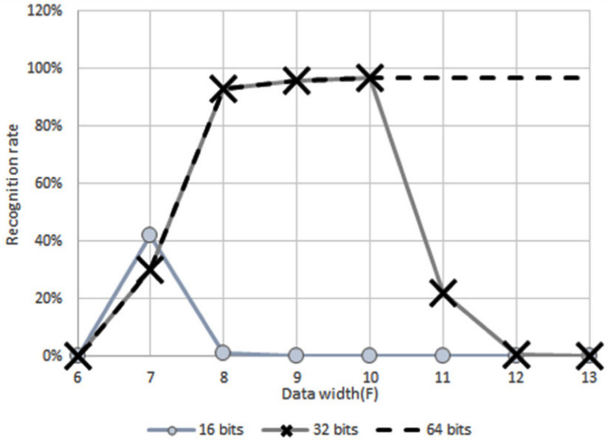

Fig. 3. The recognition rate between 16 bits, 32 bits, and 64 bits.

Table 2. Comparison between floating point and fixed point

	Floating point	Fixed point
Time (ms)	5.24	4.09
FF	1975	1828
LUT	2439	2043
DSP	0	0
BRAM	14	14

In Table 2, we compare the implementation results for the fixed-point with F=10 and floating-point in case of the max-pooling layer of our HHR model. As you can see in Table 2 below, the execution time of a fixed point is 22% less than that of floating point.

2) Loop Pipelining

The loop pipelining technique allows operations in a loop to be implemented in a concurrent manner. Therefore, it improves the throughput of the loop iterations. In the HLS C library, we simply write '#pragma HLS PIPELINE' right after loop initial declaration to apply the loop pipeline technique.

You can see the effectiveness of loop pipelining in Fig. 4. We applied two different conditions to the last fully-connected layer. By the loop pipeline technique, the computation time reduced to 7 ms, achieving 85% computing time reduction for the fully-connected layer implementation.

3) Loop Unrolling

The loop unrolling technique is a way of reducing a number of sequential iterations by building operators in the loop of several hardware instances processing in a parallel manner. We simply write '#pragma HLS factor = #' as a similar way as loop pipelining to do loop unrolling.

In Fig. 5, we can see the experimental result of loop unrolling comparing two different conditions to the first convolutional layer. The execution time is reduced by

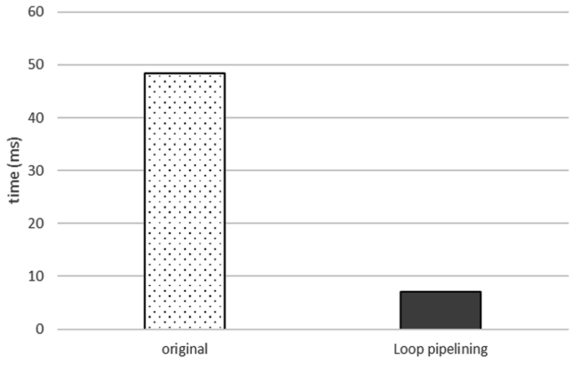

Fig. 4. The experimental result of loop pipelining.

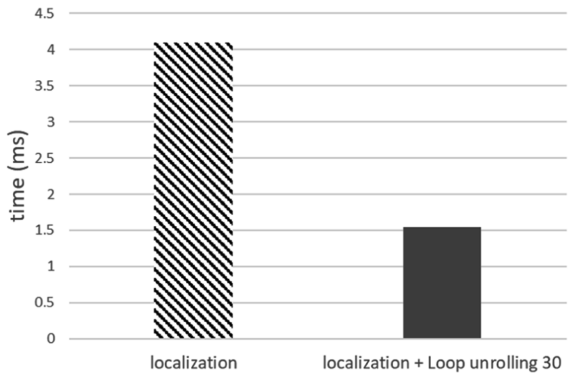

Fig. 5. The experimental result of loop unrolling.

63% from 4.09 ms to 1.54 ms.

Note that in a nested loop case, if the inner loop fails to be unrolled, the unrolling of outer loop will not work hence the effect of the unrolling technique will be halved.

4) Loop Partitioning

This approach is effective in optimizing memory bandwidth. The loop partitioning technique splits a bigger memory into multiple smaller memories. To be specific, arrays are usually implemented in an FPGA by on-chip memory (BRAM) which has two data ports, the maximum, limiting the throughput of a read/write (or load/store) intensive algorithm. The on-chip memory bandwidth can be improved by splitting the array (a single big BRAM) into multiple smaller arrays (multiple small BRAMs), effectively increasing the number of access ports.

The loop partitioning technique should be applied with loop unrolling. Because of the memory bandwidth problem, operators within a loop that are synthesized with loop unrolling technique cannot access on-chip memory in parallel because there are only two data ports with a BRAM. Therefore, loop unrolling should be used with loop partitioning to achieve better performance. To apply loop unrolling with loop partitioning, we should use both pragma commands: both memory partitioning pragma and loop unrolling pragma.

Table 3 shows the comparison between loop unrolling and combination of loop unrolling and loop partitioning for the last fully-connected layer. With applying only the loop unrolling technique, the execution time results in 39.98 ms. On the other hand, with the combination of loop unrolling and loop partitioning, the execution time results in 0.75 ms. The combination of the two techniques showed 54 times faster than loop unrolling alone. In addition, we found that if the number of parallel divisions of the loop and the number of partitions of on-chip memory are the same in number, the performance becomes the highest. Therefore, we apply the same number to the loop unrolling factor and memory partitioning factor.

Table 3. The comparison between loop unrolling and combination of loop unrolling and loop partitioning

	Fully connected layer (10th) speed comparison	
	Recognition time (ms)	**Speed comparison**
Partial localization + loop unrolling (Factor: 513)	39.98	×1
Partial localization + loop unrolling (Factor: 513) + memory partitioning (Factor: 513)	0.75	×54

5) Loop Reordering

The loop reordering technique is for reordering the loops in the nested loops. Because optimization techniques for the loop can be applied sequentially from the innermost loop first, nested loops should be reordered so that the loop which applies the optimization techniques can be placed on the innermost part of loops.

We utilize data sharing relations proposed by Zhang et al. [8] to apply the loop reordering technique. The study classifies the buffer and processing element (PE) relations into four categories: relevant, irrelevant, dependent, and independent [8]. The irrelevant relation means broad connections between buffers and PEs. The dependent relation means there are complex connections between buffers and PEs. The independent relation means direct one-to-one connections between buffers and PEs. Therefore, if we want to optimize to the fullest, the loop without dependent relations should be placed on the innermost.

Based on data sharing relations, we apply loop reordering technique to the convolutional layer as shown in Fig. 6. You can see that 'ip' loop variable has independent relation with input and weights, and irrelevant relation with output. Therefore, we locate 'ip' loop in the innermost part of the loop for optimization.

Loop reordering itself is not powerful than computational optimization. However, if this approach combines with the previously introduced techniques such as loop pipelining or loop unrolling, it can be greatly effective in computational optimization.

6) Loop Tiling

The loop tiling technique is a way of dividing the loop into smaller iteration units. As the number of loop iterations increases, more hardware resources are consumed. Due to the limitation of FPGA resource, if the number of loop iterations exceeds a certain threshold, the technique cannot be effective. To overcome the problem, we engage loop tiling technique.

```
for(op=0 ; op < NoOutputPlane; op++){
  for(m=0 ; m < maskSize ; m++){
    for(to=0; to < outputSize ; to++){
      int i = index(m,to);
      for(ip =0; ip < NoInputPlane ; ip++){
        output[op][to] += input[ip][i] * weights[op][ip];
      }
    }
  }
}
```

Fig. 6. Pseudo code of convolution layer after loop reordering.

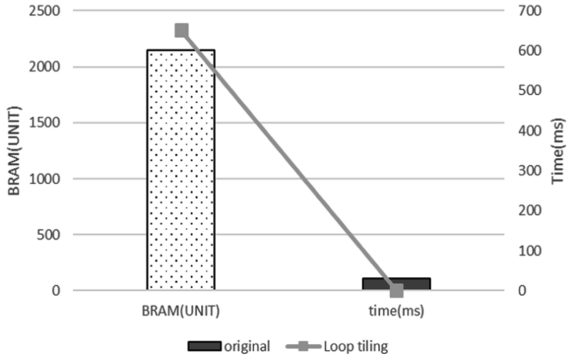

Fig. 7. The experimental result of loop tiling.

The experimental result of loop tiling is shown in Fig. 7. After applying loop tiling with the other loop optimization techniques, only 651 units of BRAM are consumed. It means that we can apply additional optimization approaches to this layer and other layers.

As shown in Fig. 6, a significant improvement is achieved compared to the case of applying only localization into the fully-connected layer. Furthermore, the execution time with loop tiling is drastically reduced from 110.87 ms to 0.41 ms.

IV. IMPLEMENTATION DETAILS

A. Implementation Environment

In Fig. 8, we briefly illustrate Xilinx SDAccel framework that is the target platform for our HHR application to be

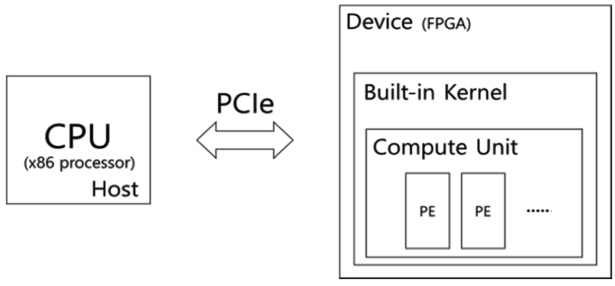

Fig. 8. SDAccel framework.

developed and tested.

In the SDAccel environment, host processor (×86 processor) and devices (accelerator, FPGA board) are connected by PCIe bus. The device consists of a set of compute units, which are again divided into many PEs. The processing elements play a key role in executing operations and achieving the computational parallelism. Kernel means a function executed by the device. Due to the limitation of FPGA resources, only the parts that needs computing acceleration by hardware are usually synthesized into FPGA device. In this work, only the DCNN model of our HHR is implemented into device.

B. Exploring Design Space by Optimization Techniques Combination

As mentioned in the previous section, combining the optimization techniques can significantly reduce the execution time of DCNN layer. We found each of the layer has 11, 11, 13, 7, 15, 7, 11, 11, 14, and 16 number of synthesizable combinations, respectively as shown in Fig. 9. The graphs in Fig. 9 show design space within each layer by different combinations of the optimization techniques. The x-axis of the graph is the FPGA BRAM resource usage and the y-axis of the graph is the execution time for each space within each layer by different combinations of optimization techniques. The x-axis of the graph is the FPGA BRAM resource usage and the y-axis of the graph is the execution time for each combination. Each dot in the graph represents a design alternative obtained by the combination of a set of optimization techniques, and we confirmed each of the synthesis result meets the FPGA resource limitation by the synthesis. In Fig. 9, all the feasible design options are listed including the original design. We can see that using more FPGA resource utilization generally reduces the execution time of the operations.

In Eq. (7), we formalize the design space which shows combination of optimization techniques for each layer.

$$\boldsymbol{L}_i = (\boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3, \boldsymbol{P}_4, \boldsymbol{P}_5, \boldsymbol{P}_6, \boldsymbol{P}_7) \tag{7}$$

where L_i denotes the combination for i^{th} layer, P_1 denotes memory localization, P_2 denotes embedding constant weights, P_3 denotes loop reordering, P_4 denotes loop pipelining, P_5 denotes loop unrolling, P_6 denotes loop tiling, and P_7 denotes memory partitioning.

We mark 0 or 1 in P_1, P_2, P_3, and P_4. Zero means the specific optimization techniques is not applied to L_i while 1 means this optimization technique is applied to

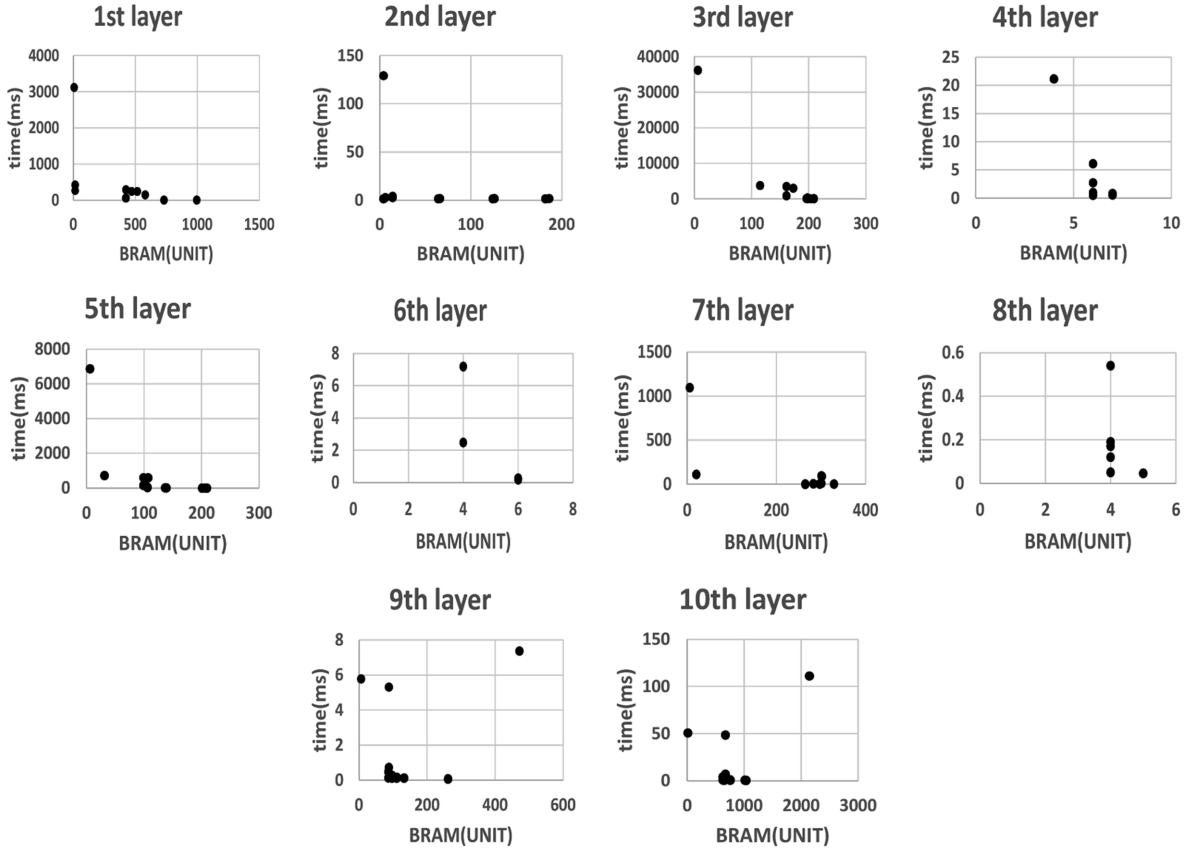

Fig. 9. The execution time and utilization BRAM for each layer when we apply various optimization approach in each layer.

Table 4. Parameter sets of each layer

Layer	P_1	P_2	P_3	P_4	P_5	P_6	P_7
L_1	0 or 1	0 or 1	0 or 1	0 or 1	(4,16), (4,32), or (4,64)	(3,10), or (3,15)	10, 15, 16, 32, or 64
L_2	0 or 1	0 or 1	0 or 1	0 or 1	(2,30), (2,60), (2,90), or (2,150)	(2,30), (2,60), (2,90), or (2,150)	30, 60, 90, or 150
L_3	0 or 1	0 or 1	0 or 1	0 or 1	(4,16), (4,32), (4,64), (3,13), or (3,26)	(3,13), or (3,26)	13, 26, 16, 32, or 64
L_4	0 or 1	0 or 1	0 or 1	0 or 1	(2,13), or (2,169)	(2,13)	13 or 169
L_5	0 or 1	0 or 1	0 or 1	0 or 1	(4,16), (4,32), (4,64), (3,5), (3,10), or (3,25)	(3,5), (3,10), or (3,25)	5, 10, 16, 25, 32, or 64
L_6	0 or 1	0 or 1	0 or 1	0 or 1	(2,5), or (2,25)	(2,5)	5 or 25
L_7	0 or 1	0 or 1	0 or 1	0 or 1	(4,16), (4,32), (4,64), (4,128), or (3,4)	(0,0)	4, 16, 32, 64, or 128
L_8	0 or 1	0 or 1	0 or 1	0 or 1	(2,16), (2,32), (2,64), (2,128), or (2,256)	(2,16) or (2,32)	16, 32, 64, 128, or 256
L_9	0 or 1	0 or 1	0 or 1	0 or 1	(2,16), (2,32), (2,64), (2,128), or (2,256)	(2,16), (2,32), (2,64), or (2,128)	16, 32, 64, 128, or 256
L_{10}	0 or 1	0 or 1	0 or 1	0 or 1	(2,16), (2,32), (2,64), (2,128), (2,256), or (2,512)	(2,16), (2,32), (2,64), (2,128), or (2,256)	16, 32, 64, 128, 256, or 512

L_i. We mark (N, M) for P_5 and P_6. N means the order of loop, and M means unrolling factors. N=0 and M=0 means no application of the optimization technique. We mark memory partitioning factors in P_7. We assume that when we mark loop tiling technique with the same unrolling factors is applicable. Table 4 shows the feasible parameter sets for each layer. Configurations come from Table 4. To build the whole DCNN, we should find a configuration by choosing one combination for each layer.

The objective is to minimize the execution time of HHR accelerators by choosing the best combination of each layer from the view of global optimality while satisfying the FPGA resource limitation.

Let us give a few examples to show how to find the optimal design configuration for the minimal execution time. We figure out the costs (BRAM resource) and benefits (execution time) for each different set of optimization techniques for each layer. For the explanation purpose, here, we mention only three layers. Tables 5–7 show the costs of each optimization technique in the first convolution layer, the first max-pooling layer, and the first fully-connected layer, respectively. In Tables 5–7, C_i, M_i, and F_i represent a combination of convolution layer, max-pooling layer, and fully-connected layer, respectively. We considered four optimization techniques. In Tables 5–7, P_4, P_5, P_6, and P_7 denote loop pipelining techniques, loop unrolling factor, loop tiling factor, and memory partitioning factor, respectively. For all cases, we assume that memory

Table 5. The costs of each combination of the first convolution layer (possible L_1 configurations)

Combination	P_4	P_5	P_6	P_7	Time (ms)	BRAM (unit)
C_1	1	-	-		59.10	424
C_2	0	(4,16)	-	16	242.55	470
C_3	0	(4,32)	-	32	238.90	516
C_4	0	(4,64)	-	64	147.55	580
C_5	1	(4,64)	-	64	61.46	422
C_6	1	-	(3,10)	10	1.97	731
C_7	1	-	(3,15)	15	2.27	995

Table 6. The costs of each combination of the first max-pooling layer (possible L_2 configurations)

Combination	P_4	P_5	P_6	P_7	Time (ms)	BRAM (unit)
M_1	1	-	-	-	2.64	14
M_2	0	(2,30)	-	30	1.54	64
M_3	0	(2,60)	-	60	1.51	124
M_4	0	(2,90)	-	90	1.52	182
M_5	0	(2,150)	-	150	2.75	6
M_6	1	-	(2,30)	30	1.63	66
M_7	1	-	(2,60)	60	1.63	126
M_8	1	-	(2,90)	90	1.63	186
M_9	1	-	(2,150)	150	1.49	4

Table 7. The costs of each combination for the first convolution layer (possible L_9 configurations)

Combination	P_4	P_5	P_6	P_7	Time (ms)	BRAM (unit)
F_1	1	-	-	-	0.73	88
F_2	0	(2,16)	-	16	0.47	86
F_3	0	(2,32)	-	32	0.28	97
F_4	0	(2,64)	-	64	0.17	111
F_5	0	(2,128)	-	128	0.12	133
F_6	0	(2,256)	-	256	0.076	262
F_7	1	(2,256)	-	256	0.05	262
F_8	1	-	(2,16)	16	0.12	86
F_9	1	-	(2,32)	32	0.11	97
F_{10}	1	-	(2,64)	64	0.1	111
F_{11}	1	-	(2,128)	128	0.1	133

localization, embedding constant weights, and loop reordering techniques are applied by default.

Table 8 shows the search results to find proper configurations by combining the optimization techniques. For the purpose of explanation, let us assume that the maximum BRAM size is 900 and we have only 3 layers. As shown in Table 8, $C_6 + M_9 + F_7$ configuration seems to be the best solution for minimum latency. However, $C_6 + M_9 + F_7$ cannot be chosen as a solution because the BRAM size of the configuration exceeds the BRAM size limitation. Therefore, we choose $C_6 + M_9 + F_{10}$ as a feasible answer.

Table 8. The results of configuration for each layer

Configuration	Time (ms)	BRAM (unit)
$C_1 + M_1 + F_1$	62.47	526
$C_1 + M_1 + F_2$	62.21	524
…	…	…
$C_6 + M_9 + F_7$	3.51	997
...	…	…
$C_6 + M_9 + F_{10}$	3.56	846
…	…	…
$C_7 + M_9 + F_{11}$	3.86	1,132

The optimal configuration search problem can be formalized as shown in Eq. (8).

$$\text{Find } \omega = \{L_i | i = 1, ..., 10\} \text{ s.t. minimize } \sum d(i, L_i) \quad \textit{while satisfying } \sum R(i, L_i) < MaxR \tag{8}$$

```
B[i] = {} for all i in 1 to 10;
MinT = maxInteger;
// Search Algorithm for the i-th layer
int Search(int i, float totalTime, float totalResource, float **T, float **R){
    float thisTime, thisResource;
    if(i > maxLayer){
        if(totalTime < MinT)
            MinT = totalTime;
      return MinT; // rearch last layer
    }

    for( index = 0; index < layerLength[index] ; index++){
        thisTime = T[i][index];
        thisResource = R[i][index];
        if(totalTime+thisTime > MinT || totalResource + thisResource > MaxR)
            continue; // terminate the search
        else{
            B[i] = index;
            MinT = Search(i+1, totalTime+thisTime, totalResource + thisResource, T, R);
        }
     }
}
```

Fig. 10. Proposed recursive algorithm for finding configuration with minimum latency.

where $d(i, L_i)$ denotes the latency time of the i^{th} layer by the L_i combination of optimization technique.

$R(i, L_i)$ is the FPGA resource usage when the L_i configuration is applied to the i^{th} layer. MaxR is the maximum resource of the FPGA chip. FPGA resource may mean LUT, FF, DSP, BRAM, and so on.

Theoretically, we can enumerate 31,336,425,120 configurations by choosing one of the combinations from each layer of our HHR. The greedy algorithm that searches the design space exhaustively is not feasible. Therefore, in Fig. 10, we propose a dynamic-programming style search algorithm to solve the search problem.

In the algorithm shown in Fig. 10, the array B variable saves the configuration index of each layer. MinT denotes the current minimum execution time found. Total time denotes the sum of the execution time of previous layers, total resource denotes the sum of resource utilization of previous layers. The single array L denotes the number of optimization technique applied to each layer. The array T denotes the execution time of indexth approach in i^{th} layer. The array R denotes the resource utilization of indexth approach in i^{th} layer, and MaxR denotes the maximum resource of the FPGA chip.

Using this algorithm, we can find the optimal design configuration for all layers with least execution time of DCNN.

V. EVALUATION

A. Experimental Setup

We implement the HHR into PEA-C8K1-115 FPGA board from COTS Technology. On the FPGA board Xilinx FPGA Kintex XCKU115 is mounted. Its working frequency is 200 MHz, it has Block BRAM of 2,160 unit. We use SDAccel Development tools (v16.1). For the comparison with the FPGA-based accelerator, we also implemented our HHR algorithm with both GPGPU and CPU. GPGPU implementation runs on NVIDIA GTX 780 Ti. The CPU-based software implementation runs on an Intel i7-4790k (@ 4.00 GHz).

Table 9. FPGA resource utilization

Resource	FF	LUT	DSP	BRAM
Used	342,068	258,813	3,273	1,365.5
Available	1,326,720	663,360	5,520	2,160
Utilization (%)	25.78	39.02	58.64	63.22

B. Experimental Results

Table 9 shows the FPGA resource utilization of HHR implementation by FPGA. As shown in Table 8, BRAM is the most limiting resources, which means larger BRAM is needed for further optimization.

Table 10 shows the comparison of the experimental results among different implementation environments. We measured the recognition time, power consumption, and energy consumption. In Table 10, CPU columns contain the results of CPU-based approach with 1 or 10 threads; meaning only single thread CPU execution and 10 multiple thread CPU execution time, respectively. GPGPU column means GPGPU-based results, and FPGA column means FPGA-based execution results. The samples per second and samples per Joule mean execution speed and energy efficiency, respectively.

As shown in Table 9, samples per second of FPGA are only 29% of GPGPU, while the energy efficiency (samples per Joule) of FPGA approach is 500% better than GPGPU and 580% better than 10-thread CPU. In summary, the total energy consumption requirement is in the order of FPGA < GPGPU < CPU.

VI. CONCLUSION

In this paper, we implemented DCNN architecture into the FPGA-based accelerator by using various optimization techniques. We have introduced our design space search algorithm for finding minimum latency design under the environment of limited resource in FPGA. Using the proposed algorithm, we achieved the energy efficiency with FPGA-based implementation 5× and 5.8× times

Table 10. Experimental results

	CPU (i7-4790k 4.00 GHZ)		GPGPU (GTX 780 Ti)	FPGA (XCKU115)
	1 thread	10 threads		
Type	Floating point	Floating point	Floating point	Fixed point
Time (ms)	74.47	22.00	3.30	11.19
Power (W)	22	66	374	22
Energy (J)	1.64	1.45	1.23	0.25
Samples per second	13.43	45.45	303.03	89.37
Samples per Joule	0.61	0.69	0.81	4.06

superior to the energy efficiency of CPU and GPGPU-based implementation approaches.

AKNOWLEDGMENTS

This work was supported by the National Program for Excellence in Software at Handong Global University (No. 2017-0-00130).

REFERENCES

1. S. S. Farfade, M. J. Saberian, and L. J. Li, "Multi-view face detection using deep convolutional neural networks," in *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, Shanghai, China, 2015, pp. 643-650.
2. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097-1105, 2012.
3. A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F. F. Li, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, Columbus, OH, 2014, pp. 1725-1732.
4. H. Takenouchi, T. Watanabe, and H. Asai, "Development of DT-CNN emulator based on GPGPU," in *Proceedings of the 2009 RISP International Workshop on Nonlinear Circuits and Signal Processing (NCSP2009)*, Honolulu, HI, 2009.
5. K. Ovtcharov, O. Ruwase, J. Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Research Whitepaper, 2015.
6. C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, 2010, pp. 257-260.
7. M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Proceedings of the 2013 IEEE 31st International Conference on Computer Design (ICCD)*, Asheville, NC, 2013, pp. 13-19.
8. C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*, Monterey, CA, 2015, pp. 161-170.
9. N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. S. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, 2016, pp. 16-25.
10. I. J. Kim and X. Xie, "Handwritten Hangul recognition using deep convolutional neural networks," *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 18, no. 1, pp. 1-13, 2015.
11. Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1-127, 2009.

Hanwool Park

Hanwool Park received his B.S. degree in computer science and engineering from Handong Global University, Korea in 2017. He is now a student of master program of computer science in Technical University of Munich, Germany.

Yechan Yoo

Yechan Yoo is a student for Bachelor's degree in computer science and electrical engineering at Handong Global University, Korea. He is currently studying under Professor Kang Yi at Handong Global University. His current research interests include parallelism computing for intelligent systems using FPGA and HLS.

Yoonjin Park

Yoonjin Park is currently a student for Bachelor's degree in computer science and electrical engineering at Handong Global University, Korea. Her research interests include embedded system programming and FPGA.

Changdae Lee

Changdae Lee is currently pursuing his Bachelor's degree in computer science at Handong Global University, Korea. He also joined Deepbio Inc. as a software engineer in 2017 and is currently working on full stack web development and machine learning.

Hakkyung Lee

Hakkyung Lee is currently an undergraduate student in electrical engineering at Handong Global University, Korea. His research interests include FPGA design and electronics device design.

Injung Kim

Injung Kim received his B.S., M.S., and Ph.D. degrees in computer engineering from KAIST, Korea; he received Ph.D. in 2001. He was a senior research engineer at Inzisoft Co. Ltd. He is currently a professor at the school of Computer Science Electrical Engineering, Handong Global University, since 2006. He received KAIST Best Dissertation Award in computer science department in 2001 and IR52 Jang-Young-Sil Award by Minister of Science and Technology in 2005. He was titled as 'A man of merit in the development of SW industries' by prime minister of Korea in 2014.

Kang Yi

Kang Yi received his B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Korea in 1990, 1992 and 1997, respectively. He joined Handong Global University in Feburary 1999 as full time lecturer in the School of Computer Science and Electrical Engineering, where he is now a full Professor since 2010. In addition, he was a visiting professor at University of California, Irvine, during 2005–2006 and was an invited professor at Center for Integrated Smart Sensors during 2013–2014. He has published and presented more than 100 papers and 10 books in the area of multimedia and low power embedded systems.