Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System*

Chi Zhang, Viktor Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, USA 90089
{zhan527, prasanna}@usc.edu

ABSTRACT

We present a novel mechanism to accelerate state-of-art Convolutional Neural Networks (CNNs) on CPU-FPGA platform with coherent shared memory. First, we exploit Fast Fourier Transform (FFT) and Overlap-and-Add (OaA) to reduce the computational requirements of the convolutional layer. We map the frequency domain algorithms onto a highly-parallel OaA-based 2D convolver design on the FPGA. Then, we propose a novel data layout in shared memory for efficient data communication between the CPU and the FPGA. To reduce the memory access latency and sustain peak performance of the FPGA, our design employs double buffering. To reduce the inter-layer data remapping latency, we exploit concurrent processing on the CPU and the FPGA. Our approach can be applied to any kernel size less than the chosen FFT size with appropriate zero-padding leading to acceleration of a wide range of CNN models. We exploit the data parallelism of OaA-based 2D convolver and task parallelism to scale the overall system performance.

By using OaA, the number of floating point operations is reduced by 39.14% $\sim$ 54.10% for the state-of-art CNNs. We implement VGG16, AlexNet and GoogLeNet on Intel QuickAssist QPI FPGA Platform. These designs sustain 123.48 GFLOPs/sec, 83.00 GFLOPs/sec and 96.60 GFLOPs/sec, respectively. Compared with the state-of-the-art AlexNet implementation, our design achieves 1.35x GFLOPs/sec improvement using 3.33x less multipliers and 1.1x less memory. Compared with the state-of-art VGG16 implementation, our design has 0.66x GFLOPs/sec using 3.48x less multipliers without impacting the classification accuracy. For GoogLeNet implementation, our design achieves 5.56x improvement in performance compared with 16 threads running on a 10 Core Intel Xeon Processor at 2.8 GHz.

*This work is supported by the US NSF under grants ACI-1339756 and CCF-1320211. This work is also supported in part by Intel Strategic Research Alliance funding. Equipment grant from the Intel Hardware Accelerator Research Program is gratefully acknowledged.

FPGA '17, February 22-24, 2017, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/3020078.3021727

Keywords

Convolutional Neural Networks; Discrete Fourier Transform; Overlap-and-Add; CPU; FPGA; Shared Memory; Double Buffering; Concurrent Processing

1. INTRODUCTION

Convolutional Neural Network (CNN)[9, 15, 17, 20] has been widely used in image recognition, video analysis and natural language processing. Its high computational complexity and need for real-time performance in many applications, has lead to several efforts to accelerate CNN. Various accelerators and libraries have been developed on FPGA[4, 13, 16, 21], GPU[14], multi-core processor[23] for both inference and training.

Due to the high computational complexity of the convolutional layer, prior work has addressed parallelism of the computation by unrolling the 2D convolution to matrix multiplication[12] or reducing the number of operations using Fast Fourier Transform[10]. However, parallelization by unrolling encounters a bottleneck due to limited on-chip memory of FPGAs. Even though FFT provides an asymptotically superior approach, the large gap between the input feature map size and kernel size makes it very inefficient. Other attempts include compressing the model using approximation[22] and data quantization techniques[13] while sacrificing some classification accuracy.

Recently, heterogeneous architectures employing FPGA accelerators have become attractive including Xilinx Zynq, Convey-HC2 and Intel QuickAssist QPI FPGA Platform[19, 8, 11]. The shared memory and high speed interconnection in these platforms makes data communication more efficient between the CPU and the FPGA compared with earlier platforms. The flexibility of CPU and massive parallelism of FPGA makes accelerating large scale CNNs promising. Our design effectively uses the CPU-FPGA platform as follows:

- We exploit the massive parallelism of FPGA to accelerate the most computationally intensive and universal operation (2D convolution) and leave the other layer specific work to the general purpose processor. This makes our approach highly flexible and applicable to a wide range of CNN architectures.
- We characterize the parallelism of our FPGA accelerator by data parallelism and task parallelism. Data parallelism is determined by the convolver design while task parallelism is determined by the number of convolvers operating in parallel. We carefully optimize the design to effectively use the available FPGA resources.

We develop a highly-parallelized convolver in frequency domain on FPGA and a software engine on CPU for inter-layer data remapping including ReLU layer. The CPU is also responsible for optional pooling layer, normalization layer and fully-connected layer after the convolutional layers. The main contributions of this paper are:

- We make a quantitative analysis of the required number of floating point operations in convolutional layers by space convolution[1] and by using Fast Fourier Transform (FFT) and Overlap-and-Add (OaA) [18].
- We propose a highly-parallelized OaA-based 2D convolver architecture on FPGA to accelerate the convolutional layers.
- To make a fair comparison among various convolvers, we use a composite performance metric in signal processing called *Delay-Multiplier (DM)* Product. We demonstrate the superiority of OaA-based convolver by showing that there always exists a FFT size such that the DM Product of OaA-based convolver is less than that of space convolver for typical kernel sizes.
- We exploit double buffering technique on FPGA to reduce the memory access latency. We make a quantitative analysis of the tradeoff between the on-chip memory consumption and the memory access latency.
- We propose a novel data layout in the shared memory for efficient data communication between CPU and FPGA. We also propose a software engine on CPU to perform data remapping between layers working concurrently with FPGA.
- We evaluate our work by implementing VGG16[15], AlexNet[9] and GoogLeNet[17] on Intel QuickAssist QPI FPGA Platform. Experimental results show that our designs achieve 2.28x $\sim$ 4.5x improvement in resource efficiency and 0.48x $\sim$ 1.90x improvement in power efficiency.

2. BACKGROUND

2.1 CNN in Frequency Domain

Typically, a CNN contains four building blocks, known as convolutional layer, ReLU layer, pooling layer and fully-connected(FC) layer. The overall architecture of a CNN is a connection of several such building blocks to form a very deep classification system.

2.1.1 Convolutional Layer

Throughout this paper, we use the following notations.

- Input feature maps of size $N_{in} \times N_{in} \times D_{in}$
- D_{out} kernels, each of size $F \times F \times D_{in}$
- Output feature maps of size $N_{out} \times N_{out} \times D_{out}$

The convolutional layer serves as a feature extractor. For each $N_{in} \times N_{in}$ input feature map[2], it performs 2D convolution with the shifting $F \times F$ kernel map of the same depth in each kernel with stride S. Then D_{in} output maps are summed up to obtain one output feature map. This process is repeated for all D_{out} kernels to obtain the complete $N_{out} \times N_{out} \times D_{out}$ output feature maps. Note that

$$N_{out} = \frac{N_{in} - F}{S} + 1 \quad (1)$$

2D Convolution and 2D FFT. 2D Convolution can be computed using 2D Fast Fourier Transform (FFT) as follows[18]:

$$y = \text{CONV2}(x, f) = \text{IFFT2}(\text{FFT2}(x). * \text{FFT2}(f)) \quad (2)$$

where x is a $N_{in} \times N_{in}$ input, f is a $F \times F$ kernel, y is the output of size $N_{out} \times N_{out}$. 2D FFT can be computed as 1D FFT of each row followed by 1D FFT of each column.

Using 2D FFT reduces the computation complexity of a convolutional layer from $\Theta(N_{in}^2 F^2)$ to $\Theta(N_{in}^2 \log N_{in})$. However, it is worth noticing that the overhead of FFT and additional operations due to zero-padding and stride cannot be neglected, especially when dealing with small kernels as in most CNNs. In traditional signal processing, Overlap-and-Add (OaA) [18] is used to efficiently calculate the discrete convolution of a long input signal with a small kernel. By using OaA, we can further reduce the computational complexity of a convolutional layer to $\Theta(N_{in}^2 \log F)$ [6].

2.1.2 Other Layers

Besides convolutional layer, CNN also contains three other layers including ReLU layer, pooling layer and fully-connected layer. It is still challenging to compute these layers in frequency domain due to the non-linearity of ReLU layer. The computation of a fully-connected layer can be viewed as large matrix-vector multiplication, which has been optimized on FPGA in earlier designs[13].

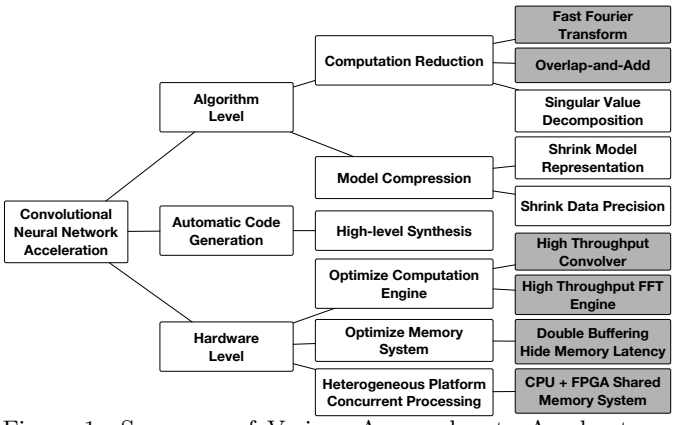


Figure 1: Summary of Various Approaches to Accelerate CNN

3. RELATED WORK

We summarize various approaches to accelerate CNN and highlight our focus in Figure 1. Currently, there are three main approaches to accelerate CNN including algorithm, automatic code generation and hardware. Using FFT to reduce the convolutional layer computation complexity is studied in [10, 6]. Using singular value decomposition (SVD) to accelerate fully-connected layer on FPGA is studied in [13].

In order to compress the CNN model, a new architecture called SqueezeNet is studied in [7]. Another effort to compress the model is to shrink the data precision from 32-bit

[1]We use space convolution to refer to direct convolution.
[2]We assume that zero-padding before the convolutional layer is included in N_{in}.

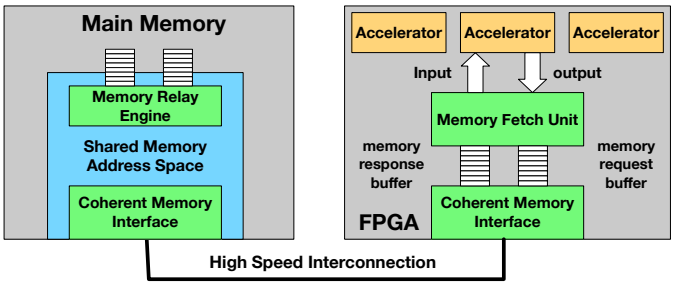


Figure 2: CPU-FPGA Shared Memory Model

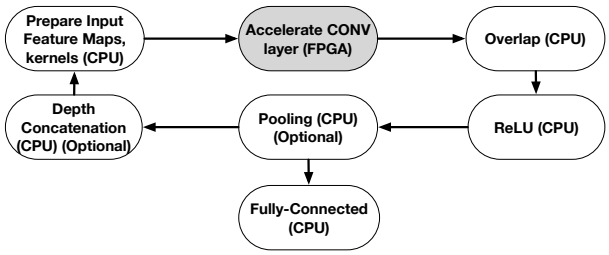


Figure 3: Proposed Mapping

floating point to 16 bit or even 8 bit fixed-point while preserving classification accuracy.

Hardware-level optimization is targeted for FPGA implementation. A high throughput convolver and a FFT engine has been studied in [2]. For large-scale CNN acceleration on FPGA, memory system optimization is the key to sustaining peak performance on a FPGA. A general strategy is to balance the computation throughput and memory bandwidth. Data buffering techniques can be used to hide the memory access latency. Concurrent processing on FPGA and CPU can further improve performance.

4. SYSTEM-LEVEL FRAMEWORK

4.1 Shared Memory Model

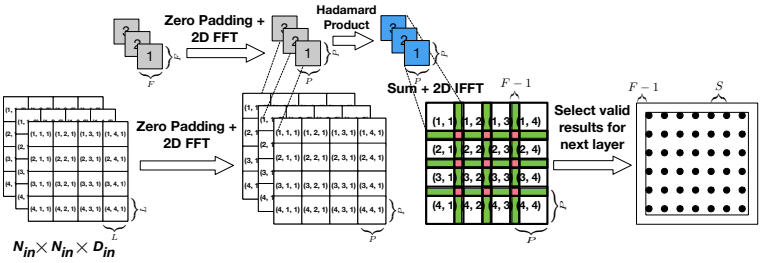
The shared memory model of CPU-FPGA platform is shown in Figure 2. Similar to the CPU load/store instruction, FPGA can send memory access request to the coherent memory system to perform read/write operations. A memory fetch unit and memory request and response buffer is implemented on FPGA.

The address space of the CPU user program is the entire main memory while the FPGA can only access a portion of it. To transfer data between the shared memory address space and the CPU thread private address space, a memory relay engine is implemented as a thread on the CPU.

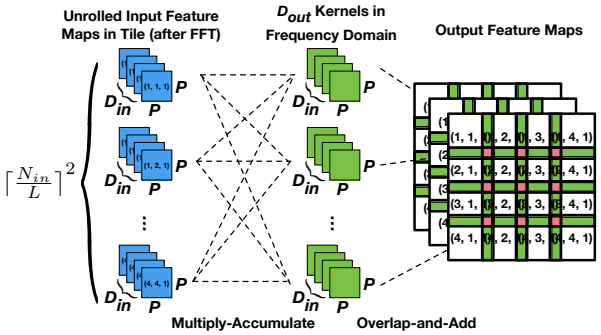
In the shared memory model, the FPGA can be viewed as a special thread, which operates on the same set of data with CPU. Using a synchronization mechanism, CPU and FPGA can process different portions of the shared data concurrently.

4.2 Mapping Choices on CPU and FPGA

The CPU-FPGA heterogeneous platform provides a flexible CPU for light-weight operations and a FPGA for high performance parallel computing. Besides, the shared memory model provides an easy and efficient mechanism for FPGA to access data. Thus, various design choices can be made to accelerate CNNs. We identify our design choices in Figure 3.



(a) 2D OaA for original input feature maps layout (Only one kernel is shown above)



(b) 2D Overlap-and-Add for unrolled input feature maps tiles

Figure 4: Illustration of 2D Overlap-and-Add

As mentioned in [21], the convolutional layer occupies more than 90% of the total computation time in a CNN. Also, the convolution operations can be fully pipelined and parallelized on FPGA when transformed into frequency domain. The convolutional layer is mandatory inside all CNNs. By using Overlap-and-Add, we can build a generic convolver on FPGA for any kernel size less than the chosen FFT size.

The overlap varies with respect to the kernel size. The computation time required by performing overlap is much smaller compared with the time to perform 2D convolution. The ReLU layer is a threshold function, which takes very little time to compute on the CPU. The pooling layer takes the maximum value over a certain window, which is also a light-weight computation. The depth concatenation is optional and can be performed while the CPU is rearranging the data output from the FPGA. The shared memory model enables concurrent processing on the CPU and the FPGA, which reduces the overall computation latency further.

5. FREQUENCY DOMAIN 2D CONVOLVER

The 2D convolver is the key computation engine on FPGA to accelerate CNN inference. A "traditional" convolver on reconfigurable architecture is studied in [1]. Note that it cannot be used for different kernel sizes at run time. Also, the data parallelism of the design is too low to support high throughput computations. To address these issues, we propose a high throughput 2D convolver based on FFT and Overlap-and-Add (OaA) [18].

5.1 2D Overlap-and-Add

In traditional signal processing, Overlap-and-Add (OaA) is used to efficiently calculate the discrete convolution of a very long input signal with a small filter. Typical kernel sizes in CNN are $1 \times 1, 3 \times 3, 5 \times 5$ and 7×7, which are much

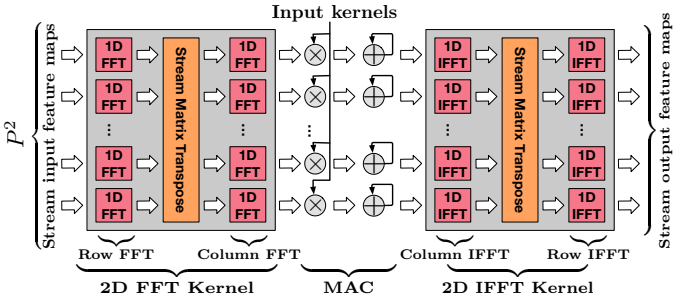


Figure 5: Diagram of OaA-based 2D Convolver

smaller compared with the input feature map size. This makes OaA suitable for this task.

The 2D Overlap-and-Add technique is illustrated in Figure 4a. The input is a $N_{in} \times N_{in} \times D_{in}$ feature map and D_{out} kernels, each size $F \times F \times D_{in}$. The steps of 2D Overlap-and-Add are:

1. Choose the row and column FFT size P, such that $P \geq F$ and P is a power of 2.
2. Pad each $F \times F$ kernel map to $P \times P$ and perform P point 2D FFT to each map inside each kernel. (Note this can be performed in advance and stored in the main memory for inference)
3. Divide the input feature map into $L \times L$ tiles (If N_{in} is not divisible by L, use zero padding), where $L+F-1 = P$. Then, pad each $L \times L$ tile to $P \times P$ and perform P point 2D FFT to each tile.
4. For each (i, j) pair, perform Hadamard Product of each input feature tile $(i, j, 1), \ldots, (i, j, D_{in})$ with corresponding kernel map $1, 2, \ldots, D_{in}$ of kth kernel. Then sum them up and perform P point 2D inverse FFT (IFFT) to obtain output tile (i, j, k).
5. Each output tile (i, j, k) overlaps with its neighboring tile with a stride of width $F-1$ as shown in Figure 4a. The green stride is overlapped by 2 tiles and red stride is overlapped by 4 tiles. Add all the overlapped items to produce one output feature map.
6. Note that not all the computed results are used as input feature maps for the next layer. First, crop out the boundary region of width $F-1$. Then, select the results with a stride of S as shown in Figure 4a.
7. Repeat Step 3 to 6 for all D_{out} kernels and concatenate the results in depth to obtain the complete output feature map.

Note that in Step 4, it is valid to perform the sum first and then perform 2D IFFT due to the linearity of IFFT. This will significantly reduce the amount of computations performed in IFFT.

5.2 OaA-based 2D Convolver

The proposed OaA-based 2D convolver is shown in Figure 5. It contains three stages including 2D FFT Kernel, Multiply-and-Accumulate (MAC) and 2D IFFT Kernel. The data parallelism of this architecture is P^2, where P is the 1D FFT size. The MAC unit must support complex numbers. A canonical complex number multiplier contains 3 floating-point multipliers [5].

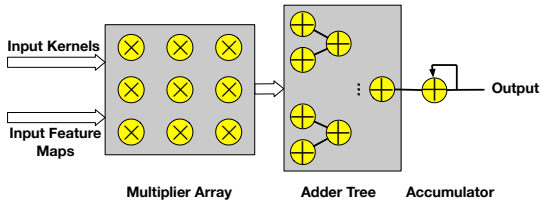


Figure 6: Diagram of Space 2D Convolver

To illustrate how the input data is fed into the convolver, we redraw the 2D Overlap-and-Add for the unrolled input feature tiles in Figure 4b. There are in total $\lceil \frac{N_{in}}{L} \rceil^2$ input feature tiles. For each kernel, we perform MAC with all the input tiles and overlap the results to obtain one output feature map. We then repeat the process for all the kernels to obtain the complete output feature maps.

The input of the convolver is one $P \times P$ tile of each input feature map and $P \times P$ map with the same depth of certain kernel. Note that the architecture shown in Figure 5 does not perform final overlapping. The reason is that the overlapped stride width is determined by the kernel size, which can change at run time for different layers. It is inefficient to build a specific hardware module compared with performing overlapping on CPU, given that it is a light-weighted computation.

The main motivation to use OaA to perform 2D convolution is to reduce the number of floating point operations asymptotically as well as increase the data parallelism. The computational complexity of space convolution and OaA convolution is $O(N_{in}^2 F^2)$, $O(N_{in}^2 \log F)$ respectively[3][6]. However, given that the kernel size is often small, the constant factor cannot be ignored and detailed analysis needs to be performed.

5.3 Performance Analysis

We assume the convolvers are all pipelined and no pipeline bubbles are fed into the convolver during any clock cycle. For space convolver, the kernels are in time domain. For OaA-based convolver, the kernels are in frequency domain.

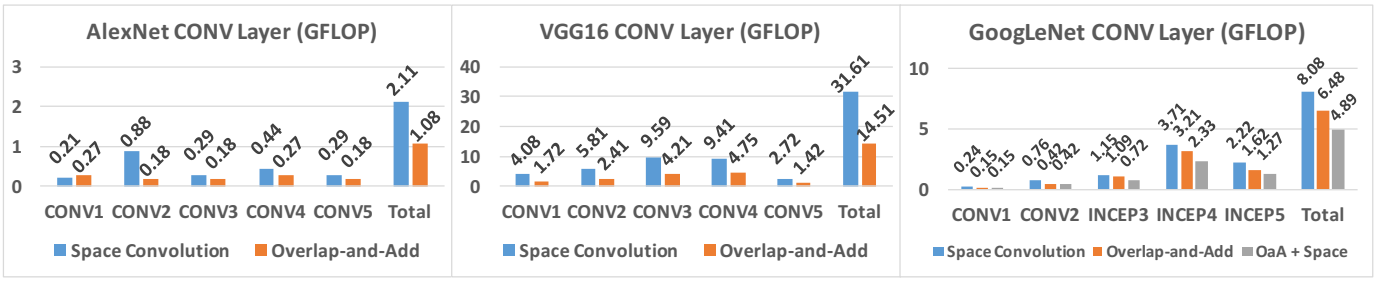
5.3.1 *Performance Metric*

We consider the *Delay-Multiplier* Product in traditional signal processing as a key performance metric to compare convolver designs employing various algorithms. It is defined as the delay (# of cycles) to process a complete convolutional layer times the number of multipliers used. Since we only compare the performance of the computation engines:

- Power consumption of the convolver is proportional to the number of multipliers.
- Area of the convolver is proportional to the number of multipliers.

Thus, the *Delay-Multiplier* Product is an approximation to *Delay-Power* Product, which is the energy dissipated by the design. Note that the *Delay-Power* Product is also an approximation to the *Area-Delay* Product, which is a key metric to evaluate the efficiency of a hardware design.

[3]We consider FFT-based convolution as a special case of OaA-based convolution, where $L = N_{in}$.



(a) AlexNet (2012) (b) VGG16 (2014) (c) GoogLeNet (2014)

Figure 7: Floating Point Operations required in various state-of-art CNNs

Table 1: # of multipliers needed for various 1D FFT kernels

FFT size	4	8	16	32
# of multipliers	0	4	24	88

Table 2: Space-OaA convolver DM ratio with various kernel and FFT sizes, $S = 1$, $N_{in} \gg F$

Kernel size	3	3	3	5	5	7
FFT size	4	8	16	8	16	8
***DM* ratio**	0.75	**1.01**	0.77	**1.25**	**1.56**	0.61
Kernel size	7	7	9	9	11	11
FFT size	16	32	16	32	16	32
***DM* ratio**	**2.12**	**2.31**	**2.25**	**3.25**	**1.89**	**4.09**

5.3.2 *Space Convolver*

We show the diagram of a space 2D convolver in Figure 6. It consumes $F \times F$ multipliers and can produce 1 valid output every $\frac{1}{D_{in}}$ clock cycle. Thus, the total latency[4] D_{space} is

$$D_{space} = (\frac{N_{in} - F}{S} + 1)^2 \cdot D_{in} \cdot D_{out} \quad (3)$$

The delay-multiplier product is

$$DM_{space} = (\frac{N_{in} - F}{S} + 1)^2 \cdot D_{in} \cdot D_{out} \cdot F^2 \quad (4)$$

5.3.3 *OaA Convolver*

According to Figure 4b, for each input tile and kernel pair, we need D_{in} cycles to process. There are $\lceil \frac{N_{in}}{L} \rceil^2 \cdot D_{out}$ such pairs. Thus,

$$D_{OaA} = \lceil \frac{N_{in}}{L} \rceil^2 \cdot D_{in} \cdot D_{out} \quad (5)$$

The number of multipliers needed in various FFT sizes is shown in Table 1. In addition, we need P^2 complex multipliers to perform MAC, each containing 3 floating point multipliers [5]. Thus, the delay-multiplier product of OaA convolver is:

$$DM_{OaA} = \lceil \frac{N_{in}}{L} \rceil^2 \cdot D_{in} \cdot D_{out} \cdot (3P^2 + 4P \cdot N_{mult}) \quad (6)$$

where N_{mult} is the number of multipliers in each 1D FFT kernel.

5.3.4 *Performance Comparison*

***DM* Product comparison.** To compare OaA convolver with space convolver, we define the space-OaA convolver DM ratio as

$$DM \text{ ratio} = \frac{DM_{space}}{DM_{OaA}} = \frac{(\frac{N_{in}-F}{S} + 1)^2 \cdot F^2}{\lceil \frac{N_{in}}{L} \rceil^2 \cdot (3P^2 + 4P \cdot N_{mult})} \quad (7)$$

We assume the stride is 1 and $N_{in} \gg F$ as in most convolutional layers. Using $P = L + F - 1$, we can simplify Eq 7 as:

$$DM \text{ ratio} = \frac{(P - F + 1)^2 F^2}{3P^2 + 4P \cdot N_{mult}} \quad (8)$$

We show the space-OaA convolver DM ratio with typical kernel and FFT sizes in Table 2. As shown in Table 2, for each typical kernel size 3, 5, 7, 9, 11, we can always find a FFT size such that the OaA convolver is superior to space convolver in terms of *Delay-Multiplier* Product as emphasized in bold. In practice, we choose the FFT size to maximize the DM ratio given on-chip resource and memory bandwidth constraints.

Computational Complexity. The total number of floating point operations required by various state-of-the-art CNNs using space convolution and OaA convolution is shown in Figure 7. For AlexNet, VGG16 and GoogLeNet, using OaA convolution can reduce the the total number of floating point operations by 48.82%, 54.10%, 19.79%, respectively, compared with the space convolution. Note that in GoogLeNet Inception Module, more than half of the convolutional layers use 1×1 kernels; in this case using overlap-and-add is not advantageous. If we use space convolution in convolutional layers with 1×1 kernels and use OaA convolution for the rest of the convolutional layers, we can reduce the the total number of floating point operations by 39.43% as shown in Figure 7c grey bars.

Reduced computational complexity can lead to less execution time and higher energy efficiency given the same computational resources (# of multipliers). We define the throughput as the number of outputs produced in each cycle by the convolver. It is determined by the hardware mapping of the algorithm. For OaA-based convolver, the throughput is $O(P^2)$ as shown in Figure 5, whereas the throughput of the adder-tree based space convolver is $O(1)$ as shown in Figure 6.

Flexibility. The OaA convolver can be exploited to accelerate **any kernel size** less than the FFT size P **at run time** if the data in the shared memory is appropriately zero-padded. This can be easily achieved by choosing the corresponding tile size L and performing data rearrangement using the CPU for different convolutional layers.

[4]We ignore the latency to fill and drain the pipeline since they are too small compared with the total processing time.

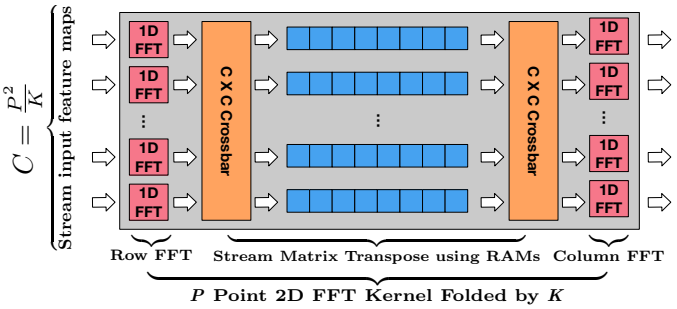


Figure 8: An example of folding 2D FFT Kernel

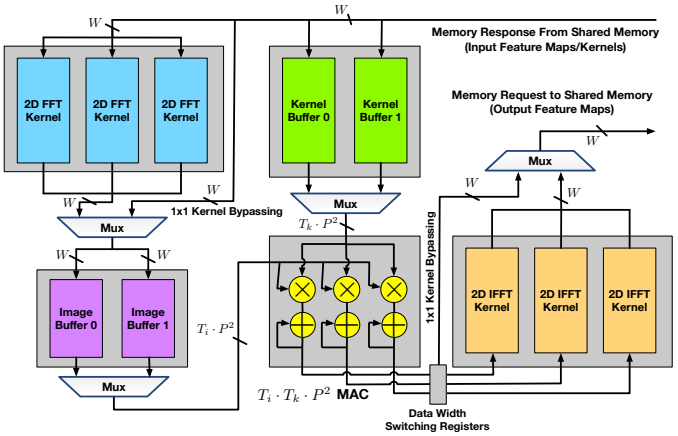


Figure 9: Overall System Design on FPGA, $T_i = 3, T_k = 1$

6. OVERALL SYSTEM DESIGN

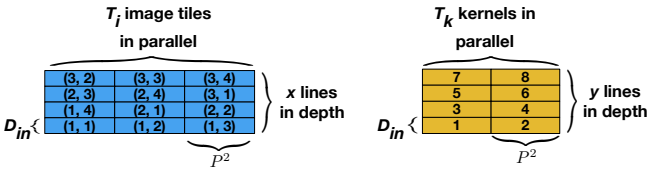
In Section 5, we assumed that the convolvers are running at the peak performance with no pipeline stalls. However, this may not be true if the design is bounded by memory bandwidth. This causes the computation engines on the FPGA to be stalled waiting for the input data from the external memory. The data parallelism of the 2D convolver may be larger than the communication channel data width. In this case, we need to fold the 2D FFT Kernel to match the communication channel data width and make the design scalable.

6.1 Folding 2D FFT Kernel

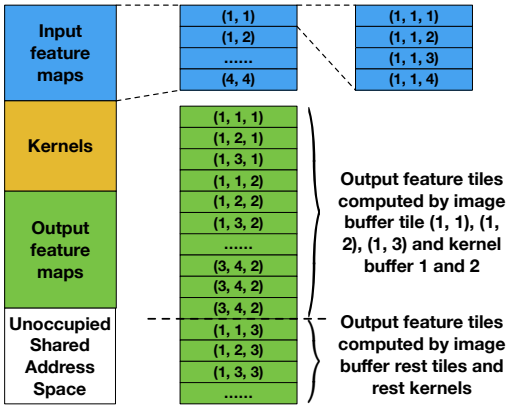
In order to match data parallelism with the communication channel data width W, we can reduce the data parallelism of the 2D FFT kernel. As shown in Figure 8, we fold it by a factor of K, where $1 \leq K \leq P$ and K is a divisor of P such that $W \approx \frac{P^2}{K}$. The data parallelism after folding is $\frac{P^2}{K}$ and the number of 1D FFT kernels required is $\frac{2P}{K}$. In order to support streaming matrix transpose for data arriving in consecutive cycles, we apply the techniques described in [3]. Matrix transpose is a special form of data permutation and by storing the data using intermediate RAMs, we can achieve matrix transpose for data arriving in consecutive cycles. The additional on-chip memory needed for folding 2D FFT kernel is $8P^2$ bytes.

6.2 Overall System Diagram

The overall system is shown in Figure 9. The input data from shared memory can be either input feature maps or kernels. The input feature map data is sent to 2D FFT Kernel and the frequency domain input feature map is stored in the image buffers. The kernel data is directly stored in the kernel buffers. The MAC array is the main computation engine, which reads data from the image and kernel buffers and performs MAC operations. Each accumulated result is sent to 2D IFFT kernel and the output data is sent to the shared memory. The data width switching registers are needed to match the MAC array output data parallelism with the communication channel data width. Note that the switcher registers will not overflow since the MAC array produces results every D_{in} cycles. Since the focus of this paper is CNN inference, we can assume that all the kernels are in frequency domain. These can be precomputed and stored in the main memory. In this section, we further explore **task parallelism** to scale the system performance. We define,



(a) On Chip Memory Layout for Image and Kernel



(b) Data Layout in Shared Memory Address Space

Figure 10: Data Layout for Convolutional Layer with 4×4 input feature map and 3×3 kernels, $T_i = 3, T_k = 2, x = 4 \cdot D_{in}, y = 4 \cdot D_{in}$

- T_i: The number of input feature maps processed in parallel in each cycle.
- T_k: The number of kernel maps processed in parallel in each cycle.

Note that we need to perform MAC operations between each input feature map and each kernel map. Thus, the total system task parallelism $T_t = T_i T_k$. Figure 9 illustrates the design for $T_i = 3, T_k = 1$.

As mentioned in Section 5, it is inefficient to perform convolution layer with 1×1 kernel in frequency domain. Thus, a **1-by-1 kernel bypassing** is added to the original data path as shown in Figure 9. In this case, the input feature maps are directly stored and the kernels are in time domain. The 2D IFFT is also bypassed and accumulated results are sent directly to the shared memory. To support 1-by-1 kernel bypassing, we design the MAC array to support both real numbers and complex numbers.

```
read_address_image = 0; read_address_kernel = 0;
for image_start_line = 0 to x − 1 by D_in
   current_image_line = image_start_line;
   for current_kernel_line = 0 to y − 1
      read_address_image = current_image_line;
      read_address_kernel = current_kernel_line;
      if current_image_line == image_start_line + D_in − 1
         current_image_line = image_start_line;
      else
         current_image_line = current_image_line + 1;
      end
   end
end
```

Code 1: Image and Kernel Buffer Read Address Computation

6.3 Optimizing Memory Subsystem

6.3.1 Data Layout

Input feature maps and kernels. We show the shared memory data layout in Figure 10b. The input feature maps and kernel data is stored tile by tile instead of the original 3D array. This enables the FPGA to access data through continuous virtual addresses instead of scattered pointers; this improves the cache performance due to high spatial locality.

On-chip memory. The data layout for on-chip input feature maps and kernels is also tile by tile as shown in Figure 10a. The P^2 data in one input feature map tile or one kernel map is unrolled and stored in parallel as shown in Figure 10a. Let x and y denote the depth of image and kernel buffer, respectively. An example of $x = 4 \cdot D_{in}$, $y = 4 \cdot D_{in}$ is shown in Figure 10a.

Output feature maps. The output feature map layout is determined by the image buffer depth x and the kernel buffer depth y. A pseudo-code to compute the read address of image and kernel buffers is shown in Code 1.

An example of the output feature map data layout in shared memory is shown in Figure 10b, where $x = 4 \cdot D_{in}$, $y = 4 \cdot D_{in}$ and the input feature map size is 4×4 and the kernel size is 3×3.

6.3.2 Double Buffering

To reduce the FPGA-memory traffic and enable full overlap of the memory access and the computation, double buffering technique is exploited as shown in Figure 9. Double buffering is only effective when the computation time is greater than the time to bring the same amount of data to on-chip memory of the FPGA. The data access pattern proposed in Code 1 satisfies this requirement because it takes $O(xy)$ time to perform computations while $O(x + y)$ time to bring the same amount of data to the on-chip memory. However, the on-chip memory will soon become the bottleneck. In this case, we can use only one image buffer and the latency to bring the input feature maps to on-chip memory cannot be fully hidden. The additional delay increased due to using only one image buffer is $4N_{in}^2 D_{in}/B$.

6.3.3 Timing Analysis

A detailed timing diagram of on-chip memory write destination, read source buffer and memory write request is shown in Figure 11. i, k denote the image buffer and the kernel buffer, respectively. At any time, only one image buffer and kernel buffer is activated to produce buffered data and the other image and kernel buffer is used for buffering data from the shared memory. Suppose the memory read bandwidth is B, the FPGA operating frequency is f. Then according to Code 1, the time to consume one kernel buffer is

$$t_{kernel,consume} = \frac{x}{D_{in}} \cdot y \cdot \frac{1}{f} \tag{9}$$

Each kernel buffer contains $T_k \cdot P^2 \cdot y$ complex numbers. Thus, the time to fill one kernel buffer is[5]

$$t_{kernel.fill} = \frac{T_k \cdot P^2 \cdot y \cdot 8}{B} \tag{10}$$

In order to hide the kernel memory access latency, the inequality $t_{kernel,fill} < t_{kernel,consume}$ must be satisfied. Thus,

$$x > \frac{8T_k P^2 D_{in} f}{B} \tag{11}$$

Another constraint is that we have to fill the vacant image buffer during the time slots between filling kernel buffer and consuming kernel buffer within one image buffer output period as shown in Figure 11. Thus,

$$(x \cdot \frac{y}{D_{in}} \cdot \frac{1}{f} - \frac{T_k \cdot P^2 \cdot y \cdot 8}{B}) \cdot \frac{D_{in} \cdot D_{out}}{y \cdot T_k} > \frac{T_i \cdot P^2 \cdot x \cdot 8}{B} \tag{12}$$

This leads to,

$$x > \frac{8P^2 D_{in} D_{out} f T_k}{D_{out} B - 8T_i P^2 f T_k} \tag{13}$$

where $D_{out} B - 8T_i P^2 f T_k > 0$. Thus,

$$x > \max(\frac{8T_k P^2 D_{in} f}{B}, \frac{8P^2 D_{in} D_{out} f T_k}{D_{out} B - 8T_i P^2 f T_k}) \tag{14}$$

where $D_{out} B - 8T_i P^2 f T_k > 0$. Note that there is no constraint on the kernel buffer depth y and it can be arbitrarily small as long as it can hold the largest kernel among all the convolutional layers. The total on-chip memory M_{total} needed by the design is

$$M_{total} = (x \cdot P^2 \cdot T_i + y \cdot P^2 \cdot T_k) \cdot 2 \tag{15}$$

6.3.4 CPU-FPGA Concurrent Processing

As shown in Figure 3, CPU performs overlap, ReLU, pooling and optional depth concatenation, which varies with different convolutional layers. These light-weight tasks are well suited for CPU and can be overlapped with the FPGA.

Synchronization Mechanism. The synchronization between the CPU and the FPGA is through a shared flag as shown in Figure 11. After FPGA completes each output tile of each output feature map, it sets a shared flag in the shared memory. Once the CPU detects the set flag, it performs overlap, ReLU, rearranges the data for the next layer.

6.4 Performance and Resource Estimation

The performance and resource consumption of our design employing double buffering and single image buffer is shown in Table 3, where N_{in} is input feature map size, L is OaA tile size, D_{in} is the number of input feature maps, D_{out} is the number of output feature maps, f is FPGA operating

[5]In this paper, we use 32-bit single precision floating-point.

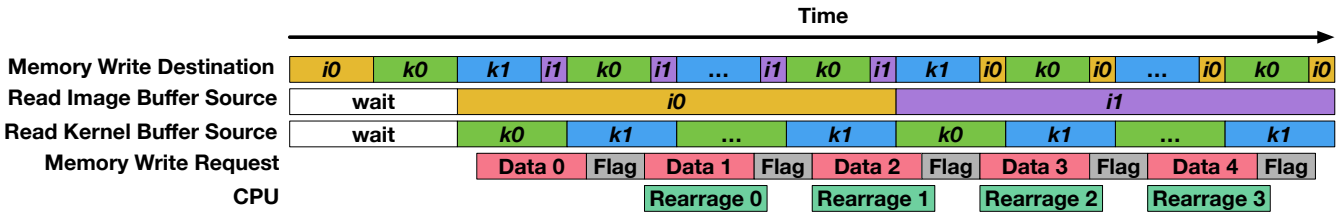


Figure 11: Timing Diagram

Table 3: Theoretical Performance and Resource Consumption

	double buffering	single image buffer
Delay	$\lceil \frac{N_{in}}{L} \rceil^2 D_{in} D_{out}/f$	$\lceil \frac{N_{in}}{L} \rceil^2 D_{in} D_{out}/f + 4N_{in}^2 D_{in}/B$
Multipliers	$3P^2 + 4PN_{mult}/K$	$3P^2 + 4PN_{mult}/K$
Memory	$P^2(2xT_i + 2yT_k + 8)$	$P^2(xT_i + 2yT_k + 8)$

frequency, B is memory bandwidth, K is 2D FFT folding factor, P is the FFT size, N_{mult} is the number of multipliers in 1D FFT kernel, T_i (T_k) is the image (kernel) task parallelism, x (y) is the image (kernel) buffer depth.

We also give an estimation of the performance and resource consumption by using single image buffer. According to the analysis in Section 6.3.3, the image buffer is often large and the kernel buffer is small. Using single image buffer can make the design practical for FPGAs with limited on-chip memory. The additional delay introduced to bring the input feature maps to FPGA is $4N_{in}D_{in}/B$, which is considerably small compared with the total processing time.

7. EXPERIMENTS AND RESULTS

7.1 Experimental Setup

We conducted our experiments on Intel Heterogeneous Architecture Research Platform (HARP), which is a preproduction of Intel QuickAssist QPI FPGA Platform[8]. It integrates 10 Core Intel Xeon E5-2600 v2 processor and Altera Stratix V FPGA with 6.25 MB BRAM on chip. The CPU and FPGA exchange data through a shared memory; this platform is an example of the model described in Section 4. The high speed interconnection in HARP is Intel QuickPath Interconnection (QPI), which achieves 5.0 GB/s bandwidth according to our own experiments. The QPI data width W is 64 bytes. A direct-mapped coherent cache of 64 KB is implemented on FPGA to reduce the data access latency. Results in this work were generated using preproduction hardware and software from Intel, and may not reflect the performance of production or future systems.

We conducted experiments on AlexNet, GoogLeNet and VGG16 using HARP. We measured the execution time of each group layer inside the three CNNs. We measured the CPU-FPGA sequential execution time; in this case the CPU waits for all the results from the FPGA and then performs overlap and ReLU. We also measured the concurrent execution time; in this case the CPU processes the partial results from the FPGA using the synchronization model discussed in Section 6.3.4.

7.2 Performance Evaluation

We show the overall design parameters in Table 4. Based on the number of DSPs available, the task parallelism is set to 1. Based on the available on-chip memory, we use a single image buffer of depth 8192. In AlexNet/VGG16, the maximum kernel depth is 512. In GoogLeNet, the maximum kernel depth is 832. Thus, we choose the kernel depth y to be 512 for AlexNet/VGG16 and 1024 for GoogLeNet. According to DM ratio analysis in Section 5, we choose FFT size to be 8, which is superior to space convolver in terms of *Delay-Multiplier* Product for most kernel sizes in CNNs in our experiments based on the available on-chip memory and measured bandwidth. The Intel HARP QPI data width supports 16 32-bit floating point data access in parallel. Thus, we can shrink the 2D FFT data parallelism to 16 and the corresponding folding factor K is 4.

Table 4: Design Parameters for AlexNet, VGG16 and GoogLeNet

Parameter	Design Value
Task Parallelism $T_t = T_i T_k$	1
# of image buffers	1
Image buffer depth x	8192
# of kernel buffers	2
Kernel buffer depth y	512 (AlexNet, VGG16) 1024 (GoogLeNet)
FFT size	8
2D FFT folding factor K	4

Table 5: Resource used in our design, 4.0 MB BRAM required by VGG16 and AlexNet. 5.0 MB BRAM required by GoogLeNet.

Resource	Registers (K)	Logic (ALM)	32-bit float Multipliers	DSP	BRAM (MB)
Used	266	200522	224	224	4.0/5.0
Available	939	234720	256	256	6.25

The resource consumption is shown in Table 5 for the design parameters in Table 4. According to Table 5, we conclude that the number of available DSPs is the bottleneck for the chosen design parameters.

Most of the kernel sizes in AlexNet, VGG16 and GoogLeNet convolutional layers is less than the chosen FFT size. Thus, they can be accelerated using the design parameters described in Table 4. Only one exception occurs in CONV1 of AlexNet, whose kernel size is 11 and stride is 4. Accelerating convolutional layers with large stride using frequency domain convolution is not attractive compared with space convolution. Hence, we choose to directly implement it on the CPU.

7.2.1 *Tradeoff Analysis*

On-chip Memory Consumption vs. Sustained Performance. The peak performance is achieved when there is no pipeline stalls in the computation engines. The sustained performance is determined by the number of the computation engine pipeline stalls and the total computation time. It

Table 6: Execution Time for VGG16 and AlexNet

Layer (Group)	VGG16 Execution Time (ms)					AlexNet Execution Time (ms)			
	FPGA (Theoretical)	FPGA (Actual)	CPU	Sequential	Concurrent	FPGA (Actual)	CPU	Sequential	Concurrent
CONV1	30.96	31.53	7.76	39.29	32.74	-	17.17	17.17	17.17
CONV2	44.36	46.01	4.18	50.19	46.48	7.86	0.09	7.95	7.94
CONV3	81.92	82.27	3.54	85.81	82.75	4.42	0.12	4.54	4.50
CONV4	81.92	82.77	1.37	84.14	82.90	6.64	0.15	6.79	6.71
CONV5	17.69	18.36	0.27	18.63	18.40	4.42	0.11	4.53	4.49
CONV Total	256.85	262.94	17.12	280.06	**263.27**	**23.34**	17.64	40.98	**40.81**

Table 7: Performance Comparison with the State-of-Art CNN Implementations on FPGA

	[21]	[13]	This Work	
Platform	Virtex7 VX485t	Zynq XC7Z045	Intel QuickAssist QPI FPGA	
Clock (MHz)	100	150	200	
Data Precision	32-bit float	16-bit fixed	32-bit float	
Bandwidth (GB/s)	12.8	4.2	5.0	
CNN Model	AlexNet	VGG16-SVD	AlexNet	VGG16
BRAM	4.5 MB	2.13 MB	4.0 MB	4.0 MB
DSP, Multipliers	2240, 747	780, 780	224, 224	224, 224
Throughput (CONV) (GFLOPs/sec)	61.62	187.80	83.00	123.48
Delay (CONV) (ms)	21.61	163.42	CPU:17.17 FPGA:23.64	263.27
Power (FPGA) (W)	18.61	9.63	13.18	13.18
Delay×Multipliers	16142	127467	9141/5295	58972
Resource Efficiency (GFLOPs/sec/Multiplier)	0.082	0.241	0.37	0.55
Power Efficiency (GFLOPs/sec/W)	3.31	19.50	6.30	9.37
Classification Accuracy	Lossless	Lossy	Lossless	
Flexibility	Any CNN	Limited	Any CNN	

Table 8: Resources for implementing multipliers

Multipliers	DSP
16-bit fixed point (Xilinx)	1
32-bit fixed point (Xilinx)	2
32-bit float point (Xilinx)	3
32-bit float point (Altera)	1

is determined by to what extent we can overlap the computation with the memory access latency. We show the impact of on-chip memory on sustained performance versus peak performance ratio in Figure 12.

Energy vs. FFT Size. It is shown in Section 5.3 that larger FFT size reduces the DM Product, which leads to the reduction of energy consumption of the convolvers. However, the energy consumed by on-chip memory to sustain peak performance increases. Thus, in an energy constrained system, we should tradeoff between the on-chip memory energy consumption and the convolver energy consumption.

7.2.2 VGG16 and AlexNet

We show the execution time of various layers in VGG16 and AlexNet in Table 6. Note that the VGG16 execution time of each layer is very close to the predicted value using our analysis. The predicted value is obtained using Table 3. The variation between predicted and actual results is due to the delay to fill and drain the pipelines. The sequential CPU-FPGA total execution time of VGG16 is 280.06 ms while the concurrent total execution time is 263.27 ms. The CPU-FPGA concurrent processing through shared memory reduces the total execution time by 6.0%. However, the advantage of concurrent processing is not noticeable in AlexNet since the execution time on CPU is much smaller than that of FPGA.

Table 9: GoogLeNet Convolutional Layer Performance

GoogLeNet (convolutional layer only) Execution Time (ms)		
Layer	**CPU (16 thds)**	**CPU (1 thd) + FPGA**
CONV1	38.53	12.70
CONV2	117.13	17.14
Inception3	91.05	16.16
Inception4	173.21	29.16
Inception5	45.45	8.48
Total	**465.37**	**83.64**
Throughput (GFLOPs/sec)	**17.36**	**96.60**

Comparison with State-of-the-Art. We show the comparison of our work with two state-of-art CNN implementations on FPGA in Table 7. Since the number of multipliers consumed is not directly available in [21, 13], we convert the DSP consumption to multipliers consumption using Table 8. Compared with the state-of-the-art AlexNet implementation, our design achieves 1.35x GFLOPs/sec and similar delay with 3.33x less multipliers and 1.1x less memory. Compared with the state-of-the-art VGG16 implementation[13], our design has 0.66x throughput with 3.48x less multipliers without sacrificing the classification accuracy.

Resource and Power Efficiency Comparison. Table 7 shows that our design improves resource efficiency by 4.5x and 2.28x compared with state-of-the-art CNN implementations[21, 13]. Compared with state-of-art AlexNet and VGG16 implementations, our design improves power efficiency by 1.90x and 0.48x. The reasons for having lower power efficiency compared with [13] is: 1) Our design consumes more BRAMs. 2) Our design uses floating-point operations, which limits the system task parallelism and overall performance. 3) Design in [13] is only optimized for VGG16 while our design is applicable to any CNN with maximum kernel size less than the chosen FFT size.

Fully-Connected (FC) Layer. We implemented the FC layer directly on CPU using 16 threads. The FC layer can be viewed as matrix-vector multiplication, which is bounded by memory bandwidth. The execution time for FC1, FC2, FC3 in VGG16 is 115.74 ms (16.10 ms with SVD), 19.30 ms and 4.71 ms, respectively. It outperforms the FC layer implementation on FPGA in [13].

7.2.3 GoogLeNet

We also implemented GoogLeNet on Intel HARP and the results are shown in Table 9. Our experiments show that we improve the performance by 5.56x compared with 16 threads running on 10 Core Intel Xeon CPU at 2.8 GHz. The software was compiled with gcc using optimization level O3 and OpenMP.

Scalability. The major issue to increase the thread-level parallelism to compute the convolutional layer lies in the

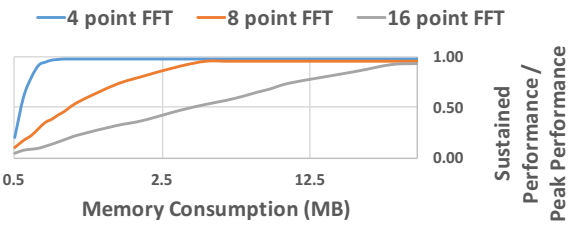


Figure 12: Sustained Performance versus Peak Performance ratio vs. Memory Consumption. The x-axis is in log scale

cost to maintain cache coherency. For FPGA based design, the parallelism can always be increased if the memory bandwidth and the FPGA on-chip resources increase.

8. DISCUSSION

Platform Choice. A CPU-FPGA based design will consume more power than FPGA-only based design. However, the CPU adds more flexibility to the design. Moreover, since most of the computational complexity is in the convolutional layers, the CPU performs simple operations and data rearrangement. Thus the power consumption of the CPU will not increase significantly as the CNN size increases.

Automatic Code Generation. Our framework provides a complete solution to accelerate CNN on FPGA including intra-layer data rearrangement. Widely used CNNs' convolutional layers mainly consist of small kernels. Thus, by zero-padding various kernel sizes to fit a chosen FFT size, and using FPGA to accelerate it by exploiting massive parallelism, we can achieve large performance improvement for various CNN models. We can use our framework to develop an automatic code generation tool so high-level users can specify CNN models and generate the design.

Fixed Point vs. Floating Point. Many previous approaches use fixed point instead of floating point for computations. The advantage is less resource consumption and higher power efficiency. However, it may penalize the classification accuracy.

9. CONCLUSION

In this paper, we first exploited Overlap-and-Add to reduce the computation complexity of the convolutional layers. Then, we proposed a 2D convolver in frequency domain to accelerate convolutional layers on FPGA. To optimize the memory system, we exploited the double buffering technique to overlap the computation with the memory access. Finally, we implemented 3 state-of-art CNN models including AlexNet, VGG16 and GoogLeNet on Intel QuickAssist QPI FPGA Platform. Future work includes employing our design to more CNN models, conducting experiments using fixed point data as well as automatic generation of optimized designs.

10. REFERENCES

[1] B. Bosi, G. Bois, and Y. Savaria. Reconfigurable Pipelined 2D Convolvers for Fast Digital Signal Processing. *IEEE Trans. On Very Large Scale Integration (VLSI) Systems*, 1999.

[2] R. Chen and V. K. Prasanna. Energy Optimizations for FPGA-based 2-D FFT Architecture. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, Sept 2014.

[3] R. Chen, S. Siriyal, and V. K. Prasanna. Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 240–249, New York, NY, USA, 2015. ACM.

[4] C. Farabet, Y. Lecun, K. Kavukcuoglu, B. Martini, P. Akselrod, S. Talay, and E. Culurciello. Large-Scale FPGA-Based Convolutional Networks. In R. Bekkerman, M. Bilenko, and J. Langford, editors, *Scaling Up Machine Learning*, pages 399–419. Cambridge University Press, 2011. Cambridge Books.

[5] M. Hemnani, S. Palekar, P. Dixit, and P. Joshi. Hardware optimization of complex multiplication scheme for DSP application. In *Computer, Communication and Control (IC4), 2015 International Conference on*, pages 1–4, Sept 2015.

[6] T. Highlander and A. Rodriguez. Very Efficient Training of Convolutional Neural Networks using Fast Fourier Transform and Overlap-and-Add. *CoRR*, abs/1601.06815, 2016.

[7] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *CoRR*, abs/1602.07360, 2016.

[8] Intel Inc. Xeon+FPGA Platform for the Data Center. https://www.ece.cmu.edu/ calcm/carl/lib/ exe/fetch.php?media=carl15-gupta.pdf.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[10] M. Mathieu, M. Henaff, and Y. LeCun. Fast Training of Convolutional Networks through FFTs. *CoRR*, abs/1312.5851, 2013.

[11] Micron Technology, Inc. The Convey HC-2 Computer. https://www.micron.com/about/about-the-convey-computer-acquisition.

[12] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, and C. Zhang. FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2016. cpe.3850.

[13] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA'16. ACM, 2016.

[14] D. Scherer, H. Schulz, and S. Behnke. *Accelerating Large-Scale Convolutional Neural Networks with Parallel Graphics Multiprocessors*, pages 82–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[15] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[16] 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*, FPGA '16, pages 16–25, New York, NY, USA, 2016.

[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *CoRR*, abs/1409.4842, 2014.

[18] Wikipedia. https://en.wikipedia.org/wiki/Multidimensional_discrete_convolution#Overlap_and_Add.

[19] Xilinx Inc. Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[20] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. *CoRR*, abs/1311.2901, 2013.

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

[22] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun. Efficient and Accurate Approximations of Nonlinear Convolutional Networks. *CoRR*, abs/1411.4229, 2014.

[23] A. Zlateski, K. Lee, and H. S. Seung. ZNN - A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-Core and Many-Core Shared Memory Machines. *CoRR*, abs/1510.06706, 2015.