Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs

Qingcheng Xiao*[1], Yun Liang†[1], Liqiang Lu[1], Shengen Yan[2,3] and Yu-Wing Tai[3]
[1]Center for Energy-efficient Computing and Applications, EECS, Peking University
[2]Department of Information Engineering, Chinese University of Hong Kong
[3]SenseTime Group Limited
{walkershaw,ericlyun,luliqiang}@pku.edu.cn,{yanshengen,yuwing}@gmail.com

ABSTRACT
Convolutional neural network (CNN) finds applications in a variety of computer vision applications ranging from object recognition and detection to scene understanding owing to its exceptional accuracy. There exist different algorithms for CNNs computation. In this paper, we explore conventional convolution algorithm with a faster algorithm using Winograd's minimal filtering theory for efficient FPGA implementation. Distinct from the conventional convolution algorithm, Winograd algorithm uses less computing resources but puts more pressure on the memory bandwidth. We first propose a fusion architecture that can fuse multiple layers naturally in CNNs, reusing the intermediate data. Based on this fusion architecture, we explore heterogeneous algorithms to maximize the throughput of a CNN. We design an optimal algorithm to determine the fusion and algorithm strategy for each layer. We also develop an automated toolchain to ease the mapping from Caffe model to FPGA bitstream using Vivado HLS. Experiments using widely used VGG and AlexNet demonstrate that our design achieves up to 1.99X performance speedup compared to the prior fusion-based FPGA accelerator for CNNs.

1 INTRODUCTION
Recently, convolutional neural networks (CNNs) are increasingly used in numerous cognitive and recognition computer vision applications [11, 13, 22]. CNN has high computation complexity as it needs a comprehensive assessment of all the regions of the input image or features maps and computes the score [7]. To overcome the computing challenge, specialized hardware accelerators designed for CNNs have emerged which deliver orders of magnitude performance and energy benefits compared to general purpose processors [4]. Among them, Field Programmable Gate Arrays (FPGAs) is an appealing solution due to its advantages of reconfigurability, customization and energy-efficiency [21, 27]. Recent progress in High Level Synthesis (HLS) has greatly lowered the programming hurdle of FPGAs [6, 15]. With the innovation of FPGA architecture and HLS, CNN inference applications are becoming commonplace on embedded systems [5, 18, 19, 21].

CNNs are composed of multiple computation layers, where the output feature maps of one layer are the input feature maps of the following layer. Prior studies have shown that the computation of the state-of-the-art CNNs are dominated by the convolutional layers [7]. For example, the convolutional layers of GoogleNet [22] occupies 90% of the total computation time. Convolutional layers can be implemented using a straightforward and general approach or other algorithms such as matrix multiplication, FFT through computation structure transformation.

More recently, `Winograd` algorithm [26] based on minimal filtering theory has been introduced for layers with small kernel sizes and strides [14]. Compared to the conventional implementation, fast `Winograd` algorithm reduces the number of required multiplications by reusing the intermediate filtering results [14]. `Winograd` algorithm is computing resource efficient but puts more pressure on the memory bandwidth. To accelerate CNNs on FPGAs, the key is to parallelize the CNNs as much as possible until either the computing resources (LUTs, BRAMs, DSPs, FFs) or memory bandwidth are exhausted. Unfortunately, homogeneous design using either conventional or `Winograd` algorithm will only exhaust one dimension of resource, leaving others under-utilized. To fully utilize FPGA resource, this work makes the following contributions:

- We present a framework that explores heterogeneous algorithms for accelerating CNNs on FPGAs. The framework employs *fusion* architecture to fuse multiple layers to save memory transfer, but efficiently utilize the computing resources.
- We design an optimal algorithm based on dynamic programming to determine the structure of the *fusion* architecture and the implementation algorithm for each layer. Given a CNN, our algorithm maximizes the throughput subject to a data transfer constraint.
- We present an automatic tool-flow to ease the mapping from the Caffe model to FPGA bitstream. The tool-flow will implement each layer and enable dataflow through Vivado HLS automatically.

Experiments using widely used VGG and AlexNet demonstrate that our techniques achieve up to 1.99x performance speedup compared to prior fusion-based FPGA accelerator for CNNs [1].

2 BACKGROUND AND MOTIVATION
2.1 Convolution Algorithms
The conventional algorithm directly convolves the input feature maps with convolutional kernels to produce the output feature maps. More clearly, N size $K \times K$ kernels with M channels are used to slide through the M size $H \times W$ input feature maps and perform the convolution. We use $D_{h,w,m}$ to denote the (h, w) element in the m_{th}

DAC '17, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062244

*This work is done during Qingcheng Xiao's internship at SenseTime.
†Corresponding Author

output feature map and $G_{n,u,v,m}$ to denote the the (u, v) element in the n_{th} kernel and m_{th} channel. Then, the computation can be formulated as follows,

$$Y_{i,j,n} = \sum_{m=1}^{M} \sum_{u=1}^{K} \sum_{v=1}^{K} D_{i*S+u,j*S+v,m} \times G_{n,u,v,m} \quad (1)$$

where S is the stride when shifting the kernels and $Y_{i,j,n}$ represents the (i, j) element in the n_{th} output feature map.

The conventional convolution algorithm is general but less efficient. As an alternative, convolution can be implemented using `Winograd` minimal filtering algorithm [14].

Let us denote the result of computing m outputs with the r-tap FIR filter as $F(m, r)$. Conventional algorithm for $F(2, 3)$ requires $2 \times 3 = 6$ multiplications. `Winograd` algorithm computes $F(2, 3)$ in the following way:

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \times \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \quad (2)$$

where

$$m_1 = (d_0 - d_2)g_0 \quad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \quad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

Now, only 4 multiplications are required. In general, the number of multiplications that `Winograd` algorithm requires is equal to the input size. The above 1D algorithm can be nested to form 2D minimal algorithms $F(m \times m, r \times r)$ as follows,

$$Y = F(m \times m, r \times r) = A^\top [[GgG^\top] \odot [B^\top dB]] A \quad (3)$$

where d is the $(m + r - 1) \times (m + r - 1)$ input tile, g is the $r \times r$ filter, G, B and A are constant matrices and $\odot$ indicates element-wise multiplication. For the 2D algorithm, each input feature map is first divided into tiles of size $(m+r-1)\times(m+r-1)$. Then, $F(m\times m, r\times r)$ is calculated with each tile and kernel for every channel. Finally, the results are accumulated to produce an output tile with size $m \times m$. The algorithm details can be found in [26].

For the implementation of convolution on FPGAs, DSPs is mainly the limiting resource as it is employed for multiplication. `Winograd` algorithm is more efficient as it performs the equivalent amount convolution operations but with less DSP resources. This algorithm can be implemented most efficiently for the cases where kernel size is small and stride is 1. There are multiple tile size choices for `Winograd` algorithm. In this paper, we use a uniform size $F(4 \times 4, 3 \times 3)$.

2.2 Motivation

Roofline model [25] has been designed to analyze the performance bottleneck by relating the attainable performance with memory bandwidth and computational roof visually. In Roofline model as shown in Figure 1, the X-axis is the computation to communication (CTC) ratio while the Y-axis represents the attainable performance. CTC ratio denotes the computation operations per transferred data. Bandwidth roof (e.g. slope) is the product of CTC ratio and off-chip memory bandwidth. Computational roof describes the peak performance provided by the available hardware resources. Obviously, the attainable performance is restricted to both the two roofs.

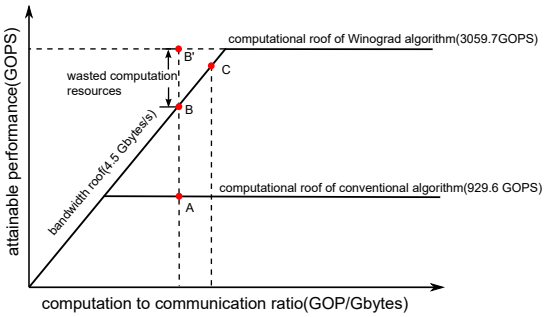

Figure 1: Motivation illustration using roofline model

We rely on the roofline model to illustrate the benefit of our heterogeneous design. The conventional and `Winograd` algorithms have different computational roofs. The conventional algorithm is known to be computation limited [7]. While the `Winograd` algorithm puts more pressure on the memory system since the computation capability is improved. In Figure 1, A represents the conventional algorithm and B represents the `Winograd` algorithm. In our system, both algorithms are implemented using the same data reuse structure. Therefore, they share the same CTC ratios.

We use B' to denote the ideal performance of `Winograd` algorithm without bandwidth roof. The performance gap between B and B' indicates the computing resource waste due to the bandwidth saturation. Let us use the 2_{nd} convolutional layer of VGGNet [20] as an example. This layer has 64 input feature maps with size 224×224 and 64 kernels with 64 channels and size 3×3. For simplicity, only DSP resources are considered when calculating the computational roofs and only the input feature maps are considered for bandwidth consumption. In Figure 1, design A yields 929.6 GOPS performance on a Xilinx FPGA chip Virtex7 485t, while design B suffers from insufficient bandwidth and achieves 2592 GOPS and 3059.7 GOPS can be realized by design B'.

In this paper, we employ *fusion* architecture to fuse multiple neighboring layers together. This design reconstructs the computation of the fused layers so that the inputs flow through the fused layers to produce the outputs, avoiding storing and reloading the intermediate feature maps. For the fused layers, we explore the conventional and `Winograd` algorithms. This helps to improve the computing resource utilization without aggravating the bandwidth. In fact, it actually increases the CTC ratio as more operations are performed for the same amount of transfer, leading to a better design C in Figure 1. Also, both conventional and `Winograd` algorithms can be implemented with different parallelism parameters, leading to different resource utilization. This adds another dimension to explore in our technique.

3 FRAMEWORK

Our framework provides a comprehensive solution that can map a great diversity of CNNs onto FPGAs. We design an automatic tool-flow to ease the mapping process as shown in Figure 3. It takes Caffe configuration file and specification of the target FPGA as inputs and generates bitstream on FPGA. Caffe is a popular deep learning infrastructure [12] and the structure of CNN can be described in its configuration file. The specification of the target FPGA includes Block RAMs (BRAMs), DSPs, off-chip bandwidth and others. The tool-flow involves three main components: architecture, optimal algorithm, and code generator.

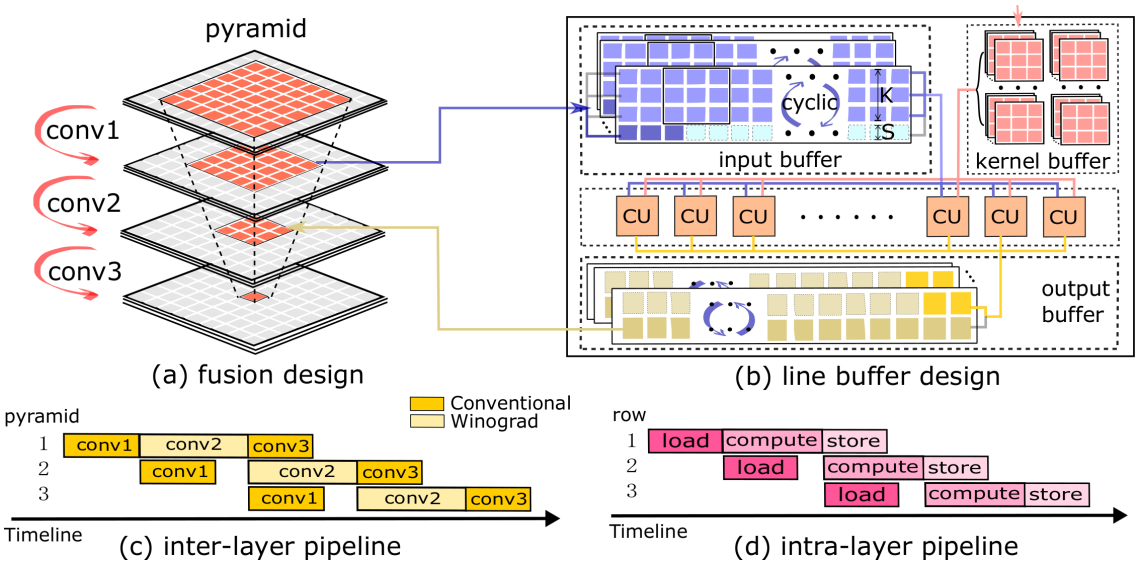

Figure 2: Architecture Details

- Architecture. Recently, a *fusion* architecture using tile-based buffers is introduced to fuse multiple layers together and save off-chip memory transfer. The tile-based reuse buffer is difficult to use as it has to deal with complex boundary conditions [1]. Instead, we propose a simple *fusion* architecture based on line buffer.
- Optimal Algorithm. We design an optimal algorithm to determine the structure of the *fusion* architecture and the implementation choice for each layer based on this architecture. The algorithm is based on dynamic programming and branch-and-bound search. Our algorithm also balances the inter-layer pipeline within a *fusion* group.
- Code Generator. We rely on HLS to generate the implementation of the optimal strategy. When generating the implementation code, templates are built in order to handle different kinds of parameters and layers. Then the source code is compiled into a bitstream using Vivado toolchain.

4 ARCHITECTURE DESIGN

4.1 Fusion Architecture

The *fusion* architecture is designed based on the fact that for convolutional operations one element in the output feature map only depends on a small region (e.g. kernel size) of the input feature map, which in turn depends on a larger region of its input layer. Figure 2 (a) shows a *fusion* example of three layers, every element of conv3 layer depends on a 3×3 tile of conv2 layer. Each element in the conv2 layer depends on a 3×3 tile of conv1 layer. Collectively, the final output element along with all the tiles it relies on compose a pyramid. Using *fusion* architecture, to compute one element in the final output layer, we only need an input tile of the first layer, all the necessary intermediate tiles in the pyramid can be computed, without storing and retrieving the intermediate data to and from off-chip. Thus, this design reduces the pressure of memory bandwidth.

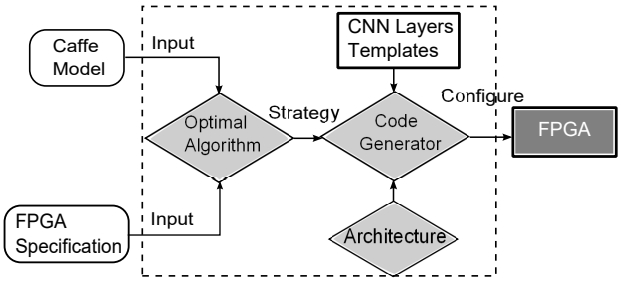

Figure 3: Framework Overview

4.2 Line Buffer Design

The pyramids of adjacent elements in the last layer overlap with each other, leading to data reuse opportunities. A detailed discussion was made about whether to reuse or recompute these values in [1]. In its final design, tile-based buffers are adopted to store those reusable data and additional layers are inserted between original layers to manage these buffers. However, complex operations are performed to update the tile-based buffers due to mutative boundary conditions. Besides, these buffers occupy additional BRAMs. In this work, we use circular line buffer for each layer as shown in Figure 2 (b), which naturally achieves data reuse without extra resources or elaborate data management efforts.

Suppose a convolutional layer has M input feature maps with size $H \times W$. It convolves with N size $K \times K$ kernels with M channels. The shifting stride of kernels is S. In our design, the whole input line buffer consists of $K + S$ lines. Initially, the first K rows of input feature maps are loaded into line $[1, K]$. After this, kernels slide through these lines to perform convolutions and produce the first row of corresponding output feature maps. Meanwhile, the next S rows are being transferred into line $[K + 1, K + S]$. Then, we convolve line $[1 + S, K + S]$, load feature maps into line $[1, S]$ and store the first output row. The next round begins as line $[1 + 2S, (K + 2S)\%(K + S)]$ are being convolved and line $[1 + S, 2S]$ are being loaded. Figure 2 (b) illustrates the process in one channel when $K = 3$ and $S = 1$.

4.3 Pipeline Design

Based on the *fusion* architecture, we employ a two-level pipeline design: intra-layer pipeline and inter-layer pipeline, as depicted in Figure 2 (c) and (d).

- Intra-layer. For each layer, it involves three phases: data load, computation, and data store. Our algorithm (section 5) determines the algorithm choice for each layer. After that, we use pipeline to hide the data load and store with computation as shown in Figure 2 (d).
- Inter-layer. When employing *fusion* design, we pipeline the layers that are fused together. Obviously, in the pipeline manner, the pipeline stage length is determined by the longest stage. It becomes more complex when different algorithms can be used for different layers. Our algorithm (section 5) will balance the latency between different layers in the same *fusion* group through resource allocation.

5 ALGORITHM DETAILS

The prior section presents a *fusion* architecture. In this section, we design an optimal algorithm that divides the CNNs into *fusion* groups and determines the implementation algorithm for each layer. The aim of the algorithm is to minimize the end-to-end latency of a given CNN. Since the computation of the given CNN is fixed, minimizing the latency is equivalent to maximizing the throughput. For each algorithm (either conventional or `Winograd`), we also explore its hardware parallelism, corresponding to the number of computing units in Figure 2. Different hardware parallelism leads to different resource usage.

DEFINITION 1. *For layer i, its implementation strategy is a triple $C_i = \langle g_i, algo_i, p_i \rangle$ in the fusion architecture, where g_i, $algo_i$ and p_i specify the fusion group, algorithm, and hardware parallelism for layer i, respectively. Accordingly, a strategy for an N-layer network is defined as a set $S = \{C_i | 1 \le i \le N\}$, representing the structure of fusion design and implementation for every layer.*

PROBLEM 1. *Given the model of an N-layer CNN and resource constraint R, the goal is to find out the optimal strategy S which minimizes the end-to-end latency of the CNN subject to data transfer constraint T.*

On FPGAs, resource constraint R is multi-dimensional including BRAMs, DSP slices and logic cells of the target device. We use T to bound the feature maps transfer only, since *fusion* design does not help to save the kernel weight transfer.

We develop a dynamic programming algorithm to solve Problem 1. Let $L(i, j, t)$ represent the latency of the optimal strategy for layers from i to j, where t is the transfer constraint. As long as t is sufficient for the minimal transfer requirement, we can either unify them as a group or find a sweet spot to split them into two groups. Therefore, we derive the following recursion formula

$$L(i,j,t) = \begin{cases} \min\{ \min\limits_{i \le k < j, x < t} \{L(i,k,x) + L(k+1,j,t-x)\}, \\ \quad fusion[i][j]\} & t \ge min\_t[i][j] \\ \infty & t < min\_t[i][j] \end{cases}$$

where $fusion[i][j]$ represents for the minimal latency of fusing the i_{th} layer to the j_{th} layer as a group under the constraints, $min\_t[i][j]$ refers to the minimal data transfer requirement, namely the sum of input feature map size of the i_{th} layer and output feature map size of the j_{th} layer.

Algorithm 1: optimal algorithm

```
   Input: N, T, R
   Output: S
1  for j = 0; j < N; ++j do
2      for i = j; i ≥ 0; −−i do
3          for t = 0; t < T; ++t do
4              if t < min_t[i][j] then
5                  L[i][j][t] = ∞
6              else
7                  min_latency = fusion[i][j]
8                  k_flag = j
9                  t_flag = t
10                 for k = i; k < j; ++k do
11                     if t < (min_t[i][k] + min_t[k + 1][j]) then
12                         continue
13                     for x = 0; x < t; ++x do
14                         sum_latency = L[i][k][x] + L[k + 1][j][t − x]
15                         if sum_latency < min_latency then
16                             min_latency = sum_latency
17                             k_flag = k
18                             t_flag = x
19                 L[i][j][t] = min_latency
20                 k_mark[i][j][t] = k_flag
21                 t_mark[i][j][t] = t_flag
22 Generate the fused design structure of S based on k_mark and t_mark
23 foreach group in S.structure do
24     update_ipls(group.start, group.end, R)
25 return S
```

Algorithm 1 gives the implementation details. The minimal latency of the input CNN is given by $L[0][N-1][T-1]$. Algorithm 1 first generates the structure of *fusion* design (line 22). Then, for each *fusion* group, we generate the implementation details of each layer (line 23-24).

To derive $fusion[i][j]$ used in Algorithm 1 (line 7), we devise a depth-first based branch-and-bound algorithm as shown in Algorithm 2. Algorithm 2 implements layer i to j as a group under the resource constraint R. Starting from the i_{th} layer, it goes deeper until reaching the j_{th} layer. Once a node in the j_{th} layer is visited, the group latency is updated using the path latency if necessary (line 7-8). Since we employ inter-layer pipeline for the layers within the

Algorithm 2: branch-and-bound algorithm

```
   Input: i, j, R
   Output: group_latency
1  group_latency = UPPER
2  ROOT = new NODE(zero_res_usgae, 0)
3  visit(ROOT, i, i, j)
4  return group_latency
5  Function visit (parent, cnt, start, end)
6      if cnt > end then
7          if parent.lat < group_latency then
8              group_latency = parent.lat
9          return
10     foreach algorithm algo for layer cnt do
11         foreach p from max to min parallelism when using algo for layer cnt do
12             if unvisited[cnt][algo][p] then
13                 ipls[cnt][algo][p] = implement(cnt, algo, p)
14                 unvisited[cnt][algo][p] = false
15             ipl = ipls[cnt][algo][p]
16             if ipl.lat > group_latency then
17                 break
18             if meet_constraints(ipl, parent, R) then
19                 child = new
                     NODE(ipl.res + parent.res, max{ipl.lat, parent.lat})
20                 visit(child, cnt + 1, start, end)
21                 delete child
```

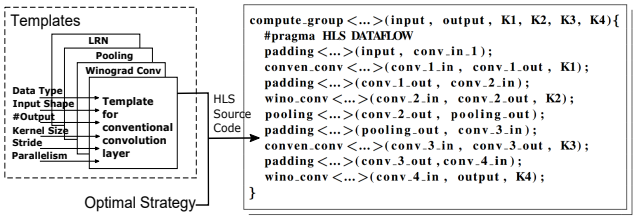

Figure 4: Generate Source Code Using Templates

same group as shown in Figure 2 (c), the path latency is the latency of the slowest layer along the path. We use the current best group latency to bound the following tree traversal (line 16-17). We will only create a new branch if the current path latency is smaller than the group latency. When implementing a layer, our framework explores different algorithms and hardware parallelisms (line 10-11). Different algorithms and parallelisms lead to different resource usage. The *implement* function evaluates the resource requirements and the expected latency of the given algorithm *algo* for the cnt_{th} layer with parallelism p (line 13). If the left resources are sufficient for the implementation, a child node would be generated and explored (line 18-20).

The complexity of Algorithm 1 is $O(N^3T^2)$. The $fusion[i][j]$ array is generated by Algorithm 2 offline.

6 CODE GENERATOR

Given the optimal strategy, the code generator generates HLS source code using templates, as depicted in Figure 4. We design templates for various type of layers including convolution, pooling, and local response normalization (LRN) layers. Moreover, for convolutional layers, we design different templates for conventional and `Winograd` algorithms. When using these templates, several parameters need to be specified such as data type, feature map shapes, kernel size, stride, and parallelism.

For the layers to be fused in a group, we wrap them with a top function as shown in Figure 4. Then, to enable the inter-layer pipeline we add DATAFLOW directive to the top function which allows the data flow through the layers. The memory channels between layers can be implemented as either ping-pong or FIFO buffers depending on the access patterns. Our architecture guarantees that both input and output data for each layer are accessed in sequential order. Thus, the FIFO channels are used. The templates carefully partition line buffers to fully exploit PIPELINE directives and elaborate sub-functions to enable intra-layer pipeline. DATAPACK directives are also used to maximize the bandwidth utilization. For the last step, the code generator employs Vivado tool-chain to compile the source code into bitstream.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Setup

For a given CNN, we apply our algorithm to obtain optimal strategy which directs the code generator. After the HLS source code is generated, we use Vivado HLS (v2016.2) to conduct C simulation and C/RTL co-simulation. Once the implementation has been validated, we employ Vivado SDSoC (v2016.2) to compile the source code into bitstream. To evaluate our framework, we use an off-the-shelf device zynq ZC706 as the experiment platform. ZC706 board is composed of dual ARM Cortex-A9 CPUs, one XC7Z045 FPGA chip, and 1 GB DDR3 memory. It provides a 4.2 GB/s peak memory bandwidth. We set its working frequency as 100 MHz for all designs and use 16-bit fixed data type.

As mentioned above, `Winograd` algorithm can be configured with different tile size. We use $F(4 \times 4, 3 \times 3)$ in this work. Thus, to complete the same amount of computation, our `Winograd` implementation uses one-quarter of the DSPs needed by the conventional algorithm while requiring 4 times higher bandwidth.

When adopting the proposed algorithm, we define the unit of transfer constraint as 10 KB and employ 8 as an upper bound for the number of layers within a *fusion* group due to memory ports limitation. For both case studies, our algorithm returns the optimal solutions within seconds. Very deep CNNs such as GoogleNet are usually based on modules and highly structured. To further improve the efficiency of our algorithm, we can treat every module as a single layer.

7.2 Case Study of VGG

We first compare our framework to the state-of-the-art fusion architecture proposed by [1] using VGG [20]. VGGNet-E consists of 16 convolutional layers, 3 fully connected layers, 3 max-pooling layers and one softmax layer. Alwani et al. [1] choose to fuse the first five convolutional layers and two pooling layers as the feature map transfer is heavy in these layers. For a fair comparison, we fuse these seven layers, too. ReLU layers can be easily integrated into convolutional layers. We implement [1] and our techniques using the same data type. Figure 5 shows the latency comparison under five different feature map transfer constraints.

Under all evaluated constraints, our framework performs consistently better than [1]. We achieve 1.42X-3.85X (on average 1.99X) performance speedup for different transfer constraints. As shown in Figure 5, when the transfer constraint is relaxed, our technique can achieve better performance. Note that without *fusion* architecture, at least 34 MB total feature map transfer is required for these layers. If we use 34 MB as the constraint, each layer forms a group in our algorithm, offering 660 GOPS effective performance*. However, [1] fails to do so as it does not provide the capability to explore the trade-off between performance and memory transfer.

Table 1 gives a detailed comparison when the transfer constraint is set to 2 MB. Our strategy uses a similar amount of resource and power but achieves much better performance compared with [1].

*effective performance = the number of total operations / the total latency.

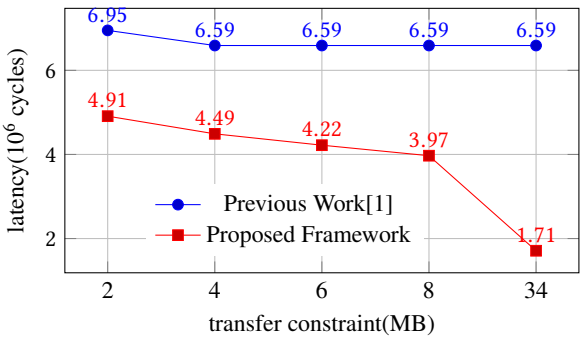

Figure 5: First five convolutional layers latency comparison of VGG between our strategies and [1].

Table 1: Detailed comparison under 2 MB transfer constraint

	Ours	[1]
BRAM18K	909	703
DSP48E	824	784
FF	120,957	90,854
LUT	155,886	118,400
Power(W)	9.4	9.4
Energy Efficiency (GOPS/W)	24.42	17.25

Table 2: Implementation details of AlexNet

Layers	Algorithm	Parallelism	BRAM	DSP	FF	LUT
conv 1	conventional	144	101	144	17,578	31,512
conv 2	`Winograd`	4	104	144	23,688	37,838
conv 3	`Winograd`	2	72	72	12,059	19,629
conv 4	conventional	192	368	192	20,005	27,613
conv 5	`Winograd`	2	112	72	10,923	17,597
other layers			144	101	11,873	14,780
Total			901	725	96,126	148,969
Available			1090	900	437,200	218,600
Utilization (%)			82.7	80.6	22.0	68.1
Latency			1.73×10^6 cycles			

The *fusion* design that we employ helps to decrease the feature map transfer, leading to great energy saving for the memory transfer part. Our *fusion* architecture leads to 94% to 0% (average 68.2%) transfer energy saving for different transfer constraints in Figure 5. Besides, our heterogeneous algorithms exploration improves the performance by 99% on average, leading to another 50% energy saving for the computing part.

7.3 Case Study of AlexNet

AlexNet [13] is composed of five convolutional layers (integrated with ReLU), three pooling layers, two LRN layers and three final fully connected layers. We omit the last three fully connected layers as the FC layers use very small feature map compared with kernel weight [1].

Given a 340KB transfer constraint (the total size of the first layer input feature map and the last layer output feature map), we are able to fuse all the layers into one group. Table 2 gives the implementation details for each layer. For this case, the second, third and fifth convolutional layers are implemented using `Winograd` algorithm, while the other layers are implemented using the conventional algorithm. The DSPs saved by `Winograd` algorithm are exploited by conventional convolutional layers, improving overall performance. In another word, our framework exploits the generality of conventional algorithm and the high performance of `Winograd` algorithm. Compared with [1], our strategy achieves 1.24X speedup due to small exploration space.

8 RELATED WORK

To overcome the computing challenge of CNNs, lots of FPGA-based accelerators have been proposed for better performance or energy-efficiency. Some works bend themselves to building frameworks. [19] develops a virtual machine and hand-optimized templates and [23] builds a component library. Some elaborate on their high-performance convolution PE designs. [2, 3, 21] design PEs which utilize parallelism in different dimensions. [27] proposes an accelerator that serves all convolutional layers owning to uniform unroll factors. Emerging convolutional algorithms also drive new PE designs. For example, [17] introduces an end-to-end accelerator based on `Winograd` algorithm. However, different from this work, [17] mainly focus on the PE design of `Winograd` algorithm. Others try to achieve higher sparsity to enhance energy-efficiency. There exist three main methods towards higher sparsity: connection pruning [9, 10], low-rank decomposing and regularizing [8, 16, 24]. These work are orthogonal to our exploration. Besides, all these work process networks layer by layer. Recently, [1] propose a *fusion* design which fuses the computation of adjacent layers. *Fusion* design reuses intermediate data and decreases feature map transfer.

9 CONCLUSIONS

In this work, we propose a framework that helps in exploring heterogeneous algorithms for accelerating deep CNNs. We first design a line-buffer-based architecture that applies to distinct algorithms and achieves intermediate data reuse naturally. Then we develop a dynamic programming algorithm to find the optimal strategy. Finally, we employ our code generator to implement the strategy. We evaluate our strategies for AlexNet and VGG on Xilinx ZC706 board to show the robustness and efficiency of our framework.

ACKNOWLEDGMENTS

The authors would like to thank Shuo Wang for his valuable suggestions.

REFERENCES

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.
[2] S. Cadambi and et al. A programmable parallel accelerator for learning and classification. In *PACT*, 2010.
[3] S. Chakradhar and et al. A dynamically configurable coprocessor for convolutional neural networks. In *ISCA*, 2010.
[4] T. Chen and et al. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.
[5] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ISCA*, 2016.
[6] J. Cong and et al. High-level synthesis for fpgas: From prototyping to deployment. *TCAD*, 2011.
[7] J. Cong and B. Xiao. Minimizing computation in convolutional neural networks. In *ICANN*, 2014.
[8] Y. Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.
[9] S. Han and et al. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.
[10] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv*, 2015.
[11] S. Ji, W. Xu, M. Yang, and K. Yu. 3D convolutional neural networks for human action recognition. *TPAMI*, 2013.
[12] Y. Jia and et al. Caffe: Convolutional architecture for fast feature embedding. In *MM*, 2014.
[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.
[14] A. Lavin. Fast algorithms for convolutional neural networks. *arXiv*, 2015.
[15] Y. Liang and et al. High-level synthesis: productivity, performance, and software constraints. *IJECE*, 2012.
[16] B. Liu and et al. Sparse convolutional neural networks. In *CVPR*, 2015.
[17] L. Lu, Y. Liang, Q. Xiao, and S. Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. In *FCCM*, 2017.
[18] J. Qiu and et al. Going deeper with embedded FPGA platform for convolutional neural network. In *FPGA*, 2016.
[19] H. Sharma and et al. Dnnweaver: From high-level deep network models to fpga acceleration. In *CogArch*, 2016.
[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.
[21] L. Song and et al. C-Brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization. In *DAC*, 2016.
[22] C. Szegedy and et al. Going deeper with convolutions. In *CVPR*, 2015.
[23] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *DAC*, 2016.
[24] W. Wen and et al. Learning structured sparsity in deep neural networks. In *NIPS*, 2016.
[25] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *CACM*, 2009.
[26] S. Winograd. *Arithmetic complexity of computations*. Siam, 1980.
[27] C. Zhang and et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.