Frequency Improvement of Systolic Array-Based CNNs on FPGAs

Jiaxi Zhang[1,*], Wentai Zhang[1], Guojie Luo[1,3,†], Xuechao Wei[1], Yun Liang[1], and Jason Cong[2]
[1]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, Beijing, China
[2]Computer Science Department, University of California, Los Angeles, CA, USA
[3]Peng Cheng Laboratory, Shenzhen, China
Email: *zhangjiaxi@pku.edu.cn, †gluo@pku.edu.cn

***Abstract*—FPGAs are commercially available off-the-shelf for implementing convolutional neural network (CNN) accelerators to trade off accuracy, performance, and power. Systolic array architecture for CNN accelerators on FPGAs has the potential to run at a high frequency due to its regular and simple interconnections. However, current FPGA CAD tools are unable to synthesize and layout systolic arrays in high quality. In this paper, we identify the reasons for the frequency degradation of systolic array designs for CNN accelerators. We also propose two methods to improve the frequency at the front-end and the back-end, respectively. The experimental results show that our methods are able to achieve $1.29\times$ higher frequency and attain 1.5TOPS for the VGG16 network on the Xilinx KCU1500 platform.**

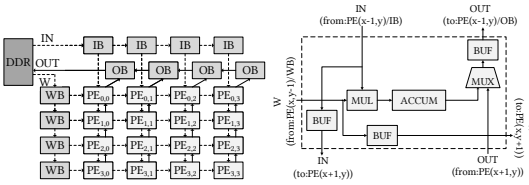
I. Introduction

There are emerging computational demands for convolutional neural networks (CNNs), which are applied for artificial intelligence in image/video classification, recognition, natural language processing, and autonomous driving. Due to the specific computing and memory-access patterns of neural networks, customized computing is a promising approach to achieve high performance and energy efficiency. And FPGA is a commodity off-the-shelf technology to implement customized computing and applies to an increased amount of CNN designs [1], [2], [3], [4], [5], [6].

Throughput is an important performance indicator for FPGA-based CNN designs. It depends on the amount of operations that a design can process in a cycle, as in the following equation:

$$\text{Throughput} = \frac{\#\text{Operations}}{\#\text{Cycles}} \times \text{Frequency}. \quad (1)$$

Thus, increasing the frequency helps improving the throughput. In some early works, researchers exploit the parallelism in computation to utilize the abundant logic resources on FPGAs. Afterwards, many works focus on minimizing data movements for the efficient use of memory bandwidth to achieve high performance. To determine the optimal performance design from a large design space with various computation rates and memory access rates, many works [2], [5], [6], [4] apply the Roofline model and develop their performance analysis and optimization methods to improve the overall throughput. On the one hand, most of previous works aim at increasing the amount of data processed per cycle (the first term on the right-hand side of Equation 1), but few focus on frequency improvement [7] (the second term on the right-hand side of Equation 1). Latte [7] features pipeline transfer controllers in four common communication and computation patterns to improve the frequency with a HLS-based implementation. On the other hand, it has been reported that Intel's latest FPGA architecture runs at a frequency of 460-920MHz [8] for some applications, such as the quantitized matrix multiplication, which reveals the opportunities in frequency improvement to increase the throughput.



(a) Systolic array Schematic (b) PE Structure

Fig. 1: Systolic array for CNN

Systolic array [9] is a homogeneous network, which has the potential to run at a high frequency due to its regular and simple interconnections. Google [10] use it as the core architecture to implement the large-scale matrix processor in TPU running at 700MHz. This architecture has also been applied to FPGA-based CNNs [2], [3], [6]. But we observe that the frequency is much lower than expected when they use the latest FPGA CAD tools. In other words, systolic array has a layout-friendly topology of its data processing elements (PEs), but current FPGA CAD tools are still unable to synthesize and layout systolic arrays in high quality. Therefore, there is plenty of room for frequency improvement to attain a high performance.

In this paper, we examine the 2-D systolic architecture and identify the reasons for frequency degradation. Then we propose two solutions to improve the frequency of systolic array designs for FPGA-based CNN accelerators. Our contributions can be summarized as the following:

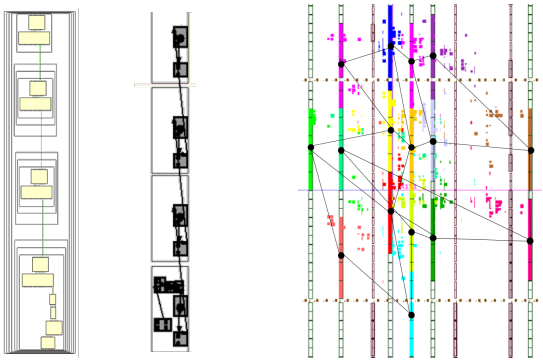
- We locate the critical paths in the implementation and analyze the causes in the FPGA CAD tools that prevent a high-quality systolic array designs for CNN accelerators. FPGA-based accelerations for CNN models are mostly DSP-rich, which results in difficulties of strict hardware resource allocation. Systolic array's rectangular structure does not benefit from the popular half perimeter wire length (HPWL)-oriental CAD tools.
- To increase the design frequency, we propose two techniques at the front-end and back-end of the FPGA CAD flow, respectively. The experimental evaluations demonstrate the effectiveness of these two strategies for performance optimization.

II. Background and Motivation

A. Background

Systolic array [9] is a highly-parallel and layout-friendly architecture due to its simple and regular topology of processing elements (PEs). This architecture has been applied to FPGA-based CNNs [3], [6]. Figure 1 (adopted from [6]) shows the 2-D systolic

978-1-7281-0397-6/19/$31.00 ©2019 IEEE



(a) A long DSP chain (b) A distorted layout

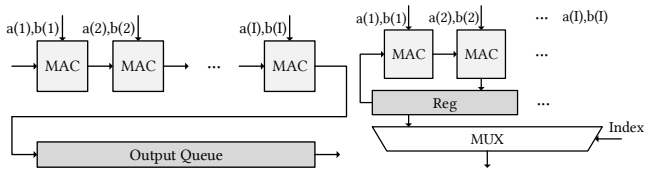
Fig. 2: Motivational Examples

array structure and the PE design for CNNs. For each $PE_{x,y}$, it receives the input features from $PE_{x-1,y}$ or IB (input buffer) as well as the weights from $PE_{x,y-1}$ or WB (weight buffer) and sums up the partial products by an accumulator. It also shifts the input features to $PE_{x+1,y}$ and the weights to $PE_{x,y+1}$. Besides, it sends the results received from $PE_{x+1,y}$ and the ones computed by itself to $PE_{x+1,y}$ or OB (output buffer) using a multiplexer. For detailed information about the computation of the systolic array cycle by cycle, please refer to [6]. This implementation has two main advantages to achieve a high performance. On the one hand, each PE transmits the data horizontally and vertically to its neighboring PEs, and all the PEs process the data flowing through them in parallel. This deeply pipelined structure is suitable for massive parallelism and increases the first term on the right-hand side of Equation (1). On the other hand, only the boundary PEs of the array require communicating with memory. This simple communication structure avoids large fan-out interconnects by using local connections between adjacent PEs, which decreases the global data transfers and offers the possibility for a high frequency and performance.

B. Timing Issues in Systolic Array-based CNN

The simple communication structure of systolic array shows a great potential to solve timing issues and obtain a high frequency. However, existing FPGA CAD tools fail to synthesize and layout this regular structure in good quality, so that the frequency is lower than expected. For example, we implement a systolic design with 4×4 PEs in Xilinx Vivado 2017.2 with the timing optimization options as in the Xilinx Vivado Design Suite User Guide UG904, such as `-hold_fix` and `-fanout_opt`. After examining the layout and the critical paths of this design, we find the implementation has mainly two issues:

- *Long data path caused by the accumulation inside a PE.* Figure 2a shows the schematic and the layout of one typical critical path of the design. This critical path consists of four DSPs, and each DSP performs a multiply-accumulate operation. The cascaded accumulation forms a long data path and lowers the frequency.
- *Distorted layout of the regular systolic array structure.* Figure 2b shows the layout of the systolic array after placement. Different colors highlight different PEs. And we draw the topology of the systolic array on top of the layout. It is obviously that the layout loses the benefits of the regular structure after placement. The PEs are not aligned into an array structure, and the distorted layout worsens the timing issue.



(a) Original DSP chain (b) Segmented DSP chain

Fig. 3: Different implementations

```
PE_MAC(a, b, outputQueue) {
    assign DSP48 as accum
    for (i=0; i<I; i++) {
#pragma HLS UNROLL
        accum += a[i]*b[i];
    }
    push accum to outputQueue
}
```

```
PE_MAC(a, b, outputQueue,
       MAC_reg, current_index) {
   //the number of MAC segments
   NMAC = I/u;
   assign DSP48 as accum_MAC[1..NMAC]
#pragma HLS array_partition \
       variable=accum_tmp complete
   for (i=0; i<NMAC; i++) {
#pragma HLS UNROLL
       accum_MAC[i] = MAC_reg[i];
       for (int j=0; j<u; ++j) {
           accum_MAC[i] += a[i*u+j]*b[i*u+j];
       }
   }
   transfer accum_MAC to MAC_reg,
   and clear MAC_reg[current_index]
   push accum_MAC[current_index] to outputQueue
}
```

(a) Primitive design (b) Segmented design

Fig. 4: Different PE designs of MAC

The detailed reasons of these two timing issues will be discussed in Section III. And we will also propose several techniques for frequency optimization.

III. Frequency Improving Methods

In this section, we present our frequency improvement techniques for systolic array-based neural networks. The front-end method reduces the length of the DSP chains inside a PE, and the back-end method imposes extra floorplanning constraints to avoid the distorted layout across PEs.

A. Front-end method

We first analyze and eliminate the issue for long DSP chains in Section II-B. A PE has a high demand in the computational resources, such as DSPs. DSP resources are not uniformly distributed on an FPGA device but are distributed in columns. Therefore, DSPs in a single PE could be placed across columns or take place in a single column after placement (see Figure 2b). In the typical accumulation implementation, DSPs are organized as an accumulation chain (see Figure 3a). This architecture uses cascaded DSPs, and has a long combinational data path that prevents a high frequency.

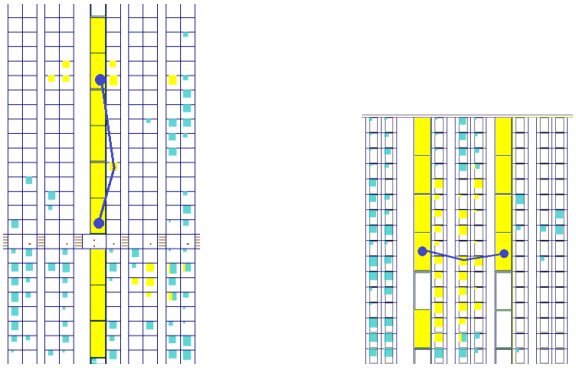
We propose to reduce the length of the accumulation chain in high-level designs at the front-end to resolve this problem. The original DSP accumulation chain computes the following quantity:

$$S_i = \sum_{j=1}^{j \le I} a_{i,j} \times b_{i,j}, \tag{2}$$

where I is the length of the accumulation, $\boldsymbol{a_i}, \boldsymbol{b_i}$ are the i-th input batch, and S_i is the returned result from the output queue. We transform this equation into:

$$S_i = \sum_{j=1}^{I/u} S'_{i,j} = \sum_{j=1}^{I/u} \left(\sum_{k=1+(j-1)\times I/u}^{k<j\times I/u} a_{i,k} \times b_{i,k} \right), \tag{3}$$

where we partition the accumulation chain into several parts $S'_{i,j}$, and u is the segment factor. $S'_{i,j}$ represents the summation of the j-th segmentation of $a_i \times b_i$. We use I/u cycles to complete the summation $S_i = \sum S'_{i,j}$, and every cycle we use MUX to select



(a) Straight line PE (b) Rectangular PE

Fig. 5: Different PE shapes after placement

one segmentation summation to output. The receiver will integrated the result using all the I/u cycles' results. The pseudocode is shown in Figure 4.

This method is applied in the high-level design. High-level synthesis tools now will synthesize the optimized source codes. Our segmented design uses additional resources: MUX for selection and registers for temporary storage. After modification, the data path of the accumulation chain is divided into I/u segments. In general, to maximize the throughput, $u = 1$ would be best choice.

B. Back-end method

The front-end optimization method can solve the long data path issue caused by the accumulation chain inside a PE, as observed in Section II-B. However, we still have issue of distorted layout for the systolic array.

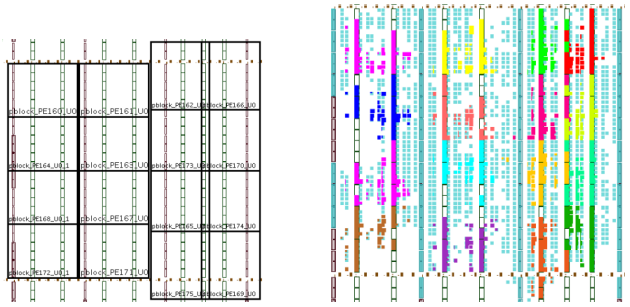
Systolic array is a homogeneous network consisting of many PEs. It has a layout-friendly structure. But in Figure 2b, we can see that the design is out of shape after placement with CAD tools' default options. On the one hand, the convolutional computation requires a large amount of DSPs, and the DSPs are distributed into columns over the whole FPGA chip. The heterogeneous resources affects the placement quality [11]. On the other hand, the total wire length is an important metric, though not the only one, for FPGA placers. We count the average number of internal and external nets of PEs and find out that there are average around 15× more internal nets than the external ones in systolic array design. We conclude that the impact of the external nets is much smaller than the internal ones, and this will result in a distorted layout.

Therefore, we perform floorplanning and set the placement constraints to solve this problem. Placement constraints restrict PEs to fixed locations or regions. Figure 5 shows unconstrained PE and constrained PE. This figure also shows the longest path in the PE, and we can see that constrained PE will have shorter critical path.

We describe the overall idea below to find a topology-aware floorplanning for systolic arrays. Given the resource usage of every PE and their connectivity after logic synthesis, we obtain the shapes and the relative locations of all the PEs after floorplanning.

First, we find a region for all the PEs to place in, and this region should provide sufficient hardware resources. In addition, this region should be as close as possible to the I/O banks that the DDR module uses. This region's location can be obtained through enumeration or greedy search.

Second, we assign PEs based on the DSP columns in the region, because DSPs are distributed into columns on an FPGA



(a) Floorplanning region (b) Layout with floorplanning

Fig. 6: Back-end improvement with floorplanning

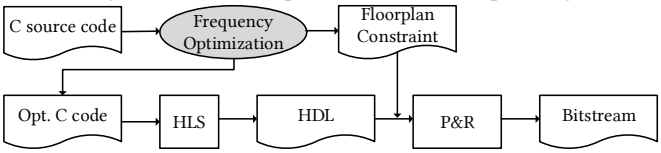


Fig. 7: Execution Flow

chip. Suppose that the systolic array has C columns of PEs and the region has C' DSP columns, the problem becomes how to allocate C columns of PEs onto C' DSP columns. We use a *max_width* variable to restrict maximum PE column's width. For a fixed *max_width*, we can assign DSP columns to PEs from left to right through greedy search. For example, if PE column c occupies DSP columns from DSP column p to q. We can enumerate the PE column $c+1$'s starting point from DSP column $p+1$, and find the minimum t and DSP columns from $p+1$ to t have enough resources for PE columns $c+1$. In addition, we also need to check if overlapped columns are resources-sufficient, such as DSP columns from p to t for PE columns c and $c+1$.

Third, we allocate all the PEs along the row direction. We set a rule that no PEs can overlap along the row direction. Then, we only need to enumerate the row starting point for each PE, and use resource usage to infer each PE's row ending point.

In summary, back-end method uses floorplanning to restrict the shapes and the relative locations of the PEs to maintain a regular array structure and make PE-to-PE nets less critical. In Figure 6, we demonstrate an example of PE assignment with 4×4 PEs. Figure 6a shows the floorplanning constraints, and Figure 6b is the placement result. We can see that floorplanning re-organizes the PEs more regularly compared to Figure 2b.

IV. Experimental Evaluations

A. Automation Flow

We implement an automated flow to generate FPGA bitstreams from a high-level design in C, as shown in Figure 7. We also use (R, C, I) to indicate the structure of PEs. R is the number of PE rows. C is the number of PE columns. I is the size of the input vector for a PE, and it also equals to the number of MAC DSPs. In the frontend, our code wrapper pre-processes the source codes to transform the original PEs into segmented PEs. After we explore the best choice of segment factor u in the design space, the transformed source codes are sent to the FPGA HLS tools. In the backend, we explore different shapes and locations of the PEs to minimize their interconnects and maitain a regular floorplan. The placement constraints generated from our floorplan are sent to the FPGA physical design tools together with the synthesized design.

TABLE I: Comparison to state-of-the-art implementations (Latency: microsecond, Throughput: GOPS)

Impl.	[2]	[4]	[5]	[6]			Ours	
FPGA	Xilinx VC709	Arria10 GX 1150	Arria10 GX 1150	Arria10 GX 1150			Xilinx KCU1500	
CNN	VGG	VGG	VGG	AlexNet	VGG	VGG	AlexNet	VGG
Frequency	150MHz	150MHz	385MHz	239.62MHz	221.65MHz	231.85MHz	290MHz	298MHz
Precision	fixed 16bit	fixed 8-16bit	fixed 16bit	float 32bit	float 32bit	fixed 8-16bit	fixed 8-16bit	fixed 8-16bit
DSP utilization	2833 (78%)	1518 (100%)	2756 (91%)	1290 (85%)	1340 (88%)	1500 (49%)	1386 (26%)	1368 (25%)
BRAM utilization	1248 (42%)	1900 (70%)	1450 (54%)	2360 (86%)	2455 (90%)	1668 (61%)	1692 (78%)	1634 (76%)
Latency	65.13	47.97	17.18	4.05	54.12	26.85	2.22	21.04
Throughput	354	645.25	1790	360.4	460.5	1171.3	830	1495

TABLE II: Frequency and resource utilization

Configurations	Freq.	BRAM	DSP	FF	LUT
(11,14,8),baseline	193MHz	78%	26%	23%	38%
(8,19,8),baseline	198MHz	76%	25%	22%	36%
(11,14,8),$u = 2$	241Mhz	78%	26%	25%	41%
(8,19,8),$u = 2$	247Mhz	76%	25%	25%	40%
(11,14,8),$u = 1$	290MHz	78%	26%	26%	43%
(8,19,8),$u = 1$	298MHz	76%	25%	26%	43%

B. Environment Setup and Experimental Results

In our experiments, we use Xilinx's FPGA CAD tools, including Xilinx Vivado 2017.2, Xilinx HLS 2017.2, and Xilinx SDx 2017.2, all with default optimization options. For higher frequency, we create microblaze- and DDR4-based projects in Xilinx Vivado, using IPs generated by Xilinx HLS. We use Xilinx Kintex UltraScale FPGA KCU1500 Evaluation Kit as the evaluation board. We implement widely-used AlexNet and VGG16 networks. We use 8-bit fixed data type for weights and 16-bit fixed data type for pixels.

We mainly compare our results with [6] to prove our frequency optimization's effectiveness. We maintain the configurations used in for AlexNet and VGG, which are $(R = 11, C = 14, I = 8)$ and $(R = 8, C = 19, I = 8)$, to control the variables. We first implement the un-optimized baseline version on our Xilinx KCU1500 board and apply our optimization to verify the frequency. This comparison can show the effect of our optimization directly. In our implementation, buffer size is determined by the real required memory bandwidth. The comparison results are listed in Table II. Frequency is improved by 50% on average, using Xilinx KCU1500 board. We only list the optimization designs with $u = 1$ because of sufficient resources. The overall comparison of CNN designs are demonstrated in Table I.

The overall comparison shows that our optimization improves the frequency by 28.5% on VGG case, which directly increases the throughput by 27.6% and reduce the latency by 21.6%. The latency here means the time need for processing once image. The design in [5] is implemented by low-level HDL (System Verilog). This ensures high frequency, but makes it depend on specific CNN models, and hard to be reused again.

The different of improvement on frequency in Table II and I is because the implementations are on different FPGA boards. Arria10 is manufactured by Intel using 20nm technology, and KCU1500 is produced by Xilinx using 20nm technology as well. However, the DSP units of Arria10 and KCU1500 may have different computational power.

V. Conclusion

In this paper, we analyze the causes in the FPGA CAD tools that prevent a high-quality systolic array designs for CNN accelerators. We also analyze how frequency optimization affects the attainable performance and point out that frequency is still effective for performance improvement even when a design reaches the memory bound. We propose two effective techniques at the front-end and the back-end of the FPGA CAD flow to improve the frequency. Evaluation results show that our methods can improve the frequency by 1.29× and attain 1.5TOP/S on the Xilinx KCU1500 platform.

Acknowledgment

This work is partly supported by Beijing Municipal Science and Technology Program under Grant No. Z181100008918015, Beijing Natural Science Foundation under Grant No. L172004, and National Natural Science Foundation of China (NSFC) under Grant 61520106004.

References

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.

[2] C. Zhang, Z. Fang, P. Pan, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*, 2016.

[3] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on Arria 10," in *FPGA*, 2017.

[4] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *FPGA*, 2017.

[5] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *FPGA*, 2017.

[6] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on FPGAs," in *DAC*, 2017.

[7] J. Cong, P. Wei, C. H. Yu, and P. Zhou, "Latte: Locality aware transformation for high-level synthesis," in *FCCM*, 2018.

[8] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. Moss, and S. Subhaschandra, "Can FPGAs beat GPUs in accelerating next-generation deep eural networks?" in *FPGA*, 2017.

[9] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," *Proc Sparse Matrix Conf*, pp. 256–282, 1978.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[11] C. Xu, W. Zhang, and G. Luo, "Analyzing the impact of heterogeneous blocks on FPGA placement quality," in *FPT*, 2014.