A Resource-Sharing & Pipelined Design Scheme for Dynamic Deployment of CNNs on FPGAs

Han-Chen Ye, Geng-Sheng Chen*

State Key Laboratory of ASIC and System, Fudan University, No.825 Zhangheng Road, Shanghai, 201203, China
* Email: gschen@fudan.edu.cn

Abstract
In this paper, we present a new design scheme for dynamic deployment of CNNs on FPGAs, to adaptively implement large CNN models on resource limited FPGAs while keeping low latency and high performance. We first propose a FPGA-based pipeline model named Resource-Sharing Pipeline in the design scheme. The novel pipeline model dynamically reconfigures pipeline stages to FPGA so that FPGA resources are shared by multiple pipeline stages at the different time. The computing parallelism of convolutional layers and the memory access efficiency are also optimized in the design scheme for performance and latency enhancement. In experiment we map all convolutional layers of VGG-16 to Xilinx VC709 platform using our new design scheme. The results show that the implementation reaches a performance of a 187.0ms latency and an 820.8GOP/s throughput under 100MHz clock frequency, achieving a promising promotion over previous works.

1. Introduction

Convolutional Neural Networks (CNN) are achieving great success in lots of Artificial Intelligence (AI) applications including image classification, speech recognition and etc. Though GPU is still the most popular device to accelerate CNNs, FPGA appears to be a more promising alternative because of its flexibility and power efficiency.

There are already many researches on FPGA-based hardware accelerators. But limited by FPGA resources, most of them had to implement a CNN model layer by layer on FPGA, and made them compute sequentially [1-4] or in a multi-FPGAs pipeline manner [5]. Other works devoted to map all layers of a CNN model on one FPGA chip and made them compute in a conventional pipeline manner [7, 8]. But also due to the limitation of FPGA resources, they could just deploy small CNN models (e.g. AlexNet), and meanwhile had to reduce the parallelism of each layer which resulted in high overall latency.

This paper proposes a novel design scheme for dynamic deployment of CNNs on FPGAs. The new design scheme provides with capability of mapping large CNN models (e.g. VGG-16), meanwhile minimizing the overall latency. This paper makes the following main contributions:

- We propose a novel resource-sharing pipeline model which allows for mapping large CNNs with each pipeline stage performing with a high parallelism.
- We use a specialized convolutional layer design to optimize intra-layer latency and throughput. Moreover, we propose a novel memory structure to enable all pipeline stages efficiently access memory.

The rest of this paper is organized as follows: Section 2 presents the details of our design scheme. Section 3 gives the experimental results. Section 4 makes the conclusion.

2. Design Scheme

2.1 Resource-Sharing Pipeline

Dynamic Partial Reconfiguration (DPR) is an FPGA technology which allows for the dynamic update of user logics in Partial Reconfigurable Blocks (PRB) while other logics on FPGA stay active. In this paper, we use its inherent resource-sharing mechanism to accelerate CNN's calculations.

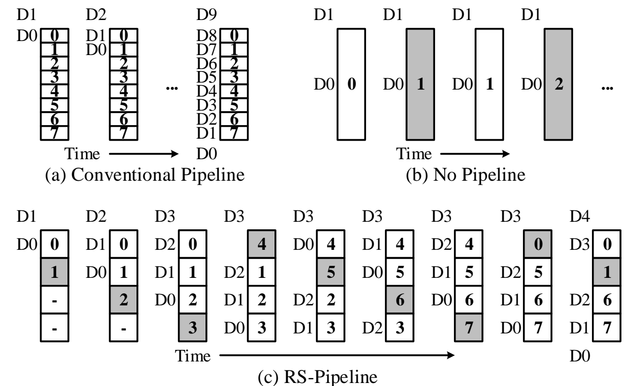

Figure 1. Resource-Sharing Pipeline

Figure 1 illustrates the using and non-using Resource-Sharing Pipeline (RS-Pipeline) methods to map the same 8-pipeline-stage model on FPGA for a comparison. Conventional pipeline showed in Figure 1(a) must map all the 8 pipeline stages on FPGA which results in low parallelism of each stage and high overall latency. Figure 1(b) shows the non-pipelined design method used in most previous CNN accelerators. This method could achieve higher overall performance than other 2 methods but cannot map all the pipeline stages at one time.

Figure 1(c) shows how our 4-stage RS-Pipeline works in this case. The 4-stage RS-Pipeline directly maps 8 pipeline stages to 4 PRBs on FPGA, where stage0 and

stage4 are mapped to PRB0, and so on. Each column in Figure 1(c) presents the status of PRBs at that moment, where white or grey indicates PRBs are in computing or reconfiguring status. At the beginning, RS-Pipeline transfers data D0 to PRB0 and computes stage0 of D0, meanwhile reconfigures stage1 into PRB1. In the next cycle, RS-Pipeline transfers D0 to PRB1 and D1 to PRB0, meanwhile reconfigures stage2 into PRB2. RS-Pipeline then computes and reconfigures in this manner until finishing all data. Hence, there are always 3 PRBs in computing status and 1 PRB in reconfiguring status on FPGA. Note that one pending data is halted in every 3 cycles (e.g. D3) until stage0 is reconfigured into PRB0.

In this way, RS-Pipeline is able to fully utilize PRB's resources sharing between multiple pipeline stages with each stage working in a high parallelism. Assuming that the latency is inversely proportional to the resources utilization with factor f. If we implement an N-stage pipeline using M-stage RS-Pipeline, thus 1 pipeline stage will utilize $1/M$ FPGA resources and have a $f \cdot M$ latency. Then the overall latency is reduced from $f \cdot N^2$ to $f \cdot MN$. Meanwhile, in RS-Pipeline, there is always 1 PRB kept in reconfiguring status while other $(M-1)$ PRBs are in computing status, which ideally will bring about a $1/M$ loss in the overall performance.

2.2 Design of Data and Control Flow

Figure 2 shows a typical design architecture on FPGA. Here, PRC (Partial Reconfiguration Controller) is responsible for loading bitstream files from DDR and reconfiguring them to target PRBs. For the data flow, host CPU downloads original data to DDR, then computing modules in PRBs fetch data from DDR and write back results after computing. For the control flow, a MicroBlaze MCU is used to handle the complicated process of RS-Pipeline. The use of MCU reduces the frequency of high-latency communication between host CPU and FPGA.

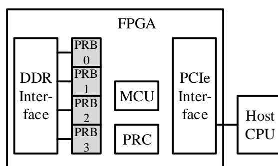

Figure 2. Design Architecture

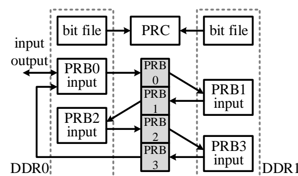

Figure 3. Memory Structure Design

2.3 Memory Structure Design

Memory bandwidth and DSPs are usually the bottleneck of overall performance in CNN accelerators. In our design scheme, both computing modules and PRC need to occupy DDR bandwidth, which makes a high-efficiency memory structure more important.

We propose a novel memory structure design as showed in Figure 3. Computing modules in even PRBs always fetch original data from DDR0 and write results to DDR1 while odd PRBs work inversely. Double-buffer technology is used in each DDR to avoid data confliction. What's more, bitstream files of even PRBs and odd PRBs are stored respectively in DDR0 and DDR1 so that PRC can always use the unoccupied bandwidth of the reconfiguring PRB.

2.4 Convolutional Layer Design

To balance the latency between reconfiguring and computing manipulations, an optimized convolutional layer design is used. We unroll the input channels loop by PI and output channels loop by PO. The minimized processing unit (PE) is therefore duplicated by $PI \times PO$ times. The input buffer, output buffer, weight buffer is accordingly divided into PI, PO, $PI \times PO$ dimensions to feed data to PEs. For all buffers in the design, we use double-buffer technology to ensure that the memory read/write and the PE's calculation are able to process concurrently. Benefited from using this design structure, we are able to adaptively scale PI and PO to optimize the parallelism of each convolutional layer.

3. Experiments and Analysis

To validate the latency and performance of our design scheme, VGG-16 is used as our sample target. We use a 2-D Winograd algorithm, a fast convolution algorithm widely used in digital signal processing applications [9], to implement convolutional layers of VGG-16. We use a Xilinx VC709 FPGA board for our experiments, which has 3600 DSPs, Gen2 x8 PCI-e and two 2GB DDR3. To measure the runtime power, we plugged a power meter in the FPGA platform.

3.1 VGG-16 Model Analysis

Our design scheme has 3 major parameters: M, PI, PO. We use VGG-16's CONV3-1 (the first convolutional layer of CONV3) as the sample to estimate and obtain these 3 optimized parameters for VGG-16 mapping. We scale PI and PO to adjust the parallelism of the layer, then measure computing latency, reconfiguring latency, and memory bandwidth along with parallelism under 100MHz clock frequency.

Figure 4 shows the result of the experiment. With the parallelism increasing, the computing latency decrease and the reconfiguring latency increase. In the case when $PI = 4$ and $PO = 2$, the computing latency and the reconfiguring latency are both about 9000us and memory bandwidth occupation for one DDR is about 13%. Thus, we use $PI = 4$, $PO = 2$ and $M = 6$ for VGG-16.

In the case when $PI = 8$ and $PO = 4$, the layer is too large to be placed into one PRB, so in this case we tune down the clock frequency to 85MHz. In the case when $PI = 2$ and $PO = 2$, reconfiguring latency is similar to

$PI = 4$ and $PO = 2$ because LUTs utilization does not significantly decrease, this overhead also makes $PI = 4$ and $PO = 2$ an optimal choice for VGG-16.

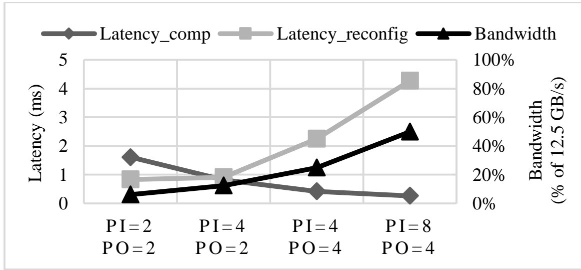

Figure 4. Parallelism vs. Latency and Bandwidth

3.2 VGG-16 Case Study

We implement all convolutional layers of VGG-16 on Xilinx VC709 platform. We divide all convolutional layers except CONV1-1 into 18 pipeline stages because CONV1-1's input feature size is too small to be pipelined. The layout of the last 6 pipeline stages is showed in Figure 5(a). The layout of static circuits (contains PCI-e, DDR interface, MCU, PRC and etc.) is showed in Figure 5(b), where the dark circuits have been locked to keep static in the place and route process.

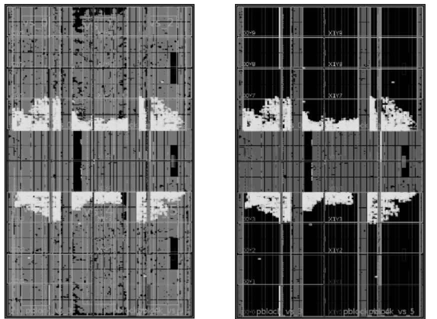
(a) Layout of Last 6 Pipeline Stages (b) Layout of the Locked Static Circuits

Figure 5. FPGA Layout of VGG-16

Table 1 shows the FPGA resource utilization, where Config0, Config1, Config2 stand for the first, the second and the last 6 pipeline stages.

Table 1. FPGA Resource Utilization

Resource	LUTs	Registers	BRAMs	DSPs
Static	89786 (21%)	88305 (10%)	299 (20%)	36 (1%)
Config0	297507 (69%)	144697 (17%)	1283 (87%)	1764 (49%)
Config1	295269 (69%)	144391 (17%)	1283 (87%)	1764 (49%)
Config2	300680 (70%)	150222 (17%)	1271 (86%)	1764 (49%)

Table 2 shows that the performance and latency of our work are out of most of previous works except a highly customized design [6]. Paper [4] reported a lower latency than us because it reused the convolution engine on chip to compute multiple convolutional layers which is a coarse-grain reconfigurable computing method and cannot be applied to other models. Our implementation reaches a 187.0ms latency and an 820.8GOP/s throughput under 100MHz frequency, achieves a promising promotion over previous works.

Table 2. Comparison with Previous Works for VGG-16

	FCCM 2017 [6]	FPGA 2017 [4]	FPGA 2016 [2]	ISLPED 2016 [5]	This Work
Platform	Xilinx ZCU102	Arria-10 GX 1150	Altera Stratix-V GSD8	Xilinx VC709	Xilinx VC709
DSPs	2520	1518	1963*	3600*	1764
Freq (MHz)	200	150	120	150	100
Precision	16b fixed	8-16b fixed	8-16b fixed	16b fixed	16b fixed
Throughput (GOP/s)	2940.7	645.25	117.8	290	820.8
Latency (ms)	-	47.97	262.9	213.6	187.0
Power (W)	23.6	-	25.8	35	27.1
DSPs Efficiency (GOP/s/DSPs)	1.16	0.425	0.06	0.08	0.465
Energy Efficiency (GOP/s/W)	124.6	-	4.57	8.29	30.29

4. Conclusion

In this paper, we present a new design scheme for dynamic deployment of CNNs on FPGAs, which helps to adaptively deploy large CNN models on resource limited FPGAs while keeping low latency and high performance. The results show that the implementation of VGG-16 by using our new design scheme has successfully reached a performance of a 187.0ms latency and an 820.8GOP/s throughput under 100MHz clock frequency, achieving a promising promotion over previous works.

References

[1] J. Qiu, et al., International Symposium on Field-Programmable Gate Arrays (FPGA), p. 26-35 (2016).
[2] N. Suda, et al., International Symposium on Field-Programmable Gate Arrays (FPGA), p. 16-25 (2016).
[3] C. Zhang, et al., International Conference on Computer-Aided Design (ICCAD), pp. 1-8 (2016).
[4] Y. Ma, et al., International Symposium on Field-Programmable Gate Arrays (FPGA), p. 45-54 (2017).
[5] C. Zhang, et al., International Symposium on Low Power Electronics and Design (ISLPED), p. 326-331 (2016).
[6] L. Lu, et al., International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 101-108 (2017).
[7] C. Huang, et al., International Conference on ASIC (ASICON), pp. 1037-1040 (2017).
[8] H. Li, et al., International Conference on Field Programmable Logic and Applications (FPL), pp. 1-9 (2016).
[9] A. Lavin and S. Gray, arXiv: 1509.09308v2, (2015).