Optimizing OpenCL Implementation of Deep Convolutional Neural Network on FPGA

Yuran Qiao, Junzhong Shen, Dafei Huang, Qianming Yang, Mei Wen, Chunyuan Zhang

▶ **To cite this version:**

Yuran Qiao, Junzhong Shen, Dafei Huang, Qianming Yang, Mei Wen, et al.. Optimizing OpenCL Implementation of Deep Convolutional Neural Network on FPGA. 14th IFIP International Conference on Network and Parallel Computing (NPC), Oct 2017, Hefei, China. pp.100-111, 10.1007/978-3-319-68210-5_9 . hal-01705448

HAL Id: hal-01705448
https://inria.hal.science/hal-01705448v1

Submitted on 9 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Optimizing OpenCL Implementation of Deep Convolutional Neural Network on FPGA

Yuran Qiao, Junzhong Shen, Dafei Huang, Qianming Yang, Mei Wen, and Chunyuan Zhang

College of Computer, National Key Laboratory of Parallel and Distributed Processing
National University of Defense Technology
Changsha, Hunan, P.R. China
Email:qiaoyuran@nudt.edu.cn

Abstract. Nowadays, the rapid growth of data across the Internet has provided sufficient labeled data to train deep structured artificial neural networks. While deeper structured networks bring about significant precision gains in many applications, they also pose an urgent demand for higher computation capacity at the expense of power consumption. To this end, various FPGA based deep neural network accelerators are proposed for higher performance and lower energy consumption. However, as a dilemma, the development cycle of FPGA application is much longer than that of CPU and GPU. Although FPGA vendors such as Altera and Xilinx have released OpenCL framework to ease the programming, tuning the OpenCL codes for desirable performance on FPGAs is still challenging. In this paper, we look into the OpenCL implementation of Convolutional Neural Network (CNN) on FPGA. By analysing the execution manners of a CPU/GPU oriented verison on FPGA, we find out the causes of performance difference between FPGA and CPU/GPU and locate the performance bottlenecks. According to our analysis, we put forward a corresponding optimization method focusing on external memory transfers. We implement a prototype system on an Altera Stratix V A7 FPGA, which brings a considerable 4.76x speed up to the original version. To the best of our knowledge, this implementation outperforms most of the previous OpenCL implementations on FPGA by a large margin.

1 Introduction

Deep Convolutional Neural Networks (CNNs) bring about significant precision gains in many fields of computer vision, such as image classification, object detection and object tracing. While deeper structures produce higher accuracy, they demand more computing resource than what today's CPUs can provide. As a result, in the present situation, graphics processing units (GPUs) become the mainstream platform for implementing CNNs [1]. However, GPUs are power-hungry and inefficient in using computational resources. For example, the CNN version based on vendor recommended cuCNN lib can only achieve about 1/3

of the peak performance of GPU [1]. Hardware accelerators offer an alternative path towards significant boost in both performance and energy efficiency.

Usually, hardware accelerators are based on ASIC [2] or FPGA [3, 4]. ASIC based accelerators provide the highest performance and energy efficiency but have to endure huge development cost. Owing to the reconfigurable nature, FPGA based accelerators are more economical considering the development expenses.

For years, FPGA developers suffer from the hard-to-use RTL (Register Transfer Level) programming languages such as VHDL and Verilog HDL. It makes programmability a major issue of FPGA. Thus, FPGA vendors begin to provide high-level synthesis (HLS) tools such as OpenCL framework [5] to enable programming FPGAs using high level languages.

Although developers can easily port codes originally designed for CPUs/GPUs to FPGAs with OpenCL framework, it is still challenging to make the OpenCL codes execute efficiently on FPGAs. The same code may exhibit different performance on different platforms due to the different architecture-related execution manners. Therefore, developers should consider the FPGA architecture when optimizing the OpenCL code.

In this paper, we make a deep investigation on how to optimize the OpenCL code on FPGA platforms. A CNN accelerator implemented in OpenCL is proposed which achieves the state of the art performance and performance density. The key contributions of our work are summarized as follows:

- We make a detailed analysis of running a CPU/GPU oriented OpenCL code of CNN on an FPGA. We explore the memory access behavior of the OpenCL FPGA implementation and point out the bottleneck of the code.
- According to the analysis result, we propose an optimized OpenCL implementation of CNN accelerator, focusing on efficient external memory access.
- We implement our design on an Altera Stratix V FPGA. A performance of 137 Gop/s is achieved using 16-bit fixed point data type. Compared with the original version, it achieves a speed up of 4.76X. To the best of our knowledge, this implementation outperforms most of the previous OpenCL based FPGA CNN accelerator.

The rest of this paper is organized as follows: Section 2 discusses the background of CNN and OpenCL framework for FPGA. Section 3 presents the performance analysis of the baseline code. The implementation details of our optimized design are demonstrated in Section 4. Section 5 provides the experiment result and Section 6 concludes the paper.

2 Background

2.1 Convolutional Neural Network

CNN is a trainable architecture inspired by the research findings in neuroscience. As Fig. 1 shows, a typical CNN structure consists of several feature extractor

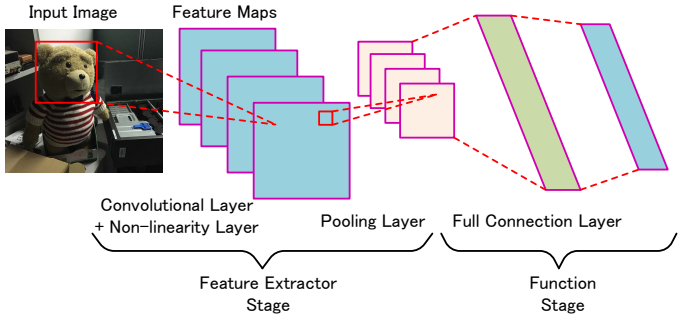

Fig. 1. A typical convolutional neural network structure

stages and a function stage. The feature extractor stages extract the input image's features and send them to the function stage. The function stage may be a classifier, a localizer or other function units customized by developers. According to the features, the function stage calculates the final result. A typical feature extractor stage consists of three layers: a convolutional layer, a non-linearity layer and a pooling layer. A function stage always consists of several full connection layers.

$$Y_r = bias + \sum_{q=0}^{Q-1} conv < X_q, K_{r,q} > \tag{1}$$

$$\begin{aligned} &conv < X_q, K_{r,q} > [m][n] \\ &= \sum_{w=0}^{ks-1} \sum_{l=0}^{ks-1} K_{r,q}[w][l] * X_q[m*s+w][n*s+l] \end{aligned} \tag{2}$$

Usually, convolutional layers generate more than 90% of the computational workload of a CNN model [6]. For a convolutional layer, Q input feature maps $X_0 \ldots X_{Q-1}$ are convolved with R*Q convolutional kernels $K_{r,q}$ (r=0,1...R-1, q=0,1...Q-1) to get R output feature maps $Y_0 \ldots Y_{R-1}$. Equations (1) (2) show the procedure. $bias$ is a value that is added to each pixel of Y_r. $conv < X_q, K_{r,q} >$ refers to the convolution between input feature map X_q and convolutional kernel $K_{r,q}$. ks is the size of the convolutional kernel and s denotes the stride that the convolutional window slides with each time. As the length limit, in this paper we only focus on the convolutional layers.

2.2 OpenCL Framework for FPGAs

OpenCL is an open standard for cross-platform parallel programming. As a high level synthesis (HLS) tool, the OpenCL framework for FPGA enables synthesizing designs described by a C-like language. It greatly improves the development productivity of FPGA. The OpenCL designs for CPU/GPU can also be easily

ported to FPGA with little efforts. The OpenCL for FPGA liberates the developers from the burden of complicated periphery circuits design (e.g. PCIe, DDR, SerDes). The details of periphery circuits are transparent to the developers thus they can concentrate on the designing of kernel logics.

The hardware infrastructure of the OpenCL framework consists of two parts, an FPGA accelerator and a host computer. The OpenCL logic in the FPGA accelerator exists as an SoC. It consists of at least a global memory controller, a link controller to the host computer and a reconfigurable fabric. Developers use HLS tools to synthesize the OpenCL codes into kernel logics and program them to the reconfigurable fabric part. The host computer communicates with the FPGA accelerator through the host-accelerator link. As a common workflow, the host computer first offloads the data to the global memory of the FPGA accelerator. Then it starts the kernel logics to process these data. At last, the host computer gets the result back. In this paper, we use an Altera FPGA development kit to build our CNN accelerator. In particular, the global memory controller is a DDR3 controller, the link controller is a PCIe controller and the host computer is a desktop PC based on x86 architecture.

However, making the OpenCL code execute efficiently on an FPGA is not easy. It requires the awareness of many details about the OpenCL framework for FPGA.

3 Performance analysis of the baseline CNN OpenCL implementation

In this section, we start with a CPU/GPU oriented OpenCL implementation of CNN provided by AMD Research [7]. We will consider it as a baseline version.

For convolutional layers, the baseline code first converts the convolutions to a matrix multiplication to utilize the efficient BLAS (Basic Linear Algebra Subprograms) library for CPU/GPU. As the computational workload of full connection layers is also matrix multiplication, this method simplifies the accelerator design.

Fig. 2 shows the procedure of converting the convolutions to a matrix multiplication. Since a two-dimensional matrix is stored as an array physically, both the convolutional kernels and the output feature maps keep their data structures unchanged. Only the input feature maps need to be reorganized as the Map_matrix. Thus the computation of a convolutional layer can be divided into two parts: reorganizing the input feature maps to Map_matrix and calculating the matrix multiplication.

In the first step, there is no arithmetic operation. The main factor that determines the execution time is the memory access. As Fig. 3 shows, the baseline version divides the Map_matrix into several column vectors, each of which consists of ks^2 pixels in a convolutional window. Each work_item loads these ks^2 pixels form a input feature map and stores them back to the Map_matrix as a column vector. We can see that the memory access of each work_item to the input feature maps is in a sliding window pattern. The memory access pattern to the Map_matrix is in per-column manner.

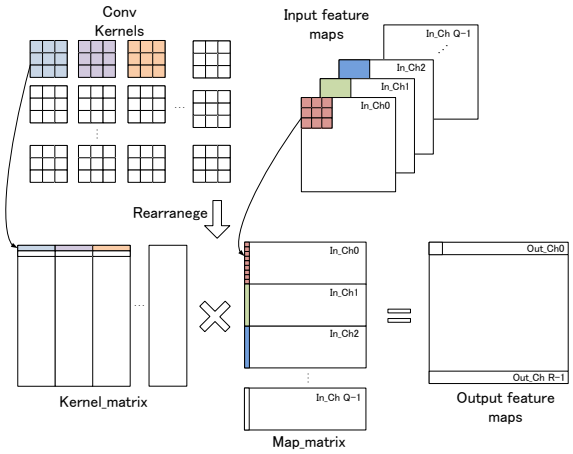

Fig. 2. The procedure of converting convolutions to matrix multiplications

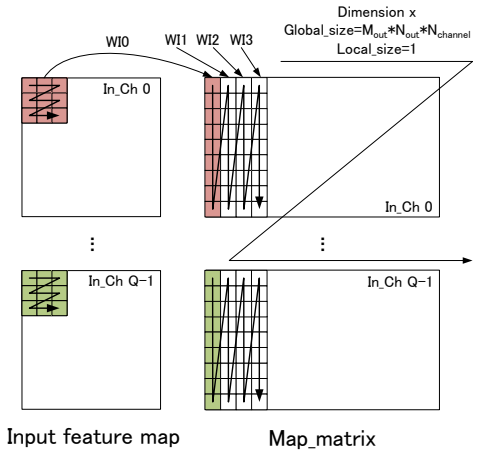

Fig. 3. The memory access pattern of the baseline version

For a DDR 3 system, the time consumption of memory access is determined by the number of memory transactions and the physical bandwidth of external memory. Each memory transaction consists of two phases: prepare phase and burst phase. The burst phase consists of several memory transmits. Assuming that the kernel has M transactions and each transactions has a burst phase with N transmits ($burst\_length = N$), the time of memory access of a kernel is:

$$T_{mem} = \sum_{j=1}^{M} T_{transaction\_j} \tag{3}$$

$$T_{transaction\_j} = T_{prepare\_j} + T_{burst\_j} \tag{4}$$

$$T_{burst\_j} = \sum_{i=1}^{N} T_{transmit\_ij} \tag{5}$$

The transmits in the same burst access memory with continuous physical addresses. Moreover, if the memory access addresses are continuous, multiple transmits can be coalesced into a single wide transmit. Also, multiple bursts can be coalesced into one longer burst. By coalescing the transmits and bursts, we can reduce the proportion of prepare phase in the the time of memory access and improve the utilization rate of memory bandwidth. Thus, continuous memory access is an essential factor for better performance.

From Fig. 3, we can see that for a single work_item, the length of the longest continuous memory access to the input feature maps is ks. However, this length cannot dominate $T_{transaction}$, and it is $T_{prepare}$ that will form most of the memory access time. In the output feature maps, there is no address-continuous memory access. Thus the single work_item can not make the best use of the memory bandwidth. Furthermore there are overlapped and adjacent data accesses between adjacent work_items. In a CPU/GPU platform, such data locality can be exploited by the cache hierarchy and multiple memory access can be merged. However, in a typical FPGA system, there is no ready-made cache to use. The OpenCL compiler can coalesce address continuous memory accesses from adjacent work_items automatically if the addresses are regular. To make things worse, the computation of addresses in the baseline version is very complicated, thus today's HLS tools can not recognize the data locality between adjacent work_items. For the reasons above, the memory access must be optimized.

The second step is a matrix multiplication. Most implementations of matrix multiplication in GPU/CPU use BLAS library to achieve a efficient execution. However, there is no BLAS library for FPGA using OpenCL so far. Therefore, we adopt the efficient FPGA OpenCL implementation from [8] to construct the baseline version. Such implementation has been well optimized for general-purpose matrix multiplication. However, the data precision of the baseline version is 32-bit floating point, but for CNN, it is redundant [6]. In the FPGA-specific design, low data precision may save DSP and logic resources.

4 Optimizing the OpenCL design of CNN accelerator on FPGA

In the last section, we discussed the baseline version and pointed out that the memory access pattern of the baseline version will cause low utilization rate of memory bandwidth. In this section, we will analyze the character of the algorithm and optimized the OpenCL implementation.

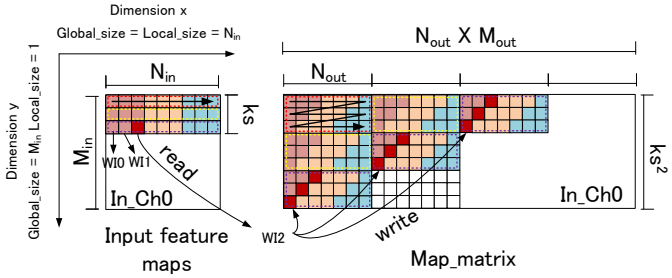

Fig. 4. The data locality in reorganizing input feature maps to map matrix and the task organization of work_items

Fig. 4 shows the relationship between input feature maps and Map_matrix. The coloured part of Map_matrix is corresponding to ks rows in input feature maps. Obviously, these rows are stored in external memory continuously. Compared to accessing these elements repeatedly, a better choice is to prefetch them to an on-chip buffer at first in an address continuous manner.

An optimized method is also shown in Fig. 4. Each work_group first reads a whole row of a input feature map and then write them in the format of the Map_matrix. For the prefetch part, the longest continuous memory access to the input feature maps in a work_group is the width of a input map (N_{in}). It is much longer than the baseline version. For the writing back part, the memory accesses between adjacent work_items are address continuous. The start addresses of the rows and the blocks can be calculated as a shared value among work_items in a same work_group. We use the local_id as the address offset so that the compiler can easily recognize the locality, thus the memory accesses in adjacent work_items can be coalesced automatically.

One pixel in the input feature maps is related to at most ks^2 elements in Map_matrix. Thus the efficiency of writing back dominates the memory access performance. We notice that in the optimized method above (OPT1 for short), the max length of continuous memory access to Map_matrix is N_{out} (the width of the output feature maps). Generally speaking, the map sizes of first several layers are large, so the burst length is large enough to make full use of the memory bandwidth. But in the last several layers, the map sizes are always small, the expense of the prepare phase becomes prominent. We can further optimize the kernel (OPT2 for short) by increasing the length of the continuous memory access.

Fig. 5 shows the further optimization of the memory access. The Map_matrix is transposed so that the pixels in a convolutional window are address continuous. The convolutional windows with the same location in adjacent channels are also address continuous. Thus we can merge $ks^2 * Mc$ pixels into one memory transaction. Mc refers to the number of adjacent channels that can be merged. The transposed Map_matrix is divided to a 2-dimension grid. $Q^*(M_{out}/Mc)$ work_items are used for the reorganizing task. Every work_item first loads ks

rows from each of the Mc input feature maps to the on-chip buffer. In the example, the Mc is 2, the ks is 3. Then each work_item writes $ks^2 * Mc * N_{out}$ pixels into the transposed Map_matrix. In Fig. 5, The pixels in one dotted box are processed by one work_item $WI(y, x)$ (x = 0, 1, ..., $M_{out} - 1$. y = 0, 1, ..., $Q/Mc - 1$).

As for the input data, the total data size of OPT2 is ks times comparing to OPT1. However, the kernel performance will not degrade heavily for two reasons. Firstly, the writing back part occupies the most time of memory access. Although we prefetch more data, the overhead is still ignorable. Secondly, the prefetch data are address continuous. The memory transactions can be coalesced. Thus the impact of the data size is not obvious.

For the output data, the maximum length of continuous memory access is Mc*ks^2. The total data size of OPT2 is the same as OPT1. Thus when Mc*ks^2 is larger than N_{out}, the kernel performance will be improved.

As the baseline version of the matrix multiplication part is already an FPGA oriented code, only minor modifications are needed to adapt it to our design. The size of matrix block of the baseline version is fixed. So we add some branch statements to enable variable matrix block size. For input data, the out-of-bounds elements are padded with zeros and written to the on-chip buffers. For the output data, before each work_item writing back its corresponding element, it will check whether the location is out-of-bounds. The out-of-bounds elements will not be written back.

The OPT2 generates Map_matrix^T, thus we need to modify the matrix multiplication kernel from $C = A * B$ to $C = A * B^T$. Each work_item can

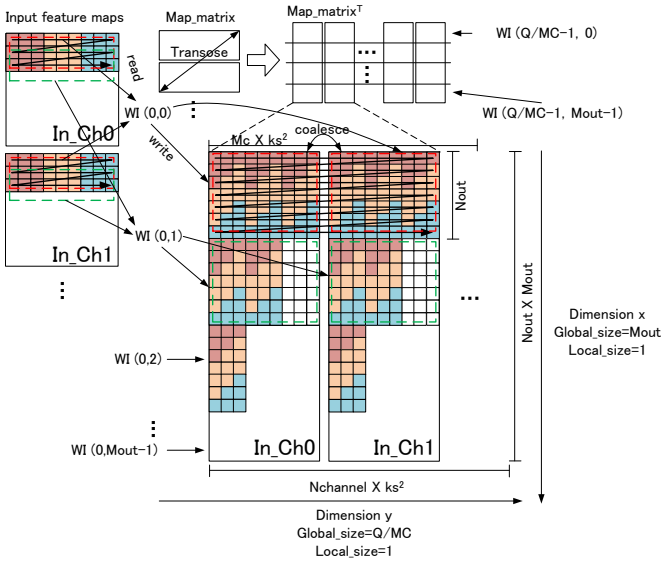

Fig. 5. A further optimizing of the memory access (OPT2)

directly read the corresponding element from the transposed matrix block, but the address will not be continuous. We change the read order to the external memory to ensure the address continuous.

The baseline version adopts 32-bit floating point data format. In fact, research has proven that such high precision is redundant for the forward propagation [4, 9] of CNN. We modify the optimized versions to 16-bit fixed point to increase the accelerator performance.

The full connection layers can be processed by the matrix multiplication kernel independently. Although our implementation can handle matrix blocks of different sizes, using batched images can reach higher performance. In the full connection layers, for a single input image, the main computation is matrix-vector multiplication. The ratio of computation/memory access of a matrix-vector multiplication is low, because every element in parameter matrix only relates to one multiply-accumulate operation. The accelerator thus needs a high external memory bandwidth to read the parameter matrix. To achieve higher bandwidth, multiple images' full connection layers can be merged through combining a batch of matrix-vector multiplications into one matrix-matrix multiplication. Every element in parameter matrix needs to be operated with batch size elements in input matrix. The ratio of computation/memory access increases when the batch size increases.

5 System Evalution

We propose a prototype of our design in a DE5-net development board. The main chip of the board is an Altera Stratix V A7 FPGA. The work frequency of out kernel logic is 185 Mhz. We implement our design using Altera OpenCL SDK 14.1. The most widely used model VGG-16 is chosen to be the benchmark.

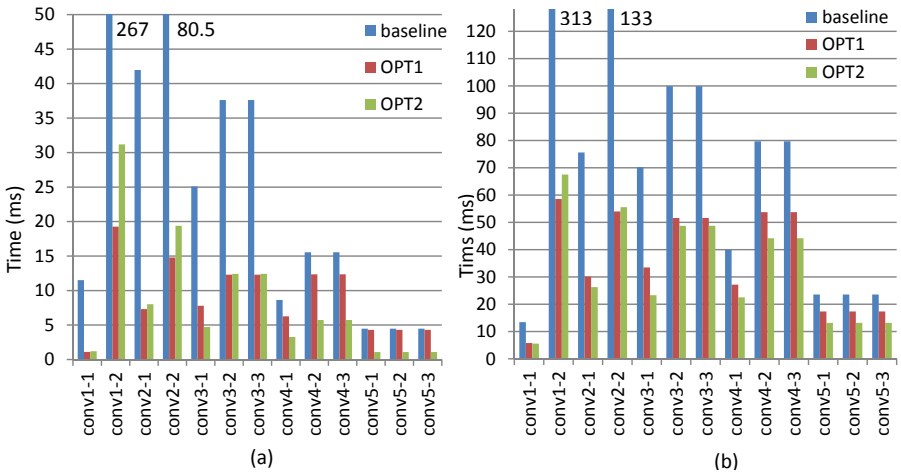

Fig. 6. Performance of the optimized versions

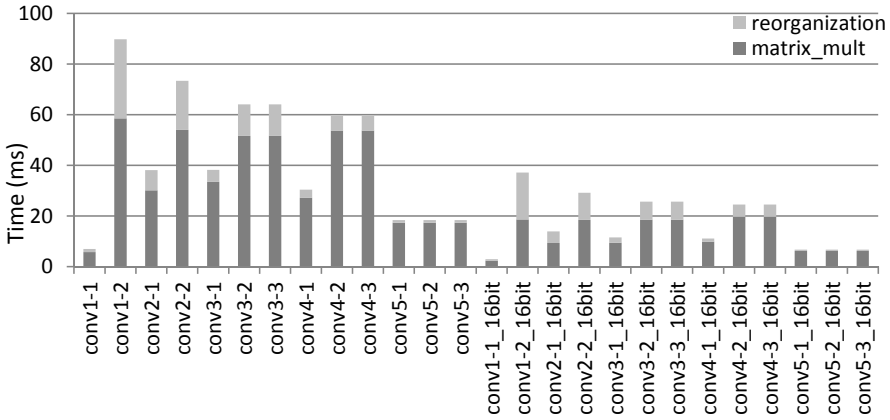

Fig. 7. Performance of different data precision

Table 1. Critical resource utilization rate in one chip

Resource	**DSP**	**BRAM**	**LUT**
Available	256	2560	234,720
Used	256	1456	179,850
Utilization	100%	57%	77%

Fig. 6 (a) shows the performance of reorganizing part. The vertical axis refers to the consumed time and the horizontal axis refers to different layers. Bars with three different colours represent the three different versions. The result shows that the performance of optimized code is improved dramatically. Both two optimized versions greatly reduce the consumed time. The OPT1 performs better in the first several layers and the OPT2 performs better in the last several layers. Our discussion in section 4 is confirmed by the experimental results.

Fig. 6 (b) shows the performance of the convolutional layers. The performance of optimized versions are still much better than the baseline version after the matrix multiplication part is added.

Fig. 7 shows the performance under different data precisions, compared with the 32-bit float-point data type, the 16-bit fixed point data type nearly doubles the performance. It is because that performing a 16-bit fixed point multiply-add operation only needs 1 DSP in Altera's FPGA while performing a 32-bit floating point multiply-add operation needs 2 DSPs. Using 16-bit fixed point computation engine needs fewer DSP units and less on-chip memory resources. Compared with the baseline version, the optimized 16-bit fixed-point version has a speed up of 4.76x.

We also make a comparison between our implementation and other state-of-the-art FPGA CNN accelerators based on OpenCL. Table 1 shows the on-chip

Table 2. Comparison between our design and existing FPGA-OpenCL based CNN accelerators

	FPGA2016[9]	**Cnnlab[10]**	**FPGA2017 [11]**	**Our design**
Precision	16-bit fixed	32-bit float	16-bit fixed	16-bit fixed
Frequency	120 MHz	171 MHz	385 MHz	185Mhz
FPGA chip	Stratix V A7	Stratix V A7	Arria 10 GX1150	Stratix V A7
DSP available	256	256	1518	256
DSP used	-	162	1378	256
CNN type	VGG-16	AlexNet	VGG-16	VGG-16
Real performance	47.5Gops	25.56GFlops	1790Gops	137Gops
Performance density	0.19Gops/DSP	0.01Gops/DSP	1.18Gops/DSP	0.53Gops/DSP
Use OpenCL only	✓	✓	✗	✓

resource utilization of our implementation (OPT2). To unify the performance metric among different FPGA devices, we use performance density as the metric, the unit is Gops/DSP. As shown in Table 2, our implementation achieves the highest performance among recent works using the same device (Stratix V A7). [9] provides a throughput-optimized OpenCL implementation. To the best of our knowledge, it is the first OpenCL implementation for an entire CNN model on FPGA. However, there is still a big gap between its real performance and the peak performance of the FPGA they used. Comparing with [9], we present a better memory access design and get a 2.88x speed up on the same device. [10] presents a CNN framework that using GPU and FPGA-based accelerators. They make more effort on compatibility while we focus on performance using FPGA. Compared with [10], our design has a speed up of 5.35x. [11] uses System Verilog to implement a CNN accelerator and package it into the OpenCL IP library. This work achieves a very high performance and gets better performance density than ours. However, in fact this work is an RTL design. Compared with HLS designs, the RTL design can exploit more hardware details. Thus it can get higher frequency and efficiency easily, whereas the major benefit of OpenCL design is better reusability and shorter development cycle.

6 Conclusion

In this paper, we have proposed an optimized CNN accelerator design using OpenCL FPGA. By analyzing the OpenCL implementation for CPU/GPU, we find that the bottleneck is the external memory access, because the memory system of FPGA is much different with CPU/GPU. Then we optimize the CNN design. Effort is made on the data re-arrangement and coalescing the memory accesses are applied for better usage of the external memory bandwidth. A prototype system is built. Compared with the baseline version, a performance speed-up of 4.76x is achieved. Our implementation on the Altera Stratix V device achieves a 137 Gop/s throughput under 185MHz working frequency. This performance outperforms most of the prior work using OpenCL on FPGA.

Acknowledgement

This research is supported by the National Key Research and Development program under No.2016YFB1000401, the National Nature Science Foundation of China under NSFC No. 61502509, 61402504, and 61272145; the National High Technology Research and Development Program of China under No. 2012AA012706; and the Research Fund for the Doctoral Program of Higher Education of China under SRFDP No. 20124307130004.

References

1. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: Cudnn: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
2. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al.: Dadiannao: A machine-learning supercomputer. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society (2014) 609–622
3. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J.: Optimizing fpga-based accelerator design for deep convolutional neural networks. In: Proceedings of the 23rd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), ACM (2015) 161–170
4. Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., et al.: Going deeper with embedded fpga platform for convolutional neural network. In: Proceedings of the 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), ACM (2016) 26–35
5. Czajkowski, T.S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P., Singh, D.P.: From opencl to high-performance hardware on fpgas. In: 22nd International Conference on Field Programmable Logic and Applications (FPL), IEEE (2012) 531–534
6. Lin, D., Talathi, S., Annapureddy, S.: Fixed point quantization of deep convolutional networks. In: International Conference on Machine Learning. (2016) 2849–2858
7. Gu, J., Liu, Y., Gao, Y., Zhu, M.: Opencl caffe: Accelerating and enabling a cross platform machine learning framework. In: Proceedings of the 4th International Workshop on OpenCL, ACM (2016) 8
8. Altera: Altera opencl design examples. https://www.altera.com/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html
9. Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J.s., Cao, Y.: Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In: Proceedings of the 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), ACM (2016) 16–25
10. Zhu, M., Liu, L., Wang, C., Xie, Y.: Cnnlab: a novel parallel framework for neural networks using gpu and fpga-a practical study with trade-off analysis. arXiv preprint arXiv:1606.06234 (2016)
11. Zhang, J., Li, J.: Improving the performance of opencl-based fpga accelerator for convolutional neural network. In: FPGA. (2017) 25–34