Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network

Jialiang Zhang and Jing Li
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
{jialiang.zhang, jli}@ece.wisc.edu

Abstract

OpenCL FPGA has recently gained great popularity with emerging needs for workload acceleration such as Convolutional Neural Network (CNN), which is the most popular deep learning architecture in the domain of computer vision. While OpenCL enhances the code portability and programmability of FPGA, it comes at the expense of performance. The key challenge is to optimize the OpenCL kernels to efficiently utilize the flexible hardware resources in FPGA. Simply optimizing the OpenCL kernel code through various compiler options turns out insufficient to achieve desirable performance for both compute-intensive and data-intensive workloads such as convolutional neural networks .

In this paper, we first propose an analytical performance model and apply it to perform an in-depth analysis on the resource requirement of CNN classifier kernels and available resources on modern FPGAs. We identify that the key performance bottleneck is the on-chip memory bandwidth. We propose a new kernel design to effectively address such bandwidth limitation and to provide an optimal balance between computation, on-chip, and off-chip memory access. As a case study, we further apply these techniques to design a CNN accelerator based on the VGG model. Finally, we evaluate the performance of our CNN accelerator using an Altera Arria 10 GX1150 board. We achieve 866 Gop/s floating point performance at 370MHz working frequency and 1.79 Top/s 16-bit fixed-point performance at 385MHz. To the best of our knowledge, our implementation achieves the best power efficiency and performance density compared to existing work.

1. INTRODUCTION

Convolutional Neural Networks (CNNs) are widely used in computer vision, speech recognition, natural language processing and text classification. Over the past decade, the accuracy and the performance of CNN has improved significantly, mainly due to the enhanced neural network structures enabled by massive datasets and increased computational resources benefits from the CMOS scaling to train the models in reasonable time.

In recent years, FPGA has become an attractive solution to accelerate CNN classification [1, 2] for its flexibility, short time-to-market, and energy efficiency [3], especially with the recent release of a new-generation high level synthesis (HLS) tool, i.e., OpenCL, which greatly reduces the time and complexity of the programming process. For instance, prior work [4] successfully demonstrated an optimized OpenCL FPGA accelerator for CNN classifiers.

While OpenCL framework provides a high-level programming interface which allows programmers to reuse their code from other platforms, thus significantly enhancing programmability and portability, this comes at the expense of performance. Therefore, it is imperative to optimize the kernel design to maximize hardware resource utilization for better performance.

In this work, to achieve a high performance CNN accelerator, we first propose an analytic model to guide our kernel design to achieve a better mapping from OpenCL kernels to FPGA hardware. We borrow the insights from the roofline model in [1] and further improve it by taking both on-chip and off-chip memory bandwidth into consideration. The core concept of the model is to introduce two key metrics, *machine balance,* and *code balance*. Such balance models quantify the difference between available resources provided by native hardware (FPGA devices) and actual resources demanded by the application (CNN classification kernel). Thus, it can help to pinpoint the performance bottlenecks in the implementation of an OpenCL-based FPGA accelerator for CNNs and provide optimization opportunities.

As a case study, we further apply the model to perform a comprehensive analysis on one of the most popular CNN models: Very Deep Convolutional Networks (VGG) [5], which has also been studied in prior works [2, 4]. From our analysis, a key learning is that the performance of existing OpenCL FPGA CNN accelerators is inherently limited by the "unbalanced" hardware resources. More specifically, the on-chip memory bandwidth cannot match the computational throughput and the off-chip memory bandwidth. Experimental results confirmed that: 1) the computational resources are heavily under-utilized and 2) on-chip and off-chip data accesses are not well balanced. Both observations are due to the inefficient bandwidth utilization of on-chip memory. However, these works either focus on optimizing the computational throughput by increasing parallelism (loop unrolling, vectorization) using the pragma directives provided by the OpenCL SDK without considering memory or optimizing the external memory access through data reuse or advanced memory modules such as Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM) [6]. Nevertheless, to our best knowledge, there are no efforts to develop a systematic and generic optimization methodology to guide accelerator design accounting for all factors, especially the on-chip memory bandwidth.

Our study further reveals that the inefficiencies of on-chip memory bandwidth are fundamentally due to the current OpenCL kernel design. While designing the kernel, explicit parallel kernel

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

DOI: http://dx.doi.org/10.1145/3020078.3021698

code is translated into a replication of custom compute units. To match the computational throughput and minimize external memory access, on-chip (local) memories will need to be replicated as well. Such a replication scheme quickly drains on-chip memory resources, making the computational-intensive applications become memory-bound. More importantly, from the hardware perspective, the native on-chip memory bandwidth has not been improved much over technology generations (Table 1), making it even more challenging to bridge the gap between computation and memory resources with existing techniques. To address this issue, we propose a novel design for a CNN classification kernel. In particular, the new kernel design is comprised of : **(i)** a two-dimensional interconnection between PEs and the local memory system, as compared with the one-dimensional interconnection in the existing OpenCL kernel design. It enables efficient data sharing without memory replication and thereby increases the effective on-chip memory bandwidth by orders of magnitude. **(ii)** a two-dimensional dispatcher to support the proposed interconnection between PEs and local memory. We also develop a work-item scheduling technique to further optimize memory resource usage. **(iii)** a shared buffer technique, which can be used in conjunction with (i) and (ii) to further reduce the external memory bandwidth requirement. We summarize the key contributions as follows:

- We propose an analytical model to quantitatively characterize the correlation between the performance of CNN classification kernels and available resources on FPGAs.
- We apply the model to perform an in-depth analysis on VGG and identify that the key performance bottleneck is on-chip memory bandwidth.
- We propose a novel kernel design to effectively address the on-chip memory bandwidth limitation and to provide an optimal balance between computation, on-chip and off-chip memory access.
- We conduct experiments to verify the effectiveness of the proposed techniques. To the best of our knowledge, our implementation achieves the **highest performance, energy efficiency and performance density** compared to state-of-the-art OpenCL FPGA CNN implementations.

The rest of the paper is organized as follows. Section 2 presents the background of OpenCL FPGA and the state-of-the-art CNN implementations. In Section 3, we present an analytical performance model and apply it to analyze the performance bottlenecks of implementing a CNN accelerator using OpenCL FPGA. In Section 4, we present a novel kernel and dispatcher design to address the performance bottlenecks. Section 5 presents a detailed case study on CNN using the proposed kernel design with an optimization technique. Section 6 presents the experimental results of our CNN design and validates proposed techniques. Section 7 concludes the paper.

2. BACKGROUND

In this section, we first provide an overview of the OpenCL stack, and then present the background of the convolutional neural network (CNN) and its state-of-art OpenCL FPGA implementations.

2.1 OpenCL Stack on FPGA

In Figure 1, we summarize the OpenCL stack on the FPGA platform, which contains two components: the host API and the

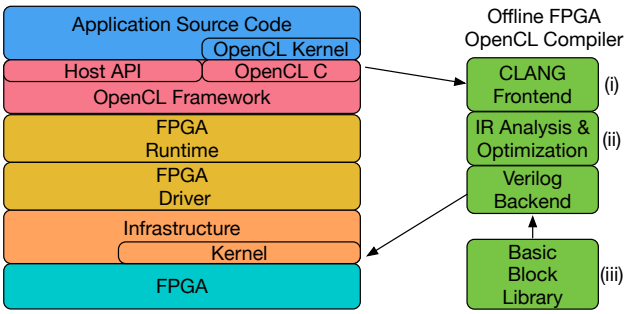

Figure 1: Overview of OpenCL FPGA stack. OpenCL FPGA provides extra flexibility in configuring the hardware architecture in an offline compilation process.

OpenCL device kernel. Under the OpenCL framework, device vendors provide two device specific layers: runtime and driver, to interface with the upper-level programming framework and the underlying hardware.

Unlike GPU and other fixed architectures, at the core, the OpenCL compilation flow for FPGAs offers additional flexibility in customizing hardware tailored for a specific application. In particular, the OpenCL kernels for FPGA must be compiled *offline* [7] into a set of configurable primitive operations provided by the vendor. We note that these primitives used in FPGA's compilation are analogous to processor instructions, e.g., load/store, arithmetic operation and synchronization operation, etc.. Nonetheless, they can be pre-compiled into custom hardware modules in FPGAs and placed in the kernel database of the system as an IP library.

As shown in Figure 1, the offline kernel compiler is based on the LLVM framework and the compilation process is composed of three components: CLANG frontend, LLVM-optimizer and a custom backend which invokes the IP library to generate custom hardware for each basic block and connect them using flexible routing resources on the FPGA. However, such flexibility also creates challenges for performing optimization in significantly enlarged design spaces in the hardware/software domain. If not handled properly, the unoptimized kernel implementation may severely degrade performance.

The OpenCL FPGA framework is depicted in Figure 2. It consists of an infrastructure region, which includes a PCIe DMA engine, an external DRAM controller, and a kernel region. The kernel connects to the SOC Bus (e.g AXI or Avalon) in the infrastructure region to interact with external DRAM and the Host. Based on the current OpenCL FPGA kernel design, *work-items* are execution instances of a kernel and are grouped by *work-groups*. The kernel region has three major components: a dispatcher, a compute subsystem organized hierarchically into compute units (CUs) and processing elements (PEs), and a local memory subsystem. The dispatcher is responsible for scheduling work-groups (work-items) to the CUs (PEs) as well as host/device memory transfers. The CUs execute work-groups in lockstep. All CUs share a global memory and constant memory while each CU has its own local memory. The PE array in each CU can be organized in one, two or three dimensions using the OpenCL attribute *num_compute_units()*. In Figure 2(c), we show a 2×2 PE organization. We note that even though PEs can be arranged in high-dimension, the interconnection between PEs and the local memory system is still one-dimensional.

2.2 FPGA Accelerator for CNN

The Convolutional Neural Network (CNN) is currently the most popular deep learning architecture in the domain of artificial intel-

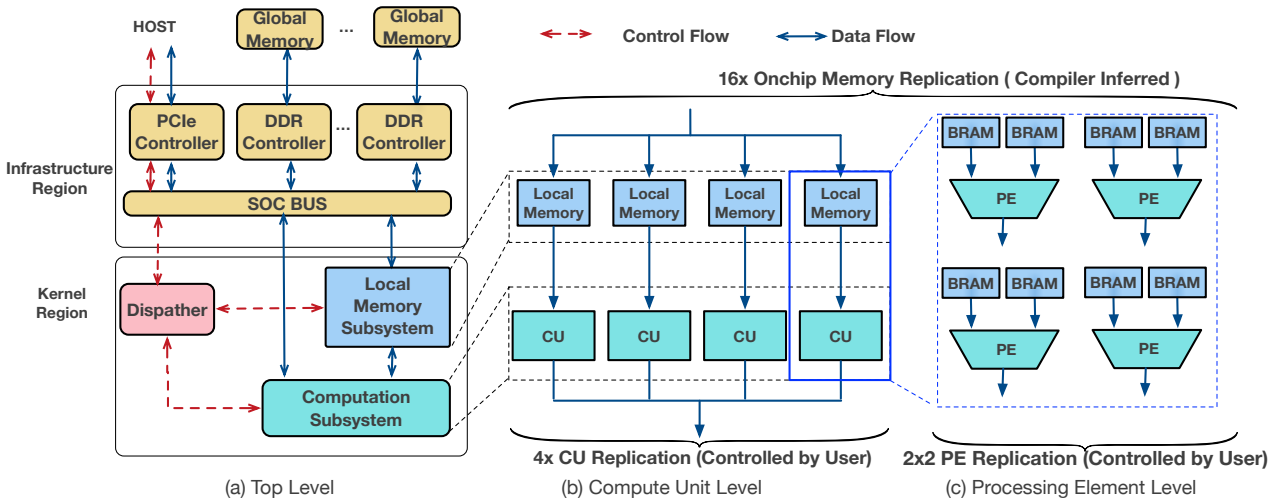

Figure 2: OpenCL FPGA framework:(a) Top level ;(b) Compute unit (CU); (c) Processing element(PE)

ligence (AI) [8] and computer vision (CV) [5, 9, 10]. CNNs are typically organized into alternating convolutional and pooling neural network layers followed by a number of fully-connected layers. The detailed operations of CNNs can be found in prior work [9]. Similar to other supervised learning algorithms, CNNs have a feed-forward path for recognition and a backward path for training. In practice, the training phase is typically conducted offline and the pre-trained model is deployed for online classification.

Most recently, the advances in FPGA hardware and design tools have reignited interests in implementing CNNs on FPGAs. For instance, [1, 2] use high-level synthesis (HLS) tools to develop CNN accelerators on FPGA with soft microprocessors and embedded hard microprocessors, respectively.

There are also efforts to develop FPGA CNN accelerators using OpenCL. For example, [4] attempts to find optimal parallel CU/PE configurations for different workloads to achieve desired computational throughput. However, in order to serve the concurrent memory requests from parallel CUs/PEs and to minimize the penalty of external memory access, the compiler automatically replicates the local memory (BRAM) by the same factor [7]. For instance, in Figure 2, we show the kernel design used by [4], with 4 CUs. Each of the CUs contains a 2×2 PE array, which provides 16x more parallelism in computation but inevitably increases the on-chip memory consumption by a factor of 16 as well. In the following section, we will analyze the impact of the PE/BRAM replication on accelerator performance.

3. DESIGN CHALLENGE OF AN OPENCL BASED FPGA ACCELERATOR FOR CNN APPLICATION

In this section, we present an analytical performance model to characterize the resource requirement of a given kernel and the available hardware resources of FPGA. Based on the model, we will perform an in-depth analysis on the design challenges of implementing a CNN accelerator using OpenCL FPGA.

3.1 Performance Model

The performance of an OpenCL FPGA accelerator design can be determined by a large number of factors. Among them, the most critical one is data access. As explained in section 1, the hardware resources in state-of-the-art FPGAs are inherently unbalanced,i.e., computation vs. memory, on-chip memory vs. off-chip memory. However, theoretical peak performance can only be approached by providing a "balanced" data flow. The problem is becoming more pronounced in implementing CNN classifiers which demand high throughput in processing.

To quantitatively evaluate the performance limiting factors in OpenCL FPGA accelerator implementations, we borrow insights from the roofline model in [1] and develop an improved analytical performance model considering both on-chip and off-chip memory bandwidth. The model is based on the following assumptions: (i) The applications of interest are bandwidth bounded, not latency bounded. A large set of streaming applications, including CNNs, fall into this category [11]. (ii) Computation and memory access overlap perfectly. External memory access is coalesced well to achieve the maximum streaming bandwidth. (iii) The performance is determined by the slowest datapath in the memory hierarchy. All faster paths are assumed to be infinitely fast.

The central concept of our model is the introduction of "machine balance" B_m, which is used to quantify the balance between memory bandwidth and computational bandwidth of a system. As shown in Equation(1), it is defined as the ratio of the maximum achievable memory bandwidth bw_{max} to peak arithmetic performance (P_{max}) provided by hardware.

$$B_m = \frac{\text{Memory Bandwidth}}{\text{Peak Performance}} = \frac{bw_{max}}{P_{max}} \quad (1)$$

To substitute the generic memory bandwidth bw_{max} with on-chip memory bandwidth bw_m^{on} and off-chip memory bandwidth bw_m^{off}, we further introduce two more metrics, B_m^{on} and B_m^{off}, defined as the machine balance for on-chip and off-chip memory respectively. Note that the machine balance B_m is largely determined by hardware and can be used to identify if a specific OpenCL FPGA platform provides adequate resources for an application.

On the other hand, from the application perspective, we introduce "code balance" B_c which reflects the memory bandwidth requirement of a specific OpenCL kernel program. In contrast to machine balance, B_c is inherently determined by the application itself. Similar to machine balance, we further define B_c^{on} and B_c^{off} as the code balance for on-chip memory and off-chip memory respectively. The calculation of B_c^{off} is given by Equation (2).

$$B_c = B_c^{off} = \frac{\text{Data Size}}{\text{Total No. of Operation}}, \quad (2)$$

Unlike B_c^{off}, the calculation of B_c^{on} can be converted into first calculating B_c^i, the code balance for each arithmetic instruction (Equation (3)) and then taking an averaged value across all the arithmetic instructions in a kernel (Equation (4)). For instance, operations like mul and add , which need two input words to perform one arithmetic operation, have a $B_c^i = 2$. Similarly, operations like *MAC*, which need two inputs to perform two consecutive multiply and add operations, have a B_c^i of 1.

$$B_c^i = \frac{\text{Instruction Input (}\textit{Words}\text{)}}{\text{No. of Operations (}\textit{Ops}\text{)}}, \quad (3)$$

$$B_c^{on} = \frac{1}{N}\sum_N B_c^i, \quad (4)$$

In addition to balancing computation to data access, the bandwidth matching between on-chip and off-chip memory is equally important for performance. For that, we further compare B_c^{on} with B_c^{off}. The difference between them indicates the degree of data locality (data reuse among instructions) that a program inherently possesses. The larger the difference, the higher the locality. In principle, one data fetched from external memory and buffered on- chip should be reused at least $\frac{B_c^{on}}{B_c^{off}}$ times to match the on-chip memory bandwidth with the off-chip memory bandwidth for maximum efficiency. We define γ as the data reuse ratio and the memory subsystem should be designed to satisfy $\gamma \leq \frac{B_c^{on}}{B_c^{off}}$ for optimal usage.

With the proposed balance model, we can compare the difference between machine balance and code balance and use $\frac{B_m^{\{on,off\}}}{B_c^{\{on,off\}}}$ to quantitatively characterize the gap between the resource requirement of an OpenCL kernel and the available hardware resources provided by the FPGA. The larger the difference, the wider the gap, and the worse the accelerator performance. As shown in Equation 5, we can use it to identify the performance bottleneck in an accelerator design, i.e., compute-bound or memory-bound. If $\frac{B_m^{\{on,off\}}}{B_c^{\{on,off\}}} < 1$, which indicates the memory bandwidth requirement of a kernel is larger than the available memory bandwidth provided by native hardware, i.e., it becomes *memory-bound*. In that case, the compute units become under-utilized and have to stall to wait for the data, resulting in performance degradation. Otherwise, it is compute-bound. Therefore, the arithmetic performance P can be expressed as

$$P^{\{on,off\}} = \begin{cases} P_{max}, & \frac{B_m^{\{on,off\}}}{B_c^{\{on,off\}}} \geq 1, \quad (\text{comp. bound}), \\ \frac{bw_{max}^{\{on,off\}}}{B_c^{\{on,off\}}}, & \frac{B_m^{\{on,off\}}}{B_c^{\{on,off\}}} < 1, \quad (\text{mem. bound}), \end{cases} \quad (5)$$

In the case of a memory-bound application, we can further provide more insights as to whether the performance is limited by on-chip memory or off-chip memory in the memory hierarchy using Equation (6).

$$P = \min\left(P^{on}, P^{off}\right) = \min\left(\underbrace{\frac{bw_{max}^{on}}{B_c^{on}}}_{\substack{\text{On-chip} \\ \text{Mem BW} \\ \text{Bounded}}}, \underbrace{\frac{bw_{max}^{off}}{B_c^{off}}}_{\substack{\text{Off-chip} \\ \text{Mem BW} \\ \text{Bounded}}}, \underbrace{P_{max}}_{\substack{\text{Comp.} \\ \text{Bounded}}}\right), \quad (6)$$

Thus, the overall performance is determined by the minimum value between the projected performance bounded by on-chip memory P^{on} and by off-chip memory P^{off}. Ideally, it is desirable to keep $\frac{B_m^{\{on,off\}}}{B_c^{\{on,off\}}}$ equal to 1 to ensure a perfect balance between computation and memory resources to achieve the best utilization and maximum throughput. However, in practice, it is not feasible as it not only varies with hardware platforms but also with workloads. To achieve the peak performance, it is desirable to keep it larger than 1 in order to make good use of FPGA's abundant computational resources for better performance. As the code balance B_c is fixed by the algorithm itself, the most effective way to improve the ratio is to improve B_m^{on} or B_m^{off}. However, as we will discuss in next subsection, the existing OpenCL FPGA implementations for applications like CNN classifiers are inherently bounded by on-chip memory ($\frac{B_m^{on}}{B_c^{on}} < 1$).

3.2 Performance Analysis

In this subsection, we will apply the balance model to analyze the performance limiting factors when implementing an OpenCL FPGA accelerator for CNN classification. We will compare the machine balance of modern FPGA devices of three technology generations (28nm, 20nm, 16/14nm) with the code balance of a CNN classifier using the VGG19 model [5], which is the winner of Imagenet 2014 competition.

In Table 1, we list the machine balance B_m^{on} and B_m^{off} for on-chip and off-chip memory respectively, in conjunction with other essential hardware resources from the largest commercially available FPGA devices over three technology generations (28nm, 20nm, 16/14nm). We note that to calculate the B_m^{on}, we adopt the memory replication strategy based on the existing kernel design in OpenCL FPGA as described in Section 2. We can see that B_m^{on} is almost constant for different technology nodes, which indicates the computational resources increase proportionally with respect to the on-chip memory bandwidth. In other words, the design space has not changed over generations of FPGA devices from the hardware perspective. However, the B_m^{off} is improved dramatically at the 16/14nm technology node due to the availability of advanced memory modules such as HBM, HMC, and bandwidth engine (BE2) [6].

In Table 2, we also list both on-chip code balance B_c^{on} and off-chip code balance B_c^{off} with workload specific information including data size, number of operations of the convolution (CONV) layer and fully-connect (FC) layer in the VGG model [5], as CONV layers and FC layers constitute the majority of the computation in the VGG model.

To gain more insights on the performance limiting factors for implementing the VGG model, we follow the methodology developed in Section 3.1 and calculate the $\frac{B_m}{B_c}$ for both CONV layers and FC layers considering both on-chip and off-chip memory access. Let us first consider off-chip memory bandwidth. In Figure 3(a), we show that $\frac{B_m^{off}}{B_c^{off}}$ for CONV layers are all greater than 1 except for the 14nm FPGA devices with DDR DRAM, which indicates the off-chip memory bandwidth is *not* the bottleneck. In Figure 3(b), we show that $\frac{B_m^{off}}{B_c^{off}}$ for FC layers is smaller than 1, which indicates its performance is limited by external memory bandwidth. However, as FC layers contribute only a small fraction to the overall computation (1.2 %), its impact on performance can be considered negligible.

Compared to off-chip memory access, we show that the on-chip memory bandwidth indeed becomes the bottleneck. As shown in Figure 3(c), $\frac{B_m^{on}}{B_c^{on}}$ for all layers is less than 1, which indicates that the CNN classifier is bounded by the on-chip memory bandwidth. To make it worse, unlike external memory bandwidth, which can be

Table 1: Comparison of resources and performance of the largest commercial FPGA over generations

	28nm	20nm	16/14nm
Amount of DSP	3600	5520	12288
DSP frequency (MHz)	741	741	891
Arith. Perf. (GFLOP/sec)	5335	8180	21897
Amount of BRAM	1470	2160	3908
Capacity of BRAM(Mb)	52.9	75.9	454.5
bw_{max}^{on} **(GWords/s)**	3586	7128	15526
bw_{max}^{off} **(GWords/s)**	70.3	76.8	96[1], 1000[2]
B_m^{on}	0.168	0.217	0.177
B_m^{off}	0.0033	0.0026	0.0010[1], 0.014[2]

[1]DDR4 SDRAM
[2]HBM

Table 2: Problem size and number of operations of CONV and FC layers in VGG model

Layer group	Size of Input Feature (k)	Size of weight (k)	Size of Output Feature (k)	Ops (GOPs)	Percent of Total Operations	B_c^{off}	B_c^{on}
CONV1	336	38	6422	3.8	8%	0.0021	1
CONV2	240	221	3211	5.5	21%	0.0010	1
CONV3	280	2064	3211	12.9	36%	0.00062	1
CONV4	140	8257	1605	12.9	26%	0.00087	1
CONV5	401	9437	401	3.6	8%	0.00277	1
FC1	25	102760	4096	0.0002	0.04 %	0.5	1
FC2	4	16777	4096	0.00003	0.04 %	0.5	1
FC3	4	4096	1000	0.000008	0.04 %	0.5	1

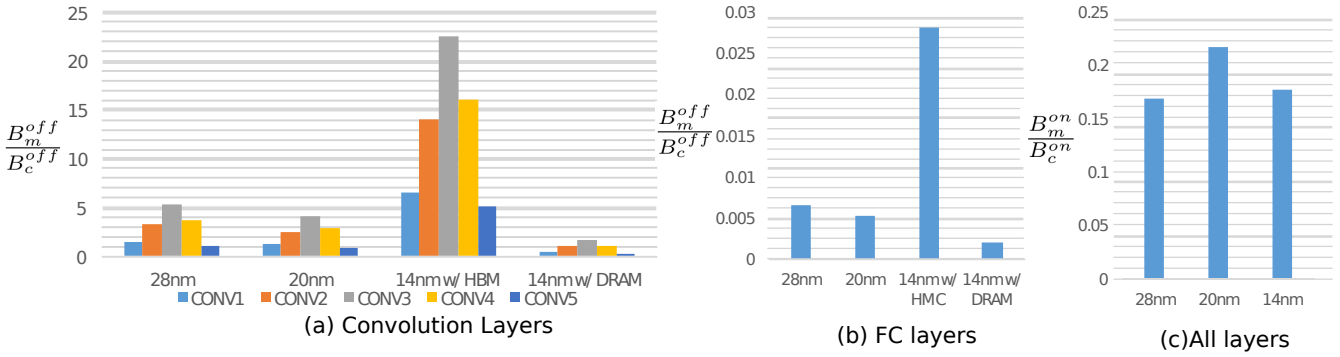

Figure 3: (a) $\frac{B_m^{off}}{B_c^{off}}$ for CONV Layers; (b)$\frac{B_m^{off}}{B_c^{off}}$ for FC layers; (c) $\frac{B_m^{on}}{B_c^{on}}$ for all layers; (d) $\frac{B_c^{on}}{B_m^{off}}$ for all layers

improved by advanced memory modules, from previous machine balance studies, we show that the on-chip memory bandwidth has not changed over generations with respect to computational bandwidth.

As discussed in Section 3.1, another key factor that should be taken into account for performance is the balance between the on-chip memory bandwidth and off-chip memory bandwidth. From Table 2, we see a dramatic difference between B_c^{on} and B_c^{off} for all CONV layers, which indicates high data locality and thus great data reuse potential. We will need to leverage it to make on-chip memory design meet the data reuse ratio requirement defined in Section 3.1 to achieve balanced on-chip to off-chip data access.

To conclude, our analysis shows that the on-chip memory bandwidth is the bottleneck for implementing the OpenCL FPGA accelerator for CNN classification and should be matched with both computational throughput and the off-chip memory bandwidth for higher performance. Hence, to fully exploit the potential of an OpenCL FPGA platform to achieve the optimal CNN implementation, we should: **(i)** Increase B_m^{on} by improving the on-chip memory bandwidth usage. **(ii)** Meet the requirement of the data reuse ratio $\gamma \leq \frac{B_c^{on}}{B_c^{off}}$ to match the on-chip memory bandwidth with off-chip memory bandwidth.

4. DESIGN METHODOLOGY

From the analysis in the Section 3.2, we know that matching the code balance B_c and the machine balance B_m is the key towards a high performance OpenCL FPGA accelerator for CNN classification. In this section, we will present a novel kernel design for CNN classification, which can achieve more efficient on-chip memory bandwidth usage by improving B_m^{on} and balance on-chip and off-chip memory access to meet the requirement of data reuse. In the rest of the section, we will present the detailed design methodology: 1) introducing a two-dimensional multi-cast interconnection between PEs and local memory; 2) developing a two-dimensional dispatcher to accommodate the change in kernel design.

4.1 CNN Classifier Kernel Design

The interconnection between compute units (CUs) and local memories (BRAMs) defined in the existing OpenCL kernel are shown in Figure 2. As we can see, even though the PEs can be organized as an array with up to three dimensions, the interconnection between local memory and PEs is still in one-dimension, i.e., each PE is connected to a dedicated BRAM port. When replicating PEs to achieve parallelism in performing computation, BRAM will have to be replicated by the same factor. Thus, the B_m^{on} of the kernel design in [4] can be calculated as follows.

$$B_m^{on} = \frac{N_{\text{BRAM}} \cdot f_{\text{DSP}}}{N_{\text{DSP}} \cdot f_{\text{BRAM}}}, \tag{7}$$

where N_{DSP} is the number of DSP slices, N_{BRAM} is the number of BRAM, f_{DSP} is the working frequency of the DSP slices and f_{BRAM} is the working frequency of BRAM as listed in Table 1.

The underlying reason for low B_m^{on} in the existing CNN implementation using OpenCL FPGA can be stated as follows: while fully exploiting data level parallelism by replicating PEs, the unnecessary memory replication quickly used up all on-chip BRAM resources. To improve B_m^{on}, the key is to improve the efficiency of BRAM usage.

On the other hand, when taking a closer look at the data access pattern of matrix multiplication which is the key kernel in CNN, as shown in Equation (8), we can see that each vector from the matrix A needs to be multiplied by n vectors from the matrix B and vice versa. Such data access patterns create opportunities for reusing the data from one on-chip BRAM port by multiple PEs.

$$(AB)_{ij} = \sum_k A_{ik} B_{kj} \qquad \forall i < m, j < n \qquad (8)$$

By leveraging such a data access pattern and the flexible FPGA routing architecture, we propose a novel two-dimensional interconnection between PEs and local memory which effectively supports the sharing of data from the same local memory port of BRAMs by multiple PEs as shown in Figure 4, as compared with the one-dimensional interconnection defined in the existing kernel design (Figure 2). In other words, we use *multicast* connections between PEs and the local memory instead of *unicast* connections. Such interconnection not only improves on-chip memory bandwidth usage without replication but also improves on-chip data reuse to meet the reuse ratio requirement for reducing external memory access.

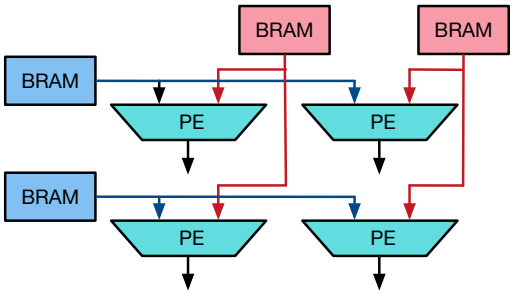

Figure 4: Proposed two-dimensional PE-to-local memory interconnection

We note that in order to support 2-D interconnection between PEs and the local memory system, the dispatcher will need to be modified accordingly. The details of necessary modifications will be explained in Section 4.2. With the proposed 2-D interconnection, the BRAM requirement for N_{DSP} PEs is reduced from $\mathcal{O}(N_{\text{DSP}})$ to $\mathcal{O}(\sqrt{N_{\text{DSP}}})$ and therefore B_m^{on} is improved by $\sqrt{N_{\text{DSP}}}$.

$$B_m^{on\prime} = \sqrt{N_{DSP}} \cdot \frac{N_{BRAM} \cdot BW_{BRAM}}{N_{DSP} \cdot f_{BRAM}} = \sqrt{N_{DSP}} B_m^{on} \qquad (9)$$

We also improve the data reuse by a factor of $\sqrt{N_{DSP}}$ without consuming extra on-chip memory resources, compared to the traditional 1-D PE-to-local memory interconnection. Using the largest FPGA device at the 14/16 nm technology node, we can increase B_m^{on} from 0.177 to 19.62 while improving data reuse by the same factor (110 ×).

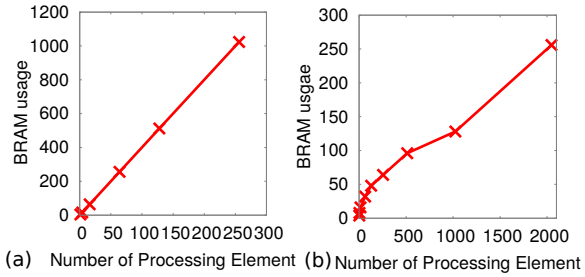

Figure 5: BRAM usage for (a) traditional one-dimensional and (b) proposed two-dimensional interconnection between PEs to local memory when varying the number of PEs using Arria10 AX1150 FPGA

We conduct two preliminary experiments to confirm the effectiveness of the proposed technique. We use the matrix multiplication kernel from the Altera OpenCL library [7] as our baseline. We vary the number of PEs in one CU and obtain the corresponding BRAM usage. In Figure 5(a), we show that the BRAM usage increases linearly when increasing the number of PEs. However, the growth stops at the point of 256 PEs, as if we further increase the number of PEs to 512, the BRAM usage exceeds the limitation of our Altera Arria10 AX1150 FPGA which has 1963 DSP slices and 1500 on-chip BRAMs. We can see that the computational resources are heavily under-utilized when BRAM usage reaches 100% using the traditional 1-D interconnection between PEs and local memory. On the contrary, in Figure 5(b), with the proposed 2-D interconnection, we can make full use of all the 1963 PEs with only 256 on-chip BRAM usage, significantly saving on-chip BRAM bandwidth. We note that the proposed PE-to-memory interconnection can also be applied to accelerate other algorithm kernels, which exhibit similar data access patterns to matrix multiplication.

We have already shown that the proposed matrix multiplication kernel design can significantly improve B_m^{on} and also the on-chip data reuse by a factor of $\sqrt{N_{DSP}}$ by using the 2-D interconnection and the 2-D dispatcher without necessarily increasing the on-chip memory usage. However, it is still not sufficient to meet the data reuse requirement targeting the Altera Arria10 AX1150 FPGA. To address this issue, we add a buffer to further improve the on-chip data reuse and to reduce external memory access. This buffer serves two purposes: **(i)** coalescing memory access from different memory banks in the shared memory system to fully utilize the external memory bandwidth. **(ii)** Leveraging the data locality in the data access stream, such as sliding windows in the convolution layer computation. In the case of CNN, we will present a line buffer design in Section 5. Note that the buffer is not necessarily needed for all FPGA devices as long as the on-chip hardware resources meet the data reuse ratio requirement.

4.2 Two-Dimensional Dispatcher

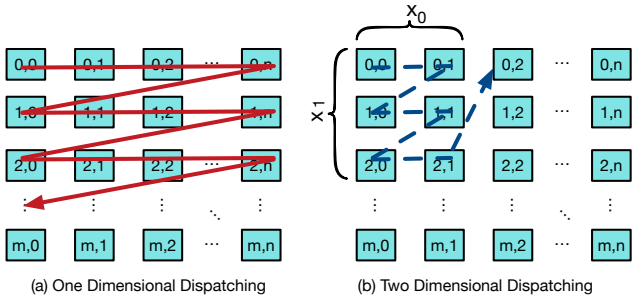

Figure 6: Conceptual diagram illustrating (a) traditional one-dimensional dispatcher and (b) proposed two-dimensional dispatcher ($x_0 = 2, x_1 = 3$)

For the conventional 1-D interconnection between PEs and local memory, the current OpenCL dispatcher always schedules the work-items along the lower dimension as shown in Figure 6(a). However, in order to accommodate our 2-D interconnection with data sharing, we propose to dispatch work-items to compute units in a two dimensional manner. The 2D dispatcher can be implemented by adding a new dispatching policy (x_0, x_1). As shown in Figure 6(b), the 2D dispatcher divides work-items into work groups with a size of x_1 in the x dimension and x_2 in the y dimension, and performs the work-item scheduling and work-group scheduling along the lowest dimension within each division. In section 5.3, we

will present the optimal scheduling policy based on the workload and available hardware resources.

5. IMPLEMENTATION OF AN OPENCL FPGA ACCELERATOR FOR CNN

In this section, we will apply the techniques proposed in Section 4 to implement a full-fledged OpenCL FPGA CNN accelerator based on the VGG model [5]. We will first present the overall architecture and implementation of each CNN layer. Then, we discuss the optimization of the scheduling policy to minimize the requirement of the external DRAM bandwidth.

5.1 Overall Architecture

The accelerator kernel has three major components as shown in Figure 2(a). The dispatcher receives pointers of kernel inputs and outputs from the PCI-e controller, and issues read and write commands to the external DRAM controller. The shared buffer uses a 512-bit Avalon memory mapped bus master to receive the data read from external DRAM. Since there is only one external memory channel on our development board, we only need to have a single global read port. The DDR4 channel has a 64-bit double data rate (DDR) bus width and the DRAM controller operates at a quarter clock rate. As we can achieve a kernel frequency larger than the output data rate of DDR memory controller ($2133\,\mathrm{MHz}/8 = 266.5\,\mathrm{MHz}$), we can fully utilize the external DRAM bandwidth.

Given the fixed PE resources (DSP slices) per FPGA device, we can vary the number of CUs and the size of the PE array per CU. As shown in Figure 7(a), in this design, we choose to assign each compute unit (CU) a 16×16 PE array. The reason to choose a dimension of 16 is that each BRAM provides data width of 32 bits and thereby 16 of them can form a 512-bit data bus which equals the bus width of external DRAM and eases the design interface between compute units and the shared buffer. As discussed in Section 4.1, with the 2D interconnection between PEs and local memory, a 16×16 PE array can achieve a B_m^{on} of 2.83 which is larger than B_c^{on} (1) in Table 2.

5.2 Detailed Implementation

A typical CNN classifier has several tens of layers, which run sequentially. Each layer reads the feature map from its previous layer and outputs a series of new feature maps, based on the parameters of the CNN model typically called "weights". Finally, the CNN classifier outputs the probability of each feature. Convolution (CONV) and Fully Connected (FC) layers are the two basic layers that constitute the majority of the computation for all CNN models. In addition, a pooling layer is typically added after convolution layers. In the rest of this section, we will present the implementation of each layer.

5.2.1 Convolution Layer

The convolution operation constitutes over 90% of the total operations in the VGG model. Therefore, to improve the overall performance of a CNN accelerator, the key is to optimize the performance of convolution. The two-dimensional convolution can be described by Equation (10)

$$out(f_o, x, y) = \sum_{f_i=0}^{N_{if}} \sum_{k_x=0}^{K} \sum_{k_y=0}^{K} wt(f_o, f_i, k_x, k_y) \times in(f_i, x + k_x, y + ky). \quad (10)$$

where $out(f_o, x, y)$ and $in(f_i, x + k_x, y + ky)$ are the output and input feature of neurons at the location (x, y) in the feature map f_o and f_i respectively. $wt(f_o, f_i, k_x, k_y)$ is the weights at the location (k_x, k_y) which get convolved with the input feature map f_i and f_o. We use the similar flatten and rearrangement method as [4] to convert the two-dimensional convolution into matrix multiplication. Figure 7(b) shows the overall data flow of performing a convolution. Weights and input feature maps are stored in DRAM and placed in the order of x,y and f_i.

As shown in Figure 8, we implement a line buffer [12] between local memory and external memory to flatten and rearrange data. The goal is to minimize the random data access penalty from external memory and to improve on-chip data reuse. The line buffer streams data from external memory which has a continuous address and converts it into the data order for 2D convolution. By leveraging the data access pattern of 2D convolution, which is a 2D sliding window and has strong data locality, the line buffer can reduce the bandwidth requirement of external memory substantially.

In addition, to hide the external memory access latency, we fill the line buffer using a ping-pong mechanism to pipeline the data access and computation. More specifically, we choose to fill the 256 memory locations at one time, as 256 is not only half of the Altera M20K memory depth but also the maximum data burst size of the DDR4 interface. Each compute unit, which has 256 floating DSP slices (PEs), calculates 256 vector products in parallel by reusing the data from 16 row memory ports and 16 column memory ports as shown in Figure 7(a).

5.2.2 Fully Connected Layer

Fully connected layers calculate the inner products between input features and weights to get the output features as shown in Equation (11),

$$out(f_o) = \sum_{f_i=0}^{N_{if}} wt(f_o, f_i) \times in(f_i) \quad (11)$$

As the inner product calculation in FC layers is also the key operation in CONV layers, we can reuse the architecture from section 5.2.1 to implement it. However, as the weights do not provide any locality, we can only apply the data sharing technique to the input feature. From the previous study in Section 3.2, we know that the performance of FC layers are bounded by external memory. Fortunately, the FC layers only contribute a small portion of computation, so they will not add too much to the runtime. In this work, we use one column of DSPs (PEs) in Figure 7(a) to implement FC layers to make the computational bandwidth match the maximum streaming bandwidth that the DDR4 memory interface can provide.

5.2.3 Pooling Layer

The pooling layer outputs the average or the maximum value of a local area of the input feature map. Pooling layers can be expressed as Equation (12),

$$out(f_o, x, y) = \max_{0<(k_x,k_y)<k} in(f_i, x + k_x, y + k_y), \quad (12)$$

where k is the pooling kernel size. We implement a similar line buffer as in Section 5.2.1, which uses the connections between different register stages to accomplish the window selection. In our design, we use a 4-input comparator to get the maximum value of a 2×2 window. It is mainly used to buffer data from the output of compute units and then to feed to the pooling layer after convolution.

5.3 Optimization of Work-item Scheduling

As shown in Section 5.2, we can reduce the external memory access to the desired number under the condition that we have un-

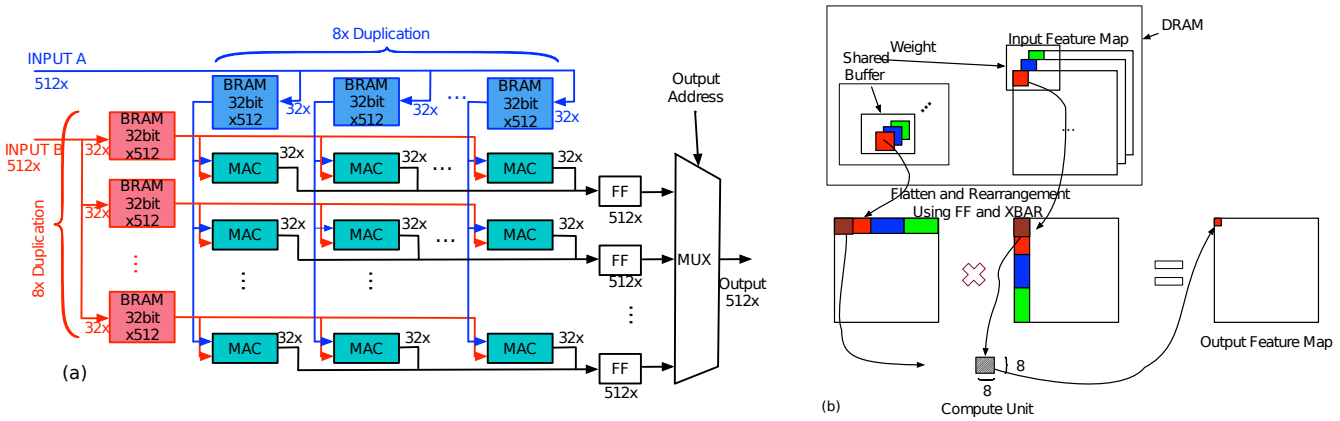

Figure 7: (a) The architecture of proposed CNN accelerator; (b) Convert 3-D convolution into matrix multiplication on proposed architecture

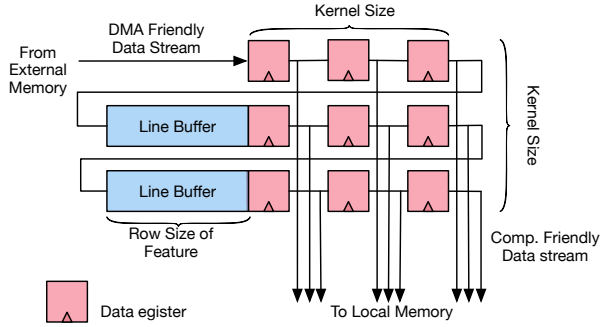

Figure 8: Design of line buffer

Table 3: Work-item scheduling optimization results of convolution layers for the VGG kernel with a 16×16 PE Array and up to 512 M20K BRAMs

Layer	Optimal $<x_1, x_2>$	Req. of DRAM BW with optimal 2-D scheduling	Req. of DRAM BW with 1-D scheduling
CONV1	<6,13>	14.5GB/s	123.9GB/s
CONV2	<6,4>	10.8GB/s	89.8GB/s
CONV3	<5,3>	11.5GB/s	92.6GB/s
CONV4	<7,9>	10.7GB/s	82.1GB/s
CONV5	<4,5>	11.6GB/s	94.6GB/s

limited on-chip memory capacity to hold all input feature maps and weights in BRAM, which in practice, however, is unlikely to be the case. In this section, we present an optimization technique to adaptively change the scheduling policy of the proposed 2-D dispatcher to trade on-chip memory bandwidth with the capacity to satisfy both requirements under device-specific constraints of on-chip memory.

Given the number of input features N_{if}, output features N_{of}, the size of input feature $size_{in}$, output feature $size_{out}$, kernel size $size_k$ and available on-chip memory resources, we minimize the external DRAM bandwidth requirement by solving the following optimization problem:

$$\begin{aligned} \min_{x_0,x_1,x_2} \quad & a\frac{x_0}{x_1} + b\frac{x_0}{x_2} + c\frac{1}{x0}, \\ s.t. \quad & x_0x_1 + x_1x_2 < size_{on-chip}, \end{aligned} \tag{13}$$

Table 4: FPGA resource utilization

	Available Resources	Proposed	Baseline	Ratio
DSP	1518	1320(86%)	576 (38%)	2.26x
BRAM	2713	1250 (46%)	1648 (60%)	0.75x
Logic	1506k (38%)	437k (43%)	237k (16%)	1.84x
Frequency	-	370 MHz	320 MHz	1.15x

$$\begin{aligned} a &= \frac{N_{of} \cdot size_{if}^2 \cdot}{size_k^2}, \\ b &= N_{of} \cdot size_{if}^2, \\ c &= N_{if} \cdot size_k^2 \cdot size_{of}^2 \cdot N_{of}, \end{aligned} \tag{14}$$

where a is the column size of the kernel buffer, b is the row size of the input feature buffer, x_0 is the column size of the kernel buffer and the row size of the input feature buffer, x_1 is the row size in kernel buffer and x_2 is the column size of the input feature buffer. Note that x_1 and x_2 are the scheduling parameters for the 2-D dispatcher. As x_0, x_1 and x_2 are all integer variables and the feasible set is small, we search over all the points in the feasible set to obtain the minimum value. In Table 3, we list the results of optimal scheduling policy of the VGG model and we can observe that the optimal scheduling policy can reduce at least 80% of the external memory bandwidth requirement.

6. EXPERIMENTAL RESULTS

In this section, we first present the modified OpenCL workflow and our evaluation setup. Then, we present the experimental results to validate the effectiveness of the proposed techniques and compare with prior work.

6.1 OpenCL Workflow and Experimental Setup

We use an Arria10 FPGA development board from Altera and list its specifications in Table 4. The board consists of an Altera Arria10 GX1150 FPGA and a 1GB DDR4 SDRAM module with 12GB/s bandwidth. The board also provides an interface to measure the voltage and current of all power rails for power monitoring.

We use Altera OpenCL SDK 16.0.211 as the OpenCL FPGA compiler and use Altera Quartus Pro 16.0.211 as the FPGA implementation tool. We implement the proposed kernel design in

Table 5: Experiment results of CNN accelerator based on VGG

		Proposed		Baseline		Speedup
Layer	**Number of Operations** (GOPs)	**Duration (ms)**	**Perf. (Gops/s)**	**Duration (ms)**	**Perf. (GOP/s)**	
CONV1	3.87	3.5	1098.04	21.3	181.6	6.03x
CONV2	5.55	4.4	1232.04	26.8	207.11	5.95x
CONV3	9.25	6.8	1329.57	41.4	223.35	5.94x
CONV4	9.25	7.4	1347.77	45.0	205.25	6.56x
CONV5	2.31	1.8	1223.32	10.9	210.73	5.82x
CONV TOTAL	30.69	23.9	1284.94	145.5	207.7	6.18x
FC6	0.029	4.1	7.25	4.8	6.09	1.19x
FC7	0.034	5.2	6.58	6.2	5.52	1.19x
FC8	0.0082	1.8	4.50	2.1	3.78	1.19x
FC TOTAL	0.073	11.1	6.6	13.2	5.52	1.19x
TOTAL	30.76	35.5	866	158.8	196	4.41x

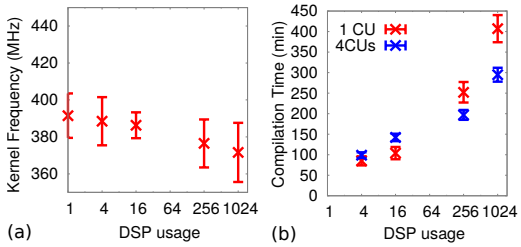

Figure 9: (a)Kernel frequency under different DSP usage; (b) Kernel compilation time under different DSP usages and CU/PE organizations

System Verilog and package it into the OpenCL IP library based on the instructions in [13]. The Verilog kernel has four 512-bit Avalon memory mapped interfaces to communicate with the external DRAM controller and uses a 6-port Avalon streaming interface to receive pointers and other parameters from the host.

The host machine is equipped with an Intel Xeon E5-1630V3 CPU with 64GB DDR4-2133 SDRAM and is running Ubuntu Linux 14.04.3. We follow the methodology in [4] and implemented the baseline design on the same Arria 10 platform. We also use the Caffe [14] convolutional learning framework as our CPU baseline. We extract the input image, pre-trained weights and output features from Caffe. We compare the result of our implementation with result from Caffe to verify the functional correctness.

6.2 Results and Discussion

In this subsection, we first compare the resource utilization with the baseline design, followed by the kernel frequency and compilation time. Then, we show the micro benchmark of our CNN accelerator and compare it with the baseline implementation. Finally, we compare our overall performance with prior work. Note that we implemented both single precision floating point and fixed point versions. Except for the data in Table 6, all the data were obtained from the floating point implementation.

The placement and routing are accomplished by Altera OpenCL SDK, and the resource utilization is reported in Table 4. To achieve a higher working frequency, we use register duplication to limit the maximum fan-out to 100. We found that the paths with the highest fan-out are the control signals, which are generated by the dispatcher and connected to all of the PEs. As the widths of control signals are typically 1 bit, register duplication will not significantly increase the usage of flip-flop. We can see that our implementation has nearly used all of the computational resources (DSPs) with no more than half of on-chip memory. On the contrary, the DSP utilization of the baseline implementation can only achieve 38% and cannot be further improved as the required BRAM usage will exceed the device capacity.

To ensure that increasing the number of PEs sharing the same BRAM port will not introduce noticeable performance degradation, we report the kernel frequency for various cases. We first fix the number of CUs to be 1 while gradually increasing the number of PEs inside a CU. We then change the synthesis seed to check if the fanout of a single memory port increases and reduce the working frequency of the kernel. We extract the working frequency from the OpenCL implementation report. As shown in Figure 9 (a), the kernel frequency slightly decreases from 390MHz to 380MHz when the number of PEs per memory port increases from 1 to 1024.

To further check if our implementation increases the place-and-route complexity, we also log the Quartus compilation time for various CU/PE organizations under the constraints of keeping the same total computational resources. In particular, we consider 1 and 4 CUs while increasing the number of PEs per CU. As shown in Figure 9 (b), the compilation time for the 1 CU design is larger than the design with 4 CUs as increasing the number of PEs per CU increases the pressure for global routing and thus increases compilation time. This result also indicates that given the same computational resource budget, we should assign a smaller number of PEs to each CU and use a larger number of CUs.

In Table 5, we summarize the execution time and performance of each layer group and compare them with the baseline implementation. We can see the performance of the CONV1 layer is much lower than other convolution layers as the performance is bounded by the external memory bandwidth. This is mainly because the Arria10 FPGA development board only uses 1 out of 4 DRAM channels. From Table 3, we can see the required DRAM bandwidth is 14.5GB/s, which is higher than the achievable bandwidth of single channel DDR4 2133 SDRAM. We can also see the proposed design outperforms the baseline implementation by 6.18x for convolution layers and achieves 4.41x improvement in overall performance. Despite the performance gain from working frequency (1.12x) and DSP utilization (2.15x), as listed in Table 4, the proposed design also benefits from the line buffer, which guarantees the DRAM accesses have continuous addresses to achieve the maximum DRAM bandwidth.

In Table 6, we compare our work with previous work. We show our CNN accelerator achieves a floating point performance of 866

Table 6: Comparison with previous CNN implementations

	ISCA2010 [15]	**FCCM16** [16]	**FPGA15** [1]	**FPGA16** [4]	**FPGA16** [2]	**Our Impl.**	
FPGA	Viretex5 SX240T	Zynq XC7Z020	Virtex7	StrtixV GSD8	Zynq XC7Z045	Arria10 GX1150	
Frequency (MHz)	200 MHz	100 MHz	100MHz	120MHz	150MHz	370MHz	385MHz
CNN size (Gops)	0.52	5.48	1.33	30	30.76	30.76	
Precision	fixed	fixed	float	fixed	fixed	float	fixed
DSP Utilization	-	95/110	1120/1400	727/1963	- /780	1320/1518	2756/3036
BRAM Utilization	-	18/280	1024/2060	1500/2567	972/2180	1250/2713	1450/2713
Performance (Gops/s)	16	12.73	61.6	47.5	136.97	866	1790
Performance Density (ops/DSPslices/cycle)	-	0.6	0.44	0.36	1.17	2.8	3.06
Power Efficiency (Gops/s/W)	-	7.27	3.31	1.84	14.22	20.75	47.78

Gop/s and 1.79 Top/s fixed point performance which significantly outperforms prior work. As different experiments use different FPGA platforms and CNN models, it is challenging to have a fair apple-to-apple comparison. To address it, in the table, we further list the performance density which is defined as the number of arithmetic operations that one DSP slice executes in one cycle to characterize the efficiency of a design, i.e., whether or not it utilizes all available computational resources on a FPGA device. The performance density also eliminates the impact of the clock frequency. As shown in Table 6, our implementation has the highest performance density. In addition, we also achieve the highest power efficiency compared to prior work.

7. CONCLUSION

In this paper, we present a new kernel design methodology to implement a high performance CNN accelerator using OpenCL FPGA. In particular, by quantitatively modeling the performance with respect to resources, we present a comprehensive case study on a popular CNN model. When implementing CNN, we find the on-chip memory bandwidth is 6x smaller than the available computational bandwidth in state-of-the-art FPGA devices and thus severely limits the performance due to heavily underutilized computational resources. To address the problem, we propose a 2-D interconnection between PEs and local memory, which can effectively reduce the on-chip memory bandwidth requirement. Furthermore, we design a 2D dispatcher with an optimized scheduling policy to further minimize the external memory requirements by leveraging the data locality of the CNN. Finally, we present the design details of a convolutional learning classification accelerator using the VGG model. Our implementation on the Altera Arria 10 achieves a 1.79 Top/s throughput under 385MHz working frequency with an energy efficiency of 47.78 Gops/s/W and outperforms prior work.

ACKNOWLEDGEMENTS

We appreciate the insightful comments and feedback from the anonymous reviewers. We thank Intel/Altera for the donation of the development tool and hardware. We especially thank Jeff Nigh for his kind help.

8. 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 *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM.

[2] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2016.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, 2014.

[4] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2016.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] Xilinx, "The rise of serial memory andthe future of ddr." http://www.xilinx.com/support/documentation/white_papers/wp456-DDR-serial-mem.pdf, 2015.

[7] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "Opencl for fpgas: Prototyping a compiler," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2012.

[8] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[10] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99, 2015.

[11] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 2010.

[12] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-d convolvers for fast digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Sept 1999.

[13] Altera, "Altera sdk for opencl best practices guide," 2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[15] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, ACM, 2010.

[16] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas,"