PLACID: A Platform for FPGA-Based Accelerator Creation for DCNNs

MOHAMMAD MOTAMEDI, PHILIPP GYSEL, and SOHEIL GHIASI, Electrical and Computer Engineering Department, University of California, Davis

Deep Convolutional Neural Networks (DCNNs) exhibit remarkable performance in a number of pattern recognition and classification tasks. Modern DCNNs involve many millions of parameters and billions of operations. Inference using such DCNNs, if implemented as software running on an embedded processor, results in considerable execution time and energy consumption, which is prohibitive in many mobile applications. Field-programmable gate array (FPGA)-based acceleration of DCNN inference is a promising approach to improve both energy consumption and classification throughput. However, the engineering effort required for development and verification of an optimized FPGA-based architecture is significant.

In this article, we present PLACID, an automated PLatform for Accelerator CreatIon for DCNNs. PLACID uses an analytical approach to characterization and exploration of the implementation space. PLACID enables generation of an accelerator with the highest throughput for a given DCNN on a specific target FPGA platform. Subsequently, it generates an RTL level architecture in Verilog, which can be passed onto commercial tools for FPGA implementation. PLACID is fully automated, and reduces the accelerator design time from a few months down to a few hours. Experimental results show that architectures synthesized by PLACID yield 2× higher throughput density than the best competing approach.

CCS-Concepts: • **Computer systems organization → System on a chip**; **Embedded hardware**;

Additional Key Words and Phrases: Convolutional neural networks, deep learning, accelerator design, design automation

ACM Reference format:
Mohammad Motamedi, Philipp Gysel, and Soheil Ghiasi. 2017. PLACID: A Platform for FPGA-Based Accelerator Creation for DCNNs. *ACM Trans. Multimedia Comput. Commun. Appl.* 13, 4, Article 62 (September 2017), 21 pages.
https://doi.org/10.1145/3131289

1 INTRODUCTION

Deep Convolutional Neural Networks (DCNNs) are extensively used in various image and video processing applications [13, 18, 28]. Despite DCNNs remarkable classification performance, the excessive runtime and energy dissipation of DCNN inference software implementations have hindered their deployment in many mobile embedded systems, such as resource-constrained robots and drones. Computation accelerators are well positioned to address this problem.

The research community has put forth a number of ideas for acceleration of DCNNs using graphics processing units (GPUs), field-programmable gate arrays (FPGAs), or application-specific integrated circuits (ASICs). Among different platform choices, FPGAs offer a favorable

Authors' addresses: M. Motamedi (corresponding author), P. Gysel, and S. Ghiasi, Electrical and Computer Engineering Department, University of California, Davis, One Shields Avenue, Davis, CA 95616; emails: {mmotamedi, pmgysel, ghiasi}@ucdavis.edu.

https://doi.org/10.1145/3131289

tradeoff between throughput, programmability, and energy efficiency. A significant drawback of FPGA-based acceleration, however, is the unproductive and slow process of hardware design, development, and verification.

At a basic level, the accelerators improve application performance by parallel execution of its operations. There are several sources of parallelism in a DCNN that can be leveraged to accelerate its computation. As the chip area and memory bandwidth are limited, it is hardly possible to fully exploit all of the parallelism existing in DCNN inference workload. To maximize the throughput, one has to strike a balance between parallel execution of operations and their demand on system resources. From that perspective, designing an accelerator for DCNNs can be viewed as an optimization problem under the two constraints of memory bandwidth and circuit area.

We present PLACID, an automated PLatform for Accelerator CreatIon for DCNNs, which offers a principled approach to the aforementioned optimization challenge. PLACID uses an analytical approach to characterize and evaluate DCNN accelerator design points, in terms of their resource requirement and computational performance. Subsequently, it explores the implementation space and designs the most efficient accelerator for the given DCNN that would be feasible on the target FPGA platform. Subsequently, the accelerator architecture is generated in the form of register-transfer level (RTL) Verilog code, which can be passed onto commercial tools to implement on the target FPGA. The main contributions of this paper are:

(1) We present a novel accelerator architecture that can utilize different sources of parallelism in a DCNN. We demonstrate that for each convolutional layer, there are two main approaches for data access optimization. We propose an architecture that supports both approaches.
(2) An accurate analytical model for execution of a DCNN on the architecture is developed. PLACID uses the model to explore the design space and to obtain the optimal architecture parameters for the given DCNN and target FPGA platform.
(3) We introduce Fat Convolution Engines (FCE). FCEs are exposed to multiple wide ports and are able to load multiple words in every cycle. This improves the classification throughput by efficient exploitation of the considerable internal bandwidth that FPGAs offer.
(4) We exploit the unique features of modern FPGAs and develop an efficient memory hierarchy that substantially reduces the required off-chip bandwidth.
(5) A working implementation is developed and tested on Altera De1-SoC board. The results of this implementation are used to validate the efficacy of the theoretical model.
(6) We present PLACID, an end-to-end hardware generator which takes a DCNN description as input and generates an RTL level architecture in Verilog.

The rest of this article is organized as follows. Section 2 offers an overview of the related work. In Section 3, DCNNs are reviewed. Section 4 gives a brief overview of the paper. In Section 5, we propose a new memory hierarchy. In Section 6, the convolution engine is explained. Section 7 reports the experimental results, and finally Section 8 concludes the paper.

2 RELATED WORK

Using GPUs is one of the most effective solutions for accelerating DCNNs. Jia et al. developed Caffe [15], which is a powerful platform for training and testing DCNNs. Caffe utilizes GPUs to accelerate DCNNs. Bergstra et al. offered Theano: a Python-based platform for deep learning on GPUs [2]. Subsequently, Abadi et al. [1] offered a tool called TensorFlow, which can be used for efficient implementation of DCNNs on different platforms from mobile devices to large-scale distributed systems. TensorFlow utilizes GPUs for accelerating the computations. Vedaldi [29] developed a toolbox for efficient implementation of DCNNs on MATLAB using GPUs. Collobert et al. offered

Torch7, which utilizes OpenMP/SSE and CUDA-based GPUs [7]. GPUs are very efficient for accelerating DCNNs, especially in the training phase. The main drawback of GPUs is their high power consumption. Therefore, they are not the best solution for using DCNNs on mobile platforms such as drones and robots where power budget is limited.

Another solution for accelerating DCNNs is designing a new ASIC chip. Chen et al. [5] developed DianNao for accelerating neural networks. Their design can achieve the performance of 452GOP/s and its power consumption is 485mW. Subsequently, they continued their research and developed another accelerator for DCNNs, which is called DaDianNao [6]. In this work, they proved that it is possible to achieve a speedup of 450× compared to a high-end GPU. DaDianNao improves DianNao from two aspects: First, the design of DaDianNao is massively biased toward on-chip storage to address the memory issues that DianNao has. Second, the processing pipeline is more powerful compared to their previous work, because it can be reconfigured for each layer and it works for both inference and training phases. Google developed a custom chip for machine learning tasks (Google TPU). The chip is tailored for TensorFlow and Google has deployed it in their data centers since 2015 [16].

The third approach for accelerating DCNNs is FPGA-based acceleration. The reconfigurability of FPGAs makes it possible to change the accelerator for different DCNNs. Moreover, the design and verification processes for a FPGA-based accelerator is time and cost effective compared to an ASIC chip. Researchers have taken a number of approaches to develop an FPGA-based accelerator for DCNN workloads. Specifically, some projects have addressed the bandwidth issue [21], while others addressed the computation aspect of DCNN inference [3, 25]. Chakradhar et al. [4] offered an accelerator for DCNNs, which utilized reconfigurability to improve the throughput. Subsequently, Zhang et al. [30] used a similar approach to accelerate DCNNs. However, they concluded that dynamic reconfigurability is not very efficient. Instead, Zhang et al. used on-chip memory to decrease the required off-chip data transfer. In our previous work, we improved the approach of Zhang et al. [30] by exploiting kernel level parallelism and offering an extended tiling technique [20]. Similar to Chen et al. [5] and Zhang et al. [30], we use tiling and on-chip memory to improve the required off-chip bandwidth. Qiu et al. [22] used kernel level parallelism to accelerate convolutional layers. This approach can be used effectively on DCNNs when their kernel sizes do not vary across layers.

This research advances the previous efforts on several fronts: In the offered architecture, in addition to the efficient use of on-chip memory, we propose Fat Convolution Engines (FCEs). FCEs exploit unique features of modern FPGAs, such as large number of distributed on-chip memory banks to alleviate the bandwidth constraint. In addition, the proposed architecture is characterized by two extra parameters (ω, bus width; P, number of ports) compared to the previous work. This further improves the on-chip bandwidth and offers a rich search space that can be explored for the optimal architecture. Moreover, PLACID does not disregard the fact that a single data access optimization approach does not always yield the highest performance. It offers two different data reusability approaches and selects the one with the higher performance based on the hyperparameters of the target DCNN.

Compression is a prerequisite for designing a high-performance accelerator. Han et al. [12] used pruning to compress the model. They also used Huffman coding and quantization for compressing parameters of DCNNs. Their approach improved the execution time by 3× to 4× and yields up to 7× improvement in energy consumption. Moreover, Gupta et al. [10] showed that networks on datasets like CIFAR-10 [17] can be trained using only 16-bit numbers. Subsequently, Courbariaux et al. [9] showed that a trimmed version of the same network needs only 7-bit digits for execution. Both Gupta et al. [10] and Courbariaux et al. [9] compress DCNNs' parameters by limiting the bit-width. Finally, Gysel et al. offered Ristretto: a platform for compressing DCNNs [11]. Ristretto

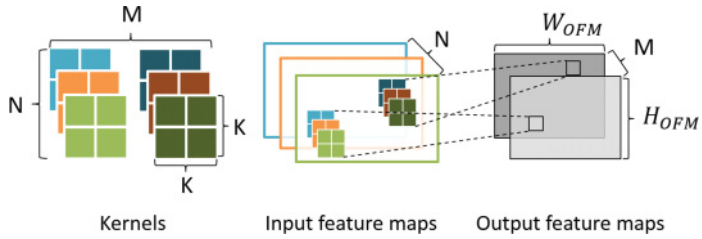

Fig. 1. Feature extraction with convolutional kernels.

Table 1. Definition of Frequently Used Acronyms

Acronyms	Definition
BCU	Buffer Control Unit
DCNN	Deep Convolutional Neural Networks
FCE	Fat Convolution Engine
IFM	Input Feature Map
MAT	Memory Access Time
MIR	Maximized Input Feature Map Reuse
MOR	Maximized Output Feature Map Reuse
OFM	Output Feature Map
PCE	Parallel Convolution Engine
TCT	Total Computation Time
TDT	Total Data Transfer

compresses the network parameters by decreasing their bit-width. We use Ristretto to compress AlexNet before designing an accelerator for it.

3 DEEP CONVOLUTIONAL NEURAL NETWORKS

Large networks have millions of parameters, which are learned during a training procedure. A network trained on images extracts features from its input and uses a classifier to predict the image label. The classification procedure only queries the DCNN's forward path. In this work, we focus on the acceleration of the forward path using reconfigurable logic.

In DCNNs, feature extraction is done by a series of L convolutional layers. Each layer l uses 2D kernel filters that extract features from Input Features Maps (IFM). The result of multiple feature extractions is summed up to form one Output Feature Map (OFM). This process is shown in Figure 1, where two filter banks, each with three kernels, are used to generate two output feature maps. The number of input and output feature maps in layer l is denoted by N^l and M^l, and the size of one OFM is $W^l_{OFM} \times H^l_{OFM}$. Acronyms and Notations are defined in Tables 1 and 2, respectively. One kernel has dimension $K^l \times K^l$, and a layer has $N^l \times M^l$ of these kernels. The feature extraction consists of a series of multiplication-accumulation (MAC) operations, as shown in Algorithm 1. Each output pixel is the sum of the 2D convolutions between the input feature maps and the respective kernels. To generate the neighboring output pixels, the kernel stack is slid across IFMs by stride S^l. In general, stride can have different horizontal (S_x) and vertical (S_y) values. However, for simplicity, we assume ($S_x = S_y$). To the best of our knowledge, in all modern DCNNs, including AlexNet [18], SqueezeNet [14], and GoogLeNet [28], this assumption is valid.

A sample convolutional neural network is shown in Figure 2. This CNN by Krizhevsky et al. [18], which won the ILSVRC 2012 competition [24] consists of eight layers. Input to the first layer

Table 2. Definitions of Frequently Used Notation

Notation	Definition
f	Clock Frequency
K	Width and Height of a kernel
N and M	Number of IFMs and OFMs
P	Number of ports in each on-chip buffer
S	Stride
T_n and T_m	Tile size over IFMs and OFMs
W_{IFM} and H_{IFM}	Width and Height of IFM
W_{OFM} and H_{OFM}	Width and Height of OFM
α	Number of rounds of buffer load/store
β	Buffer size
γ	# Cycles for Convolution
ω	Width of the input bus in FCE

ALGORITHM 1: Pseudo-code of Forward Evaluation of One Convolutional Layer

```
for (m = 0; m < M; m++) do
    for (n = 0; n < N; n++) do
        for (h = 0; h < H_OFM; h++) do
            for (w = 0; w < W_OFM; w++) do
                for (i = 0; i < K; i++) do
                    for (j = 0; j < K; j++) do
                        OFM[m][h][w] += IFM[n][h * S + i][w * S + j] * kernel[m][n][i][j];
                    end
                end
            end
        end
    end
end
```

is a 224 × 224 RGB image. The first five layers are convolutional layers, which extract features from the input image. Layers six and seven are fully connected layers, which serve the same purpose. Finally, layer eight is a classifier with 1,000 output nodes, each of which is a prediction for the corresponding image category.

Neural network parameters such as M and N have different values in different layers. In this section, we used superscript l to show this. As in the remainder of the article our discussion is focused on one specific convolutional layer, we do not use the superscripts l for simplicity.

Given that more than 90% of forward execution time of a DCNN is spent in its convolutional layers [30], in this article, we focus our discussions on their acceleration. That is, our discussion disregards other DCNN layers that are not critical to its computational performance, for example, pooling layers. DCNNs include millions of parameters, hence, they do not typically fit in the on-chip memory. Having an efficient memory hierarchy is a key factor in the performance of FPGA-based accelerators. Such a hierarchy is detailed in Section 5.

4 TEMPLATE ARCHITECTURE

PLACID utilizes an efficient architecture template to synthesize an accelerator. Before detailing this architecture, it is beneficial to briefly overview its components.

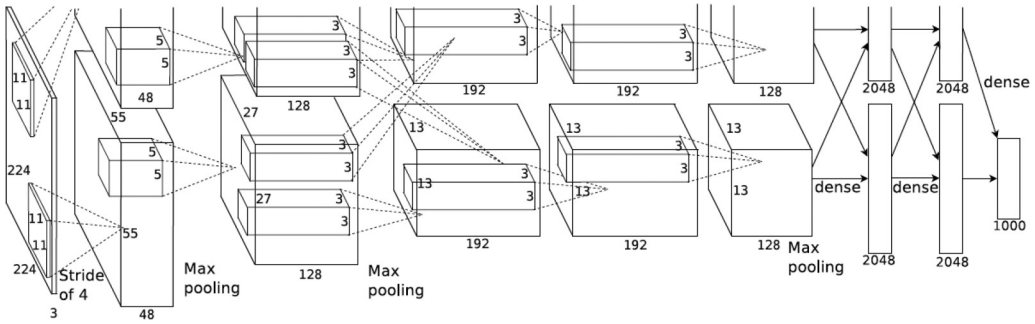

Fig. 2. AlexNet—A sample DCNN that won the ILSVRC 2012 contest [18]. It includes five convolutional layers, two fully connected layers, and one classifier. More than 90% of inference run time is spent in convolutional layers.

- **Memory Hierarchy:** As a DCNN's parameters do not typically fit into on-chip memory, we can only keep a small working set in the on-chip memory. Therefore, we utilize tiling for off-chip data transfer. That is, instead of loading all IFM, we fetch only a subset of them. Moreover, we load a subset of kernels and use the loaded data to perform the convolution. This process continues until all OFM are generated. Tiling is detailed in Section 5.1.
- **On-chip Buffers:** PLACID synthesizes many on-chip buffers. Each IFM, kernel and OFM resides in its dedicated on-chip buffer. The distributed memory hierarchy fits FPGAs very well as they have a large amount of distributed but small BRAMs (Section 5.3).
- **Interface to Processing Elements:** Each processing element is exposed to three different on-chip buffers for IFMs, OFMs and kernels. Access to each buffer takes one clock cycle. Buffers have P ports each of which is ω words wide. Therefore, the processing elements can load $P \times \omega$ words per cycle (Section 5.3). Variables P and ω are architecture parameters. PLACID determines them during design space exploration.
- **Processing Elements:** Convolution consist of a series of MAC operations. Each processing element performs a number of MAC operations in parallel using multiple multiplication units and an adder tree. We refer to the processing elements as fat convolution engines (Section 6.2).
- **Parallel Convolution Engine:** The convolution operation can be applied to different IFMs in parallel. Moreover, we can compute different OFMs independently. We leverage these two sources of parallelism to maximize the throughput. PLACID convolves T_n different IFMs to generate T_m unique OFMs in parallel (Section 6.3).
- **Design Space Exploration:** The proposed architecture is parametrized by (T_n, T_m, P, ω). PLACID performs an exhaustive search in the design space to find the set of parameters that maximizes the throughput (Section 6.7). Subsequently, it synthesizes the optimal architecture.

5 MEMORY HIERARCHY

In this section, a memory hierarchy is proposed that uses the available on-chip memory to minimize off-chip data transfer. The memory hierarchy is built out of distributed Block RAM (BRAM) memory modules that are available in modern FPGAs.

The number of parameters of DCNNs is typically larger than the available on-chip memory. Hence, it is required to divide them into small portions and only load the working set. This process is called tiling.

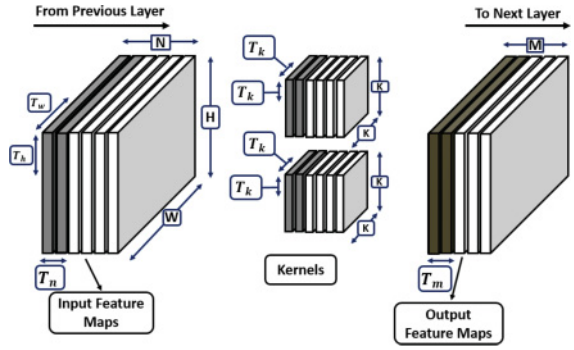

Fig. 3. Tiling in one convolutional layer. Tile sizes for M, N, W, H, and K are shown by T_m, T_n, T_w, T_h, and T_k, respectively.

5.1 Tiling Possibilities and Choices

As proposed in References [4, 20, 30], tiling can be used over M, N, W, H, and K. Figure 3 illustrates these tiling possibilities. However, in practice, tiling over all parameters is not required.

Tiling over K indicates that in each round, instead of loading the entire kernel, one part of the kernel with the size of $T_K \times T_K$ gets loaded (T_K rows and T_K columns). Likewise, tiling over W and H means that instead of loading an IFM, we load T_h rows and T_w columns of that IFM. Tiling over N means loading T_n IFMs instead of N IFMs. Finally, tiling over M means loading $T_m \times T_n$ different kernels and generating T_m different OFMs instead of loading all kernels and generating all of the OFMs.

In our previous work [20], we proposed the use of tiling over K for DCNNs with large kernel size. However, recent trends in DCNNs [13, 26, 28] suggests that kernel sizes are not large, and their size is not increasing. Therefore, using tiling over K is not beneficial.

Zhang et al. [30] proposed using tiling over row and columns of IFMs. In practice, as the size of on-chip memory, even on small platforms, is much larger than the size of an IFM in any DCNN, tiling over rows and columns is not necessary.

Moreover, another important issue with this proposal is that it scrambles the data access pattern. Normally, IFMs are stored in row-major or column-major format. In either case, reading an IFM requires access to continuous memory addresses. Hence, it is possible to use burst, and memory access is very efficient. However, as it is shown in Figure 3 using tiling over rows and columns destroys this pattern and causes inefficient memory accesses. It is worth mentioning that memory bandwidth varies with access pattern. Accessing scattered memory locations decreases the bandwidth drastically. Note that due to kernel overlap, off-line data alignment for improving access pattern is not very efficient. Note that typically stride is smaller than kernel dimensions. As a result, computation of adjacent pixels in an OFM would require loading of overlapping regions of IFMs.

Finally, reproducing the results that are reported in References [20, 30] makes it clear that the optimal solutions of both works are achieved without tiling over W, H, and K.

Even though tiling over W, H, and K is not necessary, tiling over M and N is essential. Tiling over M and N means that in each round T_n unique IFMs and $T_m \times T_n$ different kernels are loaded and T_m different OFMs are generated. Thus, the required on-chip memory, β, for IFMs, OFMs, and kernels can be computed using Equations (1), (2), and (3), respectively:

$$\beta_{IFM} = T_n \times W_{IFM} \times H_{IFM}, \tag{1}$$

$$\beta_{OFM} = T_m \times W_{OFM} \times H_{OFM}, \tag{2}$$

$$\beta_{Ker} = T_m \times T_n \times K^2. \tag{3}$$

In the aforementioned equations β_{IFM}, β_{OFM}, and β_{Ker} are the required buffer sizes. For the rest of the parameter definitions, please refer to Table 2.

5.2 Tradeoff Between IFM versus OFM Reuse

One of the byproducts of tiling is the creation of intermediate results, which complicates data management. It seems intuitive to keep the intermediate results in the on-chip memory as long as it is required and write them back to the off-chip memory only after they are finalized. However, as this solution decreases the reusability of IFMs and kernels, maintaining intermediate results in the on-chip memory is not always the best approach. In other words, if the size of IFMs is larger than the size of intermediate results, it might be inefficient to keep partial results in the on-chip memory. In what follows, we offer two different solutions: in the first solution IFMs are reused, and in the second solution OFMs are reused. We show that the performance of the two solutions depends on the DCNN characteristics and the available resources on the target FPGA platform. PLACID can generate an accelerator based on either approach.

5.2.1 Maximized IFM Reuse (MIR). In this case, the intention is to load each IFM only once. Therefore, when an IFM gets loaded, all convolutions that require any data from that specific IFM should finish before removing it from the on-chip memory. Depending on the number of kernels and the target FPGA, some partial results may need to be written back to the memory. The pseudocode that is shown in Algorithm 2 demonstrates the implementation of this approach: IFMs are loaded in the outermost loop and are convolved with all of the kernels before a new set of IFMs are loaded. In this scenario Equations (4), (5), and (6) show the number of rounds that are required to load/store IFMs, OFMs, and kernels, respectively. The tile size for IFMs is T_n. That is, only T_n IFMs can reside in the on-chip memory simultaneously. Hence, it takes $\frac{N}{T_n}$ rounds to load all IFMs. Note that this parameter that is shown by α_{IFM} is not necessarily an integer number. As a case in point, when N is 5 and T_n is 2, the value of α_{IFM} will be 2.5. That is, in the first two rounds IFM buffers will be loaded at their full capacity and in the last round IFM buffers will be loaded at their half capacity.

In each round of loading IFMs, it is necessary to load all kernels. For M kernels, when the tile size is T_m, this process takes $\frac{M}{T_m}$ steps. Hence, Equation (6) computes the total number of times that kernels are loaded. The same analysis is valid for Equation (5). The coefficient of two in this

ALGORITHM 2: MIR Pseudocode. IFMs are Loaded in the Outermost Loop and are Convolved with all Kernels before a New Set of IFMs are Loaded. Store_OFMs() Stores Values of OFM Buffers to the Off-chip Memory Which may be Partial or Final Results.

```
for (n = 0; n < N; n += Tn) do
    load_IFMs();
    for (m = 0; m < M; m += Tm) do
        load_kernels();
        load_OFMs(); //partially computed OFMs
        convolution();
        store_OFMs(); //may be partial or final
    end
end
```

ALGORITHM 3: MOR pseudocode. The Main Idea is to Avoid Generating Partial Results. It is Required to Load IFMs in the Innermost Loop. Therefore, Neither IFMs nor Kernels are Reused in Computing Different OFMs.

```
for (m = 0; m < M; m += Tm) do
    for (n = 0; n < N; n += Tn) do
        load_IFMs();
        load_kernels();
        convolution();
    end
    store_OFMs();
end
```

equation indicates that the partial OFM results will be loaded and stored again:

$$\alpha_{IFM} = \frac{N}{T_n}, \tag{4}$$

$$\alpha_{OFM} = 2 \times \left\lceil \frac{N}{T_n} \right\rceil \times \frac{M}{T_m}, \tag{5}$$

$$\alpha_{Ker} = \left\lceil \frac{N}{T_n} \right\rceil \times \frac{M}{T_m}. \tag{6}$$

5.2.2 Maximized OFM Reuse (MOR). In this case, the intention is to avoid writing back partial results. The pseudocode shown in Algorithm 3 demonstrates the implementation of this approach. As we do not want to write back partial results, it is required to load IFMs in the innermost loop. Therefore, neither IFMs nor kernels get reused in computing OFMs in different rounds. Similar to MIR approach, the number of rounds that are required to load IFMs, OFMs, and kernels are shown in Equations (7), (8), and (9), respectively:

$$\alpha_{IFM} = \left\lceil \frac{M}{T_m} \right\rceil \times \frac{N}{T_n}, \tag{7}$$

$$\alpha_{OFM} = \frac{M}{T_m}, \tag{8}$$

$$\alpha_{Ker} = \left\lceil \frac{M}{T_m} \right\rceil \times \frac{N}{T_n}. \tag{9}$$

MIR and MOR are two different approaches for data reuse. The choice of data reuse policy impacts the architecture of a DCNN accelerator. In this article, we use MIR architecture to refer to an accelerator that is designed based on MIR data reuse. Likewise, MOR architecture is used to refer to a design based on the MOR data reuse approach.

5.3 Memory Hierarchy Architecture

PLACID exploits the unique features of modern FPGAs, specifically the large number of distributed but small BRAM memory modules, to design a highly efficient memory hierarchy.

In what follows, we describe the memory architecture that PLACID generates. This architecture utilizes many small on-chip buffers. Each buffer has P ports that are ω words wide. Hence Equation (10) gives the total number of words that convolution engine can fetch in each cycle:

$$\#words\ per\ cycle = (T_n + T_m + T_m \times T_n) \times P \times \omega. \tag{10}$$

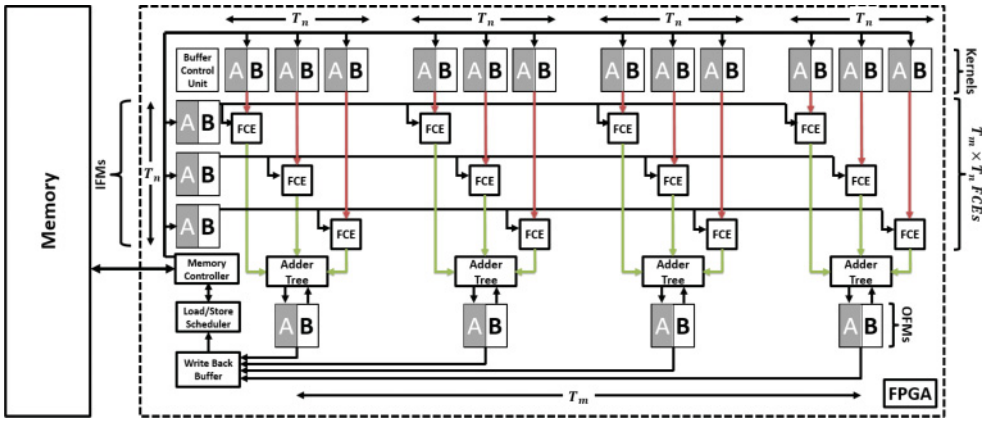

Fig. 4. Memory Hierarchy for $P = 1$, $T_m = 4$, and $T_n = 3$. Different IFMs, OFMs, and kernels reside in separate on-chip buffers. There are two buffer banks: The gray bank (A) and the white bank (B). While the first buffer bank is engaged in the convolution operation, data is loaded to the second bank. Convolution is performed in Fat Convolution Engines (FCE). FCEs are discusses in Section 6.2.

Even though the number of loads and stores in the MIR and MOR architectures are different (Sections 5.2.1 and 5.2.2), the required on-chip memory for both is the same (Section 5.1). Figure 4 depicts the memory architecture. The size of each IFM buffer is $H_{IFM} \times W_{IFM}$, and there are T_n independent buffers that each of them holds one IFM. Using T_n separate buffers is a better choice compared to a single buffer that is T_n times larger, because it provides a larger on-chip bandwidth. Disregarding the size, each on-chip memory at most has P ports. Therefore, using a large on-chip buffer decreases the parallelism. The size of the smallest on-chip memory that can be used conveniently is $H_{IFM} \times W_{IFM}$. Using smaller blocks is possible but needs a sophisticated boundary control unit. Likewise, for kernels, there are $T_n \times T_m$ on-chip memories with size K^2. Finally, there are T_m memory blocks with the size of $H_{OFM} \times W_{OFM}$ for storing OFMs. Different IFMs, OFMs, and kernels reside in separate on-chip buffers. The proposed memory system provides the convolution engine with the luxury of simultaneous access to all on-chip buffers.

5.3.1 Overlapping Communication and Computation. To hide the latency of off-chip memory, we overlap computation with communication. To realize this, PLACID implements two buffer banks. In Figure 4, bank A is colored gray, and bank B is colored white. While convolution is performed between IFMs and kernels from one buffer bank, another buffer bank is idle. Hence, the next tile of data can be loaded in the idle buffer at the same time.

5.3.2 Load/Store Scheduler. The Load/Store Scheduler (LSS) is in charge of providing the memory controller with the required addresses. LSS is aware of the global state of the system. Therefore, it can generate appropriate sets of addresses for loading different parameters (IFMs, OFMs, or kernels) when they are required. LSS also generates handshaking signals, which are required for communication with the memory controller.

5.3.3 Buffer Control Unit. The Buffer Control Units (BCU) are responsible for providing required signals for on-chip buffers. A BCU is aware of the global state of the system and ensures that each buffer is in an appropriate state all the time. For instance, when the data that belongs to IFM#0 is on the bus, the BCU needs to make sure that: (1) None of the buffers except IFM#0 is

in the write state. (2) IFM#0 has the correct address. Moreover, the LSS ensures that the buffers connected to convolution engine are meeting convolution engines's data requirements.

5.3.4 Write Back Unit. The Write Back Unit (WBU) writes partial and final results to the off-chip memory. Depending on the architecture (MIR or MOR), the write back unit writes OFMs back to memory at different times. WBU is aware of the global system state to make this decision. In the write back task, each time the memory controller asserts the wait-request signal, WBU stops the write back process and resumes it when the wait-request signal is de-asserted. However, each new read request from an on-chip memory has a few cycle initial latency. Therefore, to avoid this latency, PLACID uses FIFOs for each write-back bus. As a result of this optimization, in the same clock cycle that the wait-request signal is de-asserted, WBU can continue the write-back process.

6 COMPUTATION ENGINE

In this section, the proposed architecture for the convolution engine is detailed. Also, we discuss why the proposed architecture is well-suited for DCNNs. The architecture parameters such as the number of memory banks vary based on the target DCNN and FPGA. PLACID determines these parameters during the Design Space Exploration (DSE) process. DSE refers to the process of investigating alternative design points before implementation.

6.1 Computational Requirements

As we showed in our previous work [20], there are three main sources of parallelism in DCNNs: intra kernel, intra output, and inter output parallelism. To explain, consider that all OFMs can be computed in parallel (inter output parallelism). Moreover, computing different pixels in the same OFM is a perfectly parallel workload (intra output parallelism). Furthermore, computing the value of each pixel is a dot product, an operation that can be performed in parallel (intra kernel parallelism). A good architecture should be able to take advantage of aforementioned parallelism sources. It is worth mentioning that due to the limited resources on an FPGA chip, it is not possible to fully utilize the aforementioned parallelism sources. The DSE process aims to strike a balance between accelerator performance and its resource requirement.

PLACID uses a Moore state machine [19] to determine which buffers are used for what operations in each state. At any given time, each active buffer is in one of the following states: data is loaded/stored or data is forwarded to computation engine. MIR and MOR architectures have different state machines that are detailed below. In both architectures, to optimize memory performance, two types of memory accesses are explored:

(1) *Non-critical Memory Access:* When loading a buffer is not critical, the load process can be split over multiple convolution rounds (i.e., in each round only a portion of IFMs gets loaded).
(2) *Critical Memory Access:* In this case, a buffer has to be loaded before the next convolution round. When loading a buffer is critical, the next convolution cannot start before the end of the current load process.

In the MIR architecture, loading IFMs (except the first round) are not critical. Loading kernels, OFMs, and storing OFMs are critical. In the MOR architecture, all stores are not critical, and all loads are critical. As a case in point, in the MIR architecture, the required time for loading IFMs can get amortized over $\lceil M/T_m \rceil$ convolution rounds. In other words, the next tile of IFMs is required only after all kernels are loaded. Hence, loading the next tile of IFMs has a lower priority compared to kernels and can be delayed.

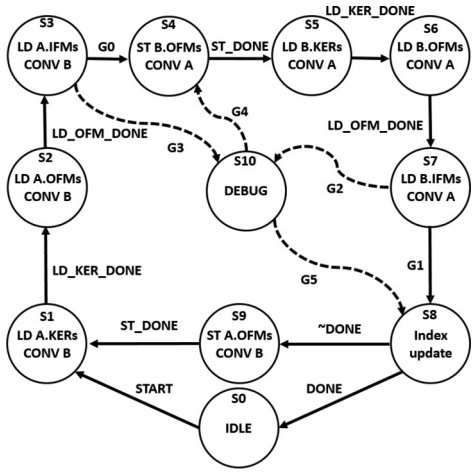

Fig. 5. MIR control logic.

6.1.1 Control Logic. Figure 5 illustrates the proposed control logic for the MIR architecture. The MOR architecture has an analogous state machine.

When there is no image to process, the accelerator is in the *IDLE* state. The *START* signal guards the transition from the *IDLE* state to S1. When the next image is ready for the process, the start signal is asserted, and the process begins loading kernels (*LD A.KERs*).

As discussed, PLACID overlaps computation with communication. Therefore, in each round while data is being loaded into one buffer bank, another buffer bank is engaged in convolution operation. However, in the first round when buffer bank A is getting loaded, convolution cannot be performed on buffer bank B, because it does not have any values yet. The process continues with loading OFMs (*LD A.AFMs*) and IFMs (*LD A.IFMs*). The guard for transiting from states *S3* to state *S4* is *G0*. To implement the non-critical memory access model, *G0* behaves as follows:

In each round, for different tiles of OFM except the last one, *G0* makes the state transition possible as soon as the *CONV B* is finished (see Algorithm 2). However, for the last tile of OFM, *G0* makes the state transition possible only if *CONV B* and *LD A.IFMs* are both finished. *G0* is defined in Equation (11). For notation definition please refer to Table 2.

$$G0 = \begin{cases} \textbf{LD\_IFM\_DONE} \ \& \ \textbf{CONV\_DONE} & m \geq M - Tm \\ \textbf{CONV\_DONE} & m < M - Tm \end{cases}. \tag{11}$$

The implementation of *G1* is analogous to *G0*. In Figure 5 all states except *DEBUG* are mandatory. Even though the state machine that is shown in Figure 5 gives a detailed understanding of the architecture, it does not include all of the details. As a case in point, *LD *.IFMs* has a few sub-states that handle loading of different IFMs. Moreover, another state machine determines the buffer that is used by convolution engines in each round. For example, in the MOR architecture, in different rounds, we switch between *A.IFM, A.KERs* and *B.IFM, B.KERS*; however, we do not necessarily switch between *A.OFM* and *B.OFM* (see Algorithm 3).

6.2 Fat Convolution Engine

Fat Convolution Engines (FCE) are the smallest computing units in the proposed architecture. These computation units exploit Intra kernel Parallelism by parallelizing each convolution operation. Figure 6 shows the architecture of an FCE. $2 \times P$ ports are concurrently exposed to the same

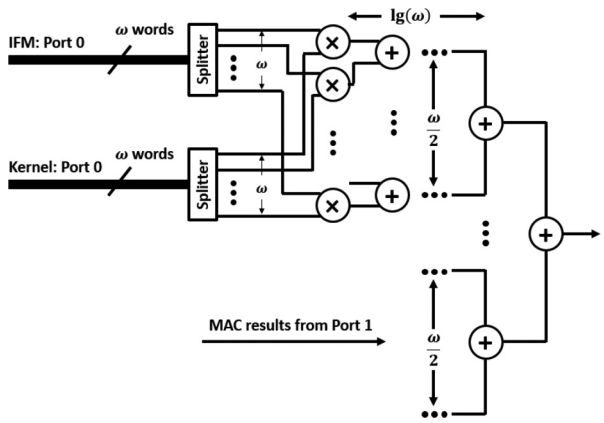

Fig. 6. Fat Convolution Engine. In every cycle, it loads $P \times \omega$ words and splits them. Subsequently, it performs multiplication and addition in a pipelined fashion.

FCE. P ports are connected to an IFM buffer and the other are connected to a kernel buffer. As Figure 6 illustrates, each input bus is ω words wide. Therefore each FCE can fetch $P \times \omega$ words per cycle, where P is the number of ports per on-chip buffer.

In every cycle, a new part of a kernel and an IFM are loaded. Subsequently, the split task occurs seamlessly via routing. Next, the multiplication is performed for different operands at the same time. Finally, the results are accumulated using an adder tree. Both multiplication and addition units are pipelined.

FCEs are designed to match the architecture of on-chip memories. They leverage wide buses to load multiple words per cycle. In addition, they can use all of the available ports of an on-chip memory module.

6.3 Parallel Convolution Engine

FCEs utilize Intra Kernel Parallelism to accelerate the computation. A Parallel Convolution Engine (PCE), which is designed using multiple FCEs, takes advantage of two parallelism sources: Intra Output and Inter Output. In the former parallelism source, multiple IFM are convolved in parallel. The results are summed up to form one OFM. In the latter parallelism source different OFMs are computed simultaneously.

Figure 7 illustrates the architecture of a PCE. In a PCE, $T_n \times T_m$ replicas of FCE work concurrently on T_n different IFMs to generate T_M different OFMs. There is a single control unit that controls the system. In each OFM buffer, one port is used for reading the previous value and the other for writing back the new value (accumulation process).

6.4 Fully Connected Layers

Fully connected layers are only responsible for 0.11% of the execution time of AlexNet [8]. This number is even smaller for more recent CNNs. As a case in point, GoogLeNet has 57 convolutional layers and only one fully connected layer [28]. In comparison, AlexNet has five convolutional layers and three fully connected layers. Hence, based on the Amdahl's law, accelerating the fully connected layers would not have a considerable impact on the total execution time.

PLACID treats fully connected layers as convolutional layers whose kernel dimensions are 1×1. In other words, a fully connected layer, is a convolutional layer in which, values of W_{IFM}, H_{IFM}, W_{OFM}, H_{OFM}, and K are equal to one (please refer to Table 2 for notation definitions). For fully

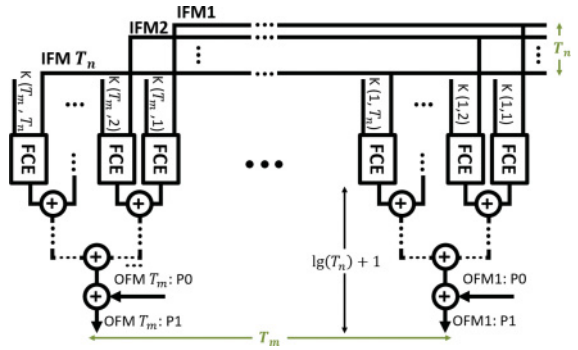

Fig. 7. Parallel Convolution Engine. $T_n \times T_m$ FCEs work concurrently to generate T_m distinct OFMs.

connected layers, each input node is an IFM and each output node is an OFM. In such layers, N and M refer to the number of input nodes and the number of output nodes, respectively. Therefore, it is possible to use the convolution engines for executing fully connected layers without any changes.

6.5 Computation Model

In this section, the throughput of the proposed accelerator is analytically analyzed. Throughput is constrained by two factors: the off-chip memory bandwidth and total computation time.

The number of cycles for one convolution round is shown in Equation (12). Equation (13) shows the total computation time for all convolutions in a layer. In what follows, we explain these two equations.

In one convolution round a kernel is convolved with one IFM. Given that the kernel size is K^2, the required number of multiplications in each convolution round is $W_{OFM} \times H_{OFM} \times K^2$. Assuming that there are P ports each of which is ω words wide, it is possible to fetch $P \times \omega$ words per cycle from on-chip buffers. Hence, $P \times \omega$ multiplications can be performed per cycle. The multiplication results are summed using an adder tree of depth $\lceil lg(P \times \omega) \rceil$. Therefore, Equation (12) shows the number of cycles for one convolution round:

$$\gamma = W_{OFM} \times H_{OFM} \times \left\lceil \frac{K^2}{P \times \omega} \right\rceil + \lceil lg(P \times \omega) \rceil. \tag{12}$$

Given that f shows the working frequency of the circuit, the required time for one convolution round is $\frac{\gamma}{f}$. As there are $\lceil M/T_m \rceil \times \lceil N/T_n \rceil$ rounds, Equation (13) gives the Total Computation Time (TCT):

$$Total\ Computation\ Time\ (TCT) = \left\lceil \frac{M}{T_m} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \frac{\gamma}{f}. \tag{13}$$

Another important factor in performance is the required data transfer. Data transfer depends on the size of buffers (β_{OFM}, β_{KER}, and β_{IFM}) and the number of times that they are loaded or written back (α_{OFM}, α_{KER}, and α_{IFM}). Equation (14) computes Total Data Transfer (TDT) to off-chip memory in words. Assuming that each word is x bytes, the required data transfer for a convolutional layer is $(TDT \times x)$ bytes. Memory Access Time for loading the required data of one convolutional layer can be computed using Equation (15). Note that the value of *bandwidth* varies based on the memory access pattern. We use an average bandwidth in a steady state mode to compute MAT. PLACID overlaps computation with communication. Hence, the required time for running a convolutional layer is $Max\ \{TCT, MAT\}$. In addition, as it is detailed in Algorithm 1, the total number of MAC operations in each layer is: $N \times M \times W_{OFM} \times H_{OFM} \times K^2$. Since each MAC consists of two basic

operations (a multiplication and an addition), Equation (16) computes the throughput:

$$TDT = \alpha_{OFM} \times \beta_{OFM} + \alpha_{KER} \times \beta_{KER} + \alpha_{IFM} \times \beta_{IFM}, \tag{14}$$

$$Memory\ Access\ Time\ (MAT) = \frac{TDT \times x}{bandwidth}, \tag{15}$$

$$Throughput = \frac{2 \times N \times M \times W_{OFM} \times H_{OFM} \times K^2}{Max\{TCT, MAT\}}. \tag{16}$$

6.6 Throughput Optimization

The problem of finding the highest throughput is an optimization problem under two constrains: bandwidth and chip area. As Equation (16) shows, the throughput is hindered by two parameters: TCT and MAT. The former is a function of the chip area and the latter is a function of both memory bandwidth and chip area. Decreasing one of these two comes at the cost of increasing the other.

Some designs yield a better TCT while others offer a better MAT. Architectures with a better TCT use the on-chip resources to create a higher number of convolution engines. This improves their computational capability, which results in a better TCT. However, some other architectures use the on-chip resources for developing advanced scratch-pad memories. Such architectures yield a better MAT by reducing the required off-chip memory accesses. For a given platform, the available on-chip resources and the memory bandwidth are constant. Therefore, the challenge in optimizing the throughput is to strike a balance between TCT and MAT. To find such a balance, it is required to thoroughly search the design space based on the design parameters (T_n, T_m, P, ω).

In what follows, we detail the process of Design Space Exploration (DSE) to obtain a design that gives the highest throughput.

6.7 Design Space Exploration

We explore the design space to determine the set of parameters which maximize the throughput of the accelerator. Our design space is parametrized by (T_n, T_m, P, ω). Moreover, we need to choose between maximizing IFMs or OFMs reuse (MIR/MOR). The design space exploration problem has the following conditions:

(1) **Number of DSP Blocks:** Each platform has a limited number of DSP blocks. The accelerator has $T_m \times T_n$ different FCEs each of which needs $P \times \omega$ multipliers (Equation (17)):

$$T_m \times T_n \times P \times \omega \leq \#DSPBlocks. \tag{17}$$

(2) **On-chip Memory:** Another constraint is the available on-chip memory. The required memory for all buffers should be less than the available on-chip memory (Equation (18)). The coefficient of 2 models double buffering:

$$2 \times (\beta_{IFM} + \beta_{OFM} + \beta_{Ker}) \leq On\text{-}chip\ Memory. \tag{18}$$

(3) **Intra kernel Parallelism:** The amount of intra kernel parallelism cannot be larger than the size of convolution kernel. Hence, it has to be constrained according to Equation (19):

$$P \times \omega \leq K^2. \tag{19}$$

(4) **Number of Ports:** Each platform has a limited number of ports per memory bank. In most FPGAs, each bank has at most two ports (i.e., $P < 2$).

Table 3. Parameters of a Sample Architecture

T_m	T_n	P	ω	MIR/MOR
8	2	2	1	MIR

Table 4. AlexNet Parameters

Layer	N	M	H_{IFM}	W_{IFM}	K
1	3	96	224	224	11
2	96	256	55	55	5
3	256	394	27	27	3
4	394	394	13	13	3
5	394	256	13	13	3

PLACID searches the design space exhaustively to find the optimal parameters for accelerator design. An FPGA chip can be responsible for accelerating all convolutional layers or one of them. Therefore, PLACID is designed to search for both layer specific and global solutions. In the global case, a static accelerator is generated that can perform the convolution for all layers. The size of design space for the static case is 115,000. Thus, the exhaustive search is fast (less than a minute).

7 EXPERIMENTAL RESULTS

In the first part of this section, the accuracy of the analytical model is evaluated. To do so, PLACID is used to generate an architecture for accelerating AlexNet on an Altera Cyclone V FPGA platform. The analytical model is used to predict the throughput. Next, the generated accelerator is synthesized and implemented on Altera DE1-SoC and its throughput is measured. Experimental results show that the implementation follows the analytical model accurately.

In the second part, PLACID is used to generate architectures for two different platforms. Finally, the theoretical performance of the generated architectures is compared with the state of the art research.

It has been proven that most neural networks can be implemented using fixed point arithmetic. Qiu et al. [22] implemented VGG [26] using 16-bit fixed point multipliers with less than 2% loss in classification accuracy. Rastegari et al. [23] offered an implementation of AlexNet with only binary weights, with 2.9% accuracy drop. We used Ristretto [11] to compress the bit-width for AlexNet from 32-bit floating point to 16-bit fixed point. In this compression, the classification accuracy dropped from 56.9% to 55.59%. We have performed all experiments using 16-bit fixed point arithmetic. It is worth mentioning that we only decreased the bit-width of AlexNet's parameters. In other words, the original architecture of the neural network is preserved.

7.1 Analysis of a Sample Implementation

PLACID is used to generate a sample architecture on Altera DE1-SoC board for accelerating AlexNet [18]. Table 3 shows parameters of this architecture and Table 4 reports AlexNet's parameters.

The Altera DE1-SoC platform is equipped with Cyclone V FPGA whose specifications are shown in Table 5. This platform has a Cortex-A9 hard processor. The processor is used only for debugging purposes.

DE1-SoC has two different memory chips: one DDR3 memory that is interfaced to the hard processor and one SDRAM memory block that is interfaced to the FPGA logic. We have used

Table 5. Altera Cyclone V Specifications

ALMs	Total Registers	BRAM (bit)	DSP Blocks
32,070	14,399	4,065,280	87

Table 6. Expected Versus Achieved Throughput

Layer	1	2	3	4	5	Average
Expected (GOPS)	4.76	3.47	1.23	1.24	1.24	11.94
Achieved (GOPS)	4.58	3.22	1.16	1.17	1.18	11.31
Difference	3.80%	7.20%	5.70%	5.65%	4.84%	5.28%

Table 7. FPGA Resource Utilization for the Architecture Shown in Table 3 on Altera Cyclon V

Resource	Adaptive Logic Module (ALM)	DSP	Block Memory (bits)	Registers
Used	11,043	28	2,139,006	14,578
Available	32,070	87	4,065,280	128,300

Table 8. Best Architecture Parameters and Theoretical Throughput for Accelerating AlexNet on Xilinx XC7VX485T. The Specifications of this FPGA Is Shown in Table 10

Layer	Layer Specific Solution							Static Solution						
	T_m	T_n	P	ω	GOPS	Type	Bound	T_m	T_n	P	ω	GOPS	Type	Bound
1	96	3	1	9	498	MIR	Computation	128	4	1	5	278	MOR	Computation
2	29	96	1	1	546	MIR	Computation	128	4	1	5	511	MOR	Computation
3	384	7	1	1	529	MOR	Computation	128	4	1	5	446	MOR	Communication
4	77	12	1	3	546	MOR	Computation	128	4	1	5	451	MOR	Computation
5	64	43	1	1	529	MOR	Computation	128	4	1	5	451	MOR	Computation
Total					536							445		

Altera SDRAM controller IP to connect the accelerator to the SDRAM. The system operates at a clock frequency of 100MHz. At this frequency, the average off-chip memory bandwidth is 146MB/s.

Equation (16) is used to compute the expected throughput for each layer of AlexNet on this accelerator. Subsequently, the resulting throughput of the implementation on Altera DE1-SoC board is measured for each layer of AlexNet. The expected throughput and achieved throughput are compared in Table 6. The analytical model predicts the throughput very accurately. The small error of 7.2% is due to fluctuation in memory bandwidth as well as control and initialization time that are ignored in the analytical model. One of the advantages of RTL level design is predictability of the architecture. Hence, it is possible to offer an accurate model for the performance. Such a model is instrumental in design optimization.

7.2 Performance Comparison

Two architectures are generated for accelerating AlexNet on Xilinx XC7VX485T and Altera Cyclone V FPGAs. Tables 8 and 9 show the optimal architecture parameters and theoretical throughput numbers. Cyclone V is a low-end FPGA chip and XC7VX485T is a high performance FPGA. We will show that PLACID achieves state of the art performance per density on both FPGAs.

Table 9. Best Architecture Parameters and Theoretical Throughput for Accelerating AlexNet on Altera Cyclone V

Layer	Layer Specific Solution							Static Solution						
	T_m	T_n	P	ω	GOPS	Type	Bound	T_m	T_n	P	ω	GOPS	Type	Bound
1	1	3	1	25	14.511	MIR	Computation	37	2	1	1	9.6	MOR	Computation
2	37	2	1	1	14.625	MOR	Computation	37	2	1	1	14.625	MOR	Computation
3	1	64	1	1	12.766	MIR	Computation	37	2	1	1	7.144	MOR	Communication
4	35	2	1	1	13.927	MOR	Computation	37	2	1	1	13.927	MOR	Computation
5	37	2	1	1	14.590	MOR	Computation	37	2	1	1	14.590	MOR	Computation
Total					14.175							12.112		

Table 10. Xilinx VX485T Specifications

LUT	FF	BRAM (bit)	DSP Blocks
303600	607200	37,969,920	2800

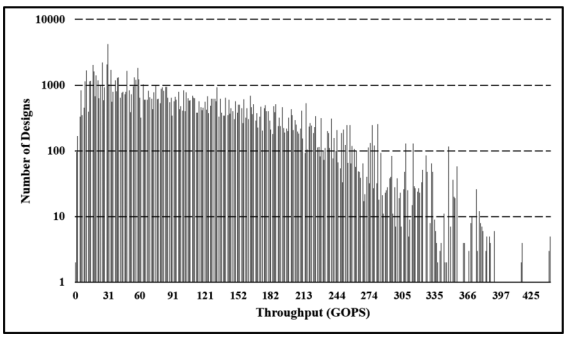

Fig. 8. A histogram of the design space for accelerating AlexNet on XC7VX485T.

Table 9 shows the architecture with the highest throughput and lowest off-chip communication. This solution is not unique. A histogram of the architecture pool for accelerating AlexNet on XC7VX485T is shown in Figure 8. There are different architectures that yield equal throughput and require equal off-chip memory access. Moreover, there are many architectures that have near optimal throughput with a smaller amount of off-chip data transfer. In our design space, there are 13 different candidates within 1% of the optimal solution. PLACID offers a large architecture pool for a given DCNN and FPGA; hence, one can further optimize the DSE process to generate an architecture that satisfies additional requirements (e.g., minimum power consumption).

Table 8 shows two solutions: layer specific and static. The layer specific solution shows architecture parameters that maximize the throughput for each layer. For example, the best architecture parameters for the first layer of AlexNet is $(T_m, T_n, P, \omega) = (96, 3, 1, 9)$. Using these parameters in a MIR architecture yields a throughput of 498 GOPS for the first layer of AlexNet. Layer specific solutions are loose upper bounds for highest achievable throughput of a convolutional layer on a certain FPGA. To realize such a solution, one has to make the architecture dynamically reconfigurable as the optimal parameters may change from one layer to the next. For example, from the first layer to the second layer, architecture parameters (T_m, T_n, P, ω) change from $(96, 3, 1, 9)$ to

Table 11. Performance Comparison Between MIR and MOR for Accelerating AlexNet on Xilinx XC7VX485T

Architecture Type	T_m	T_n	P	ω	Throughput (GOPS)
MIR	32	17	1	5	247
MOR	128	4	1	5	445

Table 12. Performance Comparison

	ISCA2010 [4]	ISFPGA2015 [30]	ICCD2013 [21]	ASPDAC2016 [20]	ISFPGA2016 [27]	Proposed Solutions	
						Cyclone V	VX485T
Precision	48-bits fixed	32-bits float	fixed point	32-bits float	8-16-bit fixed	16-bits fixed	16-bits fixed
Frequency	200 MHz	100 MHz	150 MHz	100 MHz	120 MHz	100 MHz	100 MHz
FPGA Chip	SX240T	VX485T	VLX240T	VX485T	Stratix-V (G8D8)	Cyclone V	VX485T
CNN Size	0.52 GOP	1.33 GOP	5.48 GOP	1.33 GOP	30.9 GOP	1.33 GOP	1.33 GOP
DSP Blocks	1056	2800	768	2800	1963	87	2800
Performance	16 GOPS	61.62 GFLOPS	17 GOPS	84.2 GFLOPS	136.5 GOPS	12.11 GOPS	445 GOPS
MOPS/DSP	15.15	22	22.14	30.07	69.54	**139.20**	**158.99**

(29, 96, 1, 1). Zhang et al. [30] proved that it is inefficient to support dynamic reconfigurability due to its heavy overhead. Hence, the selected architecture must have a fix set of parameters across different layers.

In the DSE process, among all proposals with a static architecture, we select the one with the highest throughput. This solution is reported under static solution in Table 8. We performed the same analysis for Altera Cyclone V platform and the results are shown in Table 9.

Based on the DSE results, the MOR architecture yields the highest throughput for accelerating AlexNet on the two considered platforms. The best MIR and MOR architectures are shown in Table 11 for comparison purposes.

Finally, Table 12 compares the performance of architectures generated by PLACID with the competing approaches. The performance density of our two architectures is at least 2× better than the state of the art solution.

The performance of accelerating AlexNet on Xilinx XC7VX485T is 7.2× improved compared to the implementation that is offered by Reference [30]. Moreover, the achieved performance on Xilinx XC7VX485T is improved 5.29× compared to the DSE performed by Motamedi et al. [20].

As Gysel et al. showed in Reference [11] using 16-bit fixed point arithmetic instead of 32-bit floating point decreases the classification accuracy by less than 1.5%. Utilizing 16-bit numbers increases the bandwidth by 2×. Moreover, each 32-bit floating point MAC operation needs two DSP units. However, each 16-bit fixed point MAC operation needs half of a DSP unit. Therefore, using 16-bit fixed point arithmetic can potentially increases the number of processing elements by 4×. Taking advantage of these opportunities is the main reason for the large gap between the proposed solution and References [20, 30]. In addition, the performance density is improved 2.3× compared to Reference [27]. This improvement is mainly due to use of FCEs, which fully exploit the available on-chip bandwidth.

We did not include Reference [22] in Table 12, since their approach relies on fixed kernel sizes. This causes a high performance in accelerating DCNNs with fixed kernel size (e.g., VGG [26]) but a poor performance in accelerating DCNNs with variable kernel size, such as AlexNet [18], GoogLeNet [28], and ResNet [13].

8 CONCLUSIONS

In this article, we presented PLACID: a platform for accelerator creation for DCNNs. We explained that accelerators that are designed by PLACID utilize various parallelism sources to effectively expedite a DCNN. Moreover, we offered a performance estimation model for computing the throughput of DCNN accelerators. Using this model, PLACID computes the throughput of all of the possible architectures for accelerating a DCNN on a given FPGA chip. Subsequently, it selects and synthesizes the one with the highest throughput. Experimental results show that accelerators designed by PLACID have a superior performance density compared to state-of-the-art approaches.

ACKNOWLEDGMENTS

This work is partly funded by National Science Foundation (NSF) under Grant No. CCF-1346812 and the Seattle Foundation. We thank Altera for providing the FPGAs that made this research possible. Moreover, we thank NVIDIA for donating GPUs, which were used for convnet compression. We are also grateful to Mr. Terry O'Neil for his insightful comments.

REFERENCES

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeronet al. 2011. Theano: Deep learning on gpus with python. In *Proceedings of the BigLearning Workshop (NIPS'11), Granada, Spain*, Vol. 3. Citeseer.

[3] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. 2010. A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 273–284.

[4] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 247–257. DOI: http://dx.doi.org/10.1145/1815961.1815993

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, Vol. 49. ACM, 269–284.

[6] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sunet al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[7] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *Proceedings of the BigLearn, NIPS Workshop*.

[8] Jason Cong and Bingjun Xiao. 2014. Minimizing computation in convolutional neural networks. In *Proceedings of the International Conference on Artificial Neural Networks*. Springer, 281–290.

[9] Matthieu Courbariaux, Jean-Pierre David, and Yoshua Bengio. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[10] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. *CoRR, abs/1502.02551* 392 (2015).

[11] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:0000.0000* (2016).

[12] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR, abs/1510.00149* 2 (2015).

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385* (2015).

[14] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50× fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[15] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, 675–678.

[16] N. Jouppi. 2016. Google supercharges machine learning tasks with TPU custom chip. *Google Blog, May* 18 (2016).
[17] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).
[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.
[19] Edward F. Moore. 1956. Gedanken-experiments on sequential machines. *Automata Studies* 34 (1956), 129–153.
[20] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi. 2016. Design space exploration of FPGA-based deep convolutional neural networks. In *Proceedings of the 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC'16)*. 575–580. DOI: http://dx.doi.org/10.1109/ASPDAC.2016.7428073
[21] Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for convolutional neural networks. In *Proceedings of the 2013 IEEE 31st International Conference on Computer Design (ICCD'13)*. IEEE, 13–19.
[22] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Songet al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.
[23] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279* (2016).
[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet large scale visual recognition challenge. *Int. J. Comput. Vision (IJCV)* 115, 3 (2015), 211–252. DOI: http://dx.doi.org/10.1007/s11263-015-0816-y
[25] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'09)*. IEEE, 53–60.
[26] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[27] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. 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 (FPGA'16)*. ACM, New York, NY, 16–25. DOI: http://dx.doi.org/10.1145/2847263.2847276
[28] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1–9.
[29] Andrea Vedaldi and Karel Lenc. 2015. Matconvnet: Convolutional neural networks for matlab. In *Proceedings of the 23rd ACM International Conference on Multimedia*. ACM, 689–692.
[30] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. 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, 161–170.

Received November 2016; revised April 2017; accepted July 2017