CONVOLUTIONAL NEURAL NETWORKS ACCELERATORS ON FPGA INTEGRATED WITH HYBRID MEMORY CUBE

by

Thibaut Lanois

B.S. in Science, INSA Lyon, 2011

Submitted to the Graduate Faculty of

Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Thibaut Lanois

It was defended on

November 27, 2017

and approved by

Jun Yang, Ph.D., Professor, Department of Electrical and Computer Engineering

Alan D. George, Ph.D., Professor, Department of Electrical and Computer Engineering

Jingtong Hu, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Youtao Zhang, Ph.D., Associate Professor, Department of Computer Science

Thesis Advisor: Jun Yang, Ph.D., Professor Department of Electrical and Computer Engineering

CONVOLUTIONAL NEURAL NETWORKS ACCELERATORS ON FPGA INTEGRATED WITH HYBRID MEMORY CUBE

Thibaut Lanois, M.S.

University of Pittsburgh, 2017

Convolutional Neural Networks (CNN) have been widely used for non-trivial tasks such as image classification or speech recognition. As the CNNs become deeper (more and more layers), the accuracy increases, but also the computational cost and the model size. Due to the numerous application of CNNs, numerous accelerators have been proposed to handle computation and memory access pattern. Accelerators use the high parallelism of FPGAs, GPUs or custom ASICs to compute CNNs efficiently. Whereas the GPU throughput is very high, the power consumption can be, in some environment, a concern. FPGA accelerators provide a good balance between time-to-market and power consumption but face several challenges. FPGA resources must be carefully chosen to match the network topologies. The hardware constraints to carefully choose are the number of DSPs/BRAMs and the external memory bandwidth. In this thesis, we present two new FPGA designs using the Hybrid Memory Cube as external memory to accelerate CNN efficiently. The first design is a 32-bit fixed point design named Memory Conscious CNN Accelerator. This design uses one Convolution Layer Processor (CLP) per layer and uses the parallelism of the HMC to supply data. To maximize the HMC bandwidth utilization, a new data scheme is proposed. This new algorithm makes data request sequential. With this new algorithm, the frequency of our accelerator can reach 300MHz and a throughput of 232 Images/sec on AlexNet. The second design is a low power CNN accelerator, where data layout is arranged before the Pipeline Execution (PE). During the layout process, Deep Compression techniques can be applied to improve performance. The PE design achieves a 66 GOPs for only 1.3 Watt of power consumption.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I would like to express my gratitude to Professor Yang, and Professor Zhang. They helped me and guided me through my research. I would like also to thank my committee members who gives me very insightful comment and pertinent question. I thank the reviewers for their good feedback. Finally, I would like to thank my family for their support.

1.0 INTRODUCTION

CNN has emerged as a new standard for image classification due to their high accuracy in recognition tasks. CNNs consist of multiple layers of filters, organized as a network. The network learns filters that maximize the activation of some type of visual structure. To improve accuracy, bigger networks are often used, which increase both the computational cost and memory demand. To overcome these costs, numerous hardware accelerators have been proposed. Among them, FPGA-based designs have shown promising results in achieving effective implementation of CNN. However, the gap between theoretical and real FPGA implementation is still large and the performance over power ratio may be inferior to GPU implementation. As each output of a convolution layers is a multiplication-accumulation calculation, the two main bottlenecks of implementing CNNs on FPGA are the DSPs efficiency and the external memory bandwidth. The common approach to calculate convolution is to transform convolution into numerous matrix multiplications that require a large amount of pseudo-random data to be moved between external memory and the FPGA. Data are often stored in local memories, and double buffering technique is used to hide the memory latency. The inadequacy between the network model size and the available on-chip storage imposes a lot of pressure on bandwidth requirement. In many designs, the FPGA frequency is limited by the external data transfer rate. This barrier where convolution goes from computation bound to memory bound will become more pronounced as the number of DSPs and the maximum frequency increase in modern FPGA.

Hybrid Memory Cube (HMC) can be a promising solution to overcome the memory bound problem. However, using the HMC also introduces different challenges. To maximize the HMC bandwidth utilization, the user request size must be as large as possible. The two designs described below request data in a sequential pattern. Whereas the first architecture, provides a new algorithm to calculate convolution in a sequential way, the second one rests upon a data reorganization realized on software to make data access sequential.

The first design uses a multi-Convolution Layer Processor (CLP) architecture, within each CLP, request pattern is sequential and follows the input mapping rather than pseudo random as in a traditional matrix multiplication approach. Due to the improved mapping and memory management, our design achieves a speedup of up to 2.23 against previous designs using a similar FPGA board.

The second design implements hardware weight sharing techniques to minimize the number of DSP required. This design consumes only 1.3 Watt of power with the throughput of 255 Images per second on the 5 convolution layers of AlexNet.

The rest of this thesis is organized as follow. Section 2.0 provides a background on CNN, HMC. Section 3.0 gives an overview of related work and state of the art design. Section 4.0 refers to the Memory conscious CNN accelerator, both description and evaluation. Section 5.0 describes and evaluate the 8-bit Deep Neural Network Accelerator. Section 6.0 shows a solution to use both the new data scheme and the weight sharing techniques. Finally, section 7.0 concludes this thesis.

2.0 BACKGROUND

A Neural Network is a computer system modeled on the human brain and nervous system. It is made up of a succession of nodes in an interconnected network. Neural Networks (NN) are in numerous applications traditionally associated with human intelligence and applications where training is needed. For instance, a NN might be used to distinguish colors in a way similar to humans, by making a comparison between the colors they have seen before.

2.1 CONVOLUTION NEURAL NETWORK

A Convolutional Neural Network (CNN) is a NN, but is especially suited for 2D input. For example, a CNN can be used for image recognition in which the input is an image and the CNN can classify the image into preconfigured categories. Today there are already numerous CNNs in application from facial recognition to self-driven cars.

CNNs use convolutional layers that take advantages of the image's locality to reduce the number of parameters. Each input of a convolutional layer is called input features map, this could be a single or many 2D matrix. A kernel, one for each input features map, is slid across each input. This operation can be repeated to obtain the desirable number of output features maps [1]. Figure 1 shows the algorithm to calculate a convolutional layer and an illustration of this layer. The calculation presented in Figure 1, can be cut in three different parts.

The bottom part, constituted of three for loops, is a 3D matrix multiplication between the weight, also called kernel and the input that will give one output. These three loops are represented by the square in the Figure 1. The black 3D matrix will give the black output pixel and the gray one will give the gray output pixel.

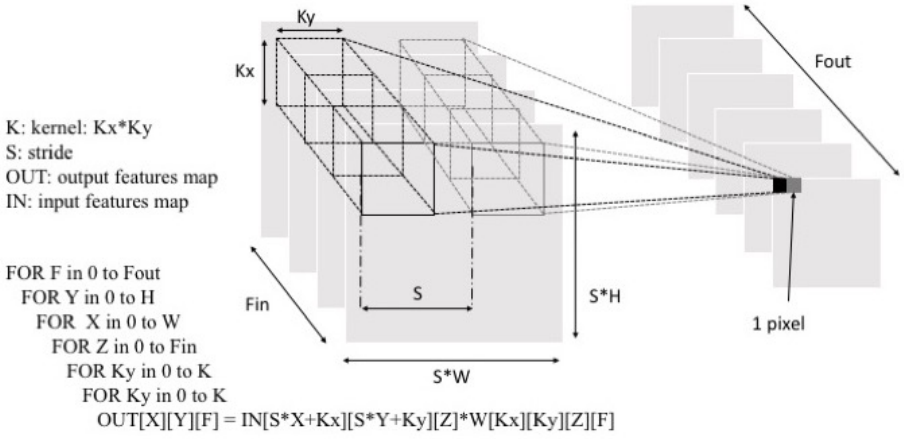

Figure 1. *An illustration of one convolution layer with the pseudo code for convolution*

The two next for loops represent the slide of the kernel across the input. The length of slide is given by the stride. In Figure 1, the stride is larger than the kernel. In practical CNNs, there is frequently an overlap between two consecutive kernels. Then the last for loop is the repetition with different kernels to reach the good number of output features maps: Fout. CNNs are formed by many convolution layers that can be followed by nonlinear functions, often by a pooling layer that takes the maximum among multiple pixels and at the end one or more fully connected layer to classify the images into pre-configured classes. The fully connected layer will give a similarity

measure between the input and each pre-configured class. In CNNs, convolutional layers represent most of the total operation, for instance, in AlexNet, 92% of the calculation is occupied by the convolution [2].

A trend to improve the accuracy of CNNs is to use deeper networks, each year the winner of ImageNet classification challenge use deeper networks. Such technique often increases the model size and the computation cost. To overcome these issues new ways of training Deep Neural Network have emerged.

2.1.1 Weight pruning and weight sharing

Deep compression proposed in [3], [4] is a new technique to reduce the size of the network by removing redundant weights and by rounding the values of the weights up or down. The idea is to transform dense matrices into sparse matrices. To do so, deep compression executes a three stages algorithm.

The first stage is to prune the network. The network is trained one first time, then weight near zeros are removed and the network is trained again. With this step, the network will learn and keep only the important connection between neurons.

The second stage is weight quantization. By this fact multiple connections will have the same weight, thus numerous weights can be shared.

The third stage is Huffman coding. This stage will not be used in our work.

According to [3], [4], the pruning stage reduces the number of connections by 9× to 13×; Quantization then reduces the number of bits that represent each connection from 32 to 5. Whereas deep compression reduces greatly the size of the model, sparsity and the encoding format can hurt

performance for CPU and GPU [5]. However, FPGAs may really take advantages of this new algorithm because FPGA can greatly take advantages of irregular parallelism and sparsity [6].

2.2 HYBRID MEMORY CUBE

Hybrid Memory Cube (HMC) is a high-performance memory technology. Through Silicon Vias (TSV) is used to stack multiple DRAM layers together with a logic layer. HMC is an attempt to overcome memory wall in multi-core processor design.

2.2.1 Architecture

As shown in Figure 2, this memory has a very hierarchical organization. Within the HMC, memory is organized vertically into vaults. Each vault contains multiple partitions and each partition is divided into multiple banks. At the bottom of each vault, implemented in the logic layer, a vault controller allows each vault to function and operate independently. The vault controller is also in charge of refresh operation that eliminates this function from the host memory controller. Each vault controller has a queue to reorder data request, in consequences responses from vault operation to host device can be out of order [7], [8]. To establish the communication between the logic layer and the host controller, the HMC uses full duplex serialized link with a data rate typically higher than 10Gbps per lane. This communication consists of a serialized physical layer and an abstracted packet interface. This protocol differs from previous DDR transaction design and has a better match with PCIe protocol [7]. HMC supports write and read data block access with 16-bit granularity form 16B to 128B of data width.

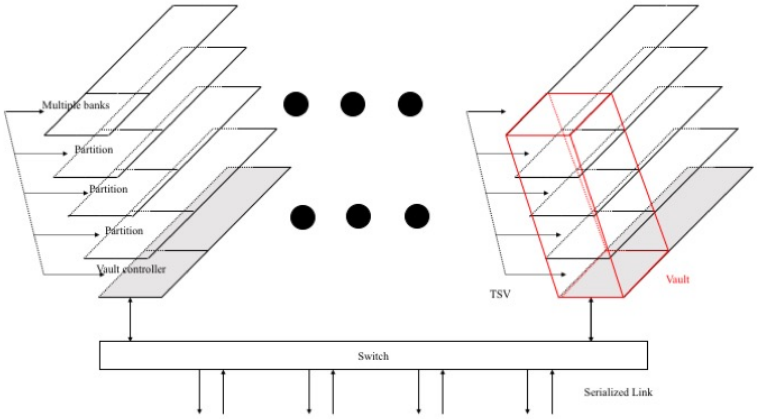

Figure 2. *HMC architecture*

2.2.2 Related work on FPGA/HMC platform

The overall configuration of the platform we used is presented in Figure 3. The server is constituted of an FPGA linked by 16 channels to the HMC. On the other hand, a PCIe connexion links the CPU and the FPGA. Inside the FPGA, there are 2 controllers, one for PCIe and another for the HMC. The rest of the FPGA is free for the user to implement their own accelerator.

The HMC controller offers 9 port of 128b to the user to request data. The HMC controller clock is 187.5 MHz and the PCIe controller clock is 100Mhz.

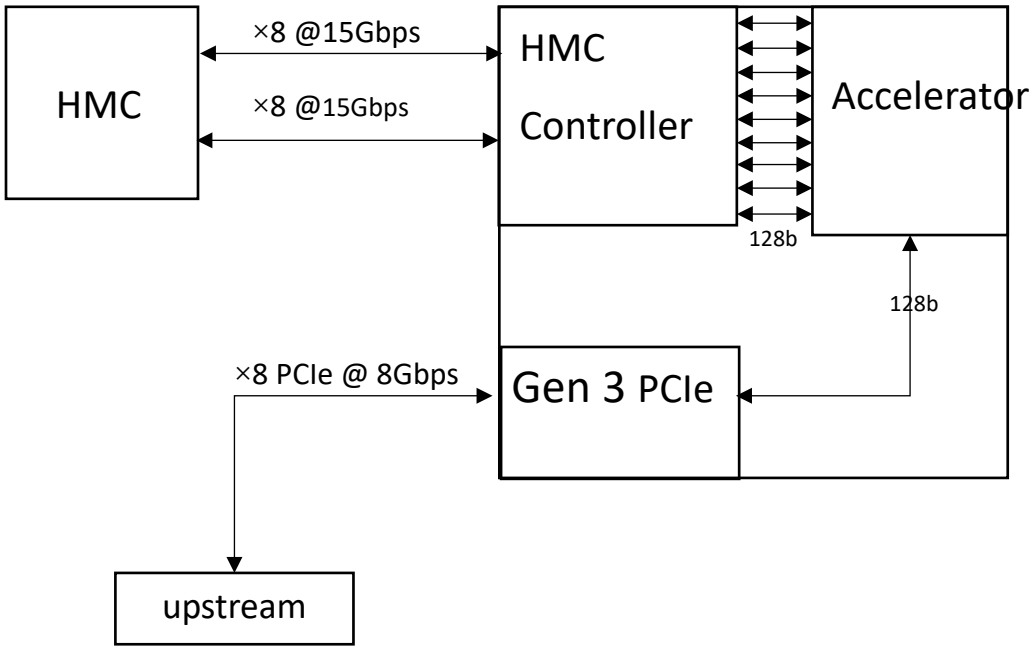

Figure 3. *Server overview*

Several works already show the benefit of using HMC in memory-heavy algorithms [9]–[12]. [9], [12] implement a Breadth First Search (BFS) algorithm. In [12], they implement a 2-level bitmap, where the first bitmap is implemented on the FPGA board and works like a cache for the HMC bitmap. In [9], they improve the design by adding a merging unit and change the layout of the graph to have a better match with the HMC layout. [11] uses the incredible parallelism performance of the HMC by implementing many bloom filter, one on each FPGA (platform constituted of 4 Stratix-V FPGAs and the HMC). [10] implements a packet matching application. They use prefetching to hide the HMC latency and duplicate the packet matching engine to use the parallel performance of the HMC.

All these works focus on memory-bound applications, where the parallel accesses and the very large bandwidth of the HMC allow this platform to perform well.

3.0 FPGA ACCELERATOR

Many CNN accelerators have been proposed on FPGA, ASIC, GPU. ASICs often offer very fair ratio between performance and power consumption, however, the time to market can be very long. Indeed, ASICs need long development cycle and expensive tools to take care to signal integrity issues, placement and routing issues... GPUs may have very large throughput. GPU takes advantages of his highly parallel structure to accelerate NN, but the power consumption can be a problem in a constraint environment (real time, server). In most cases, GPUs are still using for the backward propagation, to train the weights. FPGAs can propose a good ratio between power consumption and performance but offer a better flexibility due to their re-programmability and a faster design cycle because of their software that handle most of the placement and routing. In this paper, we will only focus on FPGA accelerators.

3.1 CONVOLUTION LAYER PROCESSOR (CLP) DESIGN

In today FPGA-accelerators, the convolution layers are processed mostly using a vector dot product or also called Matrix Multiplication design. A vector dot product is made of two components. The first step is the multiplication between each input and the corresponding weight. The second step is the addition of each level of multiplier outputs. All products are added together so there is only one output at the end of the tree of adders (black box in Figure 4) [2], [13]–[18].

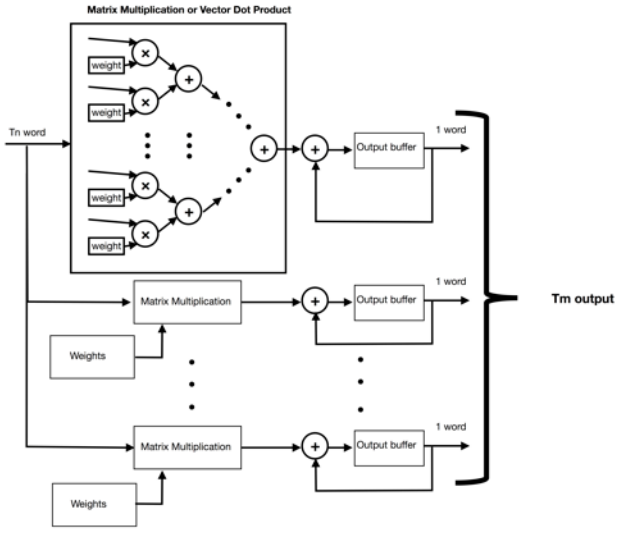

Figure 4. *Array of Dot Vector Product*

As the kernel could have a large number of input feature maps, an output buffer is added at the end of the vector dot product, in order to be able to add the previous result with the current output. Then, this small component is parallelized through the output features maps of the convolution to maximize the throughput by filling all the DSPs in the FPGA. There are two parameters for this type of CLP, the number of input (Tn) and the number of output (Tm). As each multiplication is added together, either on the tree of adders or on the output buffer, each input must be inside the actual kernel which means input request will be very scattered. In order to limit the bandwidth requirement, Tn has to be small which means a lot of output features map will be processed at the same time. Indeed, output features maps have all the same input data, only weights are different. In consequences, input data requests will be smaller, but output data will be calculated via the output features map. Therefore, output data will be very scattered. Increasing Tm may also lead to increase the number of BRAM used. To store weights and output features maps data more BRAM have to be used. Increasing the number of BRAM could lead to larger

FPGA utilization or worst to overload the FPGA capacity. Regarding external memory, DDR is not especially well suited to handle very scattered data request. For pseudo-random request, DDR3 will only sustain around 29% of it reachable bandwidth [8]. A lot of work should be done to store and reuse most of the data on chip to avoid memory bottleneck [13]–[15]. As the number of DSP slices and the reachable frequency increase very quickly in modern FPGA, DDR memory will not be able to sustain the FPGA bandwidth requirement for CNNs. At 170Mhz, with a Xilinx 690T FPGA, the bandwidth of the design described above can be up to 20.5 GB/s of pseudo random-access [16].

3.2 MULTI-CLP DESIGN

The best parameters Tn and Tm are input, output, stride and kernel dependent. For a specified network, Tn and Tm are calculated to have the better throughput for the whole network, however using only one CLP for a whole CNN leads to not optimal parameters for each layer. These parameters affect throughput and power by configuring the bandwidth requirement and the number of DSP used. For instance, if Tm is not a divisor of the number of output features maps, then the last epoch will be underutilized. In [18], they show the cross-layer optimization is a Tm of 64, however, the first layer has only 48 output features maps, so 16 outputs features are calculated but never used. The Tn parameter may also create underutilized hardware resources. If Tn is not a common factor with the kernel size, some DSPs will be only used a small portion of the time. This arithmetic utilization problem or DSP efficiency leads to an inefficient design where the outputs of DSPs are calculated, but not used to compute the convolution. In [16], they observed that the worst case is running AlexNet on a Virtex 7 FPGA with 16-bit fixed point precision. This

configuration leads to only 24% of DSP utilization. To improve resources utilization, a multi-CLPs design can be used, where the available resources are partitioned across several smaller CLPs rather than a single large one [16]. Figure 5 shows the schedule of a 3-CLPs design with a three layers' network. In Figure 5, the 3-CLPs pipeline the image.

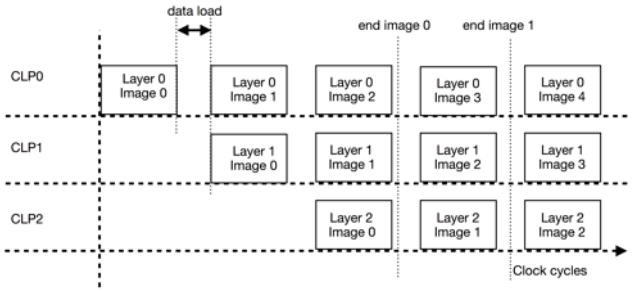

Figure 5. *3-CLPs design*

Each CLP works on a different image concurrently. Multi-CLPs design resolves the DSP utilization problem, but intensify the pseudo-random request and increase the number of BRAM used. Indeed, each CLP works on different images, so on different addresses on the memory. Moreover, multiple parallel data requests, one by CLP, lead to complex scheduler, deep queues, and high reordering [8]. Whereas DDR is well suited for large sequential linear data accesses, convolution with a matrix multiplication approach ask for pseudo-random data accesses. Multi-CLPs design emphasizes the pseudo-random accesses and puts another constraint on parallel accesses. Thus, DDR may not be the best memory to handle multiple CLPs. To overcome these issues, we propose to use HMC to handle parallel data requests. However, matrix multiplication approach is not the best algorithm to maximize the HMC bandwidth. To maximize the bandwidth of the HMC, we propose a new memory conscious CNN accelerator.

3.3 STATE OF THE ART MEMORY OPTIMIZATION FOR CNN

Many efforts have been done to overcome the memory bandwidth issue on FPGA. [13] used fused-layer to reduce bandwidth requirement by computing the convolution, not layer by layer but by pyramid-shaped multi-layer sliding window. In [14], they implemented a one CLP design, where they performed enhancement to minimize bandwidth requirement and maximize throughput. [19], [20] are techniques to decrease the number of calculation without losing accuracy. [20] suppresses ineffectual multiplications by data quantization, whereas [19] deletes it by a value-based approach. In [17], they work on efficiently match the design with the batch size to perform efficiently the matrix multiplication operation.

4.0 MEMORY CONSCIOUS CNN ACCELERATION ON FPGA INTEGRATED WITH HYBRID MEMORY CUBE

In this section 4.0 , we present our new data request scheme to optimize high bandwidth memory like the HMC.

4.1 PIPELINE CONVOLUTION DESIGN

In this part, we will first present the approach of our new design and then show the implementation, the design choices, and the interface with the HMC controller.

4.1.1 Principle

Our design is a multi-CLPs design where each CLP is supplied by one HMC port. The HMC controller handles all data requests in parallel and sends back data to the corresponding CLP on the FPGA. HMC has higher bandwidth when the request size is higher [21]. The approach where convolution is transformed into matrix multiplication may not be the best way to have large request size. Our new design maximizes the packet size by proceeding the convolution in an out of order way. Instead of fetching data kernel after kernel, the design issues the data requests through the x-axis first, then through the z-axis and finally through the y-axis. Figure 6 presents the data requests which are sent to the memory for a 4×4×1 input, a 2×2 kernel and a stride of 1.

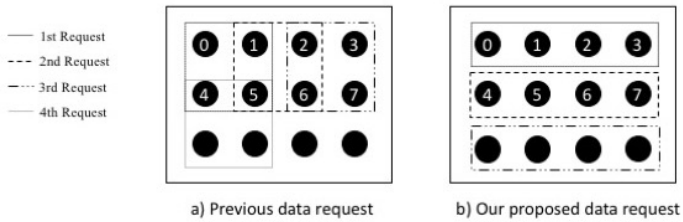

Figure 6. *Data request pattern*

Whereas Figure 7 gives a pseudo-code for our principle. In Figure 6, in both examples, the external memory bus width is 4 words. Each black dot represents a pixel or a word. Part a describes requests of the previous design and Part b displays requests of our proposed design. On the left, the legend indicates in which order the requests are made.

```
for (f = 0; f < Fout; f++) do          // output features map
    for (y = 0; y < S× H; y++) do
        for (z = 0; z < Fin; z++) do   // input features map
            for (x = 0; x < S× W; x++) do
                for (j = 0; j < K1; j++) do
                    // #pragma pipeline
                    for (i = 0; I < K0; i++) do
                        // #pragma unroll

                        Compute Address[i];
                        T[Addr[i]] =
                          IN[x][y][z] × W[i][j][z][f];
                        add ;      // add together temp(T)
                         with the same address
```

Figure 7. *Pseudo algorithm*

Our design does not use the matrix multiplication approach but a linear approach where all the calculations are done out of order. When the last data of a row is fetched then all the partial outputs are reordered. This reordering is done by the functions “compute address” and “compute add” in Figure 7. “Compute add”, add all the temporary results with the same address into one output (more information on the next section). In most of the cases, each input pixel will have more than one kernel on it. So, every input will be multiplied by more than one weight. The numbers of multiplications each input has, depends on the stride and the kernel. The number of multiplications through the x-axis and y-axis is given respectively by K_0 and K_1 in Figure 7. The design avoids data movement by processing all the multiplications for one input data at once. For example, in Figure 6, a 2×2 kernel is represented and the black pixel number 5 will be inside four different kernels. Only three requests (1,2,4) are represented in Figure 6 for more visibility. In some matrix multiplication approaches, the input with four multiplications will be inside four different matrices and will be brought to the CLP four different times. For our design, each input will be brought to the CLP only one time. In this example, K_0 and K_1 will be equal to 2. When the data have been brought from external memory to the FPGA, multiplications will be calculated simultaneously and pipeline through K_1. Doing one input data movement for these $K_0 \times K_1$ multiplications allows a new type of parallelism and avoid useless data movement. For unitary kernel with a unique input feature map, previous data requests and our proposed data requests will be the same, since both designs will be in a sequential way (in Figure 6, black pixel 0,1,2,3 is the first request then 4,5,6,7). If Tn is different to 1, that means the design computes more than one input at the time. The previous design will need Tn different weights and Tn different inputs. While our design will need Tn weights but only one input.

Our design changes where the parallelism can be found. Parallelism is behind the number of multiplication each pixel has and not via the matrix multiplication approach as it was before. This new parallelism allows large input data requests since data requests are sequential.

4.1.2 The design

This part will describe our new CLP to implement the principle described above into an FPGA. For the rest of the paper, our design proposition will be named Pipelined Convolutional Layer Processor (PCLP). The implementation will be a 32-bit fixed point convolution. Figure 8 shows a simplified design overview of a 2×2 kernel with a stride of 1. The kernel dimension and the stride set the number of multiplication each pixel has. For instance, 3×3 with a stride of 1 will have 9 multiplications by pixel ($K_0 = K_1 = 3$), or 3×3 with a stride of 2 will have either two or one multiplication by pixel ($K_{0even} = K_{1even} = 2$, $K_{0odd} = K_{1odd} = 1$). The link between the number of multiplication, the stride, and the kernel size makes the elementary design only stride and kernel dependent. Data input comes directly from the HMC, the width is 16B so 4 words of 32b. The interface between the HMC controller and the design is only a buffer. This buffer is a kind of FIFO. As the data from the HMC could be out of order, the buffer must reorder the data before sending them to the design. This buffer is also used to do the interface between the frequency of the HMC and the frequency of the PCLP.

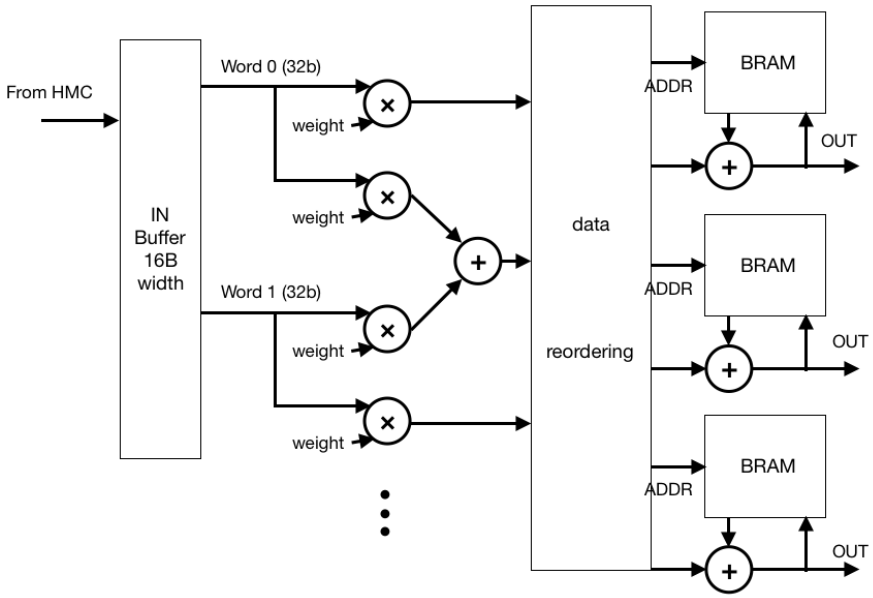

Figure 8. *Design overview*

Then all the 4 words (only 2 are represented in Figure 8) are multiplied with all the weights (in Figure 8, two weights by word with two cycles of pipeline). The design only uses the size of the x-axis of the kernel DSP (K_0) and then pipeline through the y-axis (K_1) to increase the time between two data requests and hide the latency of the HMC. For the rest of this paper, the pipeline through the y-axis will be called kernel cycle. The outputs of the multipliers will be reordered to assure that the same address always goes to the same BRAM. To avoid using multiple ports to address the BRAM, consecutive outputs of the multipliers are added together. Lastly, adders accumulate data with the same address. The address calculation and the data reordering are the most important points of our design. These two steps guarantee that the convolution is well done. Data reordering is calculated by taking the modulo of the address to always have the same BRAM for the same address. This heavy technique asks for a 32-bit modulo, dispatch into two clocks cycles, but assures that the same address will always go to the same BRAM. The address is

calculated by the formula in (1): where X is the actual X coordinates on the input image, Y the Y coordinates of the input image minus the Y of the kernel. The Y of the kernel will be always between [0: kernel size − 1]. POSITION, the position in the array of multipliers, in others words the X of the kernel. And OFFSET is the offset when the stride is not unitary. The offset is only a constant because when the stride is not unitary odd and even pixel will not have the same number of multiplications. This formula has been only validated for an even stride due to the 16B aligned data from the HMC. This constraint on the stride has no impact on well-known networks [22]–[24]. As we have minus sign, the address could be negative. When this case appears, the valid bit of the address goes down to 0. As we are going to discuss later, some border pixels of the image have not the same number of multiplications than other pixels. These multiplications are calculated but never used.

$$\mathrm{Addr} = \frac{\mathrm{Y} * \mathrm{X}_{\text{size 16B aligned}} + \mathrm{X} - \mathrm{POSITION} + \mathrm{OFFSET}}{\mathrm{STRIDE}} \quad \textbf{(1)}$$

When the stride is not unitary, each of the 4 inputs will have less multiplication to do. However, the design still works since each of the 4 inputs are independent and can be seen as 4 different kernels. A kernel of 5×5 with a stride of 2 can be seen as a kernel of 3 for the first input, a kernel of 2 for the second input, a kernel of 3 for the third input and a kernel of 2 for the last input word. As the stride is 2, there is a repetition of the two first words to the next 2. This vision only means that all odd number will have two multiplications by clock cycle and even number will have three multiplications by clock cycle. For the stride in the vertical direction, some kernel cycle will be two cycles of pipeline, whereas others will have three cycles of pipeline depending on either the coordinate is odd or even. This configuration means for even pixel in the x-axis and even pixel in the y-axis the kernel will be 3×3, odd pixel in the x-axis and even pixel in the y-axis the kernel will be 2×3, even pixel in the x-axis and odd pixel in the y-axis the kernel will be 3×2, odd

pixel in the x-axis and odd pixel in the y-axis the kernel will be 2×2. The data reordering will also be stride dependent. As illustrate, in Figure 8 when the stride is 1, two consecutive results from two different multipliers can be added together and the two extremities go directly to the reordering. However, when the stride is 2 the address layout is not the same and same words on other multipliers are added. This different layout explains why the design is also stride dependent. In case the addresses between two consecutive results that should be added together are different, that means one of the two addresses is not valid, then a multiplexer can choose to send only the valid data. This multiplexer located just before the adder is not represented in Figure 8 for more visibility. To illustrate how the design works, an example is presented with a 2×2×2×1 kernel (2) with an input image of 4×2×2 (3), to keep things clearer we only illustrate for one output feature map. The same reasoning can be used since output features map is only a repetition with different weight values.

$$W_0 = \begin{pmatrix} W_{0,0,0} & W_{1,0,0} \\ W_{0,1,0} & W_{1,1,0} \end{pmatrix}, \qquad W_1 = \begin{pmatrix} W_{0,0,1} & W_{1,0,1} \\ W_{0,1,1} & W_{1,1,1} \end{pmatrix} \tag{2}$$

$$\mathbf{I_0} = \begin{pmatrix} \mathbf{I_{0,0,0}} & \cdots & \mathbf{I_{3,0,0}} \\ \vdots & \ddots & \vdots \\ \mathbf{I_{0,1,0}} & \cdots & \mathbf{I_{3,1,0}} \end{pmatrix}, \qquad \mathbf{I_1} = \begin{pmatrix} \mathbf{I_{0,0,1}} & \cdots & \mathbf{I_{3,0,1}} \\ \vdots & \ddots & \vdots \\ \mathbf{I_{0,1,1}} & \cdots & \mathbf{I_{3,1,1}} \end{pmatrix} \tag{3}$$

Calculation steps are resumed in Table 1. In this table, the order of the multiplications is presented and addresses are in parenthesis. If we add together all the calculations with the same address, we obtain the good calculation for every output pixel. X means that the data is not valid and the address will not be calculated. X cases are cases where the multiplier is used, but the result means nothing. These cases are often when the address is negative or superior to the image size. In other words, these cases happen when the input pixel is at the border of the input.

Table 1. *Calculation steps for 2*2 kernel*

cycles	1	2	3	4	5	6
Word 0	$W_{000} \times I_{000}$ (0)	$W_{001} \times I_{001}$ (0)	$W_{000} \times I_{010}$ (4)	$W_{010} \times I_{010}$ (0)	$W_{001} \times I_{011}$ (4)	$W_{011} \times I_{011}$ (0)
	$W_{100} \times I_{000}$ (X)	$W_{101} \times I_{001}$ (X)	$W_{100} \times I_{010}$ (3)	$W_{110} \times I_{010}$ (X)	$W_{101} \times I_{011}$ (3)	$W_{111} \times I_{011}$ (X)
Word 1	$W_{000} \times I_{100}$ (1)	$W_{001} \times I_{101}$ (1)	$W_{000} \times I_{110}$ (5)	$W_{010} \times I_{110}$ (1)	$W_{001} \times I_{111}$ (5)	$W_{011} \times I_{111}$ (1)
	$W_{100} \times I_{100}$ (0)	$W_{101} \times I_{101}$ (0)	$W_{100} \times I_{110}$ (4)	$W_{110} \times I_{110}$ (0)	$W_{101} \times I_{111}$ (4)	$W_{111} \times I_{111}$ (0)
Word 2	$W_{000} \times I_{200}$ (2)	$W_{001} \times I_{201}$ (2)	$W_{000} \times I_{210}$ (6)	$W_{010} \times I_{210}$ (2)	$W_{001} \times I_{211}$ (6)	$W_{011} \times I_{211}$ (2)
	$W_{100} \times I_{200}$ (1)	$W_{101} \times I_{201}$ (1)	$W_{100} \times I_{210}$ (5)	$W_{110} \times I_{210}$ (1)	$W_{101} \times I_{211}$ (5)	$W_{111} \times I_{211}$ (1)
Word 3	$W_{000} \times I_{300}$ (3)	$W_{001} \times I_{301}$ (3)	$W_{000} \times I_{310}$ (7)	$W_{010} \times I_{310}$ (3)	$W_{001} \times I_{311}$ (7)	$W_{011} \times I_{311}$ (3)
	$W_{100} \times I_{300}$ (2)	$W_{101} \times I_{301}$ (2)	$W_{100} \times I_{310}$ (6)	$W_{110} \times I_{310}$ (2)	$W_{101} \times I_{311}$ (6)	$W_{111} \times I_{311}$ (2)

When data are calculated, they are stored inside the BRAM. A counter, one by address, keeps an update of the number of addition that has been made. When the counter reaches the good number of addition, a flag will go up to 1. This flag drives the output to be valid and resets the accumulator in the BRAM and the counter to 0. On the next clock cycle, the design will be ready to handle the next output pixel line. As we can see in Table 1, as the data request is linear, results will come in the same way. As we have 4 words in parallel, the outputs will come 4 by 4 every kernel cycle following the X-axis. In Table 1, we have the 4 output pixels at cycle 6. The equation (4) reveals the reordering for the first output pixel. This reordering is calculated by adding results of the multiplication with the same address.

$$OUT[0] = w_{0,0,0} * I_{0,0,0} + w_{1,0,0} * I_{1,0,0} + w_{0,0,1} * I_{0,0,1} + w_{1,0,1} * I_{1,0,1} + w_{0,1,0} * I_{0,1,0} + w_{1,1,0} * I_{1,1,0} + w_{0,1,1} * I_{0,1,1} + w_{1,1,1} * I_{1,1,1} \quad (4)$$

Formula (4) shows the first pixel of the convolution, so every input and weight have the same index. This design has a fixed number of multipliers that it needs to use. This number depends on the kernel and the stride. The previous example needs two multipliers by word and by clock cycle, so 8 multipliers. However, this elementary design can be parallelized through the output features maps to match the number of DSP available. In order to parallelize through the output features maps, after each elementary PCLP, a FIFO is added. This FIFO can be either used as waiting list to write to the memory or as a buffer to reuse data in the next epoch. Indeed, as input and output data go through a FIFO by adding only one multiplexer, the multiplexer can choose data between input FIFO or other PCLPs output FIFO. Data can be reused very easily as a kind of double buffering. The first output data of the previous layer will be kept into the FIFO, giving a window of time to the HMC to fill the other FIFO.

4.1.3 Matching available resources

Our accelerator instantiates multiple PCLPs to find the best configuration. This configuration has several constraints and tries to maximize the throughput. Most of the constraints are on the hardware either on the FPGA side (DSP, BRAM) or on the HMC side. For instance, HMC has 9 ports available and the bandwidth of each port depends on the layout of the data and the user request size. Figure 9, shows the bandwidth of the HMC for different configurations. The number of tags that can be used has been set to the maximum available (64 tags are available), the number of ports to 9 (the maximum available), the bandwidth is for read request only [25]. In the random mode, there is no conflict between vault and bank in the data request mapping. In the second curve, each address has the same vault. And finally, the last curve is calculated by requesting the same address for each port which creates both a bank and a vault conflict [21], [26].

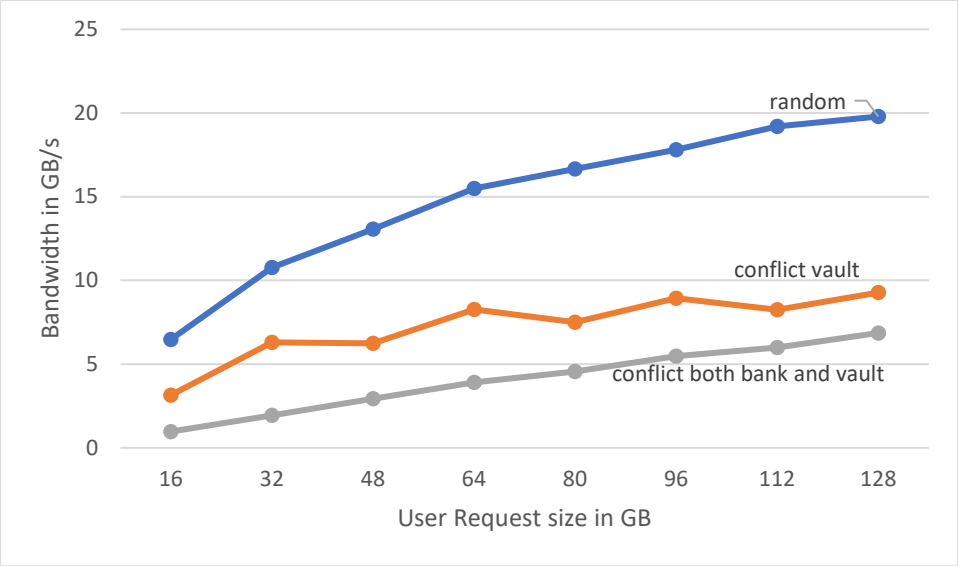

Figure 9. *HMC bandwidth for 9 ports*

As shown in Figure 9, random data requests are a lot better than request where a conflict occurs. The explanation is linked to the HMC architecture. In random data request, each vault of the HMC can be accessed in parallel hiding the memory latency. For a 128B request size, there is almost a 3-time factor between the random-access pattern and the conflict in both bank and vault. As our PCLP design maximizes the user request size by processing data request in a sequential way, our design can use the 128B size request. As each PCLP works on different images at the same time, the nearest mode is the random access. That gives for our design a bandwidth of 19.79 for 9 ports which means a 2.2GB/s bandwidth for each port. This number is the maximum bandwidth that each PCLP can have for the input data. Increase data reused by storing more data in the output FIFO can be used to decrease the input bandwidth. Weight data are brought through PCIe interface, where double buffering is used to hide memory latency. In terms of BRAM utilization, there are some BRAMs used which will be always utilized, no matter if we reuse data or not. Weight Memory is reliable on the latency of the PCIe network and BRAM of the adder is

dependent on the input size. On the other hand, output and input FIFO is clearly dependent on how much data will be reused. To hide memory latency, these BRAMs have the smallest capacity. Except for this initial capacity, the number of entries of the BRAM will be the number of data that can be reused. Equation (5) provides the relationship between the data reuse and the bandwidth requirement. F is the frequency of the PCLP and N the number of clock cycle to compute the layer.

$$\textbf{bandwidth requirement} = \frac{(\textbf{layer size} - \textbf{data reuse}) * \textbf{F}}{\textbf{N}} \quad (5)$$

Among these constraints, heuristic method is used to find an acceptable solution.

4.2 EVALUATION

We implement our accelerator using Vivado 2016.4, the code is written in VHDL and Verilog (the interface with the HMC). We analyze first, the PCLP design and then try this accelerator in well-known CNNs. This analyze uses only synthesis, simulation, implementation on Vivado and all the HMC behavior are extracted from the HPCC random benchmark (GUPS) sample provided by Micron. The GUPS is a sample that can be fully configured to approximate very precisely the real number of the HMC for a given application [25]. Working has been verifying in hardware, but numbers come from simulation tools.

4.2.1 Pipeline Convolution Analysis

As shown in Table 1, to process data in a sequential way, some DSPs execute useless multiplications. These useless calculations are only for pixels located on the border of the input.

These pixels have fewer multiplications to do than other pixels, however as the number of multipliers is fixed and the same for all the input, such pixel creates useless calculations. In Table 1, the useless multiplications are indicated by an X in the address field. The first input will be only multiplied one time, whereas we have two multipliers available. The number of useless multiplication in a layer will be input, kernel and stride dependent. Figure 10 provides the DSP utilization in function of the kernel size and the size of the input. The DSP utilization is the percentage of calculations that is useful to finish the convolution. The input size is expressed in term of the number of input pixel in one dimension. Each input size has a 16 input features maps. For instance, an abscissa of 20 in Figure 10 means an input of 20×20×16. For more visibility, each stride is unitary.

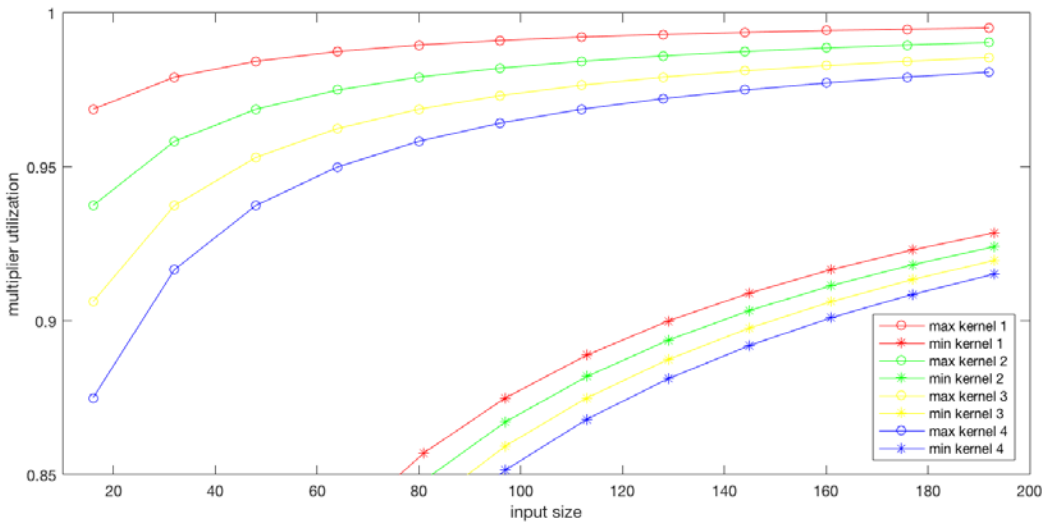

Figure 10. *DSP utilization vs kernel size*

Because our new request scheme is 16B aligned, we will have the best multiplier utilization when the input size is also 16B aligned represented by the circle in Figure 10. On the opposite, when the input size will be 16B + 1, the multiplier utilization will be the worst and it is represented by the asterisk. As Figure 10 shows, the DSP utilization rises when the input size increases. The redundant multiplications are done in the border of the input. Pixel 0,3,4,7 in Figure 6 will have only one multiplication, whereas all others have more than one multiplication. While, with a 3*3 kernel (yellow curves), the two first pixels will have at least one useless multiplication. As the input size raises, the ratio between useful and useless multiplication increases. Indeed, the number of useless multiplications is only kernel dependent so it is always the same. Nonetheless, the number of useful multiplications reliant on input and kernel, and increases with the input. So, the DSP efficiency will grow with the input size. As mention before the number of useless multiplications is kernel dependent. In Figure 10, the DSP utilization decreases when the kernel size increases. This trend can be explained by noting that the number of pixels with not the same number of multiplications than the others, will grow with the kernel size. In the unitary kernel, the three first points are not in maximum DSP utilization, whereas the design should reach 100%. We use only four DSPs one by input that process only the useful calculation. However, to obtain the best performance, our design implement a very long pipeline and as the number of clock cycles to process very small input is very low. The number of stages of pipeline is not negligible for small inputs (almost 20% for the first point on Figure 10), but is negligible as soon as the input becomes larger than 50 pixels (less than 0.2%). As mention earlier, when the stride is not unitary, each input word may have a different configuration. Each configuration is smaller than the initial kernel so the DSP utilization will be also higher. For example, for an input size of 27×27 a kernel of 3×3 with a stride of 2, the DSP utilization will reach 87% (see Table 2), whereas for the same input

and the same kernel but with a stride of one, the DSP utilization will only be 85%. Our pipeline convolution performs well for small kernel or larger kernel with not unitary stride. In each case, larger is the input better is the DSP utilization. This idea works well with recent CNNs, where small kernels are used to extract fine detail of the image at a higher resolution and large kernel with a stride of 2, or sometimes more, are used to find relationship inside the images [27]. To avoid being input dependent and intensifying the design complexity of the data reused, all the input FIFOs have a 4 words width. However, this configuration leads to the inadequacy of the input size. Even though some of the input words may be outside the scope of the matrix, they will be calculated. Table 2 and Table 3 have a row to show the reachable performance without the loss due to the 16B aligned. This mismatch can be easily solved since each data movement goes through a FIFO.

4.2.2 Multi-Pipeline Convolution Analysis

We use a Kintex Xilinx FPGA (XCKU060) and synthesize, implement for two CNNs (SqueezeNet [22] and AlexNet [23]). Then we compare our result with the previous state-of-the-art accelerators. As we explain in the previous section, our PCLP sacrifices some DSP utilization in order to simplify memory access, to paralyze through the current features maps, and to improve overall performance. In this section, we will first compare our DSP utilization to show that the loss is reasonable and then compare overall performance. The DSP utilization is compared with the implementation proposed in [16], best paper for DSP utilization so far. Table 2 presents a detailed performance analysis for the AlexNet. On the left, the simulated implementation from the multi-CLPs design proposed in [16]. On the left center the simulation for their work, but with bandwidth limitation for random access data. On the right center our proposed design and on the right the

reachable performance of our design. DSP util stands for DSP utilization. As we implement 32-bit fixed point, whereas [16] implement a 32-bit floating point design, we take it into account in the DSP utilization calculation. On this Table 2, we can see the overall number of clock cycles needed to complete all the layers in AlexNet.

Table 2. *AlexNet, performance comparison with* [16]

AlexNet model	ISCA paper			ISCA paper with real bandwidth		Our pipeline convolution			Reachable design W/O 16B aligned	
Layer	Tm	cycles *1000	DSP util	cycles *1000	DSP util	Tm	cycles *1000	DSP util	DSP util	cycles *1000
1a, 1b	96	1 098	83%	1 861	50%	10	1 063	88%	89%	1 058
2a, 2b	64	1 166	95%	1 976	57%	26	1 015	83%	88%	961
3a, 3b	64	1 168	93%	1 980	56%	26	1 076	85%	89%	1 036
4a, 4b	96	1 168	91%	1 980	54%	12	1 033	74%	81%	964
5a, 5b	64	1 168	87%	1 980	52%	8	1 033	74%	81%	964
Whole Network	384	1 168	90%	1 980	54%	766	1 076	81%	85%	1 058

As each CLP works concurrently, this number is the maximum among all the CLPs. This number is 1076 thousand clock cycles for our design against 1168 thousand clock cycles for [16]. Our design presents a 1.08 speedup if the two designs work at the same frequency (throughput and frequency analysis are presented on the 4.2.3 section). Two columns are reported for our design, the first one is the actual implemented design and the second one is the reachable performance our new architecture can reach. The reachable design is the design without the constraint on the 16B aligned. In this case, the PCLP will be reliable on the input since the output of the FIFO needs to be a common factor with the input size. The bandwidth limitation on the multi-CLPs design, due to the random access and the 16B user request size, decreases a lot the performance and the DSP utilization. In this case, the overall performance is only 4547 thousand clock cycles, so our design achieves a 4.22 speedup. Table 2 also demonstrates where the parallelism is found. The two designs use almost the same number of DSPs; however, the Tm parameter is very different. As Tm represents the parallelization through the output features maps, a small Tm means a parallelization through the input features maps. In term of DSP utilization, the reachable DSP utilization of our design is 85%, only 3% behind. In the previous design, when Tm equals 1, the DSP efficiency is 100%. However, the very high Tm leads to inefficient parallelism which decreases the overall utilization. Tm is too large to find a near common factor between the output features maps and the number of DSPs available. In our accelerator, we have more parallelism inside the kernel calculation which leads Tm to be smaller, so the Tm parallelism becomes more efficient. In our design, the DSP utilization is almost the same as when Tm equals to 1 which leads to close results with [16].

Table 3 shows the overall performance of the pipeline convolution for the SqueezeNet network. In SqueezeNet, we search for the configuration that maximizes both the DSP utilization

and the throughput. To have the best DSP utilization, we must find the higher common factor between Tm and the output features map of each layer. The better DSP utilization can also be easily explained by the network topology. Indeed, SqueezeNet is constituted of a lot of small kernels with large input sizes where the PCLP performs well. SqueezeNet have a lot of layers with the same kernel, as our design does not change in function of the input, there are a lot of layers that can be grouped in the same PCLP allowing to find better parallelism via the output features maps. For instance, as we only use 6 ports (6 PCLPs), 4 other ports are available. However, using the 4 other ports may lead to inadequate parallel factor in consequences to worse throughput.

Table 3. *SqueezeNet, overall performance*

CLP	layer	Cycles *1000	Tm	DSP Utilization	Reachable DSP utilization
CLP0	1	1 621	8	96%	96%
CLP1	2.2.b, 3.2.b, 4.2.b, 5.2.b, 8.2.b, 9.2.b	1 667	16	93%	92%
CLP2	6.2.b, 7.2.b	1 549	8	81%	91%
CLP3	10	1 679	16	80%	100%
CLP4	2.1, 2.2.a, 3.1, 3.2.a, 4.1, 4.2.a, 5.1, 5.2.a, 6.1	956	16	98%	100%
CLP5	All others	1 100	6	98%	100%
All	Whole network	1 679		91%	97%

Table 4 presents the overall result of our accelerator for SqueezeNet and AlexNet. Table 4 reports the hardware resources (BRAM and DSP) used, the frequency, the bandwidth, the number of clock cycles needed to finish the convolution and the throughput. The configuration of our design is to not reuse data. That means each data is sent back to the HMC and then bringing back to the FPGA. That also means each input FIFO is empty when a new layer begins. This configuration is the worse in terms of bandwidth but the best in terms of the BRAM utilization. As the demands for bandwidth is too large in this configuration with the maximum reachable frequency, the CLP frequency has been reduced to match available bandwidth.

Table 4. *FPGA utilization for AlexNet and SqueezeNet*

Network	DSP	BRAM	frequency in MHz	bandwidth by CLP in GB/s	Cycles	throughput in Img/sec
AlexNet	2739 (99%)	561 (52%)	200	2.13	1 291 000	154.92
SqueezeNet	2246 (82%)	876 (81%)	130	2.08	1 679 000	77.43

As Table 4 demonstrates, SqueezeNet is more memory heavy than AlexNet. Indeed, the small kernels of SqueezeNet ask for a lot of data to be brought to the PCLP. This constraint is strengthened by our design, where the size of the kernel directly impacts the number of requests that must be made. To avoid data movements, our design does all the calculation for one input on the same kernel cycle. As the kernel size decreases, the number of calculation in each kernel cycle also decreases which means a higher bandwidth requirement. In terms of BRAM usages, our

design performs well where only 561 BRAMs are used for AlexNet, whereas 1,238 BRAMs are needed in [16]. The large difference in terms of BRAM usages in our design between SqueezeNet and AlexNet is mainly due to two factors. First, SqueezeNet uses one more PCLP than AlexNet. In consequences, SqueezeNet design has one fifth more BRAM than AlexNet design. Secondly, we have larger Tm in total: 80 for SqueezeNet against 71 for AlexNet. To guarantee that we do not lose data, when the calculation is done, there is a FIFO at the end of each PCLP to store output result, increasing the number of BRAM. The number of clock cycles to finish the convolution is higher than in Table 2 due to the limited size of our FPGA. While on real hardware our FPGA board is limited to 2760 DSPs, in simulation (Table 2) our design can use 3146 DSPs to make a fair comparison with [16].

4.2.3 Comparison with State-of-the-art Design

In this section, we will compare our design with the best design so far in terms of throughput, power, and efficiency. Even if the design presented in [16] presents the best results in terms of DSP efficiency so far, as their frequency is relatively low (100Mhz for AlexNet), the design is not competitive in terms of throughput. We choose 4 different accelerators to compare with: first, the DLA design proposed in [14], the caffeine design proposed in [17] and a work on GPU proposed in [28]. For our design, we decided to reuse data to be able to increase the PCLP frequency, while keeping a sustainable bandwidth. By reusing the data, the CLP frequency for AlexNet reaches 300Mhz. This number is the maximum frequency of our design. For SqueezeNet, the maximum frequency is limited to 235MHz due to the limitation in the number of BRAM available for our FPGA.

Table 5. *Comparison between our pipeline convolution and state-of-the-art design*

Design	TP (Img/s)	Power (W)	Img/s/w
Pipeline convolution	232	20	12
DLA [14]	1020	45	23
TitanX [28]	5882	227	26
Caffeine [17]	104	25	4

Table 5 compares these 4 different accelerators in terms of throughput, power and the throughput divided by the power. The best design in terms of throughput is the GPU design, however, the power consumption can be crippling in a constraint environment. Comes after the DLA with a throughput of 1020 Images/sec on AlexNet against 232 for our design. There are two main differences between these two designs. They implemented a 16-bit floating point design when we use 32-bit fixed point. Secondly, the device they use is a lot larger than our device. The obtainable performance on Intel Arria 10 is 1.5 TFLOPS [29] whereas is only 0.3 TFLOPS for our Xilinx Ultra-Scale [30]. By proportionality, with the same hardware, we could reach 1160 Img/sec. As the power of an FPGA is directly linked to the frequency of working and the FPGA utilization, the power consumption with the Arria 10 FPGA will be very close to 45W, which leads to an efficiency of 26 Img/sec/W. Caffeine design uses the same FPGA than our design. We can see that our design achieves better throughput, more than two times higher throughput and have better power consumption.

4.2.4 High End Server projection

Using the Xilinx data-sheet obtained from [30], we simulate our design in advanced Xilinx Ultra-Scale FPGAs. Results are presented in Table 6 for the Kintex Ultra-Scale 60, the Kintex Ultra-Scale 115 and the Virtex Ultra-Scale 13. By using the latest Xilinx board, the FPGA design reaches a throughput of 1163 Images per second for AlexNet. This number is obtained by the assertion that we can have the same frequency (300Mhz) with newer FPGA devices. This assertion could be conservative since reachable frequency increase as the technology improves. Moreover, as our design uses the HMC device, multiple FPGA can be plugged into the same HMC device. Each FPGA may work concurrently on the same image. Instead of having multi-CLPs by FPGA, we could have only one CLP by FPGA, with many FPGAs link to the same HMC. This kind of configuration, where every resource of one FPGA is concentrated on one layer, will bring higher throughput, but also higher power consumption.

Table 6. *Projected throughput on Ultra-Scale FPGA*

FPGA model	DSP slices	throughput (Img/s)
KU060	2 760	232
KU115	5 520	465
VU13P	12 228	1163

4.2.5 Discussion about the Memory Conscious accelerator

Our design performs well for a 32-bit fixed point convolution, nevertheless, the throughput over power ratio is lower than the GPU one or on the same magnitude for the latest FPGA devices. The weakest of our design is the following. The frequency limit on our design is due to the reorder address scheme, modulo, and large multiplexer. Our design does not use sparsity to decrease computational cost. A 32-bit computation may not be optimized to process convolution. Due to the remarks above, we decided to present a new implementation where the reordering will be done before the PE either in software or in the buffer module. To improve performance, deep compression technique can be applied during the reordering part. Finally, the multiplications are done with 8-bit precision, and accumulations are done on 16-bit precision. With all these features, the accelerator has only 1.3 Watt of power on the FPGA and can achieve of throughput of 255 Images per second on the 5 convolutions layer of AlexNet.

5.0 AN 8-BIT DNN FPGA ACCELERATOR INTEGRATED WITH HYBRID MEMORY CUBE

Our 8-bit DNN accelerator is specially conceived with weight sharing and weight pruning in mind. Weight sharing is integrated into the multiplication pipeline. Special data layout is used to improve sparse matrix multiplication. Whereas AlexNet could fit on-chip for custom ASIC [31], larger networks even with deep compression may overflow FPGA BRAM capacity, so we decided to use the HMC as an external memory. The only limitation regarding the size of the network is the larger layer must fit inside the HMC. Figure 3 shows an overview of the configuration of our server for the 8-bit DNN architecture. For testing the PE, the CPU is used as data fetch and data layout.

The next section 5.1 will give a presentation of the accelerator we propose. The hardware implementation is first discussed in section 5.1.1. This section will give an overview of the architecture of our accelerator. 5.1.1.1 gives a detail implementation of our multiplier and 5.1.1.2 discusses the tree of adders. Section 5.1.2 provides the reordering implementation. Finally, 5.2 gives an evaluation of our design.

This 8-bit DNN accelerator can work both on fully connected layers and on convolution layer. Only the data layout change in function of the type of layer.

5.1 DESIGN IMPLEMENTATION

This section will provide information about the realization of our 8-bit DNN accelerator. The PE implementation is presented in Figure 11.

5.1.1 Hardware design

The high-level hardware architecture is given in Figure 11. Figure 11 displays the four different clock domains in our design. The rx and tx clocks from the HMC, either for read and write, have a frequency of 187.5 Mhz. These two clocks may be out of phase. The next clock domain is the PCIe interface, with the frequency of 100Mhz. This clock is used to bring weights and addresses on the FPGA. Finally, the design clock, with the frequency of 500Mhz. This last clock is where the operations for the matrix calculation are done. To cross different clock domains, a memory with 2 ports is used. The input buffer is the more complicated interface since the HMC data can be out of order. Double buffering or ping pong algorithm is also used in the input buffer to hide memory latency. Each input buffer has two ports of the HMC to bring data on the chip. With this two ports, the bandwidth will be enough to sustain the dataflow requirement. Indeed, some stalls in the main matrix multiplication pipeline will decrease the data request for our PE and make the ratio two ports for one buffer sustainable.

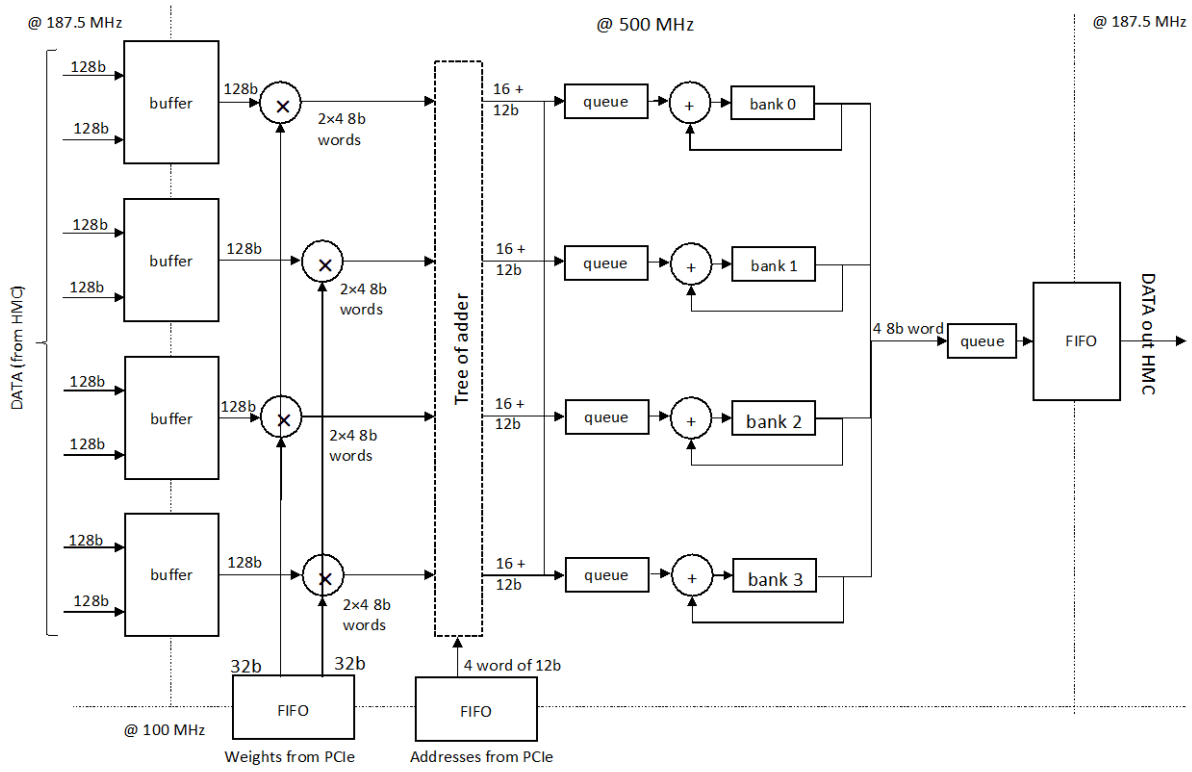

Figure 11. *High level hardware architecture*

Multiplier and adder will be more detailed in the next section (5.1.1.1, 5.1.1.2 respectively). Multiplications are done on 8-bit format but accumulations are done in 16-bit format. At the end, 16-bit format words are transformed to 8-bit format words, without any check for overflow. Addresses are brought form CPU to PCIe, and encode two pieces of information, the destination addresses and if this addition is the last operation we should do on the corresponding output pixel. To use only one memory port, the addressable space is divided into 4 different banks. The lower address bit of the address is used to determine in which bank the data will go. In case, multiple data have to go to the same bank, a queue is added before each bank. Bank finalize the accumulation by adding the data with the same address.

5.1.1.1 Multiplier implementation

The multiplication has been optimized to decrease the number of DSPs used. The design can perform 4 multiplications and 2 additions by using only one DSP, two carry-8, some LUTs and some registers. With this optimization, our design can reach a 66 Giga Operation Per Second (GOPS) and uses only 16 DSPs and 96 carry-8.

The multiplier module calculates two outputs per clock cycle and effectuates the following calculation: $out_1 = (In_0 + In_1) * W$ and $out_2 = (In_2 + In_3) * W$. The implementation of this optimization is presented in Figure 12.

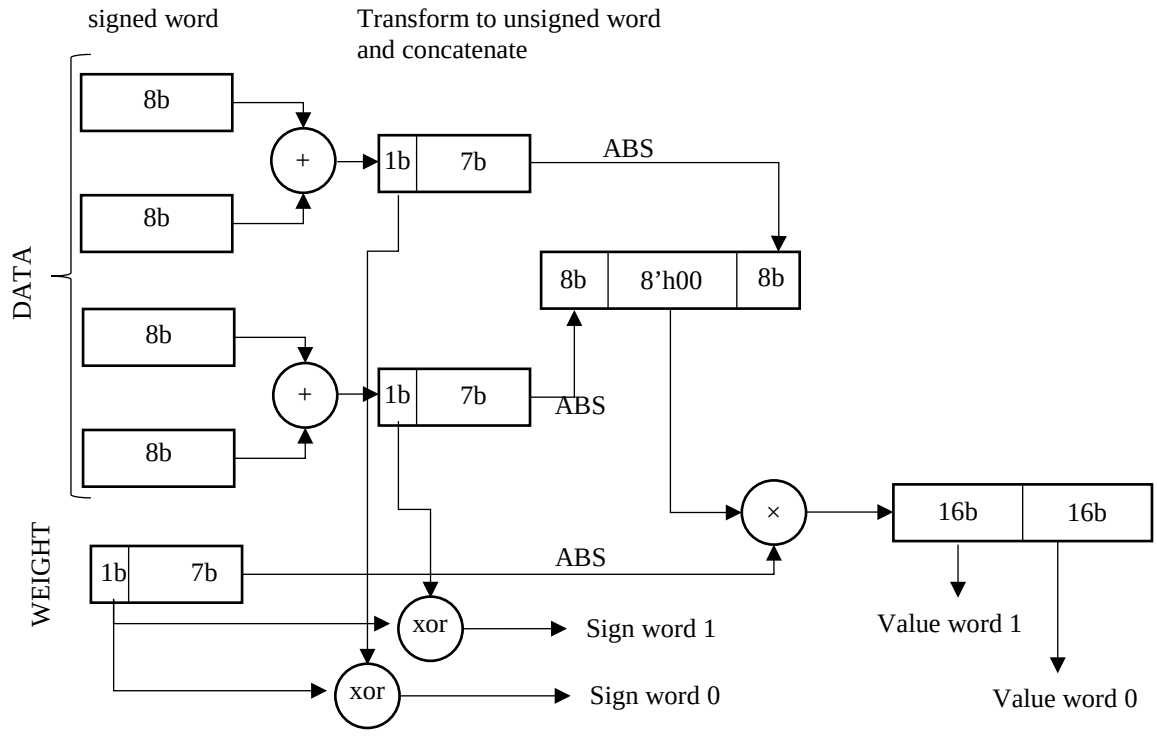

Figure 12. *Multiplier module*

Two 8-bit signed numbers are added together, then a bit sign is calculated and the two values are transformed into unsigned representation. The two 8-bit words can be concatenated with padding between them to form a 24-bit word. This number is multiplied by an unsigned weight to form a 32-bit word with half of this word representing the value of out_1 and the other half represents the value of out_2. The sign is calculated by looking if the bit signed are equals or not between the two sums and the weight. On Figure 12, registers are not displayed to make the figure clearer. The module is implemented on a 3-stage pipeline.

The optimization is designed with weight sharing in mind where multiple input data will have the same weight. This technique is particularly useful with Xilinx DSP, where the DSP can perform a 27×18 two complement multiplication [32]. So, the 24-bit for our multiplication fits perfectly in the DSP width. Due to the width of the DSP, the maximum weight size will be 11 bit. The 8-bit encoding precision should be sufficient to have very small accuracy degradation [33]. In case of the user wants more precision format, if the inputs are still 8-bit width, the weight can be 11-bit width then 11-bit multiplications are performed.

5.1.1.2 Tree of adders

The tree of adders is responsible to decrease the number of different addresses to a reasonable number to address the memory. To satisfy the two ports by memory (one for read and one for write) we use multiple banks to have more virtual ports available. A queue is added at the beginning of each port in case multiple data with the same addresses arrived on the same clock cycle. The number and the order of valid data which arrive inside the queue are random. In consequences, the number of combination the queue must handle is proportional to the number of input of the queue. To keep this process in one clock cycle at 500MHz, this number is limited to a queue with four inputs.

The tree of adders must start with 32 inputs, 16 multipliers with 2 output, and add the same addresses together to out a maximum of four outputs. The layout of the address can be either fix or random with the only constraint to have the maximum of four addresses by clock cycle.

A random address distribution for the 32 input of the tree of adders leads to near 200% of overhead for register and LUT and almost 100% for carry-8 compare to a fix distribution of address. Due to the large overhead to implement a tree of adders with random distribution addresses, the tree of adders will add data with a fix position. The chosen mapping is to add 8 (one quarter) with a stride of 2. Half of the even number will be added together, and half of the odd together will be added together… The idea is to have the four addresses in every clock cycle to maximize the bank utilization. Even positions are added together because convolution has by nature weight sharing since 2 consecutive output will have the same weights.

5.1.2 Data layout implementation

The software implementation is responsible for the mapping of the address, the fetch of the data and the weight. Figure 13 shows our algorithm to encode the weight sharing for a 3 by 3 weight matrix. The weight matrix is transformed from sparse matrix to one vector containing the value and one other matrix containing the position in the sparse matrix according to the value.

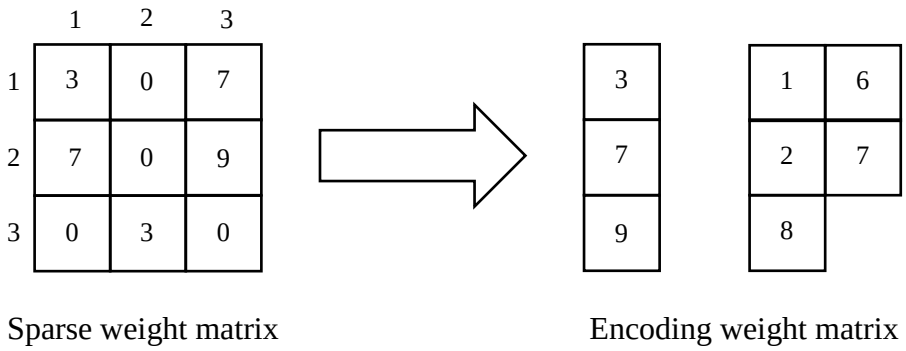

Figure 13. *Encoding weight sharing*

Then using Matlab, a direct indexing is used between the encoding weight matrix and the Im2Col from the input. The address contained in the encoding weight matrix fetch the corresponding data in the Im2Col representation.

In case, the data layout is done in CPU, the software data movement will imply a large overhead in term of timing as we can see in the 5.2.1 section. Realizing this section on the FPGA platform should bring better performance. A prototype of Verilog code has been done to test the portability of the algorithm to FPGA device.

5.2 EVALUATION

We test our accelerator using Vivado 2016.4 in simulation, synthesis, and implementation.

5.2.1 Weight pruning and weight sharing

We compare the size of data we must transfer for 3 different layouts, results for the convolution layer of AlexNet are summarized in Figure 14. The first technique is the classical image to column (Im2Col) algorithm. The input matrix is cut into smaller matrices with the same size than the weight. Then, input and weight matrices are reshaped to a vector format. All data, even the data corresponding to a zero-weight value are transmitted. The two other techniques are to use the encoding presented in 5.1.2. The first one is this technique applied after deep compression algorithm and the second one is for dense integer matrix. In these two techniques, only useful data (non-zero weights) are transmitted and weight sharing is used.

For the first layer, Deep Compression technique does not bring a lot of benefits. Indeed, even after deep compression, 84% of the weight is still non-zero value. For this layer, the three techniques are about 100MB of data to be transferred. The two encoding techniques have a small overhead due to 16B alignment from the HMC. Layer 2, 3, 4, 5 present the same pattern, where Im2Col is 1.1 to 1.5 larger than the encoding techniques alone and 2.3 to 3.2 larger than the encoding techniques combined with the deep compression algorithm.

Except for the first layer, the encoding techniques present 2 times smaller data size when the deep compression algorithm is used. The deep compression algorithm alone reduces significantly the model size to 35% for these layers. Our encoding technique can keep almost the same compression result than deep compression algorithm. This close result between the deep

compression on the model size and on our data size shows the loss due to fixing address position inside the tree of adders is very small and is not worth the large hardware overhead to implement random distribution address in the tree of adders.

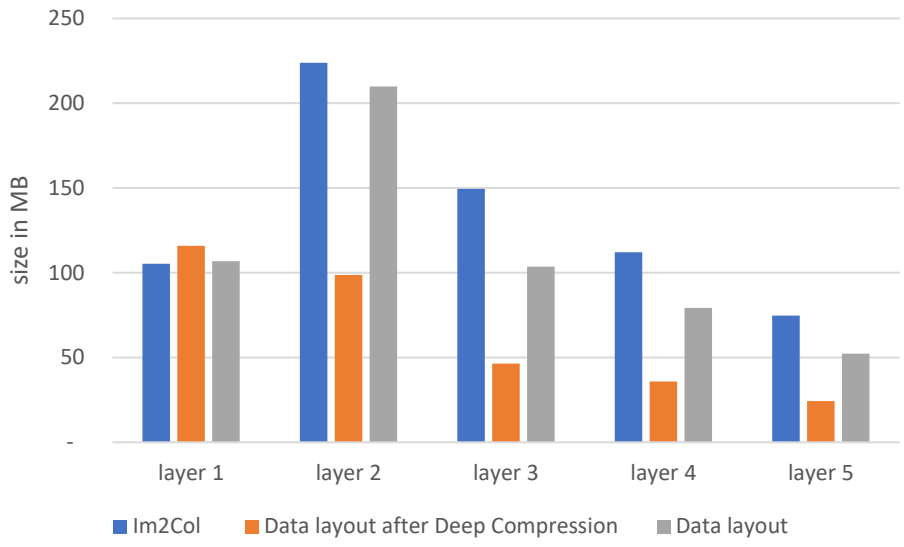

Figure 14. *Data size after encoding techniques*

The performance of our PE is directly linked to this size by a linear ratio. A smaller data size means also a shorter execution time. Execution time for our PE is presented in Table 7.

Table 7. *Execution time for our 8-bit PE*

	execution time in ms with deep compression	execution time in ms without deep compression
layer 1	1.26	1.16
layer 2	1.07	2.27
layer 3	0.50	1.12
layer 4	0.39	0.86
layer 5	0.26	0.57
total	3.48	5.97

With this Table 7, we can see that our design finish the 5-convolution layers of AlexNet 1.7 faster when the data layout used the deep compression technique.

Comparing to the size of the input matrix, our data layout even with deep compression is in order of magnitude of 1000 bigger than the input size. The number of data that have to be transferred from the HMC to the FPGA, may cause a very large overhead. For this reason, we believe that the data layout should be integrated to the FPGA.

5.2.2 Comparison with State-of-the-art Design

The power and throughput of our PE are reported in Table 8. The power consumption is only 1.3 Watt according to the Vivado power simulator. Among this 1.3 Watt, half of it comes from static power.

We compare our work with 2 others accelerator proposed in [14], [28]. We did not evaluate our design with ternary or binary implementation since these implementations decrease the accuracy of the neural network, whereas our work should keep the same accuracy.

We want to moderate the result presented in Table 8 because this table only reports the PE for our design and does not report the fetch of the data. The data layout should increase the power consumption and may slightly decrease the throughput.

Table 8. *Comparison with states of the art*

	Throughput (Img/sec)	power (W)	Images/sec/W
Our PE w Deep compression	255	1.3	196
Our PE wo Deep compression	101	1.3	78
DLA [14]	1020	45	23
TitanX [28]	5882	227	26

DLA works with 16-bit fixed point precision, and the implementation with TitanX works on 32-bit floating point, however, our design achieves the same accuracy than this two implementations. With the precaution state above, we can see that our PE have a 9.6 better throughput over watt than previous best FPGA implementation, and overcome for this metric the GPU implementation.

Table 9 shows the execution time from our PE and EIE presented in [31] for Fully Connected Layer (FC). Even with deep compression, EIE performs better than our PE for FC, our PE is 1.45 and 1.25 slower than EIE for FC1 and FC2 of AlexNet respectively. EIE has also 0.59

Watt of power compare to 1.3 for our design. However, EIE is implemented on ASIC and has a frequency of 800Mhz, whereas our FPGA is limiting to 500Mhz. EIE also performs only for FC layer and cannot handle Convolution Layer.

Table 9. *Fully Connected layer execution time*

	execution time in µs for FC1	execution time in µs for FC2
PE with deep compression	41	15
PE without deep compression	385	171
EIE (64PE) [31]	28	12

6.0 SCALING UP THE DESIGN

The new accelerator which performs a 4:1 ratio between multiplications and DSP is hard to scale up because the new encoding scheme is difficult to realize on FPGA platform. In order to scale up more easily, we propose to relax the constraint on the PE side to be able to use the new data request scheme proposed in 4.0 . The new PE, only have a 2:1 ratio between multiplications and DSP and performs the calculation presented in equation 6.

$$mult = |W1| \times (|In_1| \times 2^{16} + |In_2|) \quad \textbf{(6)}$$

By combining both the new data scheme proposed in 4.0 and a 2:1 ratio between multiplications and DSP, the design can be easily scaled up as Figure 15 and Figure 16 show.

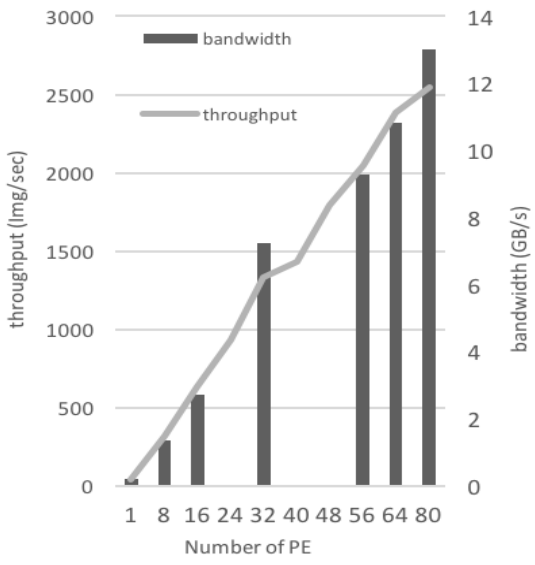

Figure 15. *AlexNet bandwidth and throughput*

To parallelize our accelerator, multiple PEs are instantiated, one by output features map. The CNN algorithm allows each output features map to be independent one to another. Because output features maps share the same inputs and only the weights are different, BRAM can be shared when we scale up our PE. Shared BRAM should be carefully selected to avoid long routing that could break the setup time. Figure 15 shows the increase in term of throughput and bandwidth when we increase the number of PE in our design. In this figure, the bins represent the bandwidth requirement whereas the curves represent the throughput. In Figure 15, we can see a pattern. The pattern consists of two phases. The first phase is the direct proportionality between the number of PE and the throughput. For AlexNet this phase goes from 1 to around 20 PEs. During this phase, the number of output features maps calculated in parallel is small compared to the number of output features maps. Thus, the parallel number of features map can be considered a common factor with the number of output features maps and so there is no redundant calculation. When we continue to increase the number of PE. The parallel factor becomes larger, and do not divide the output features maps so the throughput evolves with a kind of step function. The throughput stays constant until the number of PE reach the next common factor with the output features maps. When we reach this number, the throughput has an improvement. Regarding the bandwidth, we fetch the same size of data no matter the number of PE we use. However, when we increase the number of PE, the execution time decreases which leads to an increase in the bandwidth requirement. There is a linear relationship between the bandwidth requirement and the decrease of the execution time.

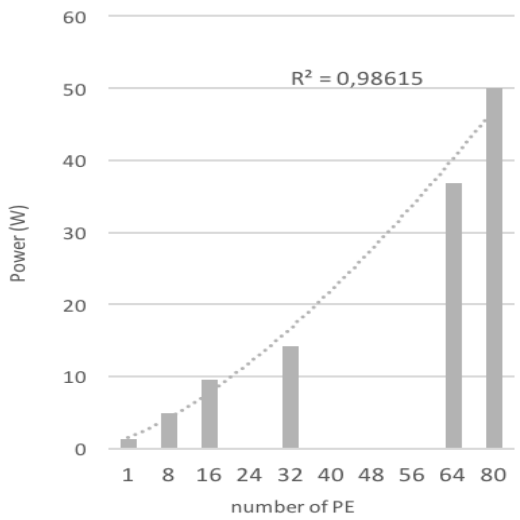

Figure 16. *Power in function of the number of PE*

Finally, Figure 16 gives the evolution of the power consumption when more PEs are added to the design. The curve is a power function approximation with a R2 = 0.98615. If we look at throughput over power ratio, it increases until 32 PEs since the throughput is linear and the power consumption is still in the asymptote curve. For AlexNet, this ratio is 40 Images/sec/w for only one PE, and it increases to 94 Images/sec/w for 32 PEs. However, if we continue to increase the number of PE, the power start the exponential growth and the throughput over power ratio decrease. The ratio for AlexNet is 51 Images/sec/w with 80 PEs.

Finally, Table 10 shows an overall comparison in term of throughput, power and throughput over power for four CNN accelerators. The first design, DLA, was proposed in [14]. It is a 16-bit CNN accelerator on a Arria 10 FPGA and uses Winograd transform to accelerate the matrix multiplication and share exponent between multiplication. The second design is a Kintex UltraScale accelerator named caffeine proposed in [17]. Finally, we compare two GPU

accelerators proposed in [28]. The first one is a 32-bit design whereas the next one is a int8 accelerator on Nvidia TitanX. We use 32 PEs to have the best throughput over power ratio.

Table 10. *A comparison between our design and the state-of-the-art for AlexNet*

Design	FPS	Power (W)	FPS/W
Our work	1335	14	94
DLA [14]	1020	45	23
Caffeine [17]	104	25	4
TitanX FP-32 [28]	5882	227	26
TitanX FP-8 [28]	18714	227	82

Among these accelerators, the highest throughput is the GPU accelerator, with int-8 calculations and 128 batch size. However, the high-power consumption of the GPU (227 Watt) can be an issue in a constraint environment and the 128-batch size used may result in high latency. Regarding the throughput over power ratio, our accelerator leads with a 1.14 performance improvement compares to the Int-8 GPU implementation. Comparing with other FPGA designs, our work shows a 1.31× better throughput that DLA, and a 4.1× improvement in term of FPS/W. Caffeine implements their design on the same FPGA platform however our work presents a 13.35× throughput improvement.

7.0 CONCLUSION AND FUTURE WORKS

We describe two novel CNN architectures using the HMC as external memory. During the first implementation, the user request has been carefully chosen to match the HMC requirement in order to maximize the bandwidth of the HMC. To do so, a new algorithm has been proposed for CNNs. This new algorithm allows data to be requested sequentially, following the input matrix layout and maximizing the user request size for the HMC. To resolve the DSP utilization problem, we propose to use a multi-CLP design where each of the CLP can be linked to one HMC port. This one to one mapping is permitted by the high HMC parallelism capacity and allows the design to have a better DSP utilization.

The aim of the second design is to continue to improve the DSP utilization and performance over power ratio. The first optimization included is to decrease the bit precision encoding format to perform multiplication and accumulation. Secondly, we re-design the PE to integrate weight sharing techniques inside the hardware and to maximize the DSP utilization. This new PE design has a 4:1 ratio for multiplication which means the design can perform 4 multiplications using only one DSP. The testing of this new PE shows great promise in term of throughput and efficiency especially when this PE is coupled with weight pruning and weigh sharing techniques.

Future work may include working on different Neural Network.

BIBLIOGRAPHY

[1] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” *Arxiv*, pp. 1–28, 2016.

[2] N. Suda *et al.*, “Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks,” *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, vol. 2016, pp. 16–25.

[3] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both weights and connections for efficient neural networks,” *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. MIT Press, pp. 1135–1143, 2015.

[4] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” Oct. 2015.

[5] J. Yu *et al.*, “Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism,” *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, 2017.

[6] E. Nurvitadhi *et al.*, “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?”

[7] P. Rosenfeld and B. Jacob, “Title of dissertation: Performance Exploration of the Hybrid Memory Cube,” pp. 1–147, 2014.

[8] J. T. Pawlowski, “Hybrid Memory Cube (HMC),” *IEEE Hot Chips 23 Symp. HCS*, vol. 115, no. 174, pp. 65–70, 2011.

[9] S. Khoram, J. Zhang, M. Strange, and J. Li, “Accelerating Large-Scale Graph Analytics with FPGA and HMC,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 82–82.

[10] D. Rozhko, G. Elliott, D. Ly-Ma, P. Chow, and H.-A. Jacobsen, “Packet Matching on FPGAs Using HMC Memory,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA ’17*, 2017, pp. 201–206.

[11] N. Mcvicar, C.-C. Lin, and S. Hauck, “K-Mer Counting Using Bloom Filters with an FPGA-Attached HMC,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 203–210.

[12] J. Zhang, S. Khoram, and J. Li, “Boosting the Performance of FPGA-based Graph Processor using Hybrid Memory Cube,” *Proc. 2017 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA ’17*, pp. 207–216, 2017.

[13] M. Alwani, H. Chen, M. Ferdman, and P. Milder, “Fused-Layer CNN Accelerators,” *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO ’16). IEEE Comput. Soc. Washington, DC, USA*, pp. 1–12, 2016.

[14] U. Aydonat, S. O ’connell, D. Capalija, A. C. Ling, and G. R. Chiu, “Deep Learning Accelerator on Arria 10,” *Proc. 2017 ACM/SIGDA Int. Symp. field-programmable gate arrays*, pp. 55–64, 2017.

[15] X. Han, D. Zhou, S. Wang, and S. Kimura, “CNN-MERP: An FPGA-Based Memory-Efficient Reconfigurable Processor for Forward and Backward Propagation of Convolutional Neural Networks,” 2017.

[16] Y. Shen, M. Ferdman, and P. Milder, “Maximizing CNN Accelerator Efficiency Through Resource Partitioning,” *Proc. ISCA ’17, Toronto, ON, Canada*, p. 13 pages, 2017.

[17] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks,” *Proc. 2016 Int. Conf. Comput. Aided Des. ICCAD ’16, New York, NY, USA, 2016. ACM.*

[18] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks,” *Proc. 2015 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, FPGA ’15, pages 161–170, New York, NY, USA, 2015. ACM.*

[19] J. Albericio *et al.*, “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, vol. 44, no. 3, pp. 1–13.

[20] Q. Jiantao *et al.*, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016.

[21] R. Hadidi *et al.*, “Demystifying the Characteristics of 3D-Stacked Memories: A Case Study for Hybrid Memory Cube,” 2017.

[22] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracywith 50X fewer parameters and <1MB model size.,” 2016.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Proc. 25th Int. Conf. Neural Inf. Process. Syst. (NIPS ’12). Curran Assoc. Inc., Red Hook, NY, USA*, pp. 1097–1105, 2012.

[24] C. Szegedy *et al.*, “Going deeper with convolutions,” *Proc. 2015 IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR ’15)*, pp. 1–9.

[25] Micron, “HPCC RandomAccess (GUPS),” pp. 1–15, 2016.

[26] Micron, “HMC Controller IP User Guide,” pp. 1–35, 2016.

[27] P. Y. Simard, D. Steinkraus, and J. C. Platt, “Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis,” *Proc. Seventh Int. Conf. Doc. Anal. Recognit. (ICDAR 2003)*, p. 6 pages, 2003.

[28] NVIDIA, “8-bit inference with TensorRT, 2017.”

[29] Intel FPGA, “Arria 10 - Overview.” [Online]. Available: https://www.altera.com/products/fpga/arria-series/arria-10/overview.html. [Accessed: 29-Aug-2017].

[30] Xilinx and Inc, “UltraScale Architecture and Product Data Sheet: Overview (DS890),” *DS890*, no. 11, 2017.

[31] S. Han *et al.*, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 243–254.

[32] Xilinx and Inc, “UltraScale Architecture DSP Slice User Guide Revision History,” 2017.

[33] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “incremental network quantization: toward lossless CNNs with low-precision weights,” *ICLR*, 2017.