electronics

MDPI

Article

A Uniform Architecture Design for Accelerating 2D and 3D CNNs on FPGAs

Zhiqiang Liu [1,*], Paul Chow [2], Jinwei Xu [1], Jingfei Jiang [1], Yong Dou [1] and Jie Zhou [1]

[1] National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China; xujinwei13@nudt.edu.cn (J.X.); jingfeijiang@nudt.edu.cn (J.J.); yongdou@nudt.edu.cn (Y.D.); zhoujie_kd@163.com (J.Z.)

[2] Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada; pc@eecg.toronto.edu

* Correspondence: liuzhiqiang@nudt.edu.cn

Received: 3 December 2018; Accepted: 3 January 2019; Published: 7 January 2019

Abstract: Three-dimensional convolutional neural networks (3D CNNs) have gained popularity in many complicated computer vision applications. Many customized accelerators based on FPGAs are proposed for 2D CNNs, while very few are for 3D CNNs. Three-D CNNs are far more computationally intensive and the design space for 3D CNN acceleration has been further expanded since one more dimension is introduced, making it a big challenge to accelerate 3D CNNs on FPGAs. Motivated by the finding that the computation patterns of 2D and 3D CNNs are very similar, we propose a uniform architecture design for accelerating both 2D and 3D CNNs in this paper. The uniform architecture is based on the idea of mapping convolutions to matrix multiplications. A customized mapping module is developed to generate the feature matrix tilings with no need to store the entire enlarged feature matrix on-chip or off-chip, a splitting strategy is adopted to reconstruct a convolutional layer to adapt to the on-chip memory capacity, and a 2D multiply-and-accumulate (MAC) array is adopted to compute matrix multiplications efficiently. For demonstration, we implement an accelerator prototype with a high-level synthesis (HLS) methodology on a Xilinx VC709 board and test the accelerator on three typical CNN models: AlexNet, VGG16, and C3D. Experimental results show that the accelerator achieves state-of-the-art throughput performance on both 2D and 3D CNNs, with much better energy efficiency than the CPU and GPU.

Keywords: 2D CNN; 3D CNN; accelerator; uniform architecture; FPGA; HLS; matrix multiplication; 2D MAC array

1. Introduction

In recent years, convolutional neural networks (CNNs) have gained great success in various computer vision applications such as image classification [1], object detection [2], and face recognition [3]. CNNs have been primarily applied on 2D images to automatically extract spatial features and have significantly enhanced the image classification accuracy. To effectively incorporate the motion information in video analysis, 3D CNNs with spatiotemporal convolutional kernels are proposed. Owing to the ability to capture both spatial and temporal features, 3D CNNs have been proved to be very effective in many video-based applications including object recognition [4], hand gesture recognition [5], and human action recognition [6].

CNNs require vast amounts of memory as there are millions of parameters in a typical CNN model. Meanwhile, CNNs are computationally intensive with over billions of operations for the inference of one input. For example, VGG16 [7], a real-life 2D CNN model for image classification with 16 layers, takes around 31 GOPs for the inference of one image. C3D [6], a real-life 3D CNN model for

human action recognition with only 11 layers, takes more than 77 GOPs for the inference of a video volume. As a result, CNN applications are mainly run on clusters of server CPUs and GPUs. What is more, with the availability of compatible deep learning frameworks, including Caffe [8], Theano [9], and TensorFlow [10], training and testing CNN models become much easier and more efficient on these platforms. While CPU and GPU clusters are the dominant platforms in CNN applications, customized accelerators with better energy efficiency and less power dissipation are still required. For example, in the case of embedded systems with limited power such as auto-piloted car and robots, higher energy efficiency is critical to increase the use of CNNs.

Owing to the advantages of high performance, energy efficiency, and flexibility, FPGAs have attracted attention to be explored as CNN acceleration platforms. Moreover, high-level synthesis (HLS) tools from FPGA vendors, such as Xilinx Vivado HLS and Intel FPGA SDK for OpenCL, reduce the programming difficulty and shorten the development time significantly, making FPGA-based solutions more popular. As reported in the recent surveys [11,12], many FPGA-based CNN accelerators have been proposed for 2D CNNs and many tool-flows for mapping 2D CNNs on FPGAs have been released. However, there are very few studies on accelerating 3D CNNs on FPGAs. Three-D CNNs are far more computationally intensive than 2D CNNs, and generate far more intermediate results during execution as the input is a video volume instead of a single image, causing greater memory capacity and bandwidth demands. In addition, the design space for 3D CNN acceleration is further expanded since the temporal dimension is introduced, making it even difficult to determine the optimal solution. Therefore, current accelerator designs for 2D CNNs are not fit for accelerating 3D CNNs directly. For example, the designs in [13,14] adopt customized computation engines to compute 2D convolutions. As there is one more dimension in 3D convolutions, new computation engines are required with this approach. The design in [15] computes 2D convolutions with the Fast Fourier Transform (FFT) algorithm. This approach is proved to be effective only for large convolutional kernels like 11×11 or 7×7 and will be less efficient for small convolutional kernels like 3×3 or 1×1. Some designs accelerate 2D convolutions by reducing the computational requirements with the Winograd algorithm [16], and the design in [17] even extends the Winograd algorithm to adapt to 3D convolutions. However, the Winograd algorithm is very sensitive to the size of convolutional kernels. For convolutional kernels with different size, different transformation matrices are required. Hence, the Winograd algorithm is perfectly suitable for CNN models with uniform-sized convolutional kernels like VGG while not suitable for CNN models with multi-sized convolutional kernels like AlexNet. Another approach is mapping convolutions to matrix multiplication operations, which is typically adopted in CPU and GPU implementations. Refs. [18,19] adopt this approach in their accelerator designs for 2D CNNs and implement accelerators on FPGAs using the OpenCL framework. A main concern of this approach is that it introduces high degree of data replications in the input features, which can lead to either inefficiency in storage or complex memory access patterns. Especially, the weight matrix and the feature matrix are both enlarged by a factor of the kernel temporal depth in 3D convolutions, which further lifts the memory requirement.

We analytically find that the computation patterns in 2D and 3D CNNs are very similar. Motivated by this finding, we attempt to design a uniform accelerator architecture for both 2D and 3D CNNs. In the case of FPGA-based clouds, a uniform architecture allows switching of acceleration services without reprogramming the FPGAs. For ASICs, which are not programmable, a uniform architecture expands the applicability of the ASIC. The uniform architecture design is based on the idea of mapping convolutions to matrix multiplication operations. The first challenge comes from the data replications when mapping the input features to the feature matrix. It will introduce multiplied memory access overheads when storing the entire feature matrix off-chip, and will cost a large amount of memory space when storing the entire feature matrix on-chip. We propose an efficient matrix mapping module that avoids data replications by reusing the overlapped data during the sliding of convolutional windows. In addition, the mapping module generates only a tiling (several columns) of the feature matrix on-the-fly instead of generating the entire one before matrix multiplication, which saves on-chip

memory consumption. The second challenge is that the weight matrix and feature matrix are enlarged by a factor of the kernel temporal depth in 3D convolutions compared to 2D CNNs. Accordingly, it lifts the memory consumption by a factor of the kernel temporal depth when storing the weight matrix and feature matrix on-chip. To guarantee the uniform architecture can be applied to large CNN models and be deployed on platforms with limited on-chip memory capacity, we adopt an effective splitting strategy. A convolutional layer with a large amount of input channels will be split into multiple convolutional layers with a smaller amount of input channels. The third challenge is how to compute matrix multiplications efficiently on FPGAs. Different to the OpenCL-based computation framework in [18,19], we adopt a 2D MAC array for matrix multiplications. The 2D MAC array is scalable and the size is mainly determined according to the hardware resources, memory bandwidth and the size of feature maps.

To summarize, our key contributions are as follows:

- We propose a uniform accelerator architecture design supporting both 2D and 3D CNNs, based on the idea of mapping convolutions to matrix multiplication operations. Special efforts are made on memory optimizations and computations to enhance throughput performance;
- We analytically model the resource utilization and throughput performance of our architecture, which helps to configure an accelerator on a specific platform within certain constraints including hardware performance, memory bandwidth and clock frequency;
- We demonstrate the architecture design by implementing an accelerator on the Xilinx VC709 board with the High-level synthesis (HLS) methodology. Three typical CNN models including AlexNet, VGG16, and C3D, are tested on the accelerator. Experimental results show that the accelerator achieves over 850 GOP/s for convolutional layers and nearly 700 GOP/s overall on VGG16 and C3D, with much better energy efficiency than the CPU and GPU.

The rest of the paper is organized as follows: Section 2 briefly introduces the basic background of CNNs and the design directions of the accelerator architecture; Section 3 presents the architecture design and the main components; Section 4 provides the implementation and optimization details; Section 5 presents the accelerator modeling; Section 6 reports the experimental results; and finally, Section 7 concludes the paper.

2. CNN Basics and Accelerator Design Directions

In this section, we briefly review the operations in convolutional layers of 2D and 3D CNNs and then the accelerator design directions on FPGAs are introduced. The fully-connected layers, activation, and pooling layers are omitted due to their relative simplicity.

2.1. Operations in Convolutional Layers of 2D CNNs

Figure 1a illustrates the process of a 2D convolution. A 2D convolution applied to an image results in another image. In convolutional layers of 2D CNNs, the input and output features are images with multiple channels. For example, the input feature of the first convolutional layer has three channels: Red, green, and blue. A 2D convolution is applied to each channel of the input feature and the generated images are then accumulated resulting in one channel of the output feature, as shown in Figure 1b. For simplicity, we use X to indicate the input feature with a size of $c \times h \times w$ and Y to indicate the output feature with a size of $m \times h \times w$. Here, c and m are the number of input and output channels, and h and w are height and width of each feature. We use W to indicate the weights, which contains $m \times c$ kernels with a size of $k \times k$. Suppose the sliding stride is 1, then each pixel $Y[mm][hh][ww]$ in the output feature is calculated by:

$$\sum_{ch=0}^{c-1} \sum_{rr=0}^{k-1} \sum_{cc=0}^{k-1} W[mm][ch][rr][cc] * X[ch][hh+rr][ww+cc] \quad (1)$$

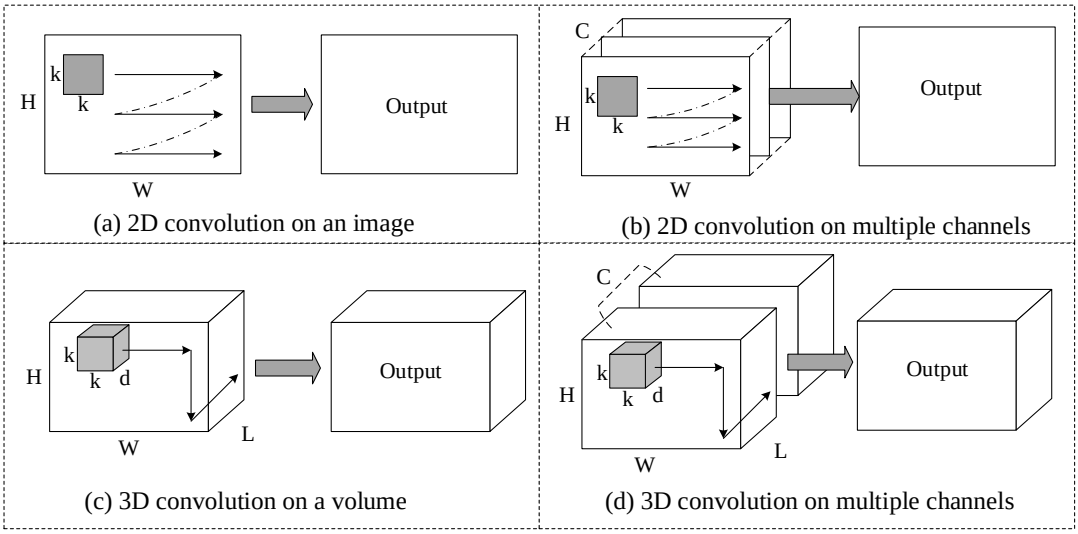

Figure 1. 2D and 3D convolution operations. Applying 2D convolution on an image (**a**) or multiple channels of image (**b**) results in an image. Applying 3D convolution on a video volume (**c**) or multiple channels of video volume (**d**) results in another volume.

2.2. Operations in Convolutional Layers of 3D CNNs

Figure 1c illustrates the process of a 3D convolution. A 3D convolution applied to a video volume results in another volume. Similarly, in convolutional layers of 3D CNNs, the input and output features are video volumes with multiple channels. A 3D convolution is applied to each channel of the input feature and the generated volumes are then accumulated resulting in one channel of the output feature, as shown in Figure 1d. The input and output features are indicated by X with a size of $c \times l \times h \times w$ and Y with a size of $m \times l \times h \times w$, where l is the number of frames while the other variables have the same meaning as above. The weights W contains a total of $m \times c$ kernels with a size of $d \times k \times k$, where d is the kernel temporal depth and k is the kernel spatial size. Suppose the sliding stride is 1, then each pixel $Y[mm][ll][hh][ww]$ in the output feature is given by:

$$\sum_{ch=0}^{c-1} \sum_{dd=0}^{d-1} \sum_{rr=0}^{k-1} \sum_{cc=0}^{k-1} W[mm][ch][dd][rr][cc] * X[ch][ll+dd][hh+rr][ww+cc] \quad (2)$$

Compared to the convolutional layers in 2D CNNs, there is an accumulation along the temporal dimension, as shown in Equation (2). By switching the order of the two outer accumulations, we get Equation (3). We can find that the inner three accumulations in Equation (3) are very similar to Equation (1) since the loop variable along the temporal dimension dd is fixed.

$$\sum_{dd=0}^{d-1} \sum_{ch=0}^{c-1} \sum_{rr=0}^{k-1} \sum_{cc=0}^{k-1} W[mm][ch][dd][rr][cc] * X[ch][ll+dd][hh+rr][ww+cc] \quad (3)$$

We can further combine the outer two loops and hence get Equation (4), which is almost the same as Equation (1) except that the number of input channels is enlarged by a factor of d. That is to say, 3D convolutions can be computed in the same way as 2D convolutions.

$$\sum_{ch=0}^{d*c-1} \sum_{rr=0}^{k-1} \sum_{cc=0}^{k-1} W[mm][ch\%c][ch/c][rr][cc] * X[ch\%c][ll+ch/c][hh+rr][ww+cc] \quad (4)$$

2.3. Convolution as Matrix Multiplication

Two-D convolutions can be mapped as matrix multiplication operations by flattening and rearranging the weights and input features. As illustrated in Figure 2, 64×3 kernels with a size of 3×3 are mapped to a rearranged matrix with dimensions of $64 \times (3 \times 3 \times 3)$. All three kernels belonging to the same group are flattened horizontally to form a row of the weight matrix. Meanwhile, all three input features with dimensions of 32×32 are mapped to a rearranged matrix with dimensions of $(3 \times 3 \times 3) \times (32 \times 32)$. All the pixels covered by the first convolution window in each channel are flattened vertically to form the first column of the feature matrix. The entire feature matrix can be generated by sliding the convolution window across the features along the column and row directions. After rearrangement, the convolutions are transformed to a general matrix multiplication. The result of the matrix multiplication is an output matrix with dimensions of $64 \times (32 \times 32)$, which is the flattened format of the output features. Notice that the number of the pixels in the feature matrix is $(k \times k)$-fold to that in the input feature, as each pixel is covered by the convolution window $k \times k$ times during sliding the convolution window across the features and hence replicated $k \times k$ times.

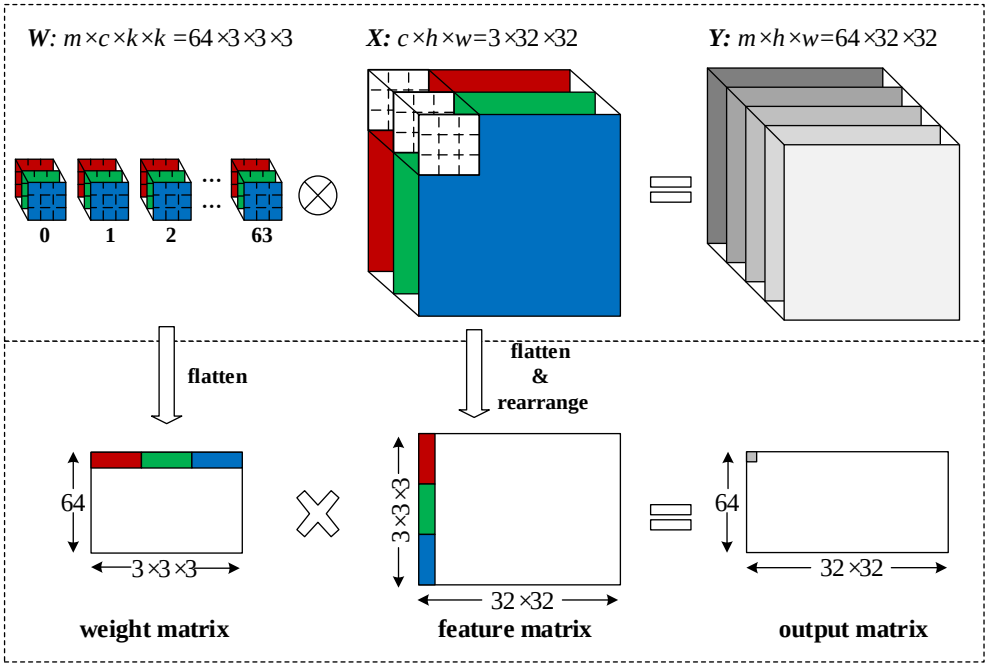

Figure 2. Mapping convolutions to matrix multiplication operations.

Similarly, 3D convolutions can also be mapped as matrix multiplications with the same method. Compared to 2D convolutions, the convolution window slides across the input features not only along the row and column directions, but also the temporal direction. Consequently, each pixel is covered by the convolution window $d \times k \times k$ times and hence replicated $d \times k \times k$ times. The number of pixels in the input feature is enlarged by a factor of $d \times k \times k$.

Here we can find that the approach of mapping convolutions as matrix multiplication operations introduces a high degree of data replications. In CPU and GPU implementations, the entire feature matrix is generated first before computing matrix multiplication. This is not a big problem to CPUs and GPUs as they have abundant memory space and bandwidth. However, to FPGAs with limited on-chip memory capacity and off-chip memory bandwidth, storing the entire feature matrix can be a critical limitation. We will show how we optimize this approach in FPGA implementations with a customized matrix mapping module in the next section.

2.4. Splitting Strategy

As shown in Figure 2, all channels of the input feature are involved to generate one column of the feature matrix. When loading the required pixels of the input feature from off-chip memory to on-chip memory, the burst access pattern is typically adopted to lift memory bandwidth utilization rate. That is to say, we have to store at least several rows for each input channel of the input feature on-chip. In a convolutional layer of a large-scale 2D CNN model, the number of input channels may be very large, 512 or even 1024, which will consume plenty of on-chip memory space. This can be even more severe for a 3D CNN model as the number of input channels is enlarged by a factor of d. Considering that the target hardware platform may have very limited on-chip memory, we adopt a splitting strategy that splits a convolutional layer with a large number of input channels to multiple convolutional layers with a smaller number of input channels. An independent sum layer is introduced to accumulate the results of two convolutional layers. Suppose the on-chip memory can store at most ic_max input channels, then a convolutional layer with c input channels will be split into $\lceil c/ic\_max \rceil$ convolutional layers with ic_max input channels each. Additionally, $\lceil c/ic\_max \rceil - 1$ sum layers will be introduced to accumulate the partial results. The splitting strategy is a kind of matrix blocking or partitioning method in essence. Different to block matrix multiplication, an independent sum layer is introduced which eliminates the need to store intermediate results on-chip and hence saves on-chip memory consumption.

3. Hardware Architecture Design

We propose a uniform architecture design for accelerating both 2D and 3D CNNs based on the idea of mapping convolutions to matrix multiplication operations. The architecture can adapt to convolutional layers with different input dimensions and kernel sizes, and can support large-scale CNN models owing to the splitting strategy. In this section, the details of the customized matrix mapping module, the computation framework, the buffers, and the whole architecture will be presented.

3.1. Matrix Mapping Module

As introduced above, the approach of mapping convolutions as matrix multiplication operations introduces a high degree of data replications, which will lift the memory access overheads when storing the entire feature matrix off-chip. We propose a customized matrix mapping module that avoids data replications by reusing the overlapped data during the sliding of convolutional windows.

Figure 3a–c illustrate how the convolution window slides across columns of the input feature. The pixel in green is involved in k convolutions. Accordingly, it appears in the feature matrix k times in k consecutive rows. We can generate k rows of data in the feature matrix by simply shifting the related row in the input feature k times, as illustrated in Figure 3d. Figure 3e shows the status after the convolution window slides vertically across the rows. There are $k - 1$ overlapped rows between two consecutive slides across the rows, which can be reused when the convolution window slides across the columns. Therefore, each pixel is used $k \times k$ times during the process.

To save on-chip memory consumption, we store only $k + stride$ rows for each channel of the input feature ($stride = 1$ in most cases) instead of the entire input feature. The first k rows are for the activated data involved in current convolutions while the left $stride$ rows are pre-cached for the next slide across the rows. We partition the input feature by the column dimension to provide enough data ports for data shifting, as shown in Figure 3f. The matrix mapping module loads k rows of each channel at the beginning, shifts each row k times to generate $c \times k \times k$ rows, and writes the feature matrix block to the feature FIFOs, as illustrated in Figure 3g. Then the matrix multiplication starts and calculates the first row of the output feature. During the process of matrix multiplication, the next $stride$ rows in each channel of the input feature will be loaded and replace $stride$ rows of data in the same channel according to the First-In-First-Out policy. The process repeats until all the pixels in the output feature are calculated.

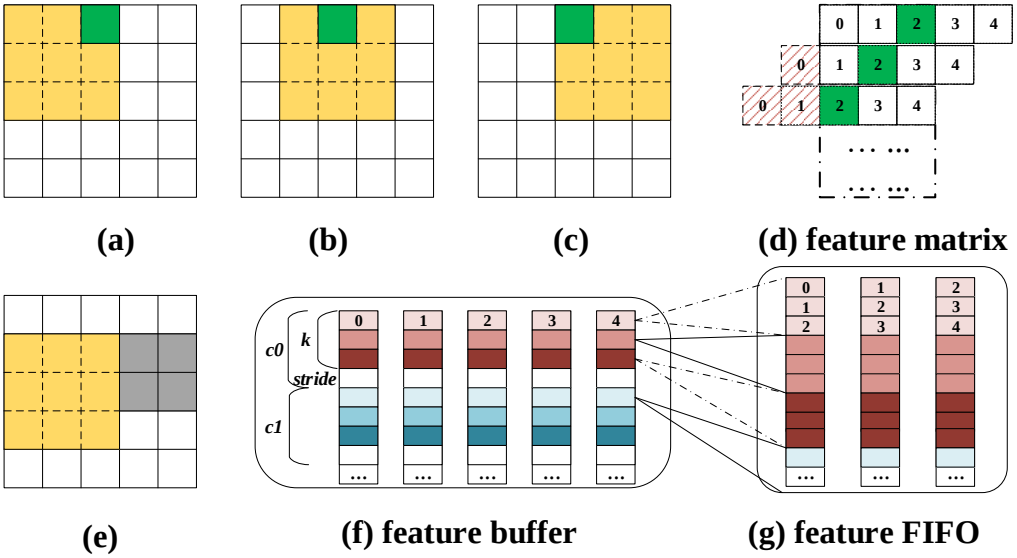

Figure 3. Generating feature matrix blocks in the matrix mapping module.

3.2. 2D MAC Array

If A and B are matrices with dimensions of $M \times N$ and $N \times L$ respectively, then the product matrix C is given by:

$$C[i][j] = \sum_{t=0}^{N-1} A[i][t] \times B[t][j] \quad (0 \leq i < M; 0 \leq j < L) \tag{5}$$

We adopt a 2D MAC array to compute matrix multiplication in the most straightforward way. A MAC unit is composed of a multiplier to calculate the products, an adder to accumulate the products, and a register to keep the partial sum. The MAC unit located at (i, j) receives operands from the i-th row of matrix A and j-th column of matrix B and generates the pixel $C[i][j]$. In the case of CNN accelerations, A is the weight matrix, B is the feature matrix, and C is the output matrix. Suppose there are mr rows and mc columns in the MAC array, then a total of $mr \times mc$ pixels can be generated at once. We adopt a simple matrix partitioning strategy to compute the whole output matrix with the 2D MAC array. As shown in Figure 4, the weight matrix is partitioned into $\lceil m/mr \rceil$ matrix blocks along the row dimension and the feature matrix is partitioned into $\lceil (h \times w)/mc \rceil$ matrix blocks along the column dimension.

The 2D MAC array exploits the parallelism of convolutions in two aspects. The output channel loop is unrolled by a factor of mr and hence mr channels of the output feature can be calculated simultaneously. The column loop of each channel is unrolled by a factor of mc and thus mc pixels in the same row of a channel can be computed in parallel. The 2D MAC array is scalable and the size is determined according to the hardware resources, memory bandwidth, feature size and the number of input channels. Hardware resources, especially the DSP slices on a FPGA chip, determine the maximum number of MAC units that can be assigned to the 2D MAC array. The width of the 2D MAC array is mainly restricted by the memory bandwidth. We can find an optimal value for the width so that the 2D MAC array is well matched with the memory bandwidth. The 2D MAC array will be under-utilized when the real width is greater than the optimal value, and the memory bandwidth is not fully exploited when the real width is less than the optimal value. Also, the width of the 2D MAC array is closely related to the feature size of a CNN model. For example, if the feature size is 56×56, it is better to deploy 28 columns of MAC units instead of 32, which achieves the same throughput performance with fewer MAC units. The height of the 2D MAC array is closely related to the number of output channels in a CNN model. A common divisor of all the output channel numbers in all

convolutional layers is most preferred. A power of two may also be a good choice for the height of the 2D MAC array.

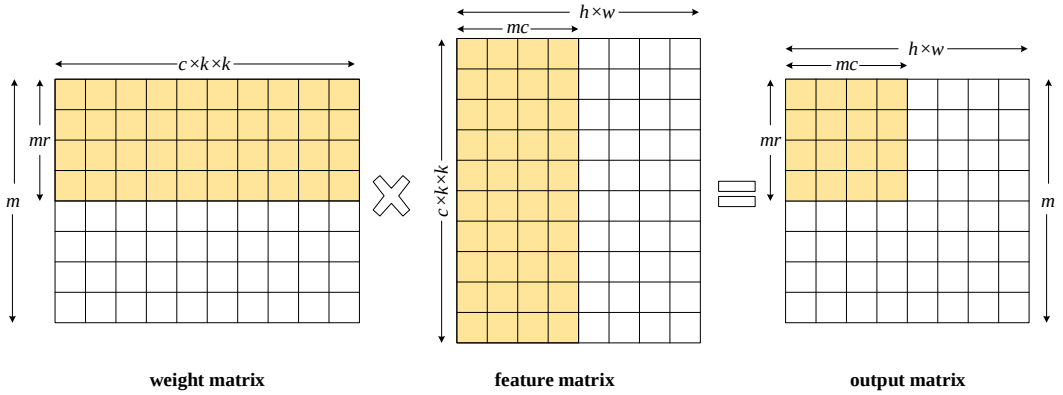

Figure 4. Partitioning weight and feature matrices to blocks to adapt to the 2D multiply and accumulate (MAC) array.

3.3. Buffer Settings

As shown in Figure 5, we deploy three buffers for caching data on-chip: The weight buffer for kernels in convolutional layers and weights in fully connected layers, the feature buffer for input features and the output buffer for output features. Each buffer is composed of multiple independent Block RAMs and the number of Block RAMs is carefully selected to perfectly match the needs of the 2D MAC array.

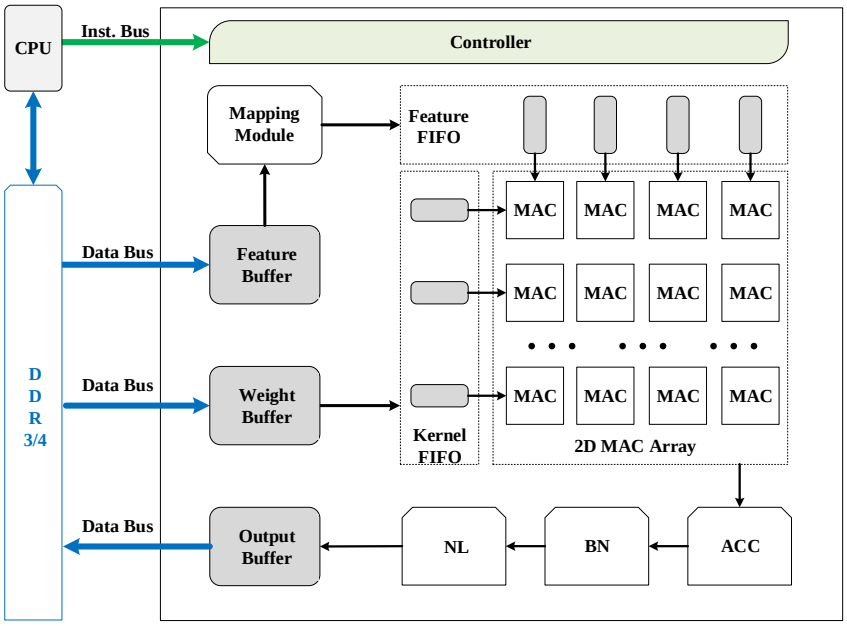

Figure 5. Our proposed uniform accelerator architecture for 2D and 3D convolutional neural networks (CNNs).

During the process of matrix multiplication, mr rows of kernel data and mc columns of feature data are required at every cycle. To offer enough data ports to the 2D MAC array, we assign mr Block RAMs in the weight buffer and $mc + 2 \times pad$ Block RAMs in the feature buffer. The additional $2 \times pad$ Block RAMs are for the padding data when the convolution window slides to the edges. As the 2D MAC array calculates $mr \times mc$ pixels of the output feature simultaneously, we assign mc Block RAMs

in the output buffer. Once the $mr \times mc$ results are generated, they will be cached in the output buffer in mr cycles, which is much shorter than the matrix multiplication latency. Meanwhile, we can adopt the burst access pattern when storing the output feature back to the off-chip memory, which lifts the memory bandwidth utilization rate.

To save on-chip memory consumption, the weight buffer stores only mr groups of kernels on-chip, the feature buffer stores $k + stride$ rows for each channel of the input feature, and the output buffer stores only mr rows (one row for each channel) of the output feature. To achieve pipelining between memory access and computation, the input buffer pre-caches $stride$ rows for each channel of the input feature during the matrix multiplication and the ping-pong strategy is used on the output buffer. Therefore, the memory access time is overlapped with the computation time to the most extent.

3.4. Accelerator Architecture

Figure 5 shows an overview of the uniform accelerator architecture for 2D and 3D CNNs. The major components include a controller for fetching instructions and orchestrating all other modules during executing instructions, three buffers for caching input features, weights, and output features respectively, a mapping module to flatten and rearrange input features to feature matrices, and a 2D MAC array to compute matrix multiplications. There are some supplementary function units: The ACC unit for accumulations of two convolutional layers, the BN unit for batch-norm and scale layers, the NL unit for nonlinear activation functions, and some FIFOs for matrix dataflow during computing matrix multiplications. All buffers are connected through data buses to external DDR3/4 memory. The input features and weights are initialized to the DDR memory by the host CPU, which can be a server CPU or an embedded processor depending on the application scenario.

Given an implemented accelerator, a specific CNN model is computed layer-by-layer by a sequence of macro-instructions. The instructions are fed to the controller directly by the host CPU through the instruction bus. Table 1 lists all the macro-instructions used in our architecture. Basically, each instruction is corresponding to a layer type in CNNs. A sum layer is specially introduced due to the splitting strategy. In the case when the input feature of a convolutional layer has too many input channels to be stored in the feature buffer, the convolutional layer will be split into multiple convolutional layers with less input channels. Sum layers are then used to accumulate the results of these convolutional layers. The batch-norm, scale, and ReLu layer can be combined with the convolutional or fully-connected layers ahead so no independent instructions are for them. As shown in Table 2, each macro-instruction is 128-bits long, including the opcode that indicates the layer type, and a series of parameters listed below. The meaning of the left parameters are the same as above.

- Ix, the height of the input feature;
- Ox, the height of the output feature;
- tm_max, the number of weight matrix blocks;
- tc_max, the number of feature matrix blocks;
- pad, the number of padding rows and padding columns;
- bn_opt, option indicating whether there are batch normalization and scale operations;
- nl_opt, option indicating whether there is non-linear function;

Table 1. Macro-instructions list.

Layer Type	Opcode	Function
Convolution	0	Processing Convolutional layers.
Pooling	1	Processing pooling layers with the 'MAX-Pooling' policy.
	2	Processing pooling layers with the 'AVERAGE-Pooling' policy.
Fully connected	3	Processing fully connected layer.
Sum	4	Accumulating the results of two convolutional layers.
Batch-Norm	-	Combined with the convolutional layer ahead.
Scale	-	Combined with the convolutional layer ahead.
ReLU	-	Combined with the convolutional or fully connected layer ahead.

Table 2. Instruction format.

127:112	111:96	95:80	79:64	63:56	55:48	47:40	39:32	31:24	23:16	15: 8	7:0
c	*m*	*Ix*	*Ox*	*tm_max*	*tc_max*	*k*	*pad*	*stride*	*bn_opt*	*nl_opt*	*opcode*

4. Accelerator Implementation

As a case study, we implement an accelerator prototype based on the uniform architecture with the HLS methodology. The pseudo-code in Figure 6 (left) demonstrates the working process of a convolutional layer. The weight buffer, feature buffer, and output buffer are declared respectively with specified data types. We adopt fixed-point arithmetic logic units in our implementation. As shown in Figure 6 (right), each kernel is represented by eight bits including one sign bit and seven fraction bits, each pixel in input and output features is represented by 16 bits including one sign bit, seven integer bits, and eight fraction bits, and each intermediate result is represented by 32 bits including one sign bit, 16 integer bits, and 15 fraction bits. The intermediate results are represented by 32 bits to preserve precision during accumulations and will be truncated to 16 bits before writing back to memory. The weight buffer is completely partitioned in the row dimension with the *array_partition* pragma. The feature buffer and output buffer are completely partitioned in the column dimension. The core functions include the load-weight function (line 10), the load-feature function (line 14), the matrix-mapping function (line 17), the matrix-multiply function (line 18), and the store-feature function (line 20). As the function name reflects, the load-weight function loads weights from the off-chip memory to the weight buffer; the load-feature function loads the input feature from the off-chip memory to the input buffer; the store-feature function stores the output feature from the output buffer back to the off-chip memory; the matrix-mapping function is corresponding to the matrix mapping module; and the matrix-multiply function is corresponding to the 2D MAC array.

```
KerType kbuf[mr][kdepth];
#pragma HLS array_partition
ImgType ibuf[idepth][mc+2*pad];
#pragma HLS array_partition
MidType obuf[odepth][mc];
#pragma HLS array_partition

for(tm = 0; tm < tm_max; tm++){
  #pragma HLS dataflow
  load_weight();
  for(tl = 0; tl < l; tl++){
    for(th = 0; th < h; th++){
      #pragma HLS dataflow
      load_feature();
      for(tc = 0; tc < tc_max; tc++){
        #pragma HLS dataflow
        matrix_mapping();
        matrix_multiply();
      }
      store_feature();
    }
  }
}
```

```
typedef ap_fixed< 8,  1> KerType;
typedef ap_fixed<16,  8> ImgType;
typedef ap_fixed<32, 17> MidType;

MidType C[mr][mc];
#pragma HLS array_partition

for(t = 0; t < c*k*k; t++){
  #pragma HLS pipeline
  for(i = 0; i < mr; i++){
    #pragma HLS unroll
    for(j = 0; j < mc; j++){
      #pragma HLS unroll
      last = (t==0)? 0 : C[i][j];
      temp = A[i][t] * B[t][j];
      C[i][j] = last+temp;
    }
  }
}
```

Figure 6. High level synthesis (HLS) pseudocode demonstrating the working process of a convolutional layer (**left**) and how to compute the matrix multiplication with the 2D MAC array (**right**).

4.1. Computation Optimization with HLS Pragmas

The *dataflow* optimization is adopted to improve the throughput performance. The *dataflow* pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, and increasing the concurrency of the RTL implementation. As shown in Figure 6 (left), the *dataflow* pragma is specified within the tc-loop, th-loop and tm-loop respectively. Figure 7a illustrates the working process of the tc-loop without dataflow pipelining. The matrix-mapping function and the matrix-multiply function are processed sequentially. With dataflow pipelining, the matrix-multiply function can begin several cycles after the matrix-mapping function begins, and are almost completely overlapped with the matrix-mapping function, as illustrated in Figure 7b. Therefore, the latency of the tc-loop with dataflow pipelining is shortened significantly. Figure 7c shows how the dataflow optimization works on the th-loop. The load-feature function and the store-feature function are fully overlapped by the tc-loop. The th-loop is pipelined and the pipeline interval equals to the maximum latency of the three parts. Figure 7d illustrates the tm-loop with dataflow pipelining. Notice that the matrix-multiply function is dependent to the weight matrix and hence the th-loop has to wait until the load-weight function is done. The latency of the th-loop is typically much longer than the latency of the load-weight function. For example, in the second convolutional layer of the C3D model, the th-loop takes 37,355 cycles while the load-weight function takes only 578 cycles. In this case, the ping-pong strategy can reduce the execution time by at most 1.5%. Therefore, the ping-pong strategy is not adopted on the weight buffer to save on-chip memory consumption. That is why the load-weight function is not fully overlapped with the th-loop.

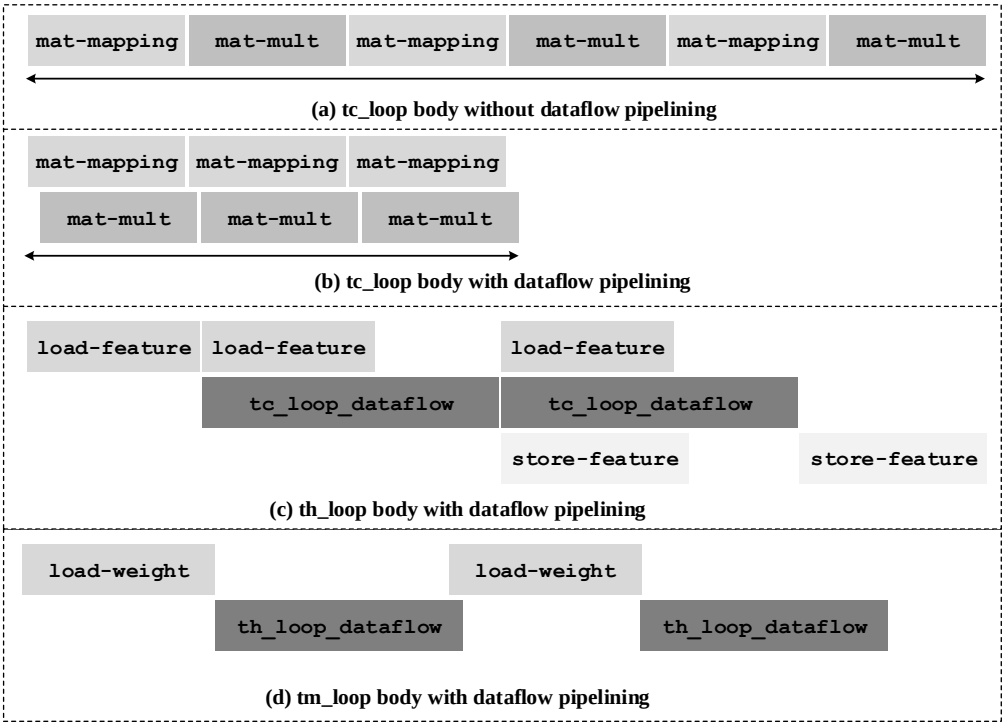

Figure 7. How dataflow optimization works on the functions and loops.

The HLS pragmas *unroll* and *pipeline* are used inside these functions to reduce latency and enhance throughput performance. Figure 6 (right) shows the HLS pseudocode of the matrix-multiply function. The *unroll* pragma enables some or all loop iterations to occur in parallel by creating multiple copies of the loop body in the RTL design. In the matrix-multiply function, $mr \times mc$ multipliers and adders are created shaping the 2D MAC array and hence $mr \times mc$ multiply-accumulations are computed concurrently. The *pipeline* pragma helps to reduce the initiation interval for a loop by allowing the concurrent execution of operations. The initiation interval in the matrix-multiply function is one cycle after optimization. To summarize, the total execution latency is greatly reduced and the system throughput is enhanced significantly owing to the HLS pragmas.

4.2. Memory Optimization

Memory optimizations are made to better use the external memory bandwidth. We expand the data access width using the HLS *data_pack* pragma, which packs the data fields of a *struct* into a single scalar with larger byte length. The *data_pack* pragma helps to reduce the memory access time and all members of the *struct* can be read and written simultaneously. In our implementation, *mr* weights are packed to a *struct* for the load-weight function, $mc + 2 \times pad$ pixels are packed for the load-feature function, and *mc* pixels are packed for the store-feature function.

In addition, we optimize the storage pattern of the features in the external memory space. Since the load-feature function loads one entire row of all input channels every time, the store-feature function stores the features back to the off-chip memory according to the frame-height-channel-width order instead of the frame-channel-height-width order. Therefore, the load-feature function and the store-feature function can transfer data between the on-chip buffer and the off-chip memory in a burst access mode, which lifts the bandwidth utilization rate significantly. The original map is still stored according to the frame-channel-height-width order as we do not want introduce additional work. The large width of the original map guarantees the burst access length in the load-feature function of the first convolutional layer. With the above memory optimizations, the memory bandwidth is

no longer the bottleneck. In our implementation, the load-feature and store-feature functions are fully overlapped by the matrix-multiply function. That is to say, the 2D MAC array is only idle during setting up and flushing the pipeline, and will be fully utilized once the pipeline is ready in convolutional layers.

5. Accelerator Modeling

5.1. Resource Modeling

An FPGA has several different types of resources of which DSP slices and on-chip memory (Block RAM) have the most effect on the configuration of a CNN accelerator. Since we are using fixed-point arithmetic in our architecture, the only component that consumes DSP slices is the multiplier in the MAC units. Each multiplier consumes one DSP slice. Hence, the total consumption of DSP slices is given by $mr \times mc$, which should be less than the total number of available DSP slices.

In terms of on-chip memory utilization, we analytically model the consumption of the weight buffer, input buffer and output buffer. There are mr Block RAMs in the weight buffer. Each weight is represented as one byte and hence the width of each Block RAM is 8. Assuming the depth of each Block RAM is $kdepth$, the total on-chip memory consumed by the weight buffer is $mr \times kdepth$ bytes. The depth of each Block RAM is given by Equation (6) if the on-chip memory is abundant. The $layer\_num$ in the equation indicates the total number of layers in a CNN model.

$$kdepth = max(c_i \times k_i \times k_i) \quad (i = 0, 1, 2, ..., layer\_num - 1) \tag{6}$$

The feature buffer stores input features in $mc + 2 \times pad$ Block RAMs. The additional $2 \times pad$ Block RAMs are introduced due to the padding required at the edges. Each pixel in input features is represented as two bytes and hence the width of each Block RAM is 16. Assuming the depth of each Block RAM is $idepth$, the total on-chip memory consumed by the input buffer can be calculated by $(mc + 2 \times pad) \times idepth \times 2$ bytes. The depth of each Block RAM is given by Equation (7) if the on-chip memory is abundant.

$$idepth = max(c_i \times (k_i + stride_i)) \quad (i = 0, 1, 2, ..., layer\_num - 1) \tag{7}$$

In the case when the on-chip memory is limited, $kdepth$ and $idepth$ are specified by users under the memory constraint. For some convolutional layers with a large number of input channels, $kdepth$ may be less than the width of the weight matrix or $idepth$ may be less than the height of the feature matrix. The splitting strategy will split the convolutional layer to multiple convolutional layers with less number of input channels to fit to the weight buffer and feature buffer.

The output buffer stores output features in mc Block RAMs. The ping-pong strategy is adopted for the output buffer. Each pixel in output features is represented as two bytes and hence the width of each Block RAM is 16. Assuming the depth of each Block RAM is $odepth$, the total on-chip memory consumed by the output buffer can be calculated by $mc \times odepth \times 4$ bytes. The depth of each Block RAM is given by Equation (8) if the on-chip memory is abundant.

$$odepth = max(mr \times \lceil w_i / mc \rceil) \quad (i = 0, 1, 2, ..., layer\_num - 1) \tag{8}$$

In the case when the on-chip memory is limited or the feature width is too large, $odepth$ can be specified by users under the memory constraint. The 2D MAC array may be under-utilized in some convolutional layers with large feature width. The real unrolling factor of the output channel loop is less than mr, which is given by the following equation:

$$mr_{real} = floor(odepth / \lceil w / mc \rceil) \tag{9}$$

5.2. Performance Modeling

The main operations in matrix multiplication are multiplications and additions, conducted by the 2D MAC array. As there are $mr \times mc$ MAC units, $mr \times mc \times 2$ operations are processed every clock cycle in the ideal case. Therefore, the peak throughput is given by $mr \times mc \times 2 \times f$ OP/s, where f is the clock frequency.

The actual throughput is the total operations divided by the total execution time. The total operations in a convolutional layer can be calculated by Equation (10). In 2D convolutional layers, $l = 1$ and $d = 1$.

$$m \times l \times h \times w \times c \times d \times k \times k \times 2 \tag{10}$$

Owing to the *dataflow* optimization, the functions are working in a pipelined way and some functions are even completely overlapped by others, as illustrated in Figure 7. The total execution cycles for a convolutional layer can be calculated by Equation (11) where ccl_{ldw}, ccl_{ldf}, and ccl_{stf} indicate the execution cycles of the load-weight, load-feature and store-feature function respectively. II_{th} indicates the pipeline interval of the th-loop in Figure 6.

$$\lceil \frac{m}{mr} \rceil \times (ccl_{ldw} + ccl_{ldf} + l \times h \times II_{th}) + ccl_{stf} \tag{11}$$

The matrix-mapping function takes $c \times k \times k$ cycles to generate a feature matrix block and the matrix-multiply function also takes $c \times k \times k$ cycles to complete a matrix multiplication. Hence, the total cycles taken by the tc-loop in Figure 6 is given by:

$$ccl_{tc} = \lceil w/mc \rceil \times c \times k \times k \tag{12}$$

The execution time of the other functions are closely related to the real memory access bandwidth. We get simplified models for the memory-related functions under sufficient memory access bandwidth: The load-weight function takes $c \times k \times k$ cycles, the load-feature function takes $c \times stride \times \lceil w/mc \rceil$ cycles, and the store-feature function takes $mr \times \lceil w/mc \rceil$ cycles. The pipeline interval of the th-loop is the maximum value of ccl_{ldf}, ccl_{stf} and ccl_{tc}.

6. Evaluation

In this section, we first introduce the experimental setup and then detailed experimental results are provided.

6.1. Experimental Setup

We evaluate our design by implementing an accelerator on a Xilinx VC709 board that contains a Xilinx XC7VX690T FPGA chip with 3600 DSP slices. We select three typical CNN models for test: AlexNet, VGG16, and C3D. AlexNet and VGG16 are 2D CNN models for image classification, and C3D is a 3D CNN model for human action recognition. The channel numbers in the convolutional layer include 64, 128, 256, and 512. The feature sizes include 224×224, 112×112, 56×56, 28×28, and 14×14. Considering the total number of available DSP slices, we assign 64 rows and 56 columns for the 2D MAC array. Accordingly, the weight buffer has 64 Block RAMs with depth of 5120, the feature buffer has 60 Block RAMs with depth of 2048, and the output buffer has 56 Block RAMs with depth of 512. The accelerator design is implemented with Vivado HLS and synthesized with Vivado Design Suite 2018.1. The software versions are implemented with Caffe [8], a deep-learning framework, running on an Intel Core i5-4440 CPU (@3.10 GHz) and an NVIDIA GPU (Titan X Pascal). The Caffe framework used in the software implementations is the original one developed by the Berkeley Vision and Learning Center (BVLC) with no special optimizations for the Intel architecture. There are no fixed-point arithmetic units in the CPU or the GPU, and hence the software versions are implemented

with floating-point arithmetic. The dataset for testing AlexNet and VGG16 is ImageNet [20] and the dataset for testing C3D is UCF1017 [21]. We use a batch size of 16 for both versions during testing.

6.2. Experimental Results

Table 3 presents the hardware resource utilization of our accelerator. The number of DSP slices consumed by our accelerator is 3595, which uses nearly 100% of the available DSP slices. The 2D MAC array consumes 3584 of the 3595 DSP slices for computing matrix multiplications and the other 11 DSPs are used for calculating addresses during memory access. In addition, the accelerator uses 391 BRAMs, which uses less than 27% of the available BRAMs. The BRAMs are mainly consumed by the weight buffer, feature buffer, and output buffer. The utilization results show that our architecture can fully utilize the computation resources while consume very few memory resources. Especially, the splitting strategy makes the architecture even less dependent on the on-chip memory capacity. Therefore, our architecture can be deployed to platforms with limited on-chip memory capacity and is very friendly for ASIC implementations.

Table 3. Field programmable gate array (FPGA) resource utilization.

Resource	DSP	BRAM	LUT	FF
Available	3600	1470	433,200	866,400
Utilization	3595	391	272,734	434,931
Utilization (%)	99.8	26.6	62.9	50.2

Figure 8 presents the evaluation results of each layer in the three CNN models. According to the performance model, the peak throughput of our accelerator at 120 MHz is 860.2 GOP/s. The conv2a layer achieves the highest throughput of 811.5 GOP/s in AlexNet, the conv1b layer achieves the highest throughput of 856.1 GOP/s in VGG16, and the conv2a layer achieves the highest throughput of 851.2 GOP/s in C3D. The first convolutional layer in all three CNN models has only three input channels which will make the 2D MAC array under-utilized. Since the C3D has a temporal depth of three, the real input channels in the first convolutional layer is nine. That is why the conv1a layer achieves a poor throughput in AlexNet and VGG16 while achieves a much higher throughput in C3D. We can also find from Figure 8 that the throughput performance of each layer decreases with the layer depth. The reason is that the feature size decreases owing to the pooling layers and hence the th-loop iterates less times in latter convolutional layers. As shown in Equation (11), ccl_{ldw} and ccl_{ldf} will account for more percentage of the total execution cycles when h decreases. In fully-connected layers, the 2D MAC array is under-utilized restricted by the memory bandwidth. Thus, the throughput of fully-connected layers is far lower than that of convolutional layers, around 40 GOP/s. That is why the average throughput of the convolutional layers is more than 407.2 GOP/s in AlexNet while the overall throughput is only 231.6 GOP/s. By comparison, VGG16 and C3D have more convolutional layers and hence fully connected layers have less effect to the overall throughput. The accelerator achieves an overall throughput of 691.6 GOP/s on VGG16 and 667.7 GOP/s on C3D.

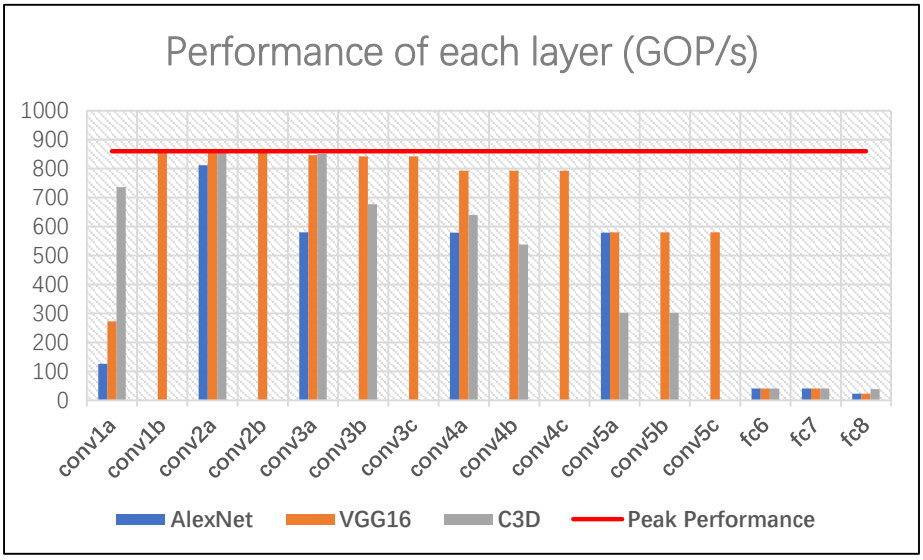

Figure 8. Throughput performance of each layer in three CNN models.

Table 4 lists the comparisons between the CPU, the GPU and our accelerator. Compared to the CPU, our accelerator achieves a 7.5-fold improvement on VGG16 and 8.2-fold improvement on C3D in terms of throughput and latency. The thermal design power (TDP) values of the CPU and the GPU are 84 and 250 W respectively. According to the power report by the Vivado Design Suite, the total on-chip power of our accelerator is only 15.8 W. We can see from Table 4 that our accelerator achieves the best power efficiency among all the platforms. Compared to the CPU, our accelerator achieves a 39.8-fold improvement on VGG16 and 43.9-fold improvement on C3D in terms of power efficiency. The Titan X Pascal GPU has a great leading advantage in terms of throughput performance: 52.4-fold to CPU and 6.9-fold to our accelerator on VGG16. By contrast, our accelerator achieves better power efficiency than the GPU: 2.3-fold on VGG16 and 2.0-fold on C3D.

Table 4. Evaluation results on the central processing unit (CPU), graphics processing unit (GPU), and our accelerators.

Platform	**CPU**		**GPU**		**FPGA**	
Vendor	Intel i5-4440		Nvidia Titan X Pascal		Xilinx VC709	
Technology	22 nm		16 nm		28 nm	
Power (W)	84		250		15.8	
CNN Model	VGG16	C3D	VGG16	C3D	VGG16	C3D
Latency (ms)	335.8	951.2	6.4	14.8	44.8	115.5
Speedup	1.0	1.0	52.4	64.3	7.5	8.2
Throughput (GOP/s)	92	81	4816	5222	691.6	667.7
Power Efficiency (GOP/s/W)	1.1	0.96	19.3	20.9	43.8	42.2
[Ratio]	[1.0]	[1.0]	[17.6]	[21.6]	[39.8]	[43.9]

We select several 3D CNN accelerator implementations and several 2D CNN accelerator implementations on FPGAs for comparison. As different FPGA platforms are used, we list the performance density in Table 5 for fair comparisons. Performance density is defined as the number of arithmetic operations that one DSP slice executes in one cycle. Note that a MAC is counted as two operations. The performance density also eliminates the impact of the clock frequency. As shown in Table 5, our accelerator is better than [18,22], and within 5% of [19] in terms of performance density. The implementation in [17] adopts the Winograd algorithm, which reduces the multiplications in 3D convolutional layers by 70.4% at the cost of 101% more additions. The additions are executed with LUTs instead of DSP slices. That is why the implementation in [17] achieves the best performance

density among all the listed implementations. In terms of throughput, our accelerator achieves state-of-the-art performance on both VGG and C3D. The implementation in [19] achieves the best throughput performance with a much higher clock frequency over other implementations.

As shown in [17], the Winograd algorithm reduces the computation complexity significantly in CNNs. However, there are still some limitations in terms of flexibility. The Winograd algorithm can only be applied when the convolution stride is 1 and the transform matrices vary with the size of convolution kernels. By comparison, our architecture with matrix multiply approach is more generic and can adapt to varied strides and convolution kernels. On the other hand, we have to acknowledge that the Winograd algorithm is perfectly fit for accelerating VGG and C3D, as they have a uniform kernel size of 3×3 and a fixed stride of 1 in all convolutional layers. However, it is not suitable for accelerating AlexNet, which has multiple kernel sizes (11×11, 5×5, and 3×3) and multiple strides (4 and 1). It is not a problem for our architecture as we have demonstrated in the evaluation. The architecture in [17] adopts a template-based approach supporting accelerating both 2D and 3D CNNs. Different configurations (e.g., unrolling factors, Winograd algorithms) have to be customized for different CNN models. That is why they implement two accelerators for accelerating VGG16 and C3D respectively. By comparison, our accelerator can accelerate different CNN models with the same configuration. The AlexNet, VGG16, and C3D are run on the same accelerator in our evaluation, and achieve very close throughput performance on convolutional layers.

Table 5. Comparisons with previous accelerator implementations.

	Ref. [18]	**Ref. [22]**	**Ref. [19]**	**Ref. [23]**	**Ref. [17]**	**Ours**	
FPGA	Altera Stratix-V	Arria10 GX1150	Arria10 GX1150	Xilinx ZC706	Xilinx XC7VX690T	Xilinx XC7VX690T	
CNN Model	VGG	VGG	VGG	C3D	C3D	VGG	C3D
Precision	fixed	fixed	fixed/float	fixed	fixed	fixed	
Clock (MHz)	120	150	385	172	150	120	
DSPs	727 (37%)	3036 (100%)	2756 (91%)	810 (90%)	1536 (42%)	3595 (99%)	
Throughput (GOP/s)	118	645	1790	142	431	691.6	667.7
Performance Density (OP/DSP/cycle)	1.35	1.42	1.69	1.02	1.87	1.60	1.55

7. Conclusions

This paper summarizes our recent work on hardware accelerator design for CNNs. The proposed uniform architecture ensures fast development of 2D and 3D CNN accelerators with state-of-the-art performance in throughput, latency, and energy efficiency. Despite the loss in performance density compared with customized accelerators using the Winograd algorithm, our architecture is more generic, which supports accelerating different 2D and 3D CNN models without reconfiguring the FPGA. This means that the architecture is also suitable for ASICs. Future work includes further demonstrations on other CNN-based applications and ASIC implementations for computer vision applications based on the FPGA prototype.

Author Contributions: Conceptualization, Z.L., J.J. and P.C.; methodology, Z.L., J.J. and Y.D.; software, Z.L., J.X. and J.Z.; validation, Z.L., J.X. and J.Z.; writing—original draft preparation, Z.L.; writing—review and editing, Z.L. and P.C.; supervision, P.C. and Y.D.

Funding: This research was funded by the National Key Research and Development Program of China (Grant No. 2018YFB1003405), the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No. 2018ZX01028-101), and the National Natural Science Foundation of China (Grant No. 61802419, 61732018, 61303070, 61602496 and 61502507). This research also supported by the Dusan and Anne Miklas and Xilinx.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

2D	Two Dimensional
3D	Three Dimensional
CNN	Convolutional Neural Network
FPGA	Field Programmable Gate Array
MAC	Multiply and Accumulate
HLS	High Level Synthesis
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ASIC	Application Specific Integrated Circuit
GOPs	Giga-Operations
GOP/s	Giga-Operations Per Second
FFT	Fast Fourier Transform

References

1. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. In Proceedings of the Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 3–6 December 2012; pp. 1097–1105.
2. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, QC, Canada, 11–12 December 2015; pp. 91–99.
3. Schroff, F.; Kalenichenko, D.; Philbin, J. Facenet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 815–823.
4. Maturana, D.; Scherer, S. VoxNet: A 3D Convolutional Neural Network for real-time object recognition. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Hamburg, Germany, 28 September–2 October 2015; pp. 922–928.
5. Molchanov, P.; Gupta, S.; Kim, K.; Kautz, J. Hand gesture recognition with 3D convolutional neural networks. In Proceedings of the Computer Vision and Pattern Recognition Workshops, Boston, MA, USA, 7–12 June 2015; pp. 1–7.
6. Du, T.; Bourdev, L.; Fergus, R.; Torresani, L.; Paluri, M. Learning Spatiotemporal Features with 3D Convolutional Networks. In Proceedings of the IEEE International Conference on Computer Vision, Santiago, Chile, 7–13 December 2015; pp. 4489–4497.
7. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* **2014**, arXiv:1409.1556.
8. Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv* **2014**, arXiv:1408.5093.
9. Team, T.D.; Alrfou, R.; Alain, G.; Almahairi, A.; Angermueller, C.; Bahdanau, D.; Ballas, N.; Bastien, F.; Bayer, J.; Belikov, A. Theano: A Python framework for fast computation of mathematical expressions. *arXiv* **2017**, arXiv:1605.02688.
10. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv* **2016**, arXiv:1603.04467
11. Abdelouahab, K.; Pelcat, M.; Serot, J.; Berry, F. Accelerating CNN inference on FPGAs: A Survey. *arXiv* **2018**, arXiv:1806.01683.
12. Venieris, S.I.; Kouris, A.; Bouganis, C.S. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Comput. Surv.* **2018**, *51*, 56. [CrossRef]
13. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the ACM International Symposium on FPGA, Monterey, CA, USA, 21–23 February 2016.

14. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; pp. 161–170.
15. Zhang, C.; Prasanna, V. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 35–44.
16. Aydonat, U.; O'Connell, S.; Capalija, D.; Ling, A.C.; Chiu, G.R. An OpenCL^{TM} Deep Learning Accelerator on Arria 10. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 55–64.
17. Shen, J.; Huang, Y.; Wang, Z.; Qiao, Y.; Wen, M.; Zhang, C. Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA. In Proceedings of the ACM/SIGDA International Symposium, Monterey, CA, USA, 25–26 February 2018; pp. 97–106.
18. Suda, N.; Chandra, V.; Dasika, G.; Mohanty, A.; Ma, Y.; Vrudhula, S.; Seo, J.S.; Cao, Y. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 16–25.
19. Zhang, J.; Li, J. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 25–34.
20. Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; et al. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.* **2015**, *115*, 211–252. [CrossRef]
21. Soomro, K.; Zamir, A.R.; Shah, M. UCF101: A Dataset of 101 Human Actions Classes From Videos in the Wild. *arXiv* **2012**, arXiv:1212.0402.
22. Ma, Y.; Cao, Y.; Vrudhula, S.; Seo, J.S. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 45–54.
23. Fan, H.; Niu, X.; Liu, Q.; Luk, W. F-C3D: FPGA-based 3-dimensional convolutional neural network. In Proceedings of the International Conference on Field Programmable Logic and Applications, Ghent, Belgium, 4–8 September 2017; pp. 1–4.