End-to-End Optimization of Deep Learning Applications

Atefeh Sohrabizadeh*
University of California, Los Angeles
Los Angeles, California
atefehsz@cs.ucla.edu

Jie Wang*
University of California, Los Angeles
Los Angeles, California
jiewang@cs.ucla.edu

Jason Cong
University of California, Los Angeles
Los Angeles, California
cong@cs.ucla.edu

ABSTRACT

The irregularity of recent Convolutional Neural Network (CNN) models such as less data reuse and parallelism due to the extensive network pruning and simplification creates new challenges for FPGA acceleration. Furthermore, without proper optimization, there could be significant overheads when integrating FPGAs into existing machine learning frameworks like TensorFlow. Such a problem is mostly overlooked by previous studies. However, our study shows that a naive FPGA integration into TensorFlow could lead to up to 8.45× performance degradation. To address the challenges mentioned above, we propose several SW/HW co-design approaches to perform the end-to-end optimization of deep learning applications. We present a flexible and composable architecture called FlexCNN. It can deliver high computation efficiency for different types of convolution layers using techniques including dynamic tiling and data layout optimization. FlexCNN is further integrated into the TensorFlow framework with a fully-pipelined software-hardware integration flow. This alleviates the high overheads of TensorFlow-FPGA handshake and other non-CNN processing stages. We use OpenPose, a popular CNN-based application for human pose recognition, as a case study. Experimental results show that with the FlexCNN architecture optimizations, we can achieve 2.3× performance improvement. The pipelined integration stack leads to a further 5× speedup. Overall, the SW/HW co-optimization produces a speedup of 11.5× and results in an end-to-end performance of 23.8FPS for OpenPose with floating-point precision, which is the highest performance reported for this application on FPGA in the literature.

KEYWORDS

FPGA, CNN, OpenPose, TensorFlow, tiling, integration
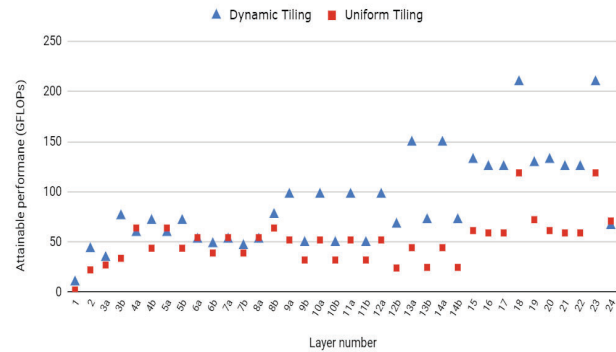
---

*Indicates the co-authors for this work.

Figure 1: Performance comparison of designs using uniform and dynamic tiling factors for the first 24 convolutional layers in the CNN network in Figure 3.

1 INTRODUCTION

Convolutional Neural Networks (CNNs) are widely used in many machine learning (ML) applications and have evolved quickly over the years. There is a growing interest in FPGA for accelerating CNN computation due to its high energy efficiency and performance (e.g., [2, 3, 10, 14, 16, 18, 21, 23, 26–28]). Furthermore, the recent advancement in CNN models and FPGA-based CNN acceleration has brought several new challenges.

Challenge 1: Performance disparity of different CNN layers: Real-world deep learning (DL) applications may have complex network architectures. In addition, many state-of-the-art efficient networks such as MobileNetV1 [11] use depth-wise separable convolutions (DSC) introduced in [19] to decrease the computation cost. MobileNetV2 [17] introduced residual bottleneck block (RBB) to further reduce the computation complexity. These layers reduce the computation cost but keep the same feature map size; this can make the layer more communication-bound and reduce the computation efficiency. Apart from this irregularity, different layers of a CNN have different characteristics in terms of their input and output number of channels, feature map size, and kernel size. This changes the computation to communication (CTC) ratio from layer to layer. Therefore, it is important to handle these layers differently given the performance disparity across these layers. We found that tiling factors can play an important role in the performance. Zhang et al. [26] showed that the CTC ratio of a single convolution layer varies with different tiling factors. Yang et al. [25] highlighted the importance of choosing proper tiling factors for the data reuse in the near and faster memory (on-chip storage for FPGAs) for the overall latency and energy efficiency. These studies lead us to consider using different tiling factors across the network. Figure 1 depicts how different tiling factors can affect the performance for each layer in one CNN network example as shown in Figure 3. We compare the
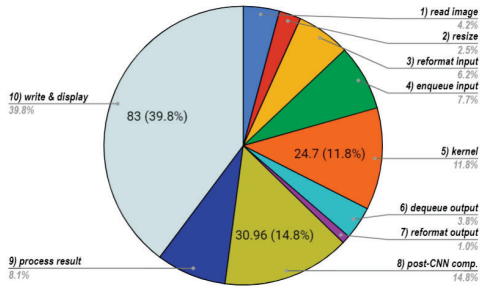
Figure 2: Runtime break down of an FPGA-based CNN acceleration pipeline in TensorFlow.

performance of using a single set of tiling factors (uniform tiling) to using different tiling factors for each layer (dynamic tiling). For the uniform tiling, we chose the tiling factor that reduces the latency of the entire network. For the dynamic tiling, we focused on each layer and selected the best tiling factor accordingly. Experimental results show that dynamic tiling can speedup the performance of the whole network by 1.7×.

Challenge 2: Integration overheads of using FPGA in ML frameworks: When processing a CNN application in modern ML framework such as TensorFlow [1], the complete stack consists of reading the input, computing the CNN, processing the result, and displaying and writing the result. Previous works have only focused on optimizing the CNN kernel on FPGA (e.g., [2, 3, 10, 14, 21, 23, 26, 28]). This is due to the fact that CNN computation is the most time-consuming step of the whole stack. Hence the rest of the overheads are ignored. While several works [10, 16] have focused on accelerator *generation* from TensorFlow-described networks, they did not address the challenges of *integrating* an accelerator into TensorFlow. By integrating our accelerator with TensorFlow, we are able to directly run networks from TensorFlow on an FPGA. Integrating FPGA into TensorFlow introduces a new set of overheads: communication between TensorFlow and FPGA and the communication between the host and the FPGA kernel itself. Figure 2 shows the break down of the end-to-end runtime for processing an 384×384 RGB image using the network in Figure 3. These steps are listed and described in Section 4. The time for CNN processing, using our accelerator denoted as the kernel, only takes 11.8% of the total runtime. This emphasizes the need for an end-to-end SW/HW co-optimization. Our experiments show that this optimization can increase the end-to-end performance of this network from 4.8FPS to 23.8FPS, leading to a 5× speedup.

To solve the challenges above, in this work, we propose an FPGA-based CNN accelerator named FlexCNN. It employs dynamic tiling and data layout transformation to adapt to the performance disparity of different CNN layers. The accelerator is further integrated into the TensorFlow framework. To mitigate the large integration overheads, we propose a two-level software pipeline to overlap the overheads with the computation.

We use OpenPose [4] as a driving application to test FlexCNN's performance. To our knowledge, there is only one prior work [2] that has implemented OpenPose on FPGA. Its CNN kernel processes an image in 42.6ms. Whereas, FlexCNN can process an image in 24.7ms, leading to 1.7× speedup.

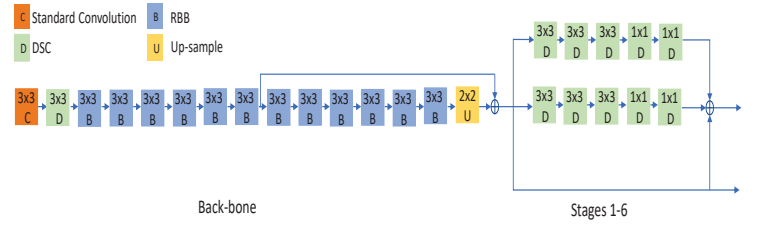In summary, the key contributions of this paper are:

Figure 3: OpenPose-V2 network topology.

- A flexible and composable accelerator architecture, called FlexCNN, supporting dynamic tiling and data layout transformation to improve computation efficiency for running CNNs.
- A TensorFlow to FPGA runtime environment for running CNN on FPGA from TensorFlow and an optimized software/hardware pipeline to mitigate the integration overheads.
- A fully automated compilation system for the FlexCNN architecture.
- The fastest FPGA accelerator to run OpenPose. FlexCNN yields a 2.3× speedup from supporting dynamic tiling and optimized data layout. Besides, our framework achieves 5× speedup from software/hardware pipelining, resulting in a final performance of 23.8FPS. In addition, FlexCNN is 3.8× more energy efficient than GPU.

2 APPLICATION DRIVER

2.1 OpenPose

OpenPose [4] is the winner of the COCO 2016 Keypoints Challenge that can detect 2D poses of multiple people in an image. OpenPose network first extracts the features of the input image using the first 10 layers of VGG-19 [20]. This is the backbone of the network. These feature maps are the inputs to a two-branch network. The first branch detects confidence maps, representing body part locations, and the second branch detects part affinity fields, a set of 2D vectors showing the location and orientation of the limbs. The results of these two branches are concatenated with the feature maps from the backbone network and form the input for the next stage. After several iterations, these branches produce final predictions.

This network is interesting to us since it has an irregular architecture compared to modern CNN-based DL applications. Instead of just a linear forward path where each layer consumes the result of its previous layer, it has concatenation layers that need extra data movement. Moreover, to reduce the computation complexity of the network, we use a modified version of OpenPose [13] that replaces the backbone with a modification of MobileNetV2 [17] and employs DSC [19] for the rest of the network, following the trend in the ML community. Figure 3 depicts the network topology of this version, we call this network OpenPose-V2. Due to the space limitation, we only show the convolutional layers. Each convolution is followed by ReLU and batch normalization layers.

2.2 New Building Blocks

2.2.1 Depthwise Separable Convolution. In a standard convolution layer (conv), the feature maps are filtered and combined in one
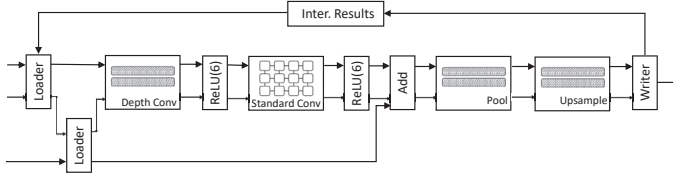
Figure 4: Accelerator overview.

Table 1: Design parameters and explanations.

Design Parameters	Explanation
$Th(k)$, $Tw(k)$, $Tn(k)$, $Tm(k)$	Tiling factors for H, W, N, and M for layer k
$SIMD$	SIMD lanes for all modules
SA_ROW, SA_COL	Rows and columns of the systolic array kernel

step. The DSC splits this step into two phases. The first phase, depthwise convolution (DW), does the filtering, and the second phase, pointwise convolution (PW), combines the produced filtered feature maps using 1×1 kernels.

A conv layer takes N feature maps as the input, each of size $H \times W$. It uses $M \times N \times K \times K$ kernels, to produce M channels for the output. The total computation cost for this layer is $M \times N \times H \times W \times K \times K$.

However, a DSC uses $N \times K \times K$ kernels for DW and $M \times N \times 1 \times 1$ kernels for PW. By applying this change, the amount of computation is reduced by a factor of $\frac{1}{M} + \frac{1}{K^2}$ [11].

2.2.2 Residual Bottleneck Block. Google introduced RBB in MobileNetV2 [17] to reduce the computation cost. It consists of a 1x1 conv followed by a 3x3 DW and then another 1x1 conv, each of which is followed by ReLU and a batch normalization layer. The 1x1 convolutions are used for dimension reduction or restoration. The nature of this block allows us to reduce the number of input and output channels. This reduces the computation intensity and makes the network more efficient.

3 THE FLEXCNN ARCHITECTURE

The basic layers in different CNNs are convolution, DW, ReLU, bias/batch normalization, downsampling/pooling, upsampling, and add. The rest of the building blocks are a combination of these layers. Thus the FlexCNN architecture has these components as building blocks and uses them to compose different parts of the network. This strategy can improve the hardware utilization on FPGA.

Figure 4 depicts the detailed architecture of the FlexCNN. It implements the dataflow architecture to process the network layer by layer. Each layer can load up to two sets of input feature maps, those from the current layer and the previous layer. Loading the feature maps from the previous layer is required for supporting convolutional layers like RBB where the results of the current and previous layers need to be added together.

The loaded feature maps will pass through modules including the depth-wise convolution module (*Depth Conv*), ReLU(6)[1] module, standard convolution module (*Standard Conv*), ReLU(6) module, add module, max-pooling module (*Pool*), and bi-linear upsampling module (*Upsample*). The final results will be written out to DRAM via *Writer*. The operations in batch normalization layer and bias

---

[1]ReLU6 outputs the minimum of the value 6 and a normal ReLU.

layer are fused into ReLU(6) modules. Each of the convolutional operations may be followed by any of the ReLU(6) or normalization layers. Hence, we put the ReLU(6) module after both the Depth Conv and Standard Conv modules.

For Standard Conv, the systolic array (SA) architecture is used. It is generated using the SA compiler in [8]. It can compute a standard convolution layer with any given kernel size. We implement line-buffer-based streaming architectures for Depth Conv, ReLU(6), Add, Pool, and Upsample modules using a similar stencil-based architecture as in [7]. All these modules are parameterized by factors as shown in Table 1, which will be explored by the design space exploration (DSE) engine covered in section 3.1.1, for the optimal performance.

Note that all the modules can be bypassed if not being used for a specific layer. We apply double buffering in both the Loader modules and the Writer module. Furthermore, if the outputs of the whole layer can fit into the on-chip buffer, the data will be pushed into on-chip buffers and directly fetched by the Loader to save the off-chip communication time.

3.1 Dynamic Tiling

Tiling is applied when processing the network for improving the data locality and minimizing the communication. Table 1 summarizes the tiling factors employed in FlexCNN. When the tiling factors are not sub-multiples of the tiled dimensions, redundant computation is introduced which degrades the performance of the design. As explained in Section 1, in a normal CNN network, the types and configurations of different layers vary from each other. Therefore, the optimal tiling factors will be different from each other as well. We have observed that using uniform tiling factor for the whole network will lead to up to 1.7× performance slowdown compared to the ideal case using different tiling factors across layers. Therefore, in this work, we apply the dynamic tiling by re-configuring the tiling factors of the accelerators on-the-fly for different layers to maximize the performance. This will bring the hardware overheads to support the dynamic tiling. However, such overheads are negligible compared to the performance improvement. Section 6 evaluates the impacts of this technique in detail.

Previous works such as [18, 22, 28] have also emphasized the need for different tiling factors across layers. Our architecture distinguishes from the previous work by changing all the tiling factors across each layer dynamically, whereas previous work only adjusted part of the tiling factors or used several accelerators, each with distinct uniform tiling factors on-chip. Eq. 1 shows the restriction on the tiling factors.

$$\begin{aligned}
Tw(k) &= c_1 \times SA\_COL \\
Tm(k) &= c_2 \times SA\_ROW \\
Tn(k) &= c_3 \times SIMD \\
Tm(k) &= Tn(k+1)
\end{aligned} \tag{1}$$

In FlexCNN, the width and output channels of the feature maps are mapped to columns and rows of the SA respectively. As a result, for each layer, $Tw(k)$ and $Tm(k)$ should be multiples of their respective SA dimension. The reduction of multiple input channels is computed in parallel inside each PE of the SA, which is defined as the SIMD lane. This implies that $Tn(k)$ should be a multiple of
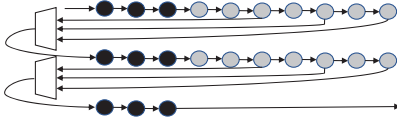
Figure 5: Architecture support for dynamic tiling in the Depth Conv module for a 3×3 **kernel with Tw of size 6/8/10.**
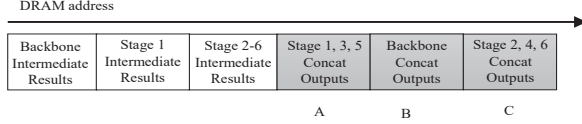

Figure 6: Data organization for OpenPose.

SIMD lane. $Th(k)$ can be any arbitrary value. As mentioned before, the computation in depth conv module can be seen as a stencil kernel. Figure 5 depicts the 3×3 stencil window connected by line buffers. We realize dynamic tiling by connecting consecutive rows of the line buffer via a MUX, enabling data feeding from different locations.

3.1.1 Design Space Exploration. Given the network, the accelerator architecture, and the FPGA's resources information, we will perform the design space exploration to select the optimal design parameters that maximize the performance on the target FPGA. Table 1 lists the design parameters to be determined.

Two analytical models *resource_est()* and *latency_est()* are built for estimating the resource usage and latency of designs. Currently, the resource model estimates block RAM (BRAM) and DSP usage that are usually the bottleneck of designs. The DSE process will sweep through the design space with all feasible combinations of design parameters. For each design parameter list, the resource usage is examined first. Designs that over-utilize the resource will be pruned away. Then, we follow a greedy algorithm to select the optimal tiling factors that minimize the latency layer by layer. The DSE process finishes within minutes on a standard workstation.

3.2 Data Layout Optimization

Data layout optimizations are applied to reduce the number of accesses to DRAM and increase the effective DRAM bandwidth. The first optimization is on the concatenation layers. A CNN network may contain blocks that concatenate the results of several layers. As shown in Figure 3, after each stage in the OpenPose-V2 network, results from two branches will be concatenated with the first outputs from the backbone network. This then serves as the inputs for the following stages. Figure 6 presents the optimized data organization of the network.

The outputs of the backbone (region *B*) and each stage (region *A, C*) are placed close to each other as shown in Figure 6. To be more specific, the outputs of Stage 1 will be written to region A. The regions A and B will serve as the inputs of Stage 2. In Stage 2, the outputs will be written to region C. The regions B and C will serve as the inputs of Stage 3 similarly. The outputs of each stage are written to regions A and C in a round-robin fashion. With this layout, the outputs of stage branches are concatenated on-the-fly, eliminating unnecessary off-chip DRAM movements.

To further improve the effective DRAM bandwidth, we change the data layout of the feature maps from $N(k) \times H(k) \times \frac{W(k)}{Tw(k)} \times$
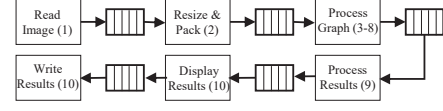

Figure 7: First level of pipeline.

$Tw(k)$ to $\frac{N(k)}{Tn(k)} \times H(k) \times \frac{W(k)}{Tw(k)} \times Tn(k) \times Tw(k)$. This allows us to increase the burst length from $Tw(k)$ to $Tn(k) \times Tw(k)$. A DSC layer can easily become communication-bound because of its low computation to communication (CTC) ratio since it is mostly using 1×1 convolution kernels. In this case, when the kernel size of the next layer is 1×1, since there is no overlapped region between different tiles, we further change the data layout to $\frac{N(k)}{Tn(k)} \times \frac{H(k)}{Th(k)} \times \frac{W(k)}{Tw(k)} \times Tn(k) \times Tw(k) \times Th(k)$. It further increases the burst length for these layers to $Tn(k) \times Th(k) \times Tw(k)$. For other kernel sizes, padding is applied because a tile of $Tn(k) \times Tw(k) \times Th(k)$ does not have all the data needed for the computation. We need to have $(p-1)$ and $((p-1) \times Th(k) + (p-1)^2)$ extra DRAM accesses with burst length of $Tn(k) \times Tw(k)$ and $Tn(k)$ respectively to fetch all the data (p denoting the kernel size). This increases the number of DRAM accesses with a burst length of $Tn(k)$, which further increases the communication time and prevents us from applying this data layout.

4 TENSORFLOW INTEGRATION

We chose TensorFlow as our ML framework since it is being widely used for inference in the ML community (e.g. [12, 15]). To invoke FPGA from TensorFlow, we redefine the nodes in the original computation graph. All computation nodes of CNN are merged into one node that is implemented by FPGA. The rest of the graph is still processed on the CPU.

When FPGA is connected to TensorFlow, the whole integration stack consists of the following steps: 1) reading the inputs of CNN, 2) pre-processing include stages like image resizing, 3) re-organizing the initial data layouts in CPU memory, 4) transferring data from CPU to FPGA device memory, 5) computation on FPGA, 6) fetching the results back via PCIe, 7) reformatting and passing it to TensorFlow, 8) non-CNN computation stages on CPU, 9) processing the results (e.g., estimating the human poses based on the attained results and drawing them for OpenPose network), and 10) writing out and displaying the results.

Figure 2 shows the breakdown of these stages in the OpenPose application for an 384×384 RGB input. Among the whole pipeline, which takes 208.8ms, the FPGA computation in Step 5 only requires 11.8% of the total time. The integration overheads have led to 8.45× performance slowdown. To reduce these overheads we have applied an optimized software/hardware pipelining.

A two-level pipelining is applied on the whole integration stack that enables the simultaneous processing of the aforementioned steps. The first level overlaps TensorFlow's overheads (steps 1, 2, 9, 10) with the rest of the steps. The second one overlaps FPGA's computation with data movement steps (steps 3, 4, 6, 7).

Figure 7 illustrates the first level of pipeline, which is applied at the TensorFlow level. The numbers in the figure show the related step number. Steps 1, 2, 9, and 10 and the rest of the steps are assigned to different processes connected by queue. Therefore, steps
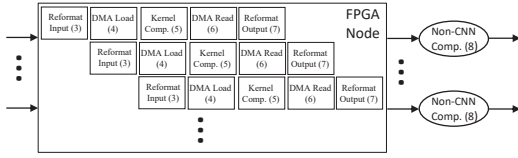
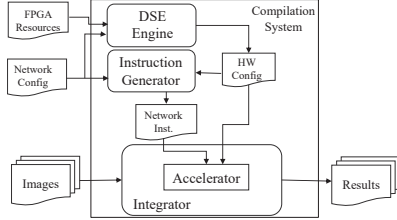**Figure 8: The overview of the Process Graph stage.**

Figure 9: Compilation system overview.

1, 2, 9, and 10 are overlapped with FPGA-related steps. The overall performance is determined by the stage with the longest latency.

To further improve the performance, we fully pipeline the communication and computation of FPGA, which consists of step 3 to 7. This builds the second level of pipeline. To allow pipelining, a batch of images are sent to FPGA. For a certain batch size, the additional latency incurred by batch processing is dissolved when the first level of the pipeline is applied. After the FPGA finishes processing the batch, the results are passed back to TensorFlow and the non-CNN computations are done in parallel for all the images. Figure 8 depicts the redefined graph that we use to achieve such a pipeline. With this optimization, the data movement steps are overlapped with kernel computation and the latency for non-CNN computation (Step 8) is amortized for the whole batch. Note that such deep software+hardware pipelining techniques were also used in [5, 9] for integrating FPGA accelerators to Spark-based applications.

5 FLEXCNN COMPILATION SYSTEM

Figure 9 depicts the overview of our compilation system, which is composed of three modules: design space exploration engine, instruction generator, and integrator.

- **Design space exploration engine.** The CNN network description file parsed from TensorFlow and the available target FPGA resources are fed into the design space exploration engine to search for the optimal hardware configuration parameters. Once found, these parameters are then used for generating the accelerator. The DSE process was explained in Section 3.1.1
- **Instruction generator.** The instruction generator takes the CNN network description file and accelerator hardware description file as the inputs and creates one instruction for each CNN layer. The non-convolutional layers (e.g., bias, ReLU(6)) are fused with adjacent convolutional layers to improve the computation efficiency. The instruction generator produces one VLIW-like instruction for each of these fused layers. This instruction contains the enable signal for each of the modules, the layer configurations, and tiling factors. An RBB contains two convolutional layers in one block. As we only have one convolution module in the accelerator, we will

Table 2: Frequency and resource utilization.

Precision	Frequency	LUT	FF	BRAM	URAM	DSP
float 32-bit	242.9MHz	43%	40%	60%	15%	50%

Table 3: Performance on OpenPose-V2.

Model	Precision	Frequency (MHz)	Runtime (ms)	
			(1)	(2)
All Uniform	float 32-bit	237	57.7	41.5
All Dynamic	float 32-bit	242.9	35.6	24.7

(2): With applying DRAM organization for concatenation layers
(1): Without applying DRAM organization for concatenation layers

divide this block into two layers. The first layer performs the first convolution and the next one computes the rest of the block.

- **Integrator.** The integrator takes in the FPGA accelerator and integrates it into the TensorFlow framework, performing the end-to-end processing task. The optimizations on the integrator were covered in Section 4.

6 EXPERIMENTAL RESULTS

6.1 Experiment Setup

The FlexCNN architecture is described using Vivado HLS C[24]. The target platform is Xilinx Virtex Ultrascale+ VCU1525. The design is synthesized and implemented using Xilinx SDAccel 2018.3. We use OpenPose-V2 network explained in Section 2 to test our work. The accelerator can get any input size. For this section, we have configured it to take an RGB image of 384×384 with floating-point precision as inputs.

6.2 Hardware Optimization

The target FPGA platform comes with four DDR banks. In our implementation, we use two DDR banks, assigning feature maps and weights (including bias) to two separate DDR banks. All the architecture choices are parameterizable and can be adjusted based on the target FPGA. We found the following configurations that work best for the OpenPose-V2 application on Xilinx VCU1525. The systolic array for our standard conv module is organized as an 8×8 array with SIMD factor of 8. For the rest of the modules we use the same SIMD factor. The maximal tiling factors for Tn, Tm, Th, Tw are 64, 64, 12, and 96, respectively. Table 2 shows the frequency and resource utilization under this configuration.

Table 3 shows the benefits of dynamic tiling and data layout transformation. We can see that these optimizations increase the performance by 2.3×. Figure 1 depicts the performance gain of using dynamic tiling in a layer-by-layer fashion for the first 24 convolutional layers. Table 4 shows how applying dynamic tiling and dynamic data layout affects the tiling factors and effective DRAM bandwidth (BW) for the first layer of the last RBB in OpenPose-V2 compared to a design without these optimizations. The kernel size for this layer is 1×1 which means it can use the optimized data layout with burst length of $Tn(k) \times Tw(k) \times Th(k)$ as described in Section 3.2. This data layout, along with the best tiling factor used for this layer increases the effective DRAM BW and CTC ratio by 2.8×. This results in 6.1× performance improvement.

We further test the DSP efficiency of our design on a given convolution layer. Of all the DSPs, 78.7% of them are used in Standard

Table 4: Performance impacts of dynamic tiling/data layout transformation.

Model	Th	Tw	Tn	Tm	Eff. DRAM BW (GB/s)	CTC	Perf. (GFLOPs)
All Uniform	12	48	32	32	4.31	14.9	24.4
All Dynamic	12	24	48	48	12.05	41.3	149.2 (6.1×)
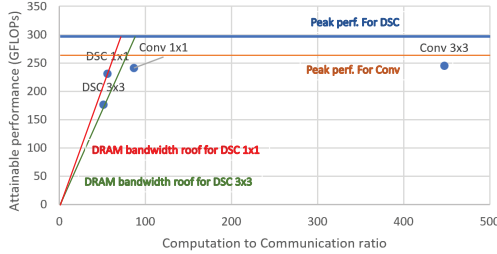
Figure 10: Layers in Table 5 under the roofline model.

Table 5: Performance on different convolutional layers.

Layer	Runtime (ms)	Perf. (GFLOPs)	DSP_{total} eff	DSP_{used} eff
Conv 3x3	709.3	245.2	73.8%	93%
Conv 1x1	80.2	240.9	72.6%	91.4%
DSC 3x3	113.4	176.3	53.1%	58.6%
DSC 1x1	84.1	230.8	69.5%	76.7%

Table 6: Performance impacts of integration optimization.

Version	Runtime frame (ms)	Perf. (FPS)	Speedup
Original	208.8	4.8	1
1st pipeline	97.1	10.3	2.1
2nd pipeline	42	23.8	5

Conv module and 11.2% in Depth Conv module. We measure DSP efficiency using two factors: the total number of DSPs in the design and the number of DSPs of the modules used by that layer. All the tests are on a $256 \times 384 \times 384$ input, producing 256 output channel. Table 5 summarizes the results. DSC layers take $K^2 \times$ less computation, making them communication-bound as shown in Figure 10. This figure depicts that DSC layers fall in the memory-bound region of the roofline model since they have less CTC ratio. Therefore, we achieve lower computation efficiency in these layers. Additionally, it shows that the data layout optimization for the DSC with the 1×1 kernel increases the burst length. This helps to increase the effective DRAM bandwidth, leading to a performance improvement over the 3×3 DSC.

6.3 Integration Optimization

In this section, we evaluate the effect of our integration optimization. FlexCNN runs at 24.7ms, which translates to a peak performance of 40.5FPS. However, without proper optimization, the direct integration into TensorFlow framework only leads to the performance of 4.8FPS, as shown in Table 6. Table 6 summarizes the impacts of two-level pipelining on the overall performance. We are using a batch of 16 for the OpenPose network to enable pipelining on FPGA since it produces the best performance and smoothest output when displaying the result. With two-level pipelining, we achieve up to 5× speedup, which leads to the final performance of 23.8FPS.

Table 7: Performance comparison of different platforms.

Platform	Frequency (GHz)	Runtime (ms)	Dynamic Power (W)
CPU	2.4	99.3	17
GPU	1.4	25.3	38
FPGA (ours)	0.243	24.7	10

6.4 Comparative Studies

To the best of our knowledge, there is only one work [2] that has implemented a variant of OpenPose on FPGA. However, they take a different approach. They reduce the computation cost of the original network by making the weights sparse and using only two stages after the backbone network. Furthermore, they quantized the data to 16-bit fixed point and stored feature maps and weights on-chip. After these modifications, they neither reported their network's computation cost nor their architecture's resource utilization. Thus, we can not compare our results to theirs directly. Instead, we have compared our results against the network implementation using TensorFlow on CPU and GPU.

The CPU is a 56-core Intel Xeon CPU E5-2680 v4 that operates at 2.40GHz. For GPU, we use the NVIDIA Tesla V100 GPU and it uses cuDNN[6] to run the network. To have a fair comparison on the latency of running the network on different platforms, we measure the runtime of a single image inference using OpenPose-V2 network. Table 7 summarizes the results. The runtime considers only the CNN inference time on RGB images of size 384×384. For both the FPGA and GPU, the time to transfer the data from host to device and device to host is excluded from the measurement. We also measure the dynamic power on each platform, which is calculated as the difference of the hardware power when running and not running the application. Both GPU and FPGA suffer from the low data reuse and degree of parallelism of this network (this is why FlexCNN's performance on this network is only 117GFLOPs). However, FlexCNN is 3.8× more energy efficient than GPU.

7 CONCLUSION

The rapid evolution of CNN networks has brought new challenges to FPGA acceleration. In this paper, we identify two major challenges including the performance disparity of different CNN layers and the high overheads of integrating FPGA into ML framework. To tackle these two challenges, we propose an accelerator named FlexCNN which employs dynamic tiling and data layout optimization to improve the hardware efficiency across layers. These two techniques achieve 2.3× speedup on the studied Openpose-V2 network. Furthermore, we propose a two-level integration pipeline to reduce the integration overheads. It adds another 5x speedup of the overall performance. At last, we are able to meet the requirement of real-time processing with 23.8FPS with these optimization techniques.

ACKNOWLEDGEMENTS

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[2] Jinguji Akira, Tomoya Fujii, Shimpei Sato, and Hiroki Nakahara. 2018. An FPGA Realization of OpenPose Based on a Sparse Weight Convolutional Neural Network. In *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 310–313.

[3] Lin Bai, Yiming Zhao, and Xinming Huang. 2018. A CNN accelerator on FPGA using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs* 65, 10 (2018), 1415–1419.

[4] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. 2017. Realtime multi-person 2d pose estimation using part affinity fields. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7291–7299.

[5] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. 2016. When Spark meets FPGAs: A case study for next-generation DNA sequencing acceleration. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[7] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[8] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[9] Jason Cong, Peng Wei, and Cody Hao Yu. 2018. From JVM to FPGA: Bridging abstraction hierarchy via optimized deep pipelining. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[10] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 152–159.

[11] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[12] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. 2018. AI benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 0–0.

[13] Ildoo Kim. 2018. tf-pose-estimation. https://github.com/ildoonet/tf-pose-estimation. (2018).

[14] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–9.

[15] De G Matthews, G Alexander, Mark Van Der Wilk, Tom Nickson, Keisuke Fujii, Alexis Boukouvalas, Pablo León-Villagrá, Zoubin Ghahramani, and James Hensman. 2017. GPflow: A Gaussian process library using TensorFlow. *The Journal of Machine Learning Research* 18, 1 (2017), 1299–1304.

[16] Daniel H Noronha, Bahar Salehpour, and Steven JE Wilton. 2018. LeFlow: Enabling flexible FPGA high-level synthesis of TensorFlow deep neural networks. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE, 1–8.

[17] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[18] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 535–547.

[19] Laurent Sifre and Stéphane Mallat. 2014. Rigid-motion scattering for image classification. *Ph. D. dissertation* (2014).

[20] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[21] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.

[22] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. 2018. TGPA: Tile-grained pipeline architecture for low latency CNN inference. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[23] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 29.

[24] Xilinx. 2018. Vivado Design Suite User Guide - High-Level Synthesis (UG902). (2018).

[25] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, et al. 2018. DNN dataflow choice is overrated. *arXiv preprint arXiv:1809.04070* (2018).

[26] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[27] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[28] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 56.