Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks

Chen Zhang[1,2,3*], Zhenman Fang[2], Peipei Zhou[2], Peichen Pan[3], and Jason Cong[1,2,3†]

[1]Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]Falcon-computing Inc., USA
chen.ceca@pku.edu.cn, {zhenman, memoryzpp, cong}@cs.ucla.edu, peichen@falcon-computing.com

ABSTRACT

With the recent advancement of multilayer convolutional neural networks (CNN), deep learning has achieved amazing success in many areas, especially in visual content understanding and classification. To improve the performance and energy-efficiency of the computation-demanding CNN, the FPGA-based acceleration emerges as one of the most attractive alternatives.

In this paper we design and implement Caffeine, a hardware/software co-designed library to efficiently accelerate the entire CNN on FPGAs. First, we propose a uniformed convolutional matrix-multiplication representation for both computation-intensive convolutional layers and communication-intensive fully connected (FCN) layers. Second, we design Caffeine with the goal to maximize the underlying FPGA computing and bandwidth resource utilization, with a key focus on the bandwidth optimization by the memory access reorganization not studied in prior work. Moreover, we implement Caffeine in the portable high-level synthesis and provide various hardware/software definable parameters for user configurations. Finally, we also integrate Caffeine into the industry-standard software deep learning framework Caffe. We evaluate Caffeine and its integration with Caffe by implementing VGG16 and AlexNet network on multiple FPGA platforms. Caffeine achieves a peak performance of 365 GOPS on Xilinx KU060 FPGA and 636 GOPS on Virtex7 690t FPGA. This is the best published result to our best knowledge. We achieve more than 100x speedup on FCN layers over previous FPGA accelerators. An end-to-end evaluation with Caffe integration shows up to 7.3x and 43.5x performance and energy gains over Caffe on a 12-core Xeon server, and 1.5x better energy-efficiency over the GPU implementation on a medium-sized FPGA (KU060). Performance projections to a system with a high-end FPGA (Virtex7 690t) shows even higher gains.

1. INTRODUCTION

In the last few years, deep learning has achieved amazing success in many areas, especially in computer vision and speech recognition. Among various deep learning algorithms, *CNN (convolutional neural networks)* has become the most popular one for visual content understanding and classification, with significantly higher accuracy than traditional algorithms in various compute vision tasks such as face recognition, image and video processing [1–3]. Now CNN is becoming one of the key algorithms in many modern applications, and is attracting enthusiastic interest from both the academic community [1, 3, 4] and industry heavyweights like Google [5], Facebook [6], and Baidu [7].

With the increasing image classification accuracy improvements, the size and complexity of the multilayer neural networks in CNN have grown significantly, and is evidenced by the rapid evolvement of real-life CNN models such as AlexNet, ZFNet, GoogleLeNet, and VGG [8–11]. This puts overwhelming computing pressure on conventional general-purpose CPUs in light of the recent slowdown of Moore's law. Therefore, various accelerators—based on GPUs, FPGAs, and even ASICs—have been proposed to improve the performance of CNN designs [12–14]. Due to its low power, high energy-efficiency, and reprogrammability, the FPGA-based approach has become one of the most promising alternatives and has stimulated extensive interest [13, 15–23].

Most prior FPGA acceleration studies on CNN [13, 15–21] mainly focus on the convolution layer in CNN, since it is computation-bound and is the most timing-consuming layer. However, this leads to three limitations. First, other unaccelerated layers in CNN cannot get the high energy-efficiency from FPGAs. Second, there is significant intermediate data communication overhead between unaccelerated layers on a CPU and the accelerated convolution (CONV) layer on an FPGA through the PCIe connection, which diminishes the overall performance gains [24]. Third, after the FPGA acceleration of the CONV layer, other layers—especially the indispensable fully connected (FCN) layer that is communication bound—can become the new bottleneck in CNN. Based on our profiling (detailed in Section 2.2), the FCN layer actually occupies more than 50% of the total execution time in CNN after the CONV layer is accelerated on an FPGA.

To address the above limitations, two of the latest studies [22, 23] start implementing the entire CNN on an FPGA. The work in [22] transforms a convolution layer into a regular matrix-multiplication (MM) in the FCN layer, and implements an MM-like accelerator for both layers. The other work in [23] takes an opposite approach: it transforms a regular MM into a convolution, and implements a convolution accelerator for both CONV and FCN layers. While these two studies make a good start on accelerating the entire CNN on an FPGA, the straightforward transformation does not consider potential optimization. They demonstrated a performance of approximately 1.2 giga fixed operations per second (GOPS), leaving large room for improvement.

In this paper we aim to address the following key challenges in uniformed and efficient FPGA acceleration of the entire CNN. First, *what is the right representation for a uniformed acceleration of the computation-bound CONV layer and the communication-bound FCN/DNN layer?*[1] Second, *how to design and implement an efficient and reusable FPGA engine for CNN that maximizes the underlying FPGA computing and bandwidth resource utilization*?

To find the right uniformed representation for both CONV and FCN layers, we first analyze the widely used *regular MM representation* in most CPU and GPU studies. These studies usually convert a convolution layer to a regular MM in the FCN layer, and lever-

*Part of this research was performed while Chen Zhang was a summer intern at Falcon Computing Solutions and a visiting student at UCLA.

†In addition to his primary affiliation with UCLA, Jason Cong is a distinguished visiting professor at Peking University and the chief scientific advisor at Falcon Computing Solutions.

ICCAD '16, November 07-10, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2966986.2967011

[1]As analyzed in Section 2.2, other layers in CNN are relatively simple and have marginal impact on the final performance and FPGA resource consumption. We do implement those layers in the same FPGA, but we will mainly discuss the CONV and FCN layers in this paper for simplicity. Note that the FCN layer is also a major component of deep neural networks (DNN) that are widely used in speech recognition. For simplicity, we just use the term "FCN".

age the well-optimized (with vectorization) CPU libraries like Intel MKL and GPU libraries like cuBLAS for a regular MM [12, 25]. However, the convolutional MM to regular MM transformation requires data duplication in CNN. According to our study, this duplication results in up to 25x more data volume for the input feature maps in the CONV layer, and thus diminishes the gains of FPGA acceleration considering that FPGA platforms have extremely limited bandwidth (about 10 to 20 GB/s [26]) compared to CPU/GPU platforms (up to 700GB/s [27]). More importantly, according to our study in Section 4.3, FPGAs' effective bandwidth is very sensitive to memory access burst lengths, which requires a more careful design for bandwidth-bound FCN layers on FPGAs.

To avoid the data duplication and improve the bandwidth utilization, we propose to use a *convolutional MM representation*. Instead of a straightforward mapping in [23], we batch a group of input feature maps in the FCN layer together into a single one in the new representation, which we call *input-major mapping*, so as to improve the data reuse of the weight kernels. Another alternative of this input-major mapping is by reversing the input feature map matrix and weight kernel matrix, which we call *weight-major mapping*, based on the observation that the latter matrix is much larger than the former one in the FCN layer. As a result, the weight-major mapping may have more data reuse, especially for the input feature maps which are easier to be reused by each weight access than that in the input-major mapping considering the hardware resource limitation. Due to more consecutive (burst) memory accesses, both mappings have better effective bandwidth utilization than the straightforward one. Considering the complex data reuse and memory burst access under the hardware resource limitation, it is quite challenging to identify which one is absolutely better between the input-major and weight-major convolutional mappings. For a quantitative comparison, we apply a revised roofline model to guide their design space explorations under different batch sizes. Our evaluation reveals that the weight-major mapping has higher computation-to-communication ratio (more data reuse and better bandwidth utilization) during the convolution, especially when the batch size is small. Therefore, *we choose the convolutional MM representation for both CONV and FCN layers, and apply a weight-major mapping in the FCN layer*.

Based on the above uniformed representation, we design and implement an efficient and reusable CNN/DNN FPGA accelerator engine called Caffeine.[2] First, Caffeine maximizes the FPGA computing capability using unroll and pipeline method similar to [13]. Second, Caffeine maximizes the underlying memory bandwidth utilization by reorganizing the memory accesses in the convolutional MM to maximize the effective memory bandwidth. As a result, Caffeine can achieve high performance for both the computation-bound CONV layer (96 GFLOPS and 365 GOPS, implementing a VGG16 network on a single KU060 FPGA board) and communication-bound FCN layer (45 GFLOPS and 173 GOPS, more than 100x speedup over prior work [23]).

Moreover, Caffeine is provided as a hardware (HW) and software (SW) co-designed CNN/DNN library that can be reused in deep learning frameworks. At HW side, we implement our systolic-like FPGA accelerator in the portable high-level synthesis (HLS) and provide tunable parameters including feature map/weight buffer size and precision (float/half/fixed<any number>), and number of processing elements. At SW side, we abstract the FPGA device driver so that various shaped CNN/DNN networks can be easily accelerated with a single FPGA bit-stream. Our SW interface also provides tunable parameters, including the number and size of input/output feature maps, shape and strides of weight kernels, pooling size and stride, ReLU kernels, and the total number of CNN/DNN layers. Once the configurations are provided to Caffeine, all the layers are computed on an FPGA without CPU interactions.

[2]The name Caffeine comes from CAFfe Fpga EngINE. But it is a generic library and not limited to the CAFFE. It can also be extended for other frameworks like Torch and TensorFlow [28, 29].

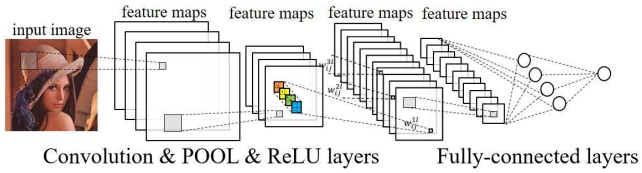


Figure 1: Inference (a.k.a feedforward) phase in CNN

Finally, to demonstrate the advantages of Caffeine, we integrate Caffeine with the industry-standard Caffe deep learning framework [12] and conduct an end-to-end comparison to existing optimized CPU and GPU solutions. Experimental results show that our Caffe-Caffeine integration achieves 7.3x performance speedup and 43.5x energy-efficiency on a medium-sized KU060 FPGA board (higher projection on the VC709 board) compared to Caffe running on a 12-core Xeon CPU. In addition, we show 1.5x higher energy-efficiency compared to the GPU solution [12].

In summary, this paper makes the following contributions.

1. A uniformed convolutional MM representation adapted for efficient FPGA acceleration of both CONV and FCN layers in CNN/DNN, where a weight-major convolutional mapping is applied in the FCN layer under the revised roofline model.
2. A HW/SW co-designed efficient and reusable CNN/DNN engine called Caffeine, where the FPGA accelerator maximizes the computing and bandwidth resource utilization, and achieves 636 GOPS peak performance for the CONV layer and 170 GOPS for the FCN layer on a VC709 board.
3. The first published attempt (to the best of our knowledge) to incorporate FPGAs into the industry-standard deep learning framework Caffe, which achieves 7.3x and 43.5x end-to-end performance and energy gains over CPU and 1.5x better energy-efficiency over GPU on a KU060 FPGA.

2. CNN OVERVIEW AND ANALYSIS

2.1 Algorithm of CNNs

As a typical supervised learning algorithm, there are two major phases in CNN: *training phase* and *inference (aka feed-forward) phase*. Since many industry applications would train CNN in the background and only perform inferences in a real-time scenario, we mainly focus on the inference phase in this paper. The aim of the CNN inference phase is to get a correct inference of classification for input images. Shown in Figure 1, it is composed of multiple layers, where each image is fed to the first layer. Each layer receives a number of feature maps from a previous layer, and outputs a new set of feature maps after filtering by certain kernels. The convolutional layer, activation layer, and pooling layer are for feature map extraction, and the fully connected layers are for classification.

Convolutional (CONV) layers are the main components of CNN. The computation of a CONV layer is to extract feature information by adopting a filter on feature maps from a previous layer. It receives N feature maps as input and outputs M feature maps. A set of N kernels, each sized in $K_1 \times K_2$, slide across corresponding input feature maps with element-wise multiplication-accumulation to filter out one output feature map. S_1 and S_2 are constants representing the sliding strides. M sets of such kernels can generate M output feature maps. The following expression describes its computation pattern.

$$Out[m][r][c] = \sum_{n=0}^{N} \sum_{i=0}^{K_1} \sum_{j=0}^{K_2} W[m][n][i][j] * In[n][S_1 * r + i][S_2 * c + j];$$

Pooling (POOL) layers are used to achieve spatial invariance by sub-sampling neighboring pixels, usually finding the maximum value in a neighborhood in each input feature map. So in a pooling layer, the number of output feature maps is identical to that of input feature maps, while the dimensions of each feature map scale down according to the size of the sub-sampling window.

Activation (ReLU) layers are used to adopt an activation function (e.g., a ReLU function) on each pixel of feature maps from previous layers to mimic the biological neuron's activation [8].

Table 1: Recent CNN models that won the ILSVRC contest

Real-life CNNs	Year	Neurons	layers	Parameters
AlexNet [8]	2012	650,000	8	250 MB
ZFNet [9]	2013	78,000,000	8	200 MB
VGG [11]	2014	14,000,000	16	500 MB

Table 2: Computation complexity, storage complexity, and execution time breakdown of CNN layers in the VGG16 model

	CONV	POOL	ReLU	FCN
Comput. ops(10^9)	$3E3(99.5\%)$	$0.6(0\%)$	$1.4(0\%)$	$12.3(0.4\%)$
Storage (MB)	113(19.3%)	0(0%)	0(0%)	471.6(80.6%)
Time% in pure sw	96.3%	0.0%	0.0%	3.7%
after CONV acc	**48.7%**	**0.0%**	**0.0%**	**51.2%**

Fully-connected (FCN) layers are used to make final predictions. A FCN layer takes "features" in a form of vector from a prior feature extraction layer, multiplies a weight matrix, and outputs a new feature vector, whose computation pattern is a dense matrix-vector multiplication. A few cascaded FCNs finally output the classification result of CNN. Sometimes, multiple input vectors are processed simultaneously in a single batch to increase the overall throughput, as shown in the following expression when the batch size of h is greater than 1. Note that the FCN layers are also the major components of deep neural networks (DNN) that are widely used in speech recognition.

$$Out[m][h] = \sum_{n=0}^{N} Wght[m][n] * In[n][h];$$

2.2 Analysis of Real-Life CNNs

State-of-the-art CNNs for large visual recognition tasks usually contain billions of neurons and show a trend to go deeper and larger. Table 1 lists some of the CNN models that have won the ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) contest since 2012. These networks all contain millions of neurons, and hundreds of millions of parameters that include weights and intermediate feature maps. Therefore, storing these parameters in DRAM is mandatory for those real-life CNNs. In this work we will mainly use the 16-layer VGG16 model [11].

Table 2 shows two key points. *First*, the CONV and FCN layers present two extreme features. CONV layers are very computation-intensive: they contain 19.3% of total data but need 99.5% of computation. FCN layers are memory-intensive: they need 0.4% of arithmetic operations but use 80.6% of the total amount of data. These two layers also occupy most of the execution time (more than 99.9%). *Second*, when CONV is accelerated, the FCN layer becomes the new bottleneck, taking over 50% of computation time. Therefore, we need to accelerate the entire CNN on an FPGA, and maximize the use of both FPGA's computation and bandwidth efficiency. While a straightforward acceleration of the POOL and ReLU layers is good enough due to their simplicity, in this paper we will mainly focus on discussing how to accelerate both the CONV and FCN layers.

3. UNIFORMED CNN REPRESENTATION

3.1 Prior Representation on CPUs and GPUs

Prior CPU and GPU studies [12, 25] most often used the regular matrix-multiplication (MM) representation so as to leverage the well-optimized CPU libraries like Intel MKL and GPU libraries like cuBLAS. To achieve this uniformed acceleration, they convert a convolutional MM in the CONV layer to a regular MM in the FCN layer. However, such a transformation comes at the expense of data duplication, which diminishes the overall performance gains in bandwidth-limited FPGA platforms [22]. Figure 2 illustrates the data duplication overhead by using MM for the CONV layer computation in AlexNet and VGG16 models. Compared to the original convolutional MM representation, the regular MM representation introduces 7.6x to 25x more data for the input feature maps, and 1.35x to 4.8x more data for intermediate feature maps and weights, which makes the CONV layer communication-bound.

3.2 New Representation Adapted for FPGAs

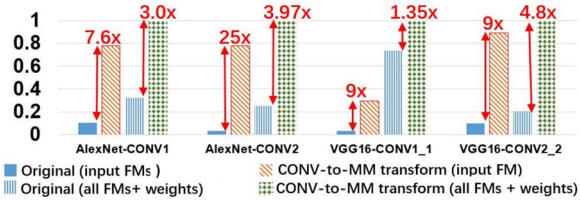


Figure 2: Data duplication by using regular MM for CONV

To avoid the data duplication overhead, we propose to use the convolutional MM representation, and transform the regular MM in the FCN layer to the convolutional MM in the CONV layer. Instead of a straightforward mapping as proposed in [23], we propose two optimized mapping to improve the data reuse and bandwidth utilization: *input-major mapping* and *weight-major mapping*.

3.2.1 Straightforward Mapping

For FCN shown in Figure 3(a), an input vector with size N will do pairwise multiplication with a weight vector of size N and accumulate the results to get one output value. There are M weight vectors and M output values. For CONV shown in Figure 3(b), similarly, N feature maps will convolve with N weight kernels, and then element-wise addition is done for the convolution results to get one output feature map. There are M sets of weight kernels, and we will get M output feature maps.

In a straightforward mapping, each element in an input $1 \times N$ vector of FCN maps to one input feature map sized as R_i=1, C_i=1 of CONV. And each element in an $1 \times N$ weight vector of FCN maps to one weight kernel of CONV sized as K_1=1, K_2=1. This can be viewed in Figure 3(c) when batch size is 1. Prior work in [23] firstly attempted to implement both CONV and FCN using a similar mapping, and demonstrated a performance of nearly 1.2 GOPS, leaving large room for improvement.

3.2.2 Input-Major Mapping

In real-life CNNs, multiple input images are processed in a batch to improve throughput. Therefore, in our *input-major mapping*, we can map a batch of elements from different input vectors in FCN to the same input feature map (FM) in CONV. As a result, the data reuse of FCN weight kernels is improved when convolving the elements from different images in the batched input FMs. When batch size is $batch$, there are $batch$ input vectors in FCN and the reuse ratio of FCN weight kernels is $batch$. Note $batch$ cannot be too large in the real-time inference phase.

To better illustrate the input-major mapping, we use Figure 3(c) to show how we map FCN to CONV when $batch$ = 2, N = 6 and M = 2. The 6 elements of the 1st input vector are mapped to the 1st element of each input FM, and the 6 elements of the 2nd input vector are mapped to the 2nd element of each input FM. Both the weight kernel size and stride size are still 1x1. While the weight kernels slide across the input FMs, they will generate $batch$ elements in each output FM. In addition to the improved data reuse for weight kernels, this batching also improves the memory access burst length of FCN input and output FMs, which improves the bandwidth utilization as explained in Section 4.3.

Another way to improve the memory burst length is to increase the weight kernel size ker and batching ker elements within a single weight (or input) vector in FCN to the same weight kernel (or input FM) in CONV. Figure 3(d) depicts an example where we change ker from 1x1 to 1x2. Compared to Figure 3(c), 2 weights are grouped in one weight kernel, and 2 input FMs are grouped into one input FM. Accordingly, stride size changes with ker to 1x2.

Table 3 *column FCN-Input* lists the parameters after input-major mapping from FCN to CONV. The number of input FMs decreases to $\frac{N}{ker}$, and the number of elements in one input FM increases to $batch \times ker$. The number of elements in an output FM is $batch$.

3.2.3 Weight-Major Mapping

As another alternative to improve the data reuse and bandwidth utilization, we propose *weight-major mapping*, where input vectors of FCN map to weight kernels of CONV, and weight vectors of

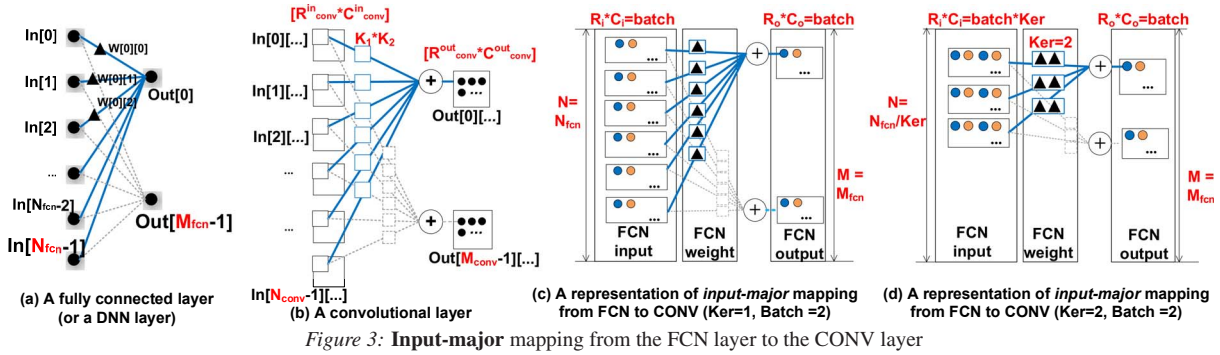


Figure 3: **Input-major** mapping from the FCN layer to the CONV layer

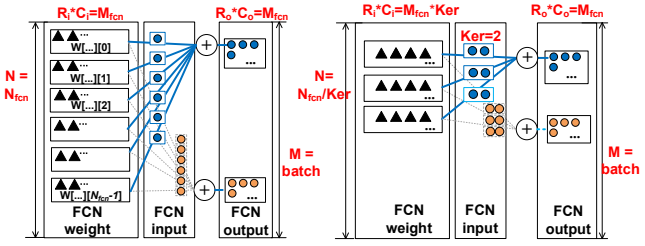


Figure 4: **Weight-major** mapping from the FCN layer to the CONV layer

FCN map to input FMs of CONV. As shown in Figure 4(a), every input vector of FCN in a batch transforms to one set of weight kernels. Weight vectors of FCN are aligned in input FMs in a way that weight elements at the same position of all weight vectors are grouped into the same input FM. Therefore, each FCN input can be reused M_{fcn} times (if it can be buffered on-chip) during the convolution, which greatly improves the data reuse. In addition, the memory burst length of FCN weights and FCN output FMs are greatly improved as well. Similarly, the *batch* size improves the data reuse of FCN weights and improves the memory burst length of FCN input FMs in weight-major mapping. In addition, it decides the number of FCN output FMs that are available to be processed simultaneously.

Similar to input-major mapping, we can increase the kernel size ker in FCN input FMs to increase the memory burst length, with an example of $ker = 2$ shown in Figure 4(b). Table 3 *column FCN-Weight* lists the parameters for weight-major mapping from FCN to CONV.

3.2.4 Uniformed Representation

Since FCN now maps to CONV, either using *input-major mapping* or *weight-major mapping*, we use a uniformed representation (column *Uniformed*) for all cases in Table 3. Considering the complex data reuse and memory burst access under different batch and kernel sizes, as well as the hardware resource constraints, it is quite challenging to identify whether input-major mapping or weight-major mapping is better. Therefore, we will conduct a quantitative design space exploration of concrete parameters in Section 5.

Table 3: Uniformed representation parameters for CONV, FCN input-major mapping and FCN weight-major mapping

	Uniformed	Conv	FCN-Input	FCN-Weight
Input FM #	N	N_{conv}	N_{fcn}/ker	N_{fcn}/ker
Input FM size	$R_i \cdot C_i$	$R^{in}_{conv} \cdot C^{in}_{conv}$	batch · ker	$M_{fcn} \cdot ker$
Output FM #	M	M_{conv}	M_{fcn}	batch
Output FM size	$R_o \cdot C_o$	$R^{out}_{conv} \cdot C^{out}_{conv}$	batch	M_{fcn}
Kernel size	$K_1 \cdot K_2$	$K_1 \cdot K_2$	ker	ker
Stride	$S_1 \cdot S_2$	$S_1 \cdot S_2$	ker	ker

4. CAFFEINE DESIGN

With our proposed uniformed representation, we design and implement Caffeine, a HW/SW co-designed CNN/DNN FPGA accelerator engine. Overall, the key features of Caffeine include:

1. **Software definable hardware accelerator**. Our accelerator can be configured by software to support various CNN/DNN designs at run time without changing the FPGA bitstream.
2. **Scalable accelerator architecture**. We implement a systolic-like processing element (PE) array in HLS that can be easily scaled up to larger FPGA devices that have more resources.
3. **Efficient computing/bandwidth resource utilization**. We optimize bandwidth utilization by reorganizing memory accesses.

4.1 HW/SW Co-Designed CNN Library

4.1.1 SW Automation and Definable Parameters

Layer configuration. Caffeine provides flexibility for users to easily configure the CNN/DNN layers they want in SW.

1. For CONV/FCN layers, the SW definable parameters include the number and size of input/output feature maps, weight kernel size and strides, as summarized in Table 4.
2. Pooling and activation layers (if used) always follow CONV layers, and can automatically derive their feature maps from prior CONV layers. $\langle flag_{pool}, flag_{act} \rangle$ parameters are provided to flag whether a pooling/activation layer is used. Moreover, kernel size and stride $\langle K_{pool}, S_{pool} \rangle$ in the pooling layer can also be configured.
3. The number of layers for each type of layer is also configurable.

SW design automation. We automate the SW design in Caffeine so users only need to provide the CNN/DNN layer configurations, input images, and HW accelerator bitstream. It works as below.

1. Parse CNN/DNN layer configurations and weight values.
2. Allocate device DRAM space for weights and feature map data. Data reorganization is used to maximize effective bandwidth. This will be presented in Section 4.3.
3. Transfer weights to FPGA and reuse them for to-be-classified images. Transfer each image to FPGA DRAM.
4. Launch FPGA acceleration job. During FPGA's computation for multiple layers, no interference is required by the host CPU. After that, transfer output result back to the host.

4.1.2 HW Definable Parameters

Due to FPGA's on-chip memory and computation resource constraints, all loops and data arrays involved in the convolution computation must be properly tiled and cached during the computation. Therefore, we provide tiling parameters for each dimension as hardware-definable parameters to define the on-chip buffers for weights, input and output feature maps and the number of parallel PEs, summarized in Table 4.

Table 4: CONV/FCN layer parameters and FPGA accelerator tiling factors

	Software definable	Hardware defineable
	Uniformed param	Tiling factors
Input/Output FM #	$N,\ M$	$T_n,\ T_m$
Input/Output FM size	$R_i \cdot C_i,\ R_o \cdot C_o$	$T_r \cdot T_c,\ T_r \cdot T_c$
Kernel/Stride size	$K_1 \cdot K_2,\ S_1 \cdot S_2$	$T_{k1} \cdot T_{k2},\ -$

4.2 Scalable Accelerator Architecture

To achieve a scalable convolution accelerator design, we design a massive number of parallel PEs to improve computation performance, and organize these PEs as a systolic array to mitigate timing issues in synthesizing a large design. We implement two levels of parallelism as suggested in [13] for the sake of better hardware utilization and circuit simplicity: 1) parallelism in computing multiple

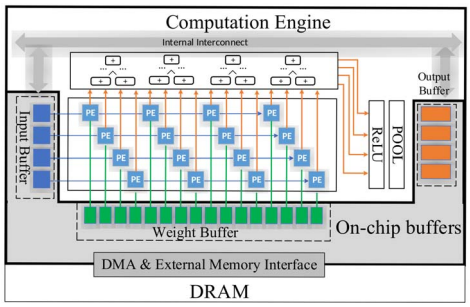


Figure 5: Scalable accelerator architecture design

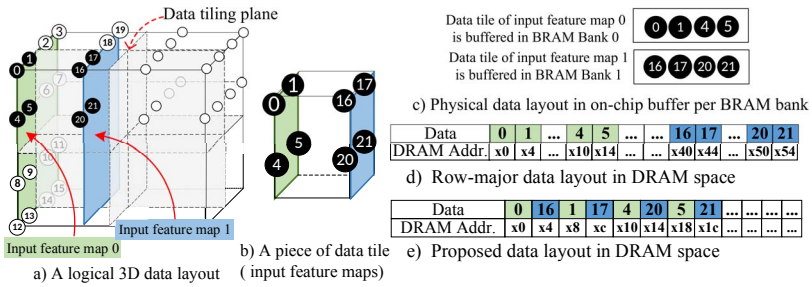


Figure 6: Bandwidth optimization by DRAM layout reorganization

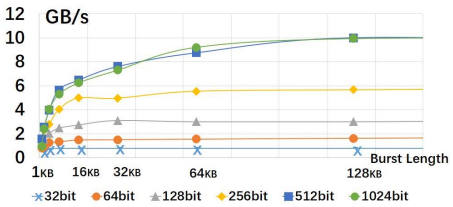


Figure 7: Effective FPGA DRAM bandwidth

output feature maps; and 2) parallelism in processing multiple input feature maps for each output feature map. Figure 5 presents an overview of our scalable accelerator architecture, together with corresponding buffers. Each PE is an arithmetic multiplication of input feature map pixels and corresponding weights. An array of adder trees sums up the convolution results. The total number of PEs is defined by $T_m \times T_n$ in Table 4.

We also implement the pooling layer using a max-pooling with kernels that are defined through software parameters, and a ReLU function for the activation layer. Either of them could be bypassed if there is no such layer following a convolution layer.

To achieve a pipeline initial interval (II) of 1, i.e., each PE processes one input data every cycle, we use a polyhedral-based optimization framework [30] to optimize the pipelining schedule by permuting the parallel loop levels to the innermost levels to avoid loop carried dependence. We also use the double buffering technique to prefetch the next data tile for each PE so that the computation time can overlap with the data transfer overhead from the device DRAM to FPGA's BRAM.

4.3 Accelerator Bandwidth Optimization

Since the FCN layer is bandwidth sensitive, we need to be careful about the accelerator bandwidth optimization. In order to have a sense of effective FPGA DRAM bandwidth under different memory access patterns, we test it on the latest Kintex Ultrascale KU060 FPGA as a representative with Xilinx SDAccel 2015.3 flow. Figure 7 plots the effective DRAM bandwidth under different memory access burst lengths and bit-widths. We make two observations in efficient FPAG DRAM bandwidth utilization. First, the effective FPGA bandwidth ('Y' axis) goes up with the increase of burst length ('X' axis) and finally flattens out above some burst length threshold. Limited burst length will greatly degrade actual bandwidth performance, like 1GB/s on 1KB memory burst access. Second, longer interface bit-width can achieve higher peak bandwidth. The maximum effective bandwidth of 10GB/s (about 83% of theoretical 12.8GB/s) can be only reached at 512 bit-width and above, when the burst length is above 128KB.

Off-chip bandwidth optimization opportunity. As analyzed earlier, the burst length and bit-width of DRAM interface are two dominating factors for FPGAs' effective bandwidth. However, the widely used data tiling technique usually results in a discontinuous DRAM access for the row-major data layout in DRAM. We illustrate this using an example in Figure 6. Figure 6.a) describes 4 input feature maps in a logical 3-dimension representation, each with a size of 4×4. Each dimension is tiled by 2 so that each tile has $2 \times 2 \times 2 = 8$ elements in total. The first tile of input feature maps is shown in Figure 6.b). Figure 6.d) presents its corresponding data layout in DRAM in a row-major representation, which results in 4 discontinues blocks. Therefore, it requires 4 DRAM accesses, each with a burst length of 2 floating points. This results in a pretty low memory bandwidth utilization and can greatly degrades the overall performance, especially for the bandwidth-intensive FCN layers.

On-chip buffer access optimization opportunity. BRAM banks are usually organized for maximum parallel data access from massive parallel PEs. As illustrated in Figure 6.c), elements (0, 1, 4, 5) from input feature map 0 should be put in bank 0, while elements (16, 17, 20, 21) from input feature map 1 should be put in bank 1. However, such requirements would cause on-chip bank write conflicts using the original DRAM organization in Figure 6.d). When loading continuous data blocks (0, 1) from DRAM to BRAM (similar for other pairs), they will be written to the same bank 0, which causes bank write conflicts and introduces additional overhead.

Optimization. To improve the effective memory bandwidth, we reorganize the DRAM layout as illustrated in Figure 6.e). First, we move the data for an entire tile to a continuous space to improve the memory burst length and bit-width. Second, we interleave the data for different BRAM banks to reduce bank read/write conflicts.

5. DESIGN SPACE EXPLORATION

In this section we discuss how to find the optimal solution of mapping a CNN/DNN onto our accelerator architecture. We explore the design space by revising a roofline model for accurate early-stage performance estimation.

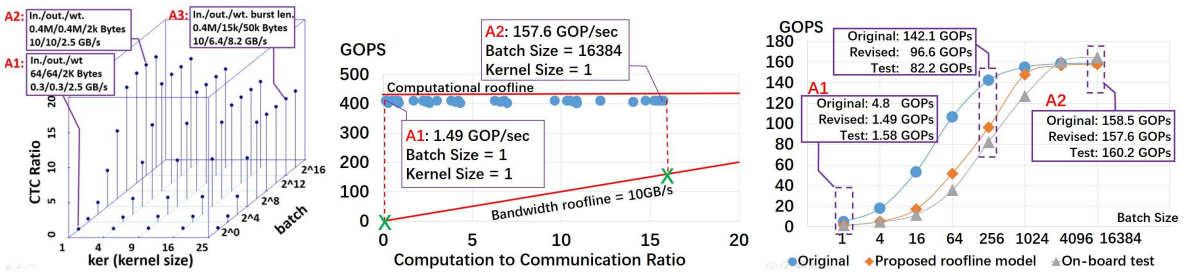
5.1 Revised Roofline Model for Caffeine

The roofline model [31] is initially proposed in multicore systems to provide insight analysis of attainable performance by relating processors' peak computation performance and the off-chip memory traffic. It is first introduce in [13] to the FPGA accelerator design for the CONV layers in CNN. Each implementation in the roofline model is described/bounded by two terms. First, the computation-to-communication (CTC) ratio, as in the 'X' axis of Figure 8(b) and Figure 9(b), features the number of operations per DRAM byte access. Second, computational performance, as in the 'Y' axis of Figure 8(b) and Figure 9(b), features the peak computation performance provided by available computational resources.

The key defect of the original roofline model used in [13] is that it ignores the fact input/output/weight arrays have different data volumes in each tile, and thus have different burst lengths and effective bandwidths. As proposed in [13], the original total number of DRAM access in one layer's computation is given by the following equation, where β denotes the size of input/output/weight data tile, and α denotes the number of times of data transfer for input/output/weight data.

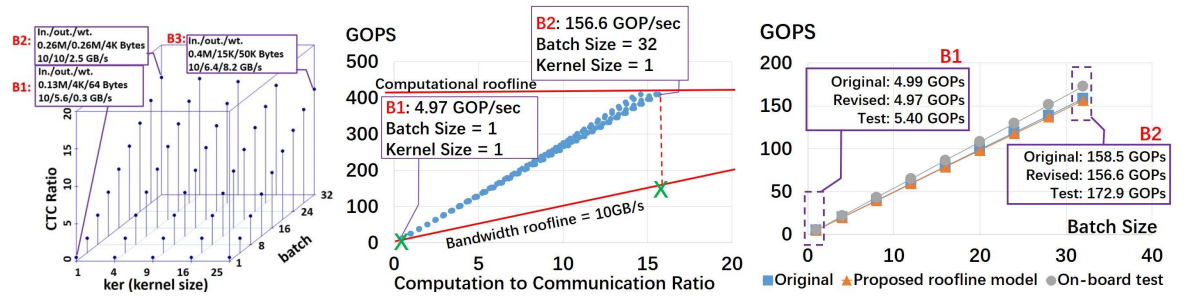
$$DRAM\_Access = \alpha_{in} \cdot \beta_{in} + \alpha_{wght} \cdot \beta_{wght} + \alpha_{out} \cdot \beta_{out} \quad (1)$$

In fact, Equation 1 does not accurately model the total DRAM traffic. For example, as shown in Figure 7, the effective bandwidth on 1KB burst DRAM access is only 1GB/s, 10x lower than the maximum effective bandwidth 10GB/s. Therefore, the original roofline model becomes extremely inaccurate in bandwidth sensitive workloads because it actually takes 10x longer time to make the data transfer than expected. So we would like to multiply a normalization factor of 10x on the original DRAM traffic.



(a) Design space in CTC ratio *(b)* Design space in revised roofline model *(c)* Comparison of original, revised roofline models and on-board test results

Figure 8: Design space exploration for FCN **input-major** mapping under various batch and kernel sizes



(a) Design space in CTC ratio *(b)* Design space in revised roofline model *(c)* Comparison of original, revised roofline models and on-board test results

Figure 9: Design space exploration for FCN **weight-major** mapping under various batch and kernel sizes

In general, we propose to normalize the DRAM traffic of input/output/weight accesses to the maximum effective bandwidth with a normalization factor γ.

$$\gamma_{in} \cdot \alpha_{in} \cdot \beta_{in} + \gamma_{wght} \cdot \alpha_{wght} \cdot \beta_{wght} + \gamma_{out} \cdot \alpha_{out} \cdot \beta_{out} \quad (2)$$

where γ is defined by the equation below. The f function is given by the curve of effective bandwidth with respect to the burst length, shown in Figure 7.

$$\gamma = max\_bandwidth/f(\beta) \quad (3)$$

5.1.1 Final Roofline Model

Given a specific set of software-definable parameters for one layer $\langle N,\ R_i,\ C_i,\ M,\ R_o,\ C_o,\ K,\ S\rangle$ and a specific hardware definable parameter $\langle T_m,\ T_n,\ T_r,\ T_c\rangle$, as described in Section 4.1, we can determine its 'X' and 'Y' axis value in the roofline model by computing its computational performance and CTC ratio.

Similar to [13], the computational performance is given by:

$$computation\ performance = \frac{total\ computation\ ops}{execution\ cycles} = \frac{2 \cdot N \cdot M \cdot R_o \cdot C_o \cdot K_1 \cdot K_2}{\lceil \frac{N}{T_n} \rceil \cdot \lceil \frac{M}{T_m} \rceil \cdot R_o \cdot C_o \cdot K_1 \cdot K_2} \quad (4)$$

Our revised CTC ratio is given by:

$$CTC\ ratio = \frac{total\ computation\ ops}{total\ DRAM\ access} = \frac{2 \cdot N \cdot M \cdot R_o \cdot C_o \cdot K_1 \cdot K_2}{\gamma_{in} \cdot \alpha_{in} \cdot \beta_{in} + \gamma_{wght} \cdot \alpha_{wght} \cdot \beta_{wght} + \gamma_{out} \cdot \alpha_{out} \cdot \beta_{out}} \quad (5)$$

5.2 Design Space Exploration

Since optimizing the CONV layer with the roofline model has been extensively discussed in [13], and there is a space constraint, we mainly focus on optimizing the mapping of the FCN layer to the uniformed representation using our revised roofline model. Specifically, it is a problem of choosing input-major/weight-major mapping methods and the optimal $batch$ and ker parameters, given the FCN layer configuration and hardware configuration.

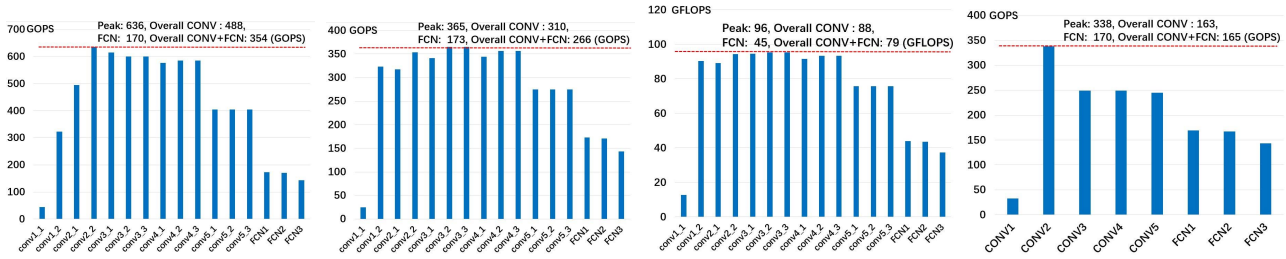
We use the VGG16 model's FCN layer 1 as an example; it has an input of 25088 (N_{fcn}) neurons and output of 4096 (M_{fcn}) neurons, whose notations follow Table 3. $Batch$ and ker are tunable parameters for mapping FCN to the uniformed representation as described in Section 3. We use the hardware configuration from Kintex Ultrascale KU060 platform and set hardware definable parameters as $\langle T_m, T_n, T_r \cdot T_c, T_{K1} \cdot T_{K2}\rangle = \langle 32, 32, 6272, 25\rangle$. We choose our tile sizes based on the guidance of [13] to maximize the FPGA resource utilization. Users can configure their own tile sizes.

5.2.1 FCN Input-Major Mapping

Figure 8(a) presents the design space of FCN input-major mapping in terms of CTC ratio under various batch ($batch$) and kernel (ker) sizes. First, given a fixed ker, the CTC ratio increases with $batch$, because $batch$ FCN inputs reuse FCN weights, and memory burst length is increased by $batch$ which results in higher effective DRAM bandwidth. The CTC ratio flattens out when $batch$ is bigger than on-chip BRAM size. Second, given a fixed $batch$, the CTC ratio increases with ker when $batch$ is small, because this increases memory burst length and thus benefits effective DRAM bandwidth. Finally, since the size of input FM is given by $batch \cdot ker$ in Table 3, the maximum $batch$ that could be cached in on-chip BRAM decreases when ker increases. Therefore, the CTC ratio decreases when ker increases on a large $batch$, because the output FM burst length (given by $batch$ according to Table 3) decreases. In the input-major mapping, the maximum CTC ratio is achieved with a parameter $\langle batch, ker\rangle = \langle 16384, 1\rangle$.

Figure 8(b) presents input-major mapping's attainable performance using our revised roofline model. Each point represents an implementation with its computation performance in GOPS and CTC ratio estimation, which are decided by parameters $\langle batch, ker\rangle$ according to our model. The red line (bandwidth roofline, slope = 10GB/s) represents the max DRAM bandwidth that this FPGA platform supports. Any point located above this line indicates that this implementation requires higher bandwidth than what the platform can provide. Thus it is bounded by platform bandwidth, and the attainable performance is then decided by the bandwidth roofline. From this figure, we can see that all implementations of FCN with input-major mapping are bounded by bandwidth. The highest attainable performance is achieved at the highest CTC ratio, where $\langle batch, ker\rangle = \langle 16384, 1\rangle$, and this $batch$ size is unreasonable in a real-time inference phase.

Figure 8(c) presents the on-board test performance of input-major mapping and the comparison between performance estimations from original and revised roofline models. Our revised



(a) VC709 VGG 16-bit fixed-point *(b)* KU VGG 16-bit fixed-point *(c)* KU VGG 32-bit floating-point *(d)* KU AlexNet 16-bit fixed-point

Figure 10: Caffeine results on multiple FPGA boards for different CNN models and data types

roofline model is much more accurate than the original one, and our estimated performance is very close to that of the on-board test.

5.2.2 FCN Weight-Major Mapping

Figure 9(a) presents the design space of FCN weight-major mapping in terms of CTC ratio under various batch ($batch$) and kernel (ker) sizes. As illustrated in Section 3.2.3, $batch$ represents the number of concurrent PEs processing different output feature maps in weight-major mapping. Due to the FPGA resource constraints, we can only put 32 such PEs in the KU060 FPGA. Therefore, we set an up-limit of 32 to $batch$ in weight-major mapping, which is pretty small. Given a fixed ker, the CTC ratio increases with $batch$ since it increases the data reuse of FCN weights and the memory burst length of FCN inputs. The size of ker has marginal impact on weight-major mapping because it has pretty good bandwidth utilization even for ker = 1.

Figure 9(b) presents weight-major mapping's attainable performance using our revised roofline model. Similar with input-major mapping, all implementations of weight-major mapping are bounded by bandwidth. In addition, small $batch$ size also leads to lower computational performance due to less number of concurrent PEs in weight-major mapping. The highest attainable performance is achieved at the highest CTC ratio, where $\langle batch, ker\rangle = \langle 32, 1\rangle$, which is reasonable in a real-time inference phase.

Figure 8(c) presents the on-board test performance of weight-major mapping and the comparison between performance estimations from original and revised roofline models. Different than input-major mapping, weight-major mapping has very good data reuse as well as good effective bandwidth as illustrated in Section 3. So the proposed roofline model is only slightly better than original model, and both models are close to the on-board test. In addition, weight-major mapping presents better performance than input-major mapping in cases of small $batch$ sizes.

Due to the advantages of weight-major to input-major mapping in small batch sizes, in the rest of this paper we will use weight-major mapping for the FCN layer with the best design point.

6. CAFFEINE RESULTS

6.1 Experimental Setup

CNN models. To demonstrate the software definable features of Caffeine, we use two CNN models—AlexNet [8] and VGG16 [11]. Users only need to write two configuration files for them.

CPU and GPU setup. The baseline CPU we use is a two-socket server, each with a 6-core Intel CPU (E5-2609 @ 1.9GHz). The GPU we use is a NVIDIA GPU K40. OpenBLAS and cuDNN libraries are used for the CPU and GPU implementations [12].

FPGA setup. The main FPGA platform we use is the Xilinx KU3 board with a Kintex Ultrascale KU060 (20nm) and a 8GB DDR3 DRAM, where SDAccel 2015.3 is used to synthesize the bitstream. To demonstrate the portability of our hardware-definable architecture, we also extend our design to the VC709 (Virtex 690t, 28nm) FPGA board. We make the IP design with Vivado HLS 2015.2 and use Vivado 2015.2 for synthesis.

6.2 Caffeine Results on Multiple FPGAs

To demonstrate the flexibility of Caffeine, we evaluate Caffeine using 1) two FPGA platforms, KU060 and VC709, 2) two data types, 32-bit floating point and 16-bit fixed point, and 3) two network models, AlexNet and VGG16, as shown in Figure 10.

First, Figure 10(a) and Figure 10(b) present the VGG16 performance on VC709 and KU060 platforms, respectively. VC709 can achieve higher peak performance (636 GOPS) and higher overall performance of all CONV+FCN layers (354 GOPS) than KU060 (peak 365 GOPS and overall 266 GOPS). Both figures show that most layers can achieve near-peak performance. Layer 1 is a special case because it only has three input feature maps (three channels for RBG pictures). For both platforms, the FCN layer's performance is quite similar (around 170 GOPS for overall performance of all FCN layers) because they are mainly bounded by bandwidth.

Second, Figure 10(b) and Figure 10(c) present the differences between a 16-bit fixed point and 32-bit floating point on KU060. Both CONV and FCN layers show a drastic increase in performance using a 16-bit fixed point. For CONV layers, fixed point saves computation resources and thus enables more parallelism. For FCN layers, fixed point saves bandwidth because of its 16-bit size.

Third, Figure 10(b) and Figure 10(d) present the KU060 platform's performance on VGG16 and AlexNet. VGG16 has better performance since it has a more regular network shape which is more suitable for accelerators (better utilization after tiling).

Finally, Table 5 presents the FPGA resource utilization of the above implementations. SDAccel uses a partial reconfiguration to write bit-stream, and thus it has an up-limit of 60% of all available resources. We use about 50% of DSP resources on the KU060 board. We use 80% of DSP resources on the VC709 board. Caffeine on the KU060 board runs at a frequency of 200MHz, and on VC709 it runs at a frequency of 150MHz.

Table 5: FPGA resource utilization of Caffeine

	DSP	BRAM	LUT	FF	Freq
VC709 fixed	2833(78%)	1248(42%)	3E5(81%)	3E5(36%)	150MHz
KU fixed	1058 (38%)	782(36%)	1E5(31%)	8E4(11%)	200MHz
KU float	1314(47%)	798(36%)	2E5(46%)	2E5(26%)	200MHz

Table 6: Comparison with other FPGA work

	[13]	[23]	[22]	Ours	
CNN models	AlexNet	VGG			
Device	Virtex 480t	Zynq XC7Z045	Stratix-V GSD8	Ultrascale KU060	Virtex 690t
Precision	float 32 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit
DSP #	2240	780	1963	1058	2833
CONV(peak) GOPS	83.8	254.8	-	365	636
CONV(overall) GOPS	61.6	187.8	136.5	310	488
FCN (overall) GOPS	-	1.2	-	173	170
CONV+FCN GOPS	-	137	117.8	266	354

6.3 Comparison with Prior FPGA Work

We compare our accelerator design to three state-of-the-art studies in Table 6. We compare four terms of performance: 1) peak CONV layer performance, 2) overall performance of all CONV layers, 3) overall performance of all FCN layers, and 4) overall performance of all CONV+FCN layers. Our work significantly outperforms all three prior studies in all terms of performance. Our FCN layer achieves more than 100x speedup over previous work.

7. CAFFE-CAFFEINE INTEGRATION

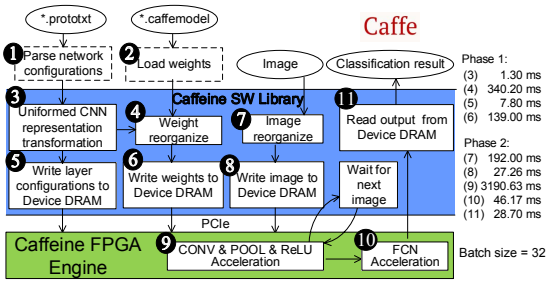


Figure 11: Caffe-Caffeine integration

To further demonstrate the advantages of Caffeine, we integrate Caffeine with the industry-standard Caffe deep learning framework [12]. Note that Caffeine can also be integrated into other frameworks like Torch [28] and TensorFlow [29]. Figure 11 (left) presents an overview of Caffeine's HW/SW library and its integration with Caffe. The integrated system accepts standard Caffe files for network configuration and weight values. All users have to provide in the integration is parsing the network configurations and loading weights (step 1 and 2) into our Caffeine software library. Caffeine will take care of the rest.

There are two major execution phases in Caffeine. In **phase 1** (steps 3 to 6), it establishes the uniformed representation and automatically decides the optimal transformation, as illustrated in Section 5, and then reorders weights for bandwidth optimization as illustrated in Section 4.3. Finally, it initializes the FPGA device with weights and layer configurations. Phase 1 only needs to execute once unless users want to switch to a new CNN network. In **phase 2** (steps 7 to 11), Caffeine conducts the CNN acceleration: in batch mode, Caffeine will accumulate multiple CONV outputs and do FCN once in a batch; in single mode, Caffeine will do CONV and FCN once for each input image. A detailed execution time breakdown of Caffeine running the VGG16 network on a KU060 platform is shown in the right part of Figure 11 with a batch size of 32, where CONV layers dominate the entire execution again.

Table 7: End-to-End comparison with CPU/GPU platforms

platforms	CPU	CPU+GPU	CPU+FPGA	
Device	E5-2609	K40	KU060	VX 690t
Technology	22nm	28nm	20nm	28nm
Freq.	1.9GHz	1GHz	200MHz	150MHz
Power(Watt)	150	250	25	26
Latency (ms) per img.	733.7	15.3	101.15	65.13
Speedup	1x	48x	7.3x	9.7x
J per image	110	3.8	2.5	1.69
Energy Efficiency	1x	28.7x	43.5x	65x

End-to-end integration results. We conduct an end-to-end comparison between Caffe-Caffeine integration with existing optimized CPU and GPU solutions [12] for VGG16 in Table 7. In order to compare CPU/GPU (using floating point) and FPGAs (using fixed point), we use giga operations per second (GOPS) as the standard metric. Using 16-bit fixed point operations can achieve comparable classification accuracy with its floating point counterparts, which has been extensively studied in prior work [22, 23]. We have similar results and omit this discussion for the sake of page limits. With on-board (KU060) testing, our integration demonstrates an end-to-end performance of 7.3x speedup over 12-core CPU and a 1.5x energy-efficiency over the Nvidia K40 GPU. We project the performance of a CPU-FPGA platform with VC709 board (it is a standalone board and we only have FPGA on-board results) and expect a 9.7x speedup over CPU and 2.2x energy-efficiency over GPU.

8. CONCLUSION

In this work we proposed a uniformed convolutional matrix-multiplication representation to accelerate both the computation-bound convolutional layers and communication-bound fully connected layers of CNN/DNN on FPGAs. Based on the uniformed representation, we designed and implemented Caffeine, a HW/SW co-designed reusable library to efficiently accelerate the entire CNN/DNN on FPGAs. Finally, we also integrated Caffeine into the industry-standard software deep learning framework Caffe. We evaluated Caffeine and its integration with Caffe using both AlexNet and VGG networks on multiple FPGA platforms. Caffeine achieved up to 365 GOPS on KU060 board and 636 GOPS on VC707 board, and more than 100x speedup on fully connected layers over prior FPGA accelerators. Our Caffe integration achieved 7.3x and 43.5x performance and energy gains over a 12-core CPU, and 1.5x better energy-efficiency over GPU on a medium-sized KU060 FPGA board.

Acknowledgments

This work is partially supported by the Center for Domain-Specific Computing under the Intel Award 20134321 and NSF Award CCF-1436827 and C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We also thank the UCLA/PKU Joint Research Institute, Chinese Scholarship Council, and AsiaInfo Inc. for their support of our research.

References

[1] Y. Taigman *et al.*, "Deepface: Closing the gap to human-level performance in face verification," in *CVPR*, 2014, pp. 1701–1708.
[2] K. He *et al.*, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *arXiv preprint arXiv:1502.01852*, 2015.
[3] R. Girshick *et al.*, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, 2014, pp. 580–587.
[4] S. Ji *et al.*, "3d convolutional neural networks for human action recognition," *TPAMI*, vol. 35, no. 1, pp. 221–231, 2013.
[5] A. Coates *et al.*, "Deep learning with cots hpc systems," in *ICML*, 2013, pp. 1337–1345.
[6] O. Yadan *et al.*, "Multi-gpu training of convnets," *arXiv preprint arXiv:1312.5853*, p. 17, 2013.
[7] K. Yu, "Large-scale deep learning at baidu," in *CIKM*. ACM, 2013, pp. 2211–2212.
[8] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.
[9] M. D. Zeiler *et al.*, "Visualizing and understanding convolutional networks," in *ECCV 2014*. Springer, 2014, pp. 818–833.
[10] C. Szegedy *et al.*, "Going deeper with convolutions," *arXiv preprint arXiv:1409.4842*, 2014.
[11] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[12] Y. Q. C. Jia, "An Open Source Convolutional Architecture for Fast Feature Embedding," http://caffe.berkeleyvision.org, 2013.
[13] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*. ACM, 2015, pp. 161–170.
[14] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.
[15] C. Farabet *et al.*, "Cnp: An fpga-based processor for convolutional networks," in *FPL*. IEEE, 2009, pp. 32–37.
[16] S. Chakradhar *et al.*, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.
[17] D. Aysegul *et al.*, "Accelerating deep neural networks on mobile processor with embedded programmable logic," in *NIPS*. IEEE, 2013.
[18] S. Cadambi *et al.*, "A programmable parallel accelerator for learning and classification," in *PACT*. ACM, 2010, pp. 273–284.
[19] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *ASAP*. IEEE, 2009, pp. 53–60.
[20] M. Peemen *et al.*, "Memory-centric accelerator design for convolutional neural networks," in *ICCD*. IEEE, 2013, pp. 13–19.
[21] K. Ovtcharov *et al.*, "Accelerating deep convolutional neural networks using specialized hardware," February 2015.
[22] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*. ACM, 2016, pp. 16–25.
[23] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*. ACM, 2016, pp. 26–35.
[24] Y.-k. Choi *et al.*, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *DAC 2016*, pp. 109:1–109:6.
[25] J. Bergstra *et al.*, "Theano: a cpu and gpu math expression compiler," in *SciPy*, vol. 4, 2010, p. 3.
[26] V. D. Suite, "Ultrascale architecture fpgas memory interface solutions v7.0," Technical report, Xilinx, 04 2015, Tech. Rep., 2015.
[27] S. Mittal, "A survey of techniques for managing and leveraging caches in gpus," *Journal of Circuits, Systems, and Computers*, vol. 23, no. 08, 2014.
[28] "Torch7," http://torch.ch.
[29] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[30] W. Zuo *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *FPGA*. ACM, 2013, pp. 9–18.
[31] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," *CACM*, vol. 52, no. 4, pp. 65–76, 2009.