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

Chen Zhang,[1,2] Guangyu Sun,[1] Zhenman Fang,[2] Peipei Zhou,[2] Peichen Pan,[3] Jason Cong[1,2,3]

[1]Center for Energy-effcient Computing and Applications, Peking University, Beijing, China
[2]Center for Domain-Specific Computing, University of California, Los Angeles, US
[3]Falcon Computing Solutions Inc., Los Angeles, US

***Abstract*—With the recent advancement of multilayer convolutional neural networks (CNN) and fully connected networks (FCN), 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 and FCN on FPGAs. First, we propose a uniformed convolutional matrix-multiplication representation for both computation-bound convolutional layers and communication-bound fully connected (FCN) layers. Based on this representation, we optimize the accelerator micro-architecture and maximize the underlying FPGA computing and bandwidth resource utilization based on a revised roofline model. Moreover, we design an automation flow to directly compile high-level network definitions to the final FPGA accelerator. As a case study, we integrate Caffeine into the industry-standard software deep learning framework Caffe. We evaluate Caffeine and its integration with Caffe by implementing VGG16 and AlexNet networks on multiple FPGA platforms. Caffeine achieves a peak performance of 1,460 GOPS on a medium-sized Xilinx KU060 FPGA board; to our knowledge, this is the best published result. It achieves more than 100x speed-up on FCN layers over prior FPGA accelerators. An end-to-end evaluation with Caffe integration shows up to 29x and 150x performance and energy gains over Caffe on a 12-core Xeon server, and 5.7x better energy efficiency over the GPU implementation. Performance projections for a system with a high-end FPGA (Virtex7 690t) show even higher gains.

***Index Terms*—convolutional neural network, deep learning, Caffe, CNN FPGA engine, hardware/software co-design.**

I. 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 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 , Facebook , and Baidu [5–7]. With the increasing image classification accuracy improvements, the size and complexity of the multilayer neural networks in CNN have grown significantly, as 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 [12–15]. Due to its low power, high energy efficiency, and reprogrammability, the FPGA-based approach is now one of the most promising alternatives and has stimulated extensive interest [13, 16–29].

Most prior FPGA acceleration studies on CNN [13, 16–22, 26] 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 that 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 [30]. 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 II-B), 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 [23, 24] started implementing the entire CNN on an FPGA. The work [23] 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 [24] 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 optimizations. They demonstrated a performance of approximately 1.2 giga fixed point operations per second (GOPS), leaving large room for improvement.

In this paper we aim to address the following key challenges in efficient FPGA acceleration of the entire CNN. First, *what is the right mathematical representation for a uniformed acceleration of the computation-bound CONV layer and the communication-bound FCN/DNN layer?*[1] Second, *how do we design and implement an efficient and reusable FPGA engine for CNN that maximizes the underlying FPGA computing and bandwidth resource utilization, while still maintaining enough programmability for various layer configurations*? Third, *how do we provide software programmers an easy-to-use interface such that they can still write high-level network definitions while taking advantage of our Caffeine FPGA engine?*

To find the right programming model and efficient implementation for CNN kernels, 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 leverage the well-optimized (with vectorization) CPU libraries like Intel MKL and GPU libraries like cuBLAS for a regular MM [12, 31]. 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 [32]) compared to CPU/GPU

[1]As analyzed in Section II-B, 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."

platforms (up to 700GB/s [33]). More importantly, according to our study in Section III-C, the FPGA 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 prior work [24], 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 achieved 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 those in the input-major mapping considering the hardware resource limitation. 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 an accurate roofline-based model to guide their design space explorations under different neural network shapes and batch sizes.

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 by optimizing multilevel data parallelism within CNN, as well as fine-grained and coarse-grained pipeline parallelism. Second, Caffeine maximizes the underlying memory bandwidth utilization by combining both on-chip and off-chip data reorganizations for the convolutional MM representation. As a result, Caffeine can achieve high performance for both the computation-bound CONV layer and communication-bound FCN layer (more than 100x speed-up over prior work [24]). To improve the portability of Caffeine across different FPGA platforms, we design our FPGA accelerator in a systolic-like micro-architecture using high-level synthesis (HLS) so that it can be easily scaled up to a larger design [36]. In addition, Caffeine also supports various CNN layer configurations with different precision requirements (i.e., both floating-point and fixed-point operations). Finally, we further provide an automation flow for software programmers so that they can easily take advantage of our FPGA accelerator engine while still programming the high-level CNN networks, just as they do for CPUs and GPUs. Our automation flow directly parses CNN network definitions—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—and compiles them into our hardware-customized instructions and reorganized weight models. Once the instructions and weights are written to the FPGA, all the layers are then computed on FPGA without the CPU interaction. As a case study, we have integrated Caffeine with the industry-standard Caffe deep learning framework [12], such that the broad Caffe users can directly benefit from the performance and energy gains of our Caffeine FPGA engine without experiencing any programming difference.

In our experiments we conduct an end-to-end comparison of the Caffe-Caffeine integration to existing optimized CPU and GPU solutions. Caffeine achieves a peak performance of 1,460 GOPS with the widely used 8-bit fixed-point operations on a medium-sized Xilinx KU060 FPGA board. Compared to Caffe running on a 12-core Xeon CPU, the Caffe-Caffeine integration achieves a 7.3x performance speed-up and 43.5x energy saving with 16-bit fixed-point operations, and 29x performance speed-up and 150x energy saving with 8-bit fixed-point operations. Compared to the GPU solution [12] running on a Nvidia GTX1080 GPU, we show 2x (batch = 16) and 5.7x (batch = 1) better energy efficiency for 8-bit fixed-point operations, and for FPGA 16-bit fixed-point operations, 0.58x (batch = 16) and 1.6x (batch = 1) energy efficiency over GPU, respectively. The performance and energy gains are even higher when projecting to a larger FPGA board, such as the Xilinx VC709 board used in this paper.

[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 [34, 35].

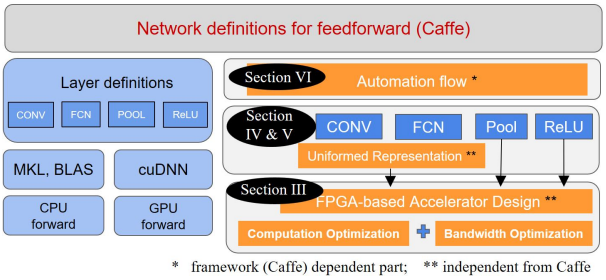


Fig. 1: Overview of Caffeine Framework

We summarize our Caffeine work in Figure 1. At the top of Figure 1 is the high-level network definitions for CNN, e.g., the definitions used in Caffe which can be compiled into the underlying hardware. On the left side of Figure 1 is the existing CNN layer representation and optimization libraries on CPU and GPU devices. Caffeine complements existing frameworks with an FPGA engine. In summary, this paper makes the following contributions.

1. We propose a uniformed mathematical representation (convolutional MM) for efficient FPGA acceleration of both CONV and FCN layers in CNN/DNN. In addition, we also propose a novel optimization framework based on the roofline model to find the optimal mapping of the uniformed representation to the specialized accelerator. Our optimization framework recommends weight-major mapping and input-major mapping according to platform constraints and NN configurations.
2. We customize a HW/SW co-designed efficient and reusable CNN/DNN engine called Caffeine, where the FPGA accelerator maximizes the utilization of computing and bandwidth resource. Caffeine achieves a peak performance of 1,460 GOPS for the CONV layer and 346 GOPS for the FCN layer with 8-bit fixed-point operations on a medium-sized FPGA board (KU060).
3. We provide an automation flow for users to program CNN in high-level network definitions, and the flow directly generates the final FPGA accelerator. We also provide the Caffe-Caffeine integration, which achieves 29x and 150x end-to-end performance and energy gains over a 12-core CPU and 5.7x better energy efficiency over a GPU.

II. CNN Overview and Analysis

A. Algorithm of CNNs

As a typical supervised learning algorithm, there are two major phases in CNN: a *training phase* and an *inference (aka feed-forward) phase*. Since many industry applications 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 2, 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

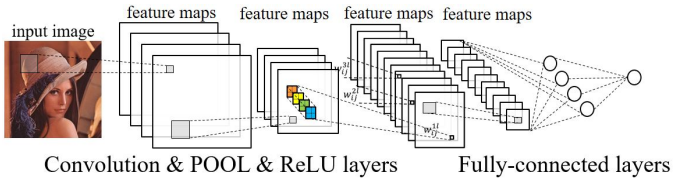


Fig. 2: Inference (aka feedforward) phase in CNN

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].

Fully connected (FCN) layers are used to make final predictions. An 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.

$$Out[m][h] = \sum_{n=0}^{N} W[m][n] * In[n][h]; \quad (1)$$

B. 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 I 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 I: 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 II 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. Since 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.

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

	CONV	POOL	ReLU	FCN
Comput. ops(10^7)	$3E3(99.5\%)$	$0.6(0\%)$	$1.4(0\%)$	$12.3(0.4\%)$
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%**

III. Specialized Convolution Accelerator

There are several design challenges that are an obstacle to an efficient convolution accelerator design on an FPGA platform. First, the organization of processing elements (PEs) and buffer banks should be carefully designed in order to process on-chip data efficiently. Second, loop tiling is mandatory to fit a small portion of data on-chip, and the computation of an entire CONV layer includes many iterations of off-chip data transfers. Improper off-chip memory accesses may degrade the utilization of bandwidth and parallelism of on-chip data processing. Third, integration with high-level frameworks such as Caffe not only needs to guarantee optimal performance with customized optimizations, but also requires enough programmability of the specialized hardware.

In the following subsections, we start from the original convolution code in Figure 3 and apply a combination of optimizations to achieve a high-performance specialized hardware accelerator design. Section III-A presents an overview of the computation model and SW/HW co-designed framework. Section III-B provides on-chip computation engine design optimization, and Section III-C provides memory access optimization.

A. Convolution Accelerator Overview

The computation pattern of a convolution layer is shown in Fig 3. Variables in red are all layer parameters, which are set in CNN training and usually differ among layers. CNN computations involve three types of data: 1) weights and biases, 2) input and output feature maps/images, and 3) CNN network definitions. Real scenario CNNs usually contain a large volume of data, ranging from hundreds of mega bytes to several giga bytes, and a large number of layers, ranging from tens to hundreds of layers.

Loop tiling and computation model on FPGA. FPGAs have limited BRAM and DSP resources. In order to support real-life CNNs with hundreds of mega bytes or even giga bytes of weights and feature maps, our CNN accelerator design puts all the data in DRAM and caches a part of weights, feature maps and layer definitions in on-chip buffers before they are fed to processing engines (PEs). We call each small part a data tile. Figure 3 shows a standard convolution layer's computation procedure. We further apply loop tiling to fit a convolution layer to the FPGA. In CNN structure designs, variables *R and C* (for the "rows" and "columns" of pixels in feature maps) range from tens to thousands; variables *N and M* (for the number of input and output feature maps) range from tens to hundreds; *KEL* (for convolution kernel size) ranges from one to ten. So we do not tile on loops "*k1 & k2*" because of their small sizes. Other loops are tiled into "tile loops" and "point loops." Point loops are for on-chip data's computation, whose optimization is discussed in Section III-B. Tile loops are for bringing data tiles on-chip, whose optimizations are discussed in Section III-C. Figure 4 shows a pseudo code of a tiled convolution layer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems


```
for(r=0; r<ROW; r++){
  for(c=0; c<COL; c++){
    for(o=0; o<OUT; o++){
      for(i=0; i<IN; i++){
        for(k1=0; k1< K1 ; k1++){
          for(k2=0; k2< K2 ; k2++){
            output[o][r][c] +=
            weights[o][i][k1][k2] * input[i][S*r+k1][S*c+k2];
} } } } } }
```

Fig. 3: Pseudo code of a convolution layer

```
External Data Transfer Optimization To be discussed in Section IV/C
for(rr=0; rr<R; rr+=Tr){
  for(cc=0; cc<C; cc+=Tc){
    for(oo=0; oo<M; oo+=To){
      for(ii=0; ii<N; ii+=Ti){
On-chip Data Computation Optimization To be discussed in Section IV/B
Data_type cache_output[ To ][ OutBRAMSize ][ OutBRAMSize ];
Data_type cache_input[ Ti ][ InBRAMSize ][ InBRAMSize ];
Data_type cache_weight[ To ][ Ti ][ KernelSize ][ KernelSize ];
for(p=0; p<K1 ; p++){
  for(q=0; q<K2  q++){
    for(r=0; r<Tr; r++){
      for(c=0; c<Tc; c++){
        for(o=0; o< To ; o++){
          for(i=0; i< Ti ; i++){
            cache_output[o][r][c] +=
            cache_weights[o][i][p][q] * cache_input[i][S*r+p][S*c+q];
} } } } } } } } } }
```

Hardware definable parameters

Software definable parameters

Fig. 4: Pseudo code of a tiled convolution layer

Software-definable parameters and accelerator design. As described in Figure 3, a convolution layer is featured by a set of parameters <M, N, R, C, K1, K2, S>. In order to enable our accelerator's programmability by software at run time without changing the FPGA bitstream, we set parameters <M, N, R, C, K1, K2, S> (which are variables in the blank rectangle in Figure 4) to be software-definable parameters. In our specialized hardware design, we make them registers to control loop pipelines and could be reset by decoding accelerator-specific instructions during runtime. Our Caffeine library provides an automation flow to translate high-level languages to the accelerator-specific instructions.

Hardware-definable parameters and accelerator design. Except for software-definable parameters, the other parameters in Figure 4 are hardware-definable parameters, which are labeled in the black rectangle "*OutBRAMSize & InBRAMSize & KernelSize*" for buffer sizes, "*To & Ti*" for parallel PEs and "*Data_type*" for floating/fixed<bit-width> point operators. Their values determine hardware design's number of parallel PEs as well as size of buffer sets on the FPGA. They are set before bit-stream synthesis. Larger values of "*OutBRAMSize & InBRAMSize & KernelSize*" result in more BRAM utilization and larger values of "*To & Ti*" result in more parallel PEs. Whenever a user wants to switch to a new FPGA device (with larger or smaller number of BRAM/DSP resources), they can simply reset hardware-definable parameters to customize a new accelerator bitstream with our library.

B. Scalable Accelerator Architecture

Figure 5 shows the original code of on-chip computation, and Figure 6 shows the computation structure after our optimization; these are described in the following paragraphs.

Multilevel data parallelism. We implement two levels of data parallelism as suggested in [13] for the sake of better hardware utilization and circuit simplicity: 1) parallelism in computing multiple output feature maps; and 2) parallelism in processing multiple input feature maps for each output feature map. 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 $T_o \times T_i$ in Figure 4.

Fine-grained pipeline parallelism. In order to fully utilize computation resource, our accelerator aims to achieve a pipeline initial interval (II) of 1; i.e., each PE is able to process a pair of input data

```
1. for(o=0; o< To ; o++){
2.   for(i=0; i< Ti ; i++){
3.     for(r=0; r< Tr ; r++){
4.       for(c=0; c< Tc ; c++){
5.         for(p=0; p< K1 ; p++){
6.           for(q=0; q< K2 ; q++){
               cache_output[o][r][c] +=
               cache_weights[o][i][p][q] * cache_input[i][ S *r+p][ S *c+q];
} } } } } } } } } }
```

Fig. 5: Pseudo code of original on-chip computation

```
1. for(p=0; p< K1 ; p++){
2.   for(q=0; q< K2 ; q++){
3.     for(r=0; r< Tr ; r++){
4.       for(c=0; c< Tc ; c++){
#pragma for LOOP PIPELINE (Fine-grained Pipeline Parallelism)
5.         for(o=0; o< To ; o++){
#pragma for LOOP UNROLL (Multilevel Data Parallelism)
6.           for(i=0; i< Ti ; i++){
#pragma for LOOP UNROLL (Multilevel Data Parallelism)
               cache_output[o][r][c] +=
               cache_weights[o][i][p][q] * cache_input[i][ S *r+p][ S *c+q];
} } } } } } } } } }
```

Fig. 6: Pseudo code of optimized on-chip computation

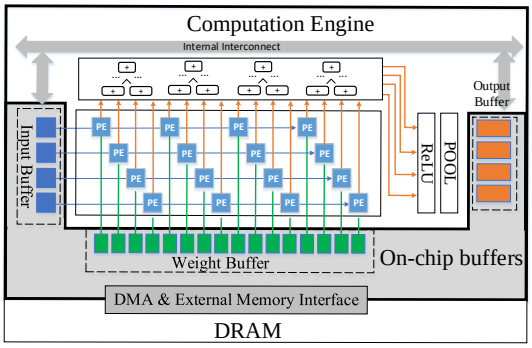


Fig. 7: Scalable accelerator architecture design

on every cycle. We use a polyhedral-based optimization framework [37] to optimize the pipelining schedule by permuting the 'p', 'q', 'r', 'c' loop levels to avoid loop carried dependence. Since pooling layers and ReLU layers are usually an optional layer following convolution layers, we also implement them in the instruction pipeline so that they can be processed immediately on convolution's output. They can also be bypassed if there is no such layer following a convolution layer They can also be configured through software-definable parameters.

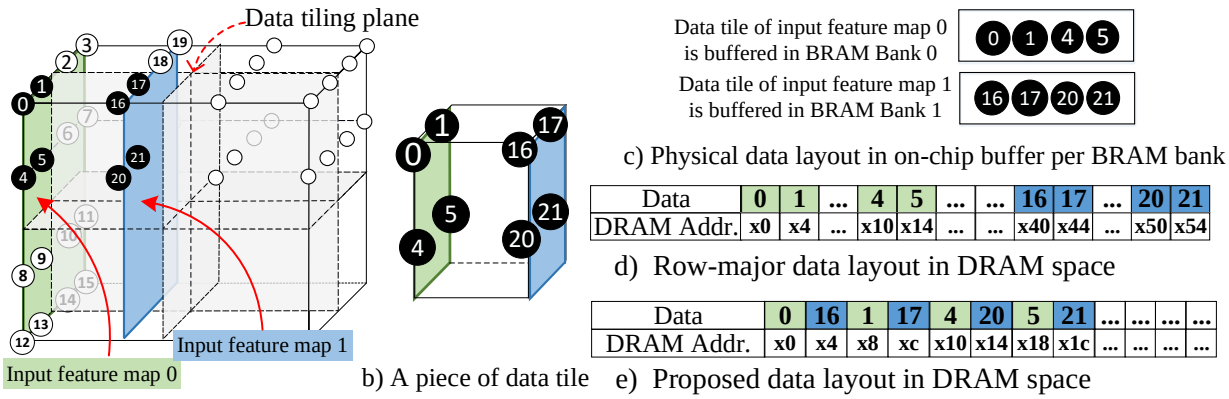
Coarse-grained pipeline parallelism. All of the layers' weight and input/output feature maps are initially stored in DRAM, and layers are computed one by one. On each layer's computation, a tile of input feature maps and weights are fetched on FPGA local buffers. We use the double buffering technique 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.

Scalable architecture. The computations shown in Figure 6 are a typical map and reduction pattern. We further use a systolic-like architecture to implement the above computations so that the hardware design could be scalable to a larger device with more parallel engines. Figure 7 presents an overview of our scalable accelerator architecture, which is designed in portable high-level synthesis (HLS). Similar methods are also explored in work[25].

C. 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 bandwidths under different memory access patterns, we test this on the latest Kintex Ultrascale KU060 FPGA as a representative with Xilinx SDAccel 2015.3 flow. Figure 9 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



Data	0	1	...	4	5	...	...	16	17	...	20	21
DRAM Addr.	x0	x4	...	x10	x14	...	...	x40	x44	...	x50	x54

Data	0	16	1	17	4	20	5	21	...	...	...	...
DRAM Addr.	x0	x4	x8	xc	x10	x14	x18	x1c	...	...	...	...

Fig. 8: Bandwidth optimization by DRAM layout reorganization

Fig. 9: Effective FPGA DRAM bandwidth under different memory access burst lengths and bit-widths

('Y' axis) goes up with the increase of burst length ('X' axis) and finally flattens out above some burst length threshold, which is about 128KB on 512-bit bitwidth in our experiment. 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 8. Figure 8.a) describes four 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 8.b). Figure 8.d) presents its corresponding data layout in DRAM in a row-major representation, which results in four 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 degrade 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 8.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 8.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 8.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. Algorithm 9 presents the method for transforming the cube indexes in Figure8.a) to indexes in linear DRAM space as shown in Figure8.e). Weight and output tensors use a similar method.

Algorithm 1 DRAM Allocation and Data Organization

Input:
Parameters for feature map tensor shape, $[M, R, C]$
Parameters for Input BRAM buffer, $[T_m, T_r, T_c]$
Output:
A linear list of tensor index in DRAM,
List = $\{a_i \mid i \in [0, M \times R \times C)\}$

for each $[i, j, k] \in [\frac{M}{T_m}, \frac{R}{T_r}, \frac{C}{T_c}]$ **do**
  for each $[i_t, j_t, k_t] \in [T_m, T_r, T_c]$ **do**
    $Tile\_Size = T_m \times T_r \times T_c$
    $Tile\_Addr = (i + j \times \frac{M}{T_m} + k \times \frac{M}{T_m} \times \frac{R}{T_r}) \times Tile\_Size$
    $Point\_Addr = i_t + j_t \times T_m + k_t \times T_m \times T_r$
    $Addr = Tile\_Addr + Point\_Addr$
    Append $Addr$ in the mappingList
  end for
end for

IV. Uniformed CONV and FCN Representation

A. Prior Representation on CPUs and GPUs

Prior CPU and GPU studies [12, 31] 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 [23]. Figure 11 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.

B. New Representation Adapted for FPGAs

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 [24], we propose two optimized mappings to improve the data reuse and bandwidth utilization: *input-major mapping* and *weight-major mapping*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

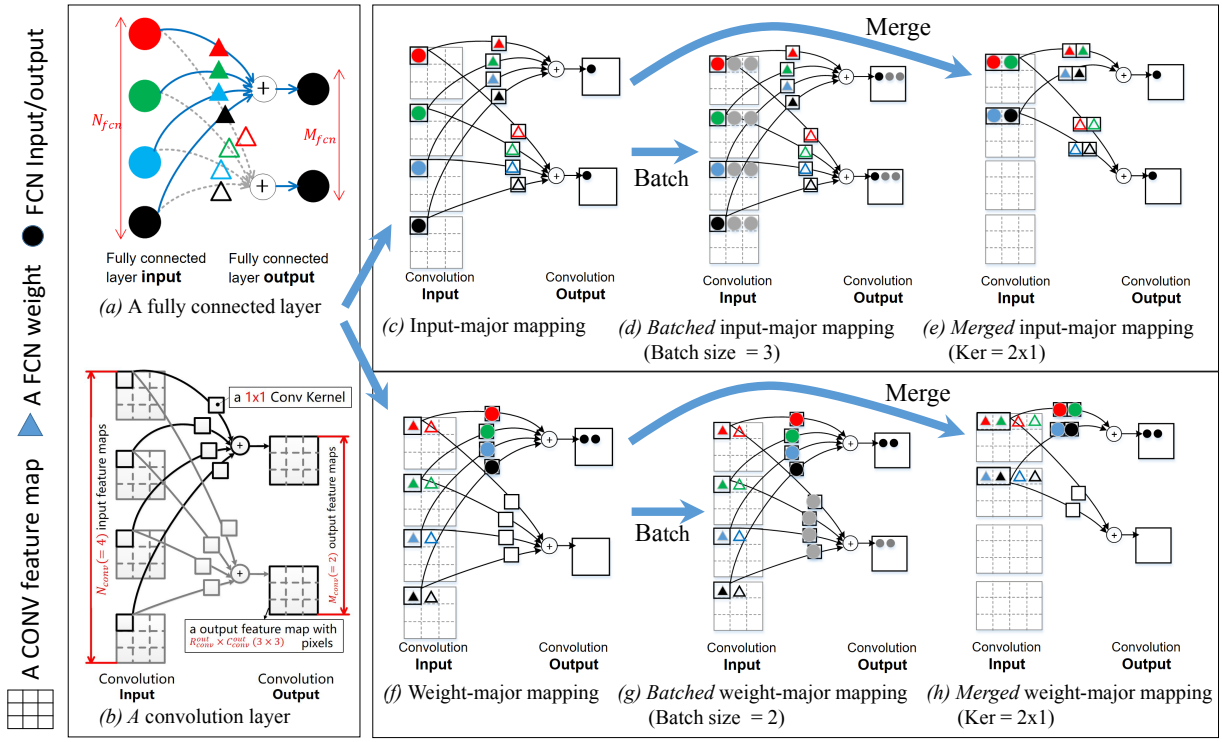


Fig. 10: **Input-major and Weight-major** mapping from the FCN layer to the CONV layer

TABLE III: Input-major mapping (batch_size = 1)

	input	weight	output
Original FCN layer	25088×1	25088×4096	4096×1
HW buffer name	input buffer	weight buffer	output buffer
HW buffer size	32×4096	32×32×3×3	32×4096
Size of data tile	32×1	32×32×1×1	32×1
Burst length	32	1024	32
# of memory access	**784**	**100,352**	**128**

TABLE IV: Weight-major mapping (batch_size = 1)

	input	weight	output
Original FCN layer	25088×1	25088×4096	4096×1
HW buffer name	weight buffer	input buffer	output buffer
HW buffer size	32×32×3×3	32×4096	32×4096
Size of data tile	32×1×1	32×4096	1×4096
Burst length	32	131072	4096
# of memory access	**784**	**784**	**1**

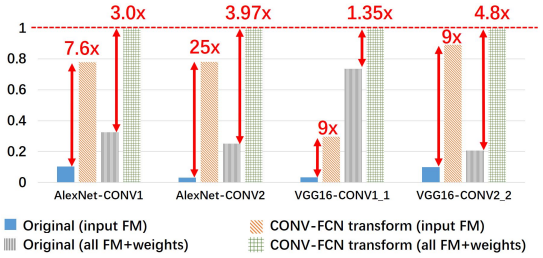


Fig. 11: Data duplication by using regular MM for CONV

1) Straightforward Mapping: For FCN shown in Figure 10(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 10(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 10(c) when batch size is 1. Prior work [24] first 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.

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 10(d) to show how we map FCN to CONV when *batch* = 3, N = 4 and M = 2. The four elements of the 1st input vector are mapped to the 1st element of each input FM, and the four 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 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 III-C.

Another way to improve the memory burst length is to increase the weight kernel size *ker* and batch *ker* elements within a single weight (or input) vector in FCN to the same weight kernel (or input FM) in CONV. Figure 10(e) depicts an example where we change *ker* from 1x1 to 1x2. Compared to Figure 10(c), two weights are grouped in one weight kernel, and two input FMs are grouped into one input FM. Accordingly, stride size changes with *ker* to 1x2.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

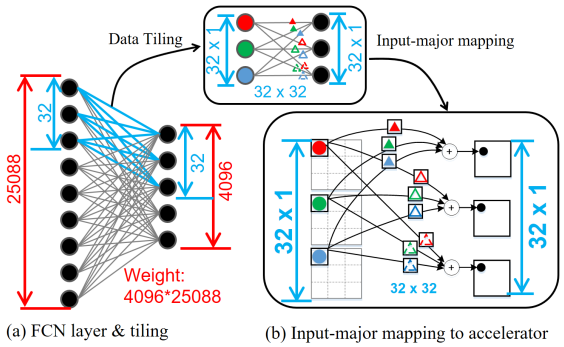


Fig. 12: Input-major mapping

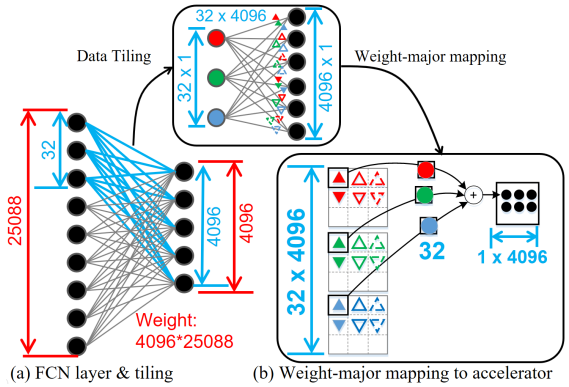


Fig. 13: Weight-major mapping

Table V *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) 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 FCN map to input FMs of CONV. As shown in Figure 10(f), 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 10(g). Table V *column FCN-Weight* lists the parameters for weight-major mapping from FCN to CONV.

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 V. 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 V.

TABLE V: 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

V. Design Space Exploration

In this section we discuss how to find the optimal solution of mapping a CNN/DNN onto our accelerator architecture. In Subsection V-A, we first use one concrete example to give readers a sense of the differences of the two mapping methods on their memory access features; and Subsection V-B gives formal formulations. In system performance, computation capability and memory access are two dominating factors to final achievable performance. We propose to use roofline models to accurately formulate the performance. In addition, as described in Fig. 9, DRAM's effective bandwidth is sensitive to access patterns. We further take DRAM bandwidth features in our formulations. In Subsection V-B and Subsection V-C, we present our systematic methods of performance analysis and design space exploration.

A. A Case Study

We use the real case of a fully connected layer from VGG16 model (FCN 1) in our case study. It has an input vector of 25,088 and an output vector of 4,096. We study the differences of two mapping methods to an accelerator with a hardware configuration of $< Tm,\ Tn,\ Tr \times Tc,\ KernelSize > = < 32,\ 32,\ 4096,\ 3 >$. In order to simplify the explanation, let's first discuss the mapping of Fig. 10(c) and of Fig. 10(f) in this subsection. More complicated situations will be discussed in Section V-B with mathematical formulations.

Figure 12(a) shows the original fully connected layer with "25,088" inputs, "4,096" outputs and "$25,088 \times 4,096$" weights. According to the input-major mapping method described in Section IV, the corresponding tiling method is 32×32, which is shown as those bold connections in Figure 12(a). Figure 12(b) shows the input/weight/output accelerator buffers and the tiled FCN layer's mapping into corresponding buffers. So the total number of memory accesses (bursts) to the input vector is $(25,088 \times 1) \div (32 \times 1) = 784$; the total number of memory accesses to the weights is $(25,088 \times 4,096) \div (32 \times 32 \times 1 \times 1) = 100,352$; the total number of memory accesses to the output vector is $(4,096 \times 1) \div (32 \times 1) = 128$. Table III summarizes the total number of memory accesses and the burst length in each memory access. Similarly, Figure 13 presents the weight-major mapping, where the tile size for the input buffer can be much larger (as discussed in Section IV). Table IV shows its corresponding data.

By comparing Table III and Table IV, we can see that the weight-major mapping has significantly less numbers of memory accesses and longer burst lengths than the input-major mapping in this case study.

B. Analytical Comparison of Two Mapping Methods

In this subsection, we give a formulation of memory access patterns by considering workload size and platform constraints. We denote the hardware configuration of our accelerator as "number of $\langle input, output, weight \rangle$ buffer = $\langle Tn, Tm, Tn \times Tm \rangle$" and "size of each buffer = $\langle Tr \times Tc, Tr \times Tc, K \times K \rangle$," which are exactly the following notations in Figure 6.

TABLE VI: Number of DRAM accesses

	Uniformed	Input-major	Weight-major
Input	$\lceil \frac{N}{Tn} \rceil \lceil \frac{R_i \cdot C_i}{Tr \cdot Tc} \rceil$	$\lceil \frac{N_{fcn}/ker}{Tn} \rceil$	$\lceil \frac{N_{fcn}/ker}{Tn} \rceil$
Weight	$\lceil \frac{N}{Tn} \rceil \lceil \frac{M}{Tm} \rceil$	$\lceil \frac{N_{fcn}/ker}{Tn} \rceil \lceil \frac{M_{fcn}}{Tm} \rceil$	$\lceil \frac{N_{fcn}/ker}{Tn} \rceil \lceil \frac{batch}{Tm} \rceil$
Output	$\lceil \frac{M}{Tm} \rceil \lceil \frac{R_O \cdot C_O}{Tr \cdot Tc} \rceil$	$\lceil \frac{M_{fcn}}{Tm} \rceil \lceil \frac{batch}{Tr \cdot Tc} \rceil$	$\lceil \frac{batch}{Tm} \rceil \lceil \frac{M_{fcn}}{Tr \cdot Tc} \rceil$

Given the uniformed representation in Table V, the number of memory accesses can be correspondingly calculated as shown in Table VI. In this table, M, N, R, C, K are following notations from Table V's collumn 2 (uniformed). When considering input-major mapping's and weight-mapping's concrete memory access behavior, we simply replace uniformed notations with the FCN-input or FCN-weight in Table V.

The remaining part of Table VI summarizes input-major and weight-major mapping of memory access. Their major differences are in their "weight" and "output" DRAM access. For "weight," input-major and weight-major mapping methods of DRAM accesses are $\lceil \frac{N_{fcn}/ker}{Tn} \rceil \lceil \frac{M_{fcn}}{Tm} \rceil$ and $\lceil \frac{N_{fcn}/ker}{Tn} \rceil \lceil \frac{batch}{Tm} \rceil$ respectively. The two formulations are almost the same except for "M_{fcn}" and "batch." Real-life network configuration's "M_{fcn}" is usually in a scale of thousands and Tm is in tens (it is constrained by DSP and BRAM resources, for example "$\langle Tm, Tn \rangle = \langle 32, 32 \rangle$" uses 1024 multiplication and accumulation operators), while the tunable parameter "batch" is smaller or equal to Tm. So $\lceil \frac{M_{fcn}}{Tm} \rceil$ would be significantly larger than $\lceil \frac{batch}{Tm} \rceil$. For the output's DRAM transfer, the considering denominator is "$Tr \cdot Tc$" which is for feature maps and usually very large. At our setting, "$Tr \cdot Tc$" is $226 \times 30 = 6780$. With similar deductions, $\lceil \frac{M_{fcn}}{Tm} \rceil \lceil \frac{batch}{Tr \cdot Tc} \rceil$ would also be significantly larger than $\lceil \frac{batch}{Tm} \rceil \lceil \frac{M_{fcn}}{Tr \cdot Tc} \rceil$.

Thus, given an accelerator information $\langle Tm, Tn, Tr, Tc \rangle$ and FCN workload configuration $\langle N_{fcn}, M_{fcn}, ker, batch \rangle$, we are able to calculate all DRAM traffic following formulations in Table VI. With the above formulations, we estimate the attainable performance by jointly considering both computation capability and bandwidth performance in the next subsection.

C. Revised Roofline Model for Caffeine

1) Original Roofline Model: The roofline model [38] 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. Eq. 2 formulates the attainable throughput of an application on a specific hardware platform. Floating-point performance (GFLOPS) is used as the metric of throughput. The actual floating-point performance of an application kernel can be no higher than the minimum value of two terms. The first term describes the peak floating-point throughput provided by all available computation resources in the system, or computational roof. Operations per DRAM traffic, or the computation-to-communication (CTC) ratio, feature the DRAM traffic needed by a kernel in a specific system implementation. The second term bounds the maximum floating-point performance that the memory system can support for a given computation to communication ratio.

$$AttainablePerf. = min \begin{cases} Computational\ Roof \\ CTC\ Ratio \times BW \end{cases} \quad (2)$$

Previous work [13] uses the roofline model to optimize the FPGA accelerator design. However, the original roofline model used in [13] ignores the fact that input/output/weight arrays have different data volumes in each tile. According to Figure 9, different burst lengths and access patterns will result in different effective bandwidths. Thus, different designs have different final bandwidth rooflines, which makes the original roofline-based method's prediction for bandwidth-intensive applications extremely inaccurate, like fully connected layers. 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 corresponding size of input/output/weight data tile, and α denotes the number of times of corresponding data transfer for input/output/weight data.

$$DRAM\_Access = \sum_i^{in,weight,out} \alpha_i \times \beta_i \quad (3)$$

In fact, Equation 3 does not accurately model the total DRAM traffic. For example, as shown in Figure 9, the effective bandwidth on 1KB burst DRAM access is only 1GB/s—10x lower than the maximum effective bandwidth of 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. Therefore, we would like to multiply a normalization factor of 10x on the original DRAM access number to approach the accurate effective DRAM traffic.

2) Revised Roofline Model for Caffeine: In general, we propose to normalize the DRAM traffic of input/output/weight accesses to the maximum effective bandwidth with a normalization factor γ.

$$DRAM\_Access = \sum_i^{in,weight,out} \gamma_i \times \alpha_i \times \beta_i \quad (4)$$

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 9.

$$\gamma = max\_bandwidth / f(\beta) \quad (5)$$

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 $\langle T_i,\ T_o,\ T_r,\ T_c \rangle$, as described in Section III-A, we can determine the '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:

$$\begin{aligned} Comput.\ Perf. &= \frac{total\ computation\ operations}{execution\ cycles} \\ &= \frac{2 \cdot N \cdot M \cdot R_o \cdot C_o \cdot K_1 \cdot K_2}{\lceil N/T_i \rceil \cdot \lceil M/T_o \rceil \cdot R_o \cdot C_o \cdot K_1 \cdot K_2} \end{aligned} \quad (6)$$

Our revised CTC ratio is given by:

$$\begin{aligned} CTC\ ratio &= \frac{total\ computation\ operations}{total\ DRAM\ traffic} \\ &= \frac{2 \cdot N \cdot M \cdot R_o \cdot C_o \cdot K_1 \cdot K_2}{\gamma_{in} \cdot \alpha_{in} \cdot \beta_{in} + \gamma_{wght} \cdot \alpha_{wght} \cdot \beta_{wght} + \gamma_{out} \cdot \alpha_{out} \cdot \beta_{out}} \end{aligned} \quad (7)$$

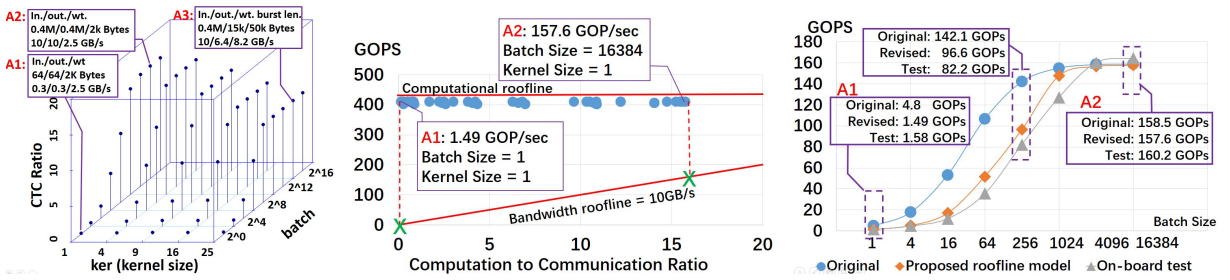
Given above revised roofline mode, we can have a design space exploration to find the optimal method mapping from the uniformed representation to our hardware accelerator.

D. 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.

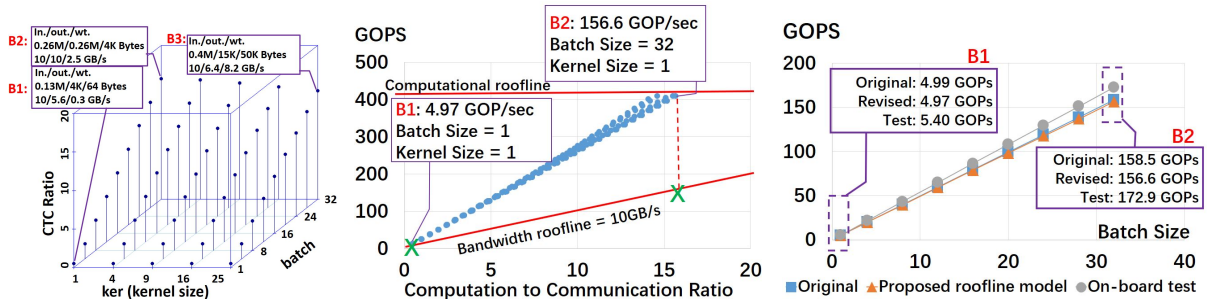
We use the VGG16 model's FCN layer 1 as an example; it has an input of 25,088 (N_{fcn}) neurons and output of 4,096 (M_{fcn}) neurons, whose notations follow Table V. $Batch$ and ker are tunable parameters for mapping FCN to the uniformed representation as described in Section IV. We use the hardware configuration from the Kintex Ultrascale KU060 platform and set hardware definable parameters as $\langle T_o, T_i, T_r \cdot T_c, T_{K1} \cdot T_{K2} \rangle = \langle 32, 32, 6272, 25 \rangle$. We choose our tile sizes based on the guidance of [13] to maximize the FPGA resource utilization. Users can configure their own tile sizes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



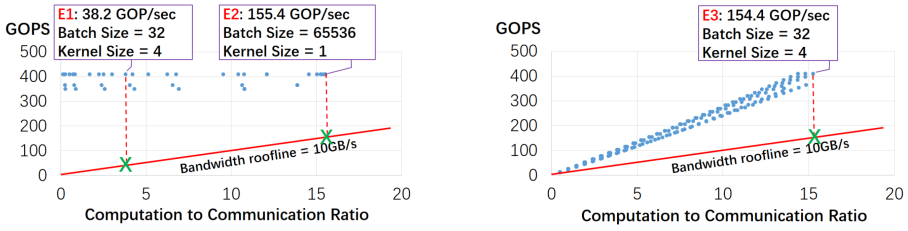
(a) Design space in CTC ratio *(b)* Design space in revised roofline model *(c)* Comparison of original, revised roofline models and on-board test results

Fig. 14: Design space exploration for FCN **input-major** mapping under various batch and kernel sizes



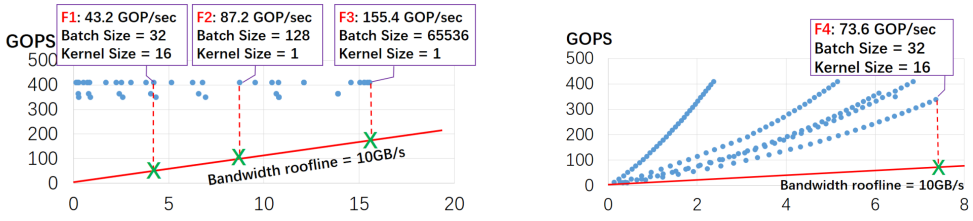
(a) Design space in CTC ratio *(b)* Design space in revised roofline model *(c)* Comparison of original, revised roofline models and on-board test results

Fig. 15: Design space exploration for FCN **weight-major** mapping under various batch and kernel sizes



(a) Input-major *(b)* Weight-major

Fig. 16: Design space exploration for hidden layers in [39]



(a) Input-major *(b)* Weight-major

Fig. 17: Design space exploration for bottleneck layers in [40]

1) FCN Input-Major Mapping: Figure 14(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 V, 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 V) decreases. In the input-major mapping, the maximum CTC ratio is achieved with a parameter $\langle batch, ker \rangle = \langle 16384, 1 \rangle$.

Figure 14(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 14(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 roofline model is much more accurate than the original one, and our estimated performance is very close to that of the on-board test.

2) FCN Weight-Major Mapping: Figure 15(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 IV-B3, $batch$ 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 15(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 14(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 IV. 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 over input-major mapping in small batch sizes, in the remainder of this paper we will use weight-major mapping for the FCN layer with the best design point.

E. Design Space Exploration on Speech Applications

Previous sections are based on convolutional neural networks, which are mainly for computer vision tasks. However, in many other areas such as speech and auto-encoder, fully connected neural network is also a major type of workload, such as networks presented on work [39][40][41][42][43][44].

Figure 16 presents a design space of a hidden layer in work [39], which has a very typical shape like other FCN workloads.

Figure 17 presents a design space of a bottleneck network, which is also frequently used in prior work[40][41][42]. Significantly less number of neurons is bottleneck layer's major difference to typical networks. Compared to regular NN layers with 2048 or more neurons, bottleneck layers usually have much less neurons, such as 20 to 40 neurons in work[40]. This will greatly influences CTC ratios. As is presented in 17(b), solution 'F4' has the highest performance in weight-major mapping method. On the same configuration ($batch\_size = 32$, $kernel\_size = 16$), weight-major mapping method wins input-major mapping. However, the highest input-major mapping solution 'F3' achieves nearly 155 GPOS, which is almost the double of that of solution 'F4'. Actually, input-major mapping wins when $batch\_size$ is larger than 128.

Under real service scenarios, there is a trade-off between low latency and high-throughput when users use small networks such as bottleneck NN. When latency is more important, we recommend weight-major mapping which achieve higher performance on small batches. Otherwise we recommend, input-major mapping for throughput. Since the choice depends on real scenarios, we left the adventure for users.

VI. From High-level Network Description to Specialized CNN Accelerator

Previous sections presented our hardware/software co-designed approach to accelerate deep learning inference with specialized hardware. However, programming hardware for non-experts is usually very difficult. Therefore, we propose an automation flow to apply our proposed optimizations discussed in those sections to compile the high-level network descriptions directly into the FPGA-based specialized hardware accelerator.

As summarized in Figure 18, our automation flow has two co-operating sides: 1) software automation, which provides a compiler to map the high-level network definitions to customized instructions for our specialized hardware; and 2) hardware automation, which is responsible for generating a new FPGA accelerator bitstream.

A. Software Automation

With the proposed software-definable accelerator design, we implement an automated flow to bridge the neural network oriented high-level domain-specific language to our customized accelerator design. Figure 18 presents the automation flow from Caffe standard inputs; these are defined in prototxt and caffemodel files in our hardware-optimized model, which includes all of the accelerator instructions (network definitions), DRAM space allocations and accelerator-specific weight data reorganizations. Overall, the key steps of our automation flow include:

1. **Network parser: network model parser and compilation.** We first parse the structure of CNN's CONV/ReLU/POOL/FCN layers from Caffe's network definition file, which is described in prototxt file, to a structured DAG-based data type to describe CNN's data flow. In addition, we read in the original CNN layers' weights and biases stored in Caffe's caffemodel file. This is the only part of our automation flow that is specific to Caffe; all other parts can be reused in other frameworks.
2. **CNN representation transformation.** In the next step we transform FCN in the CNN DAG to a convolution MM format with roofline-based optimization techniques (as described in Section IV and Section V). After the transformation, we generate accelerator-customized instructions to describe the whole CNN for the FPGA accelerator.
3. **Optimizer: weights transformation.** In this step we prepare the CNN layers' weights and biases and transfer them into a format which is specifically optimized for our customized accelerator, as described in Section III-A. This transformation includes static FPGA DRAM space allocation, weights and biases reorganization, and floating-point to fixed-point format transformation when the accelerator is defined as fixed-point by the user.

The above transformed layer definitions and weights and biases data will be generated for a new CNN once and written into FPGA DRAM through the PCIe interface. It will be reused for all following input images, and there will be no further weights or instructions communication. For each input image, the FPGA accelerator will start from reading the first CNN layer instructions stored in FPGA DRAM and stop to reach CPU until the last layer instructions are finished.

B. Hardware Automation

In the analysis of CNN's computation model in Section III-A, we discussed an application-specific hardware design with a series of computation and memory optimization techniques. Our hardware

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2017.2785257, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

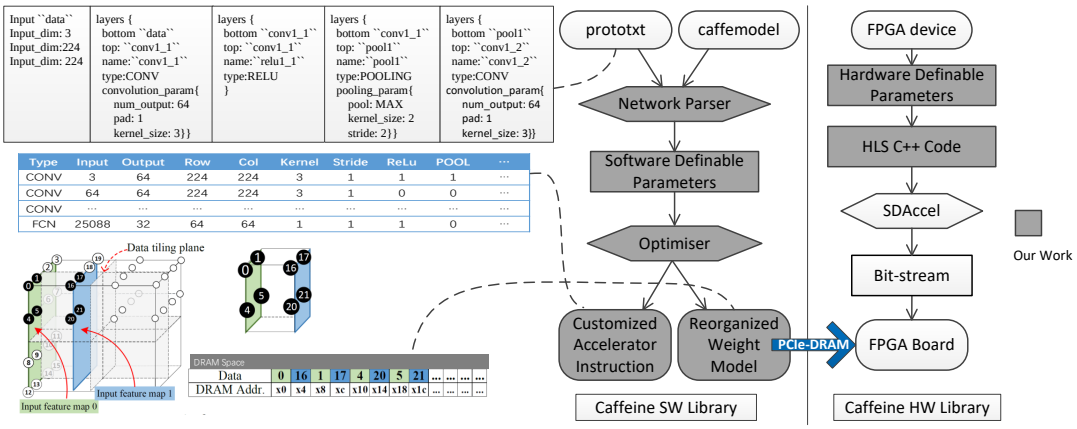


Fig. 18: Automation flow from high-level defined networks (Caffe) to hardware optimized accelerator

automation plan is to build an easy-to-use tool with such optimizations for users to customize the hardware design for their own FPGA devices.

The key required information is the number of DSP resources, on-chip storage capacity, and external memory bandwidth provided by the platform; these are the constraints to the performance of the accelerators. The output is a set of hardware-definable parameters which have been depicted in Figure 4. With a highly structured hardware template, we use high-level synthesis to generate the customized RTL as well as device-specific bitstream with Xilinx's SDAccel tool. The optimized microarchitecture proposed in Section III-B ensures its scalability to larger devices to overcome the difficulties in placement and routing.

C. Caffe-Caffeine Integration

As a case study, we integrate Caffeine with the industry-standard Caffe deep learning framework [12]. Note that Caffeine can also be integrated into other frameworks like Torch [34] and TensorFlow [35]. Figure 19 (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. As discussed earlier, the only part that is Caffe-specific 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 V, and then reorders weights for bandwidth optimization as illustrated in Section III-C. 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, it will accumulate multiple CONV outputs and execute FCN once in a batch; in single mode, it will execute 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-hand part of Figure 19 with a batch size of 32, where CONV layers dominate the entire execution again.

VII. Caffeine Results

A. 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). We

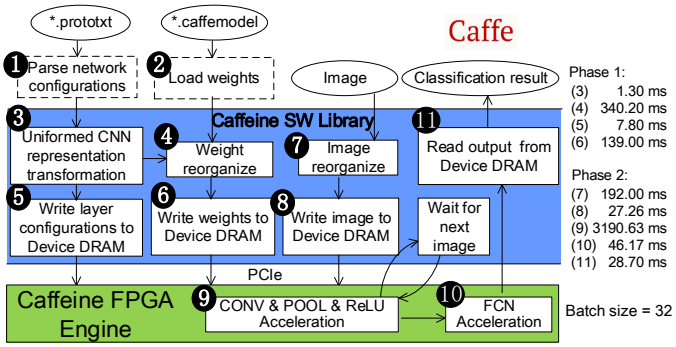


Fig. 19: Caffe-Caffeine integration

use a NVIDIA GPU GTX1080 in our experiments. OpenBLAS and cuDNN 8.0 libraries are used for the CPU and GPU implementations [12]. In the following experiments, cuDNN is set to CUDNN_CONVOLUTION_FWD_ALGO_DIRECT mode, which the library is optimized on the original 6-loops shown in Figure 3.

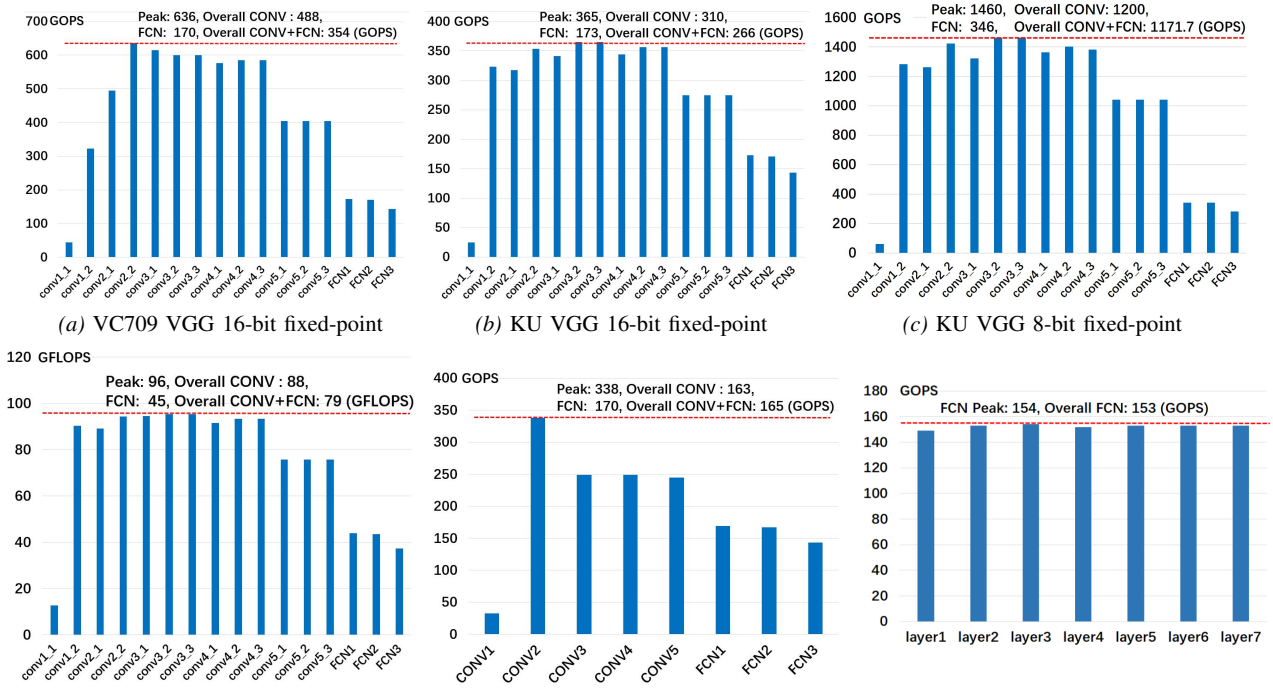
FPGA setup. The main FPGA platform we use is the Xilinx KU3 board with a Kintex Ultrascale KU060 (20nm) and a 8GB DDR3 DRAM, where SDAccel 2015.3 is used to synthesize the bitstream. To demonstrate the portability of our hardware-definable architecture, we also extend our design to the VC709 (Virtex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

B. Caffeine Results on Multiple FPGAs

To demonstrate the flexibility of Caffeine, we evaluate Caffeine using 1) two FPGA platforms, KU060 and VC709, 2) three data types, 32-bit floating-point, 16-bit and 8-bit fixed-point, and 3) two network models, AlexNet and VGG16, as shown in Figure 20.

First, Figure 20(a) and Figure 20(b) present the VGG16 performance for 16-bit fixed-point 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's 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 it is mainly bounded by bandwidth.

Second, Figure 20(b), Figure 20(c), and Figure 20(d) present the differences between the 16-bit fixed-point, 8-bit fixed-point, and 32-bit floating-point on KU060. Both CONV and FCN layers show a drastic increase in performance from 32-bit floating-point to 16-bit fixed-point. For CONV layers, fixed-point saves computation



(a) VC709 VGG 16-bit fixed-point *(b)* KU VGG 16-bit fixed-point *(c)* KU VGG 8-bit fixed-point

(d) KU VGG 32-bit floating-point *(e)* KU AlexNet 16-bit fixed-point *(f)* KU Speech FCN 16-bit fixed-point

Fig. 20: Caffeine results on multiple FPGA boards for different CNN models and data types

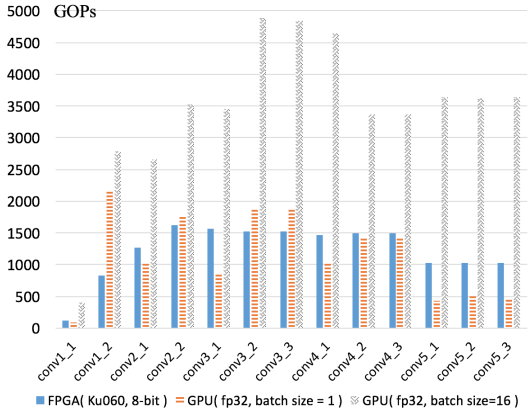


Fig. 21: GPU vs. FPGA performance

resources and thus enables more parallelism. For FCN layers, fixed-point saves bandwidth because of its fewer bits. The KU060 board with 8-bit operation can achieve as high as 1.46 TOPS peak performance for the CONV layer.

Third, Figure 20(b) and Figure 20(e) 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).

Fourth, experimental results show that our results are quite near FPGA's peak performance. For the KU060 FPGA case in Figure 20(b), the theoretical peak performance with 1024 DSPs on a 16-bit fixed-point accelerator is "$1,024 \times 2 \times 0.2GHz = 409.6\ GOPS$," while our attainable end-to-end test is 365 GOPS of peak performance. For KU060 FPGA with single-precision float in Figure 20(d), theoretical peak performance is "100 $GFLOPS$," while our evaluation peak performance is 96 GFLOPS.

Fifth, Figure 20(f) shows experimental results on the fully connected network for speech[39]. With our approach, it achieves nearly 150 GOPS performance.

TABLE VII: Comparison with other FPGA work

	Zhang[13]	Qiu[24]	Suda[23]	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

C. Comparison with Prior FPGA Work

We compare our accelerator design to three state-of-the-art studies in Table VII. 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 speed-up over previous work. In addition, very-low bit (binarized) network technique [27] is orthogonal to our work.

D. End-to-End Comparison with CPUs and GPUs

We conduct an end-to-end comparison between Caffe-Caffeine integration with existing optimized CPU and GPU solutions [12] for VGG16 in Table VIII. For fair comparison, we use giga operations per second (GOPS) as the standard metric. With on-board (KU060) testing, our integration using 8-bit fixed-point operations demonstrates an end-to-end performance of 29x speed-up and 150x energy efficiency over 12-core CPU, and 5.7x and 2x energy efficiency over batch = 1 and batch = 16 cuDNN implementations respectively. Figure 21 shows detailed layer-wise performance comparison between 8-bit fixed Ku060 FPGA implementation and GTX1080 GPU cuDNN. Our FPGA implementation has approximately similar performance to GPU when batch = 1. But batch = 16 GPU implementation has much higher performance (lower energy efficiency).

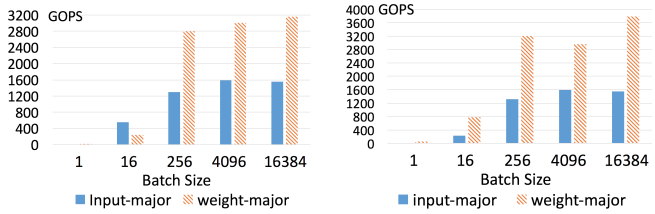
TABLE VIII: End-to-end comparison with CPU/GPU platforms

Platforms	CPU	GPU		CPU+FPGA		
Device	E5-2609	GTX1080		KU060	KU060	VX 690t
Precision	float	float	float	fix16	fix8	fix16
Technology	22nm	16nm	16nm	20nm	20nm	28nm
Freq.(GHz)	1.9	2.1	2.1	0.2	0.2	0.15
Power(Watt)	150	180	180	25	25	26
Batch Size	1	16	1	1	1	1
Latency/img.(ms)	733.7	8.13	23.5	101.15	25.3	65.13
Speedup	1x	90x	31.2x	7.3x	29x	9.7x
J per image	110	1.46	4.23	2.5	0.73	1.69
Energy Efficiency	1x	75x	26x	43.5x	150x	65x

Finally, Table IX 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. Note that for the 8-bit fixed-point implementation, it is more resource efficient and mainly uses the LUT resources. Caffeine on the KU060 board runs at a frequency of 200MHz, and on VC709 it runs at a frequency of 150MHz.

TABLE IX: FPGA resource utilization of Caffeine

	DSP	BRAM	LUT	FF	Freq
VC fix-16	2833(78%)	1248(42%)	3E5(81%)	3E5(36%)	150MHz
KU fix-16	1058 (38%)	782(36%)	1E5(31%)	8E4(11%)	200MHz
KU fix-8	116(4%)	784(36%)	2E5(60%)	1.4E5(20%)	200MHz
KU float	1314(47%)	798(36%)	1.5E5(46%)	2E5(26%)	200MHz



(a) kernel size = 1*1 *(b)* kernel size = 4*4

Fig. 22: GPU impl. of input- and weight-major mappings

E. Input-major/Weight-major Mapping on GPUs

We further verify our idea on GPU implementations. We took one optimized implementation on the original 6-loops as shown in Figure 3 from cuDNN library. We transform the VGG-16 FCN-2 layer to a convolutional layer using both input-major and weight-major mappings. Figure 22 shows that for most of the cases under 1*1 and 4*4 kernel sizes, weight-major mapping outperforms input-major mapping.

F. Comparison with TPUs

Google's Tensor Processing Unit [44] cites work [13] and argues that their systolic micro-architecture design is more friendly for frequency tunning. In this work, we also improves and use systolic design. However, TPU's performance on MLP for speech are greatly degraded because of strict bandwidth constraints. Our proposal of input-major/weight-major mapping in this paper can be helpful for TPU to optimize the computation and communication ratio and thus improve overall performance.

VIII. 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 provide an automation flow to integrate 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 1,460 GOPS on a KU060 board with 8-bit fixed-point operations, and more than 100x speed-up on fully connected layers over prior FPGA accelerators. Our Caffe integration achieved 29x and 150x performance and energy gains over a 12-core CPU, and 5.7x 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 (CDSC) industrial sponsors, including Fujitsu Labs, Huawei, Intel, Mentor Graphics, and NEC, and the NSF China under award No.61572045. The authors would also like to thank the UCLA/PKU Joint Research Institute, Chinese Scholarship Council, and AsiaInfo Inc. for their support of our research.

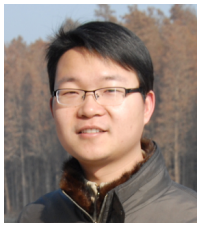
References

[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. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," 2016.
[16] C. Farabet *et al.*, "Cnp: An fpga-based processor for convolutional networks," in *FPL*. IEEE, 2009, pp. 32–37.
[17] 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.
[18] D. Aysegul *et al.*, "Accelerating deep neural networks on mobile processor with embedded programmable logic," in *NIPS*. IEEE, 2013.
[19] S. Cadambi *et al.*, "A programmable parallel accelerator for learning and classification," in *PACT*. ACM, 2010, pp. 273–284.

[20] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *ASAP*. IEEE, 2009, pp. 53–60.
[21] M. Peemen *et al.*, "Memory-centric accelerator design for convolutional neural networks," in *ICCD*. IEEE, 2013, pp. 13–19.
[22] K. Ovtcharov *et al.*, "Accelerating deep convolutional neural networks using specialized hardware," February 2015.
[23] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*. ACM, 2016, pp. 16–25.
[24] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*. ACM, 2016, pp. 26–35.
[25] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 29:1–29:6.
[26] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 45–54.
[27] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 15–24.
[28] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network." in *FPGA*, 2017, pp. 25–34.
[29] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 35–44.
[30] Y.-k. Choi *et al.*, "A quantitative analysis on microarchitectures of modern cpu-fpga platforms," in *DAC 2016*, pp. 109:1–109:6.
[31] J. Bergstra *et al.*, "Theano: a cpu and gpu math expression compiler," in *SciPy*, vol. 4, 2010, p. 3.
[32] V. D. Suite, "Ultrascale architecture fpgas memory interface solutions v7.0," Technical report, Xilinx, 04 2015, Tech. Rep., 2015.
[33] S. Mittal, "A survey of techniques for managing and leveraging caches in gpus," *Journal of Circuits, Systems, and Computers*, vol. 23, no. 08, 2014.
[34] "Torch7," http://torch.ch.
[35] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[36] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *Field-Programmable Technology (FPT), 2011 International Conference on*. IEEE, 2011, pp. 1–8.
[37] W. Zuo *et al.*, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *FPGA*. ACM, 2013, pp. 9–18.
[38] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," *CACM*, vol. 52, no. 4, pp. 65–76, 2009.
[39] Z.-J. Yan, Q. Huo, and J. Xu, "A scalable approach to using dnn-derived features in gmm-hmm based acoustic modeling for lvcsr." in *Interspeech*, 2013, pp. 104–108.
[40] F. Grézl, M. Karafiát, S. Kontár, and J. Cernocky, "Probabilistic and bottle-neck features for lvcsr of meetings," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 4. IEEE, 2007, pp. IV–757.
[41] J. Gehring, Y. Miao, F. Metze, and A. Waibel, "Extracting deep bottleneck features using stacked auto-encoders," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 3377–3381.
[42] D. Yu and M. L. Seltzer, "Improved bottleneck features using pretrained deep neural networks," in *Twelfth Annual Conference of the International Speech Communication Association*, 2011.
[43] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.
[44] N. P. Jouppi, C. Young, N. Patil, and D. e. a. Patterson, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12.

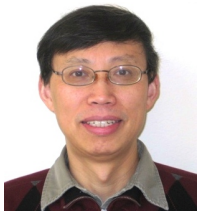
Chen Zhang received his B.S. degree in Electronic Engineering from the University of Electronic and Science of Technology of China in 2012 and Ph. D. degree in Computer Science Department from Peking University in 2017. He is now an associate researcher (II) at Microsoft Research Asia. His major research interests are high performance and energy-efficient computer architectures and systems in deep learning. He is a member of IEEE and ACM.

Guangyu Sun is an associate professor of Center for Energy-efficient Computing and Applications (CECA) at Peking University. He received his B.S. and M.S degrees from Tsinghua University, Beijing, in 2003 and 2006, respectively. He received his Ph.D. degree in Computer Science from the Pennsylvania State University in 2011. His research interests include computer architecture, electronic design automation, and acceleration system for modern applications. He is now serving as an AE of ACM JETC and TECS. He is a member of IEEE, ACM, and CCF.



Zhenman Fang recently joined Xilinx after a 3-year postdoc at UCLA. His research lies at the intersection of heterogeneous and energy-efficient computer architectures, big data workloads and systems, and system-level design automation. He has a PhD in computer science from Fudan University, China. He is a member of the ACM and IEEE.

Peipei Zhou received her B.S. degree in Electrical Engineer from Southeast University, Chien-Shiung Wu Honor College in 2012, and M.S. degree in Electrical Engineer from the University of California Los Angeles in 2014. Currently, she is a Ph.D. student at the UCLA Computer Science Department, under supervision of Professor Jason Cong. Her research interests include parallel/distributed architecture and programming, performance and energy model for computer architecture design.



Peichen Pan received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, IL, USA, in 1995. He is Vice President of Engineering at Falcon Computing Solutions Inc. His current research interests include system-level and high-level synthesis, and FPGA acceleration of big-data applications such as machine learning and genomic data processing. Dr. Pan received David J. Kuck Outstanding Ph.D. Thesis Award from UIUC in 1996.

Jason Cong received his B.S. degree in computer science from Peking University in 1985, his M.S. and Ph. D. degrees in computer science from the University of Illinois at Urbana-Champaign in 1987 and 1990, respectively. Currently, he is a Chancellors Professor at the Computer Science Department and the Electrical Engineering Department, of University of California, Los Angeles. He was elected to an IEEE Fellow in 2000 and ACM Fellow in 2008 and the National Academy of Engineering in 2017.