Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA

Kaiyuan Guo, *Student Member*, *IEEE*, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, *Senior Member*, *IEEE*, Huazhong Yang, *Senior Member*, *IEEE*

***Abstract*—Convolutional Neural Network (CNN) has become a successful algorithm in the region of artificial intelligence and a strong candidate for many computer vision (CV) algorithms. But the computation complexity of CNN is much higher than traditional algorithms. With the help of GPU acceleration, CNN based applications are widely deployed in servers. However, for embedded platforms, CNN-based solutions are still too complex to be applied. Various dedicated hardware designs on FPGAs have been carried out to accelerate CNNs, while few of them explore the whole design flow for both fast deployment and high power efficiency.**

In this paper, we investigate state-of-the-art CNN models and CNN based applications. Requirements on memory, computation and the flexibility of the system are summarized for mapping CNN on embedded FPGAs. Based on these requirements, we propose Angel-Eye, a programmable and flexible CNN accelerator architecture, together with data quantization strategy and compilation tool. Data quantization strategy helps reduce the bit-width down to 8-bit with negligible accuracy loss. The compilation tool maps a certain CNN model efficiently onto hardware. Evaluated on Zynq XC7Z045 platform, Angel-Eye is 6× faster and 5× better in power efficiency than peer FPGA implementation on the same platform. Applications of VGG network, pedestrian detection and face alignment are used to evaluate our design on Zynq XC7Z020. NIVIDA TK1 and TX1 platforms are used for comparison. Angel-Eye achieves similar performance and delivers up to 16× better energy efficiency.

***Index Terms*—Embedded FPGA, convolutional neural network, design flow, hardware/software co-design**

I. Introduction

Convolutional Neural Network (CNN) is one of the state-of-the-art artificial intelligence algorithms. With a large model and enough training data set, CNN generates complex features for certain tasks, which outperforms traditional handcrafted features. Thus CNNs can help achieve the top performance in regions like image classification [1] [2], object detection [3] and even stereo vision [4]. Some audio algorithms also involves CNN as one of the feature extraction steps [5].

Despite the outstanding performance, CNNs are hard to be implemented in daily applications and devices, because of its high computation complexity. Large CNN models can involve up to about 40G operations (multiplication or addition) [2] for the inference of one 224×224 image. Larger images in real applications can scale this number up. Thus CNN based applications are usually implemented as a cloud service on large servers. For personal devices, traditional CPU platforms are hardly able to handle CNN models with acceptable processing speed. For tasks like object detection where real-time processing is required, the situation is worse.

GPUs offer a high degree of parallelism and are good candidates for accelerating CNN. GPUs have been widely applied to the training and inference of CNN. The high utilization of GPU relies on large batch size, which is the number of images processed in parallel. Large batch size is not practical for real-time inference. For applications on video stream like object tracking, input images should be processed frame by frame. The latency of the result of each frame is critical to the application's performance. Using batch in video processing can greatly increase latency. In some tracking algorithms, the result of one frame affects the process of the next frame. This requires that the frames are processed one by one.

On the other hand, one can design dedicated architecture for CNNs and parallelize the CNN computation within a frame. The flexibility of FPGA makes it a good candidate for CNN acceleration. With a scalable design, we can also implement CNN accelerator on embedded FPGAs. Several designs have been proposed for CNN acceleration [6] [7] [8] but few of them discusses the overall design flow for mapping CNN onto embedded FPGAs.

Considering the high computation and storage of CNN, mapping it onto embedded FPGAs without simplification is not feasible. Recent works on CNN have shown that the data format can be compressed from 32-bit floating point to fixed point. This greatly reduces the power and area cost of the hardware. We have shown that 8-bit fixed point is enough for VGG network [8]. Han et al. [9] compressed the data to 4-bit by weight sharing. Recent work even tries 1-bit weight for classification [10].

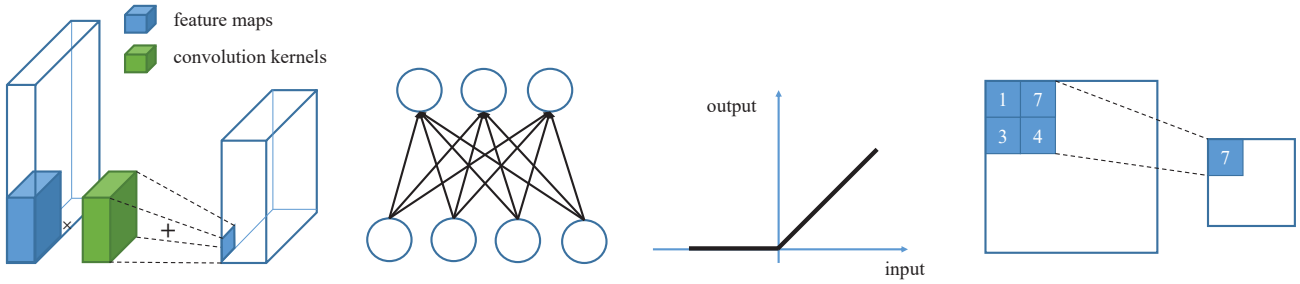
Various hardware architectures have been proposed to accelerate CNN on FPGAs. Most of the works manually map a target CNN model to hardware structure. Zhang et al. [7] explore the design space for the accelerator of AlexNet [1] and

This work was supported by 973 project 2013CB329000, and National Natural Science Foundation of China (No.61373026, 61622403), and Tsinghua University Initiative Scientific Research Program, and Joint fund of Equipment pre-Research and Ministry of Education (No. 6141A02022608).

K. Guo, J. Qiu, J. Yu, Y. Wang, H. Yang are with the Department of Electronic Engineering, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: yu-wang@tsinghua.edu.cn).

L. Sui, J. Wang, S. Yao are with Deephi Technology Co., Ltd, Beijing 100083, China (e-mail: songyao@deephi.tech).

S. Han is with the Department of Electrical Engineering, Concurrent VLSI Architecture (CVA) group, Stanford University, Palo Alto, CA, 94305, USA (e-mail: songhan@stanford.edu).



(a) Convolution layer (b) Fully Connected Layer (c) Rectified Linear Unit (d) Max-pooling Layer

Fig. 1: Typical layers in CNN: (a) Convolutional layer; (b) Fully-Connected layer (dense matrix multiplication); (c) Non-linear layer with Rectified Linear Unit; (d) Max-pooling layer with 2×2 kernel.

propose a floating-point accelerator on FPGA. [11] and [12] implement automatic design tools but are targeting a single network. This pushes the hardware performance to the extreme while sacrificing the flexibility to different networks.

With our investigation on CNN models and CNN based applications, which will be introduced in section III, we show that targeting a certain network may not be a good choice for accelerator design. Another choice is to use a flexible hardware structure and mapping different networks onto it by changing the software. We adopt this choice and design instructions such that we still provide good hardware efficiency. This can response to the changes in network topology quickly and support switching between different networks at run-time.

In this paper, we extend our previous work [8] to a complete design flow for mapping CNN onto embedded FPGA. Three parts are included in this flow:

- A data quantization strategy to compress the original network to a fixed-point form.
- A parameterized and run-time configurable hardware architecture to support various networks and fit into various platforms.
- A compiler is proposed to map a CNN model onto the hardware architecture.

Our experiments on FPGA show that the proposed design flow delivers CNN acceleration with high energy efficiency. The rest of this paper is organized as follows. Section II introduces the background of CNN. The motivation and design target is introduced in Section III. Details of the flow are shown in Section IV. We show the experimental results in Section V. Section VI reviews previous work. Section VII concludes this paper.

II. Preliminary of CNN

A CNN consists of a set of layers. As the name suggests, the most important layers in CNNs are the convolution(Conv) layers. Besides, fully connected(FC) layers, non-linearity layers, and pooling layers (down-sampling layer) are also essential in CNN.

Conv layer applies 2-d convolution with trained filters on input feature maps to extract local features. Multiple Conv layers are usually cascaded to extract high-level features. An example is shown in Figure 1 (a), where the feature maps are blue, and the 3-D Conv kernel is green. Each pixel of each output feature map is the inner product of a part of input with a 3-D convolution kernel.

FC layer applies a linear transformation on the input feature vector. It is usually used as the classifier in the final stage of a CNN. A simple FC layer with four input and three output are shown in Figure 1 (b) where each connection represents a weight of the model.

Non-linearity layer helps increase the fitting ability of neural networks. In CNN, the Rectified Linear Unit (ReLU), as shown in Figure 1 (c), is the most frequently used function [1]. Hyperbolic tangent function and sigmoid function are also adopted in various neural networks.

Pooling layer is used for down-sampling. Average pooling and max pooling are two major types of pooling layers. For a pooling layer, it outputs the maximum or average value of each sub-area in the input feature map. The pooling layer can not only reduce the feature map size and the computation for later layers, but also introduces translation invariance. A simple max pooling layer with a 2×2 kernel is shown in Figure 1 (d).

A practical CNN for face alignment is shown in Figure 2. It calculates the coordinates of 5 character points of human face given the face image, two points for eyes, two points for mouth and one point for nose. Conv layers, Pooling layers, and Non-linearity layers are interleaved to extract features. An FC layer at the end generates the coordinates of these points from extracted features. We also use this network to evaluate our hardware design.

III. Motivation

Before introducing the details of the design flow, we first investigate state-of-the-art CNN models and CNN based applications to see the required features for our design flow.

A. CNN Models

State-of-the-art CNN models differ greatly from the earlier networks in topology. **Recent work is focusing more on the design of Conv layers than on FC layers.** As in VGG network [2], 3 FC layers with more than 0.12 billion weights are used for the final classification. ResNet [14], the winner of Image-Net Large-Scale Vision Recognition Challenge (ILSVRC) 2015, implements 152 layers where only the last layer is a fully-connected layer. Shortcut structure is also

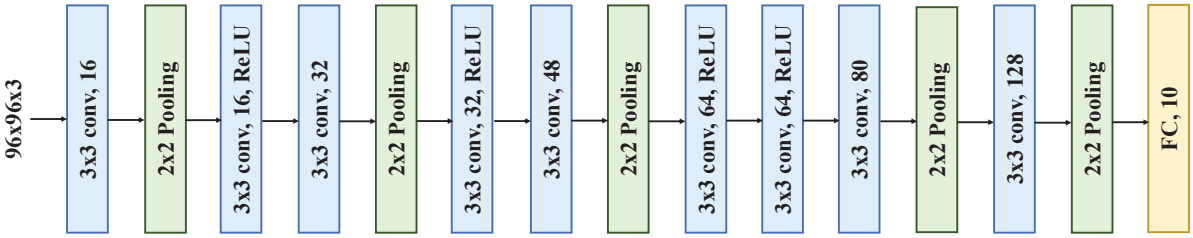


Fig. 2: A practical CNN model for face alignment. For each layer, the kernel size, output channel number and non-linearity type are given.

TABLE I: Distribution of MAC for different convolution kernel and FC layer in state-of-the-art CNN models

Model	# Computation Complexity (GMAC)						
	11*11	7*7	5*5	3*3	1*1	FC	Total
AlexNet [1]	0.105(6.55%)	0	0.929(58.00%)	0.509(31.77%)	0	0.059(3.68%)	1.6
VGG-11 [2]	0	0	0	7.49(98.37%)	0	0.124(1.63%)	7.61
VGG-16 [2]	0	0	0	15.3(99.20%)	0	0.124(0.80%)	15.5
VGG-19 [2]	0	0	0	19.5(99.37%)	0	0.124(0.63%)	19.6
SqueezeNet [13]	0	0.177(21.20%)	0	0.447(53.53%)	0.211(25.27%)	0	0.836
ResNet-34 [14]	0	0.118(3.24%)	0	3.53(96.75%)	0	0.0005(0.01%)	3.64

introduced in Conv layers to reinforce the learning ability. Networks with no FC layer are also proposed [15] [13]. One of the most successful applications of CNN is object detection. R-CNN [16] extracts proposals with traditional computer vision algorithm and gives each one a category and confidence with a CNN. Fast R-CNN [17] takes the full image as the input of a CNN and extracts proposals on the output features to reduce the redundant calculation on overlapped proposals. Recent work even uses a fully convolutional network for the complete flow [3].

Convolution kernels in CNN are also changing. **Recent CNN models prefer smaller convolution kernels than larger ones.** Early CNN designs [1] [18] adopt convolution kernels of size 11×11 for Conv layers, which are much larger than the 3×3 kernels in VGG networks [2]. The 152-layer ResNet also uses 3×3 kernels in all the layers except for the first one. SqueezeNet [13] even uses 1×1 kernels to further reduce computation complexity. Experimental results show that this kind of structure achieves comparable classification accuracy with AlexNet [1] while the parameter size is 50× fewer. Using smaller kernels in Conv layers can reduce the computation complexity while the network performance remains. Statistics on how the MAC operations distribute in state-of-the-art CNN models is shown in Table I. We can see that convolution layers, especially those with small kernel size like 3×3 are most popular in module design. So the proposed hardware adopts a 3×3 convolution kernel design to fit into most of the layers in state-of-the-art CNN models.

Table I also shows the overall computation complexity of these models. Usually, giga-level MAC is included in a CNN model. Using these models on embedded platforms is not feasible without acceleration or simplification.

Besides computation complexity, storage complexity of CNN is also high. For the CNN models listed in Table I,

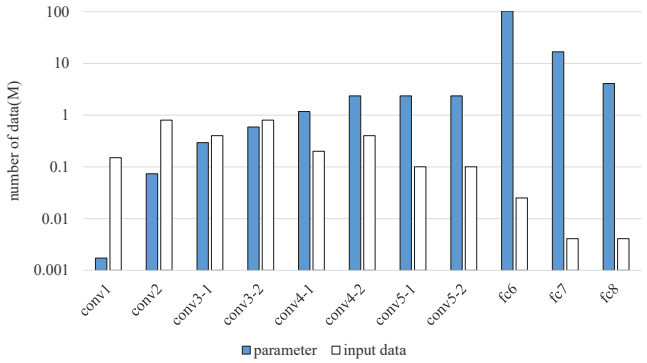


Fig. 3: Statistics on number of input data and parameters for each layer in VGG-11 model

we also investigate the size of intermediate data between different layers and the parameter of each layer. A sample statistics of the VGG-11 model is shown in Figure 3. For convolution layers, the maximum size of the feature maps or the convolution kernels of a single layer reaches MB level, which is hard to be totally cached on-chip for embedded FPGA. Thus effective memory management and data reuse strategy should be explored.

B. Hints from Application

In many applications, like object detection [3], face recognition [19] and stereo vision [4], CNN has shown its power and beats traditional algorithms where handcrafted models are used. Implementing this kind of algorithms on mobile devices will do great help to the robot or smart camera manufacturers. But in some cases, more than one network is needed in the algorithm. In [19], a cascaded CNN structure is proposed for face detection. In this algorithm, the first CNN goes over the

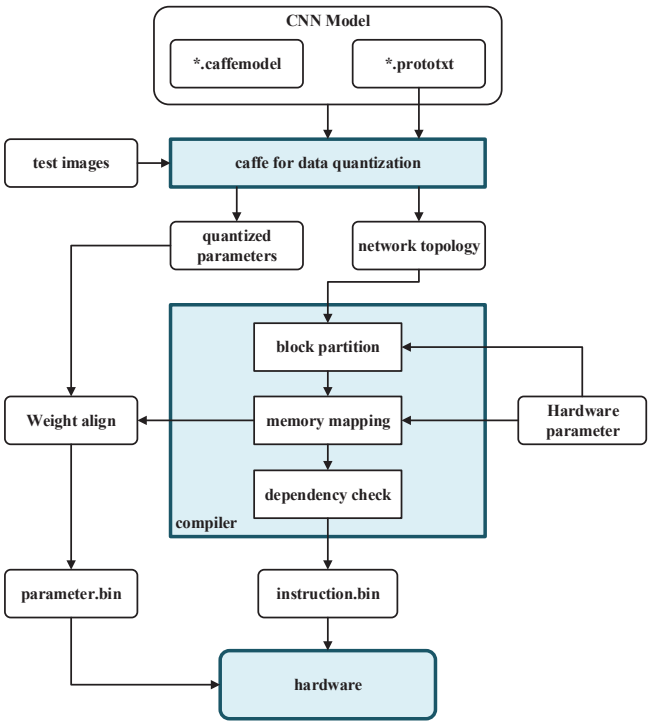


Fig. 4: Design flow from CNN model to hardware acceleration

whole image to drop useless proposals. The second CNN is applied on the preserved proposals. More proposals are dropped in this step. In this case, more than one CNN model is needed in the algorithm. The results of CNN influence the control flow of the algorithm. Simply implementing a CNN accelerator for this kind of application is not enough.

In this case, using multiple accelerators is possible but not a scalable solution if more models are involved. **So the CNN accelerator should be configurable at run-time.** As the execution of CNN can be decided by run-time results, **a host controller is needed to handle the control flow**.

IV. Flow Description

The overall structure of the design flow is shown in Figure 4. First, to deal with the high computation complexity of CNN models, data quantization is proposed to compress the data bit-width to reduce the workload. Second, to deploy the model to hardware accelerator, a compiler is proposed to automatically generate an instruction sequence to describe the process of CNN execution. Details of the three steps: block partition, memory mapping, and dependency check will be discussed in section IV-C. A hardware accelerator is proposed to support the instruction interface. To better describe the behavior of the compiler, hardware architecture will be introduced before the compiler.

A. Data Quantization

As introduced in the previous section, the high computation complexity of CNN models makes it hard to be deployed on embedded platforms. Compressing the model is a good choice.

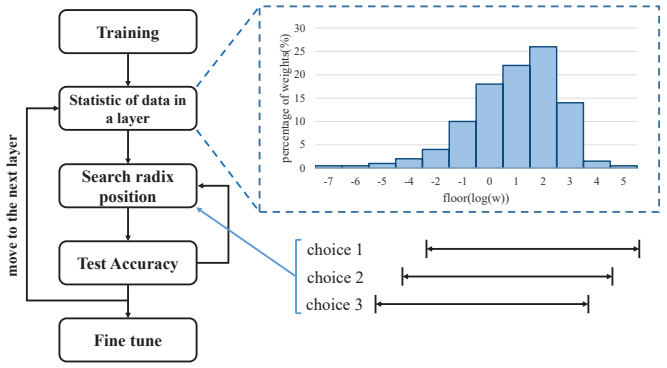


Fig. 5: Data quantization flow for CNN. We add fine-tune to the flow in [8] to further increase accuracy.

A straightforward way to compress a network is to reduce the bit-width for computing. This utilizes the flexibility of FPGA or ASIC design compared with GPU or CPU. It is also proved to be an effective way in the previous work [6] [20] [21] but limited to 16-bit or 12-bit.

Usually, a CNN is trained with 32-bit floating point data on GPU. Latest GPU can handle 16-bit floating-point format, but still complex compared with fixed-point data format. Compressing the bit-width means doing coarse data quantization. The dynamic range of data across different layers in a CNN is usually large. Thus a uniform quantization with fixed point data format for all the layers may incur great performance loss. To address this problem, we propose a quantization strategy with which the radix position of the fixed point data in each layer is chosen differently. The strategy tries to find the best radix point position in each layer given the bit-width. This is hardware friendly because only extra shifters are needed to align the data. Fixed-point adders and multipliers remain unchanged.

The quantization flow is shown in Figure 5. The network is first trained with floating point data format. Then for each layer, we first collect the statistics on the feature maps and network parameters to get a histogram of their logarithm value. This inspires how we can choose the radix point position. For each possible solution, we apply it to the network to get a fixed-point format layer and test the accuracy after quantization. Overflow and underflow may occur in this step. For the overflow data, we keep its sign and set its absolute value to the maximum. For underflow data, we set 0. Half-adjust is used to convert the floating-point data to fixed-point format. The quantization result with the best accuracy is kept. After quantization on all the layers, we apply fine tuning to further improve the accuracy. The network is converted back to floating point format to be fine tuned: the gradient, weight, activations are all floating point numbers during fine-tuning for both feed-forward and back propagation. The fine tune result is then converted to fixed-point format with the chosen positions of radix points for each layer.

Note that we use a greedy strategy by optimizing the radix position layer by layer. If we optimize all the layers together, the solution space is exponential to the number of layers, which will be too computation consuming. Our experimental

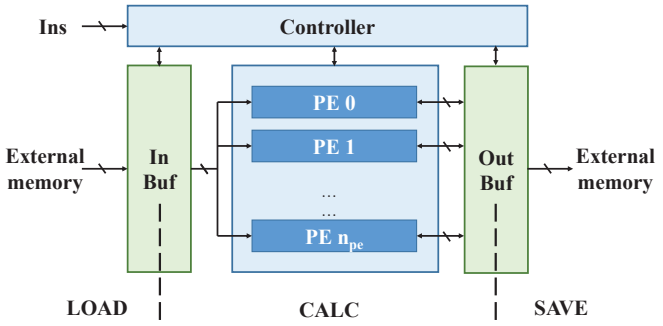


Fig. 6: Overall architecture of Angel-Eye

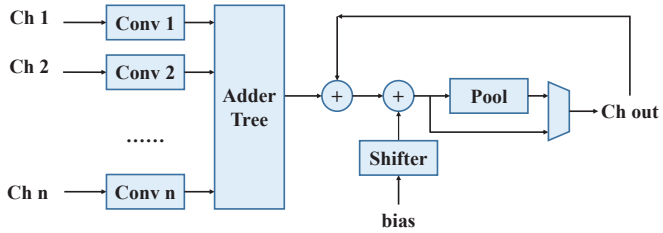


Fig. 7: Structure of a single PE

results show that this greedy strategy can simplify state-of-the-art network to 8-bit fixed point format with negligible accuracy loss.

After data quantization, all the data in the network is set to fixed-point format of the same bit-width. But the result of each layer is extended to wider bit-width after multiplication and accumulation.

B. Hardware Architecture

As discussed in section III, the CNN accelerator should be run-time configurable. Our previous work [8] is limited to VGG models. In this work, a flexible instruction interface is proposed. The calculation of CNN is described with three kinds of instructions: LOAD, SAVE and CALC, corresponding to the I/O with external memory and the convolution operation. Most of the variations of state-of-the-art CNN models are covered with this instruction set. Each instruction is 128-bit or 192-bit and contains the following fields:

- **Operation code** is used to distinguish different instructions.
- **Dependency code** sets the flags for inter-instruction dependency and helps to parallelize different kinds of instructions. This enables scheduling before instruction execution.
- **Parameter** contains specific fields for each kind of instruction. For LOAD and SAVE instructions, address and size description for the data block in external memory and on-chip memory is set. Offer the address interface of on-chip memory helps the software fully utilize the limited on-chip memory. For CALC instructions, data block address and size in on-chip memory are set. Other flags for pooling, bias, and padding are also set.

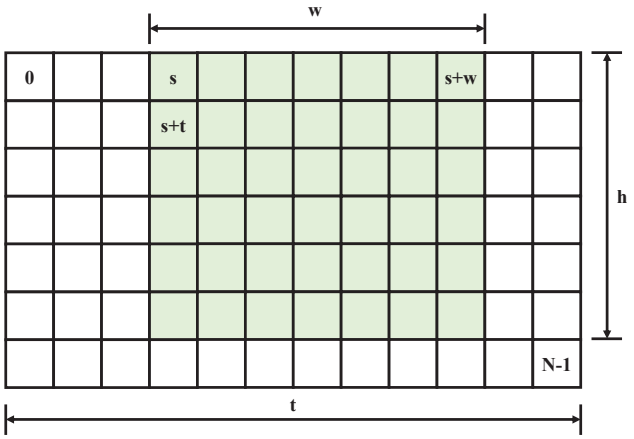


Fig. 8: A 2-D data description example. An image of width w and height h is stored in a 1-D buffer of size N at start address s with line step t. The colored blocks denote the image.

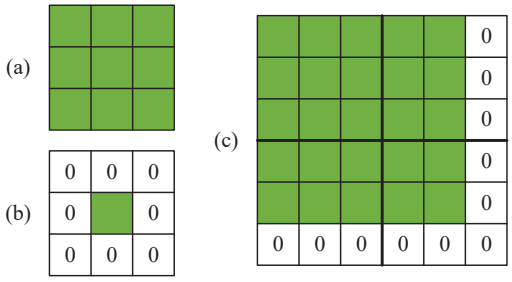


Fig. 9: Using 3×3 convolver for general convolution: (a) 3×3 kernel (b) 1×1 kernel by padding (c) 5×5 kernel by 4 3×3 kernels and padding

A hardware architecture is proposed as shown in Figure 6 to support this instruction interface. It can be divided into four parts: PE array, On-chip Buffer, External Memory and Controller.

PE Array: The PE array implements the convolution operations in CNN. Three levels of parallelism are implemented by PE array:

- **Kernel level parallelism.** Each PE consists of several convolution engines. Each convolution engine computes the inner product of the convolution kernel and a window of the image in parallel.
- **Input channel parallelism.** Different convolution engines in each PE do convolution on different input channels in parallel. The results of different input channels are added together as CNN defines.
- **Output channel parallelism.** Different PEs share the same input channels, but not the convolution kernels, to compute different output channels in parallel.

A detailed structure of a single PE is shown in Figure 7. Within each PE, different convolvers calculate 2D convolution on different input channels in parallel. As introduced in section III, in state-of-the-art CNN models, the most popular convolution kernel is of size 3×3. So we adopt the 3×3 convolution kernel in our hardware based on the line buffer

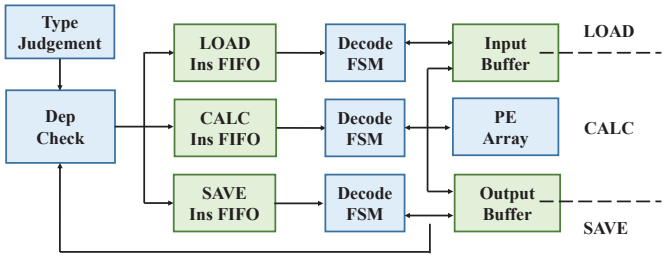


Fig. 10: Structure of Controller

design [22] This achieves the kernel level parallelism and makes good reuse of image data. Though the kernel is fixed, we are still available to support other kernel sizes as shown in Figure 9 For smaller kernels like 1×1 ones, the kernel is padded to 3×3 to be supported. For larger kernels like 5×5 ones, multiple 3×3 kernels are used to cover it. This means doing 3×3 convolution on the same image with slight deviation and add the result together.

With the help of data quantization, the multipliers and adders can be simplified to use fixed-point data with certain bit-width. To avoid data overflow, bit-width is extended for intermediate data. For our 8-bit design, 24-bit intermediate data is used. Shifters are used to align the bias with the accumulated data and cut the final result according to data quantization result for each layer.

On-chip Buffer This part separates PE Array with External Memory. This means data I/O and calculation can be done in parallel. Output buffer also offers intermediate result to PE Array if more than one round of calculation is needed for an output channel. As mentioned in section III, CNN is memory intensive. Thus we need to efficiently utilize on-chip buffer. We introduce a 2-D description interface to manage the data, which is shown in Figure 8. Each of the image in the buffer is described with the following parameters: start address, width, height, and line step. This enables that software can fully utilize the on-chip buffer for different feature map sizes. With this interface, software can also implement the ping-pong strategy on these buffer by splitting the matrix with address.

External Memory For state-of-the-art CNN and the currently available embedded platforms, On-chip Buffer is usually insufficient to cache all the parameters and data. External memory is used to save all the parameters of the network and the result of each layer. In the proposed system, external memory is also used for the communication between the CNN kernel and the host CPU. Using a shared memory for data communication has the chance of reducing abundant data transportation.

Controller This part receives, decodes and issues instructions to the other three parts. Controller monitors the work state of each part and checks if the current instruction to this part can be issued. Thus the host can send the generated instructions to Controller through a simple FIFO interface and wait for the work to finish by checking the state registers in Controller. This reduces the scheduling overhead for the host at run-time. Other tasks can be done with the host CPU when CNN is running.

Figure 10 shows the structure of this part. Parallel execution of instructions may cause data hazard. In hardware, an instruction is executed if: 1) the corresponding hardware is free and 2) the instructions it depends on have finished. Condition 1 is maintained by LOAD Ins FIFO, CALC Ins FIFO and SAVE Ins FIFO as shown in Figure 10. The instructions in the FIFOs are issued when the corresponding hardware is free. Condition 2 is maintained by checking the dependency code in Dep Check module.

C. Compiler

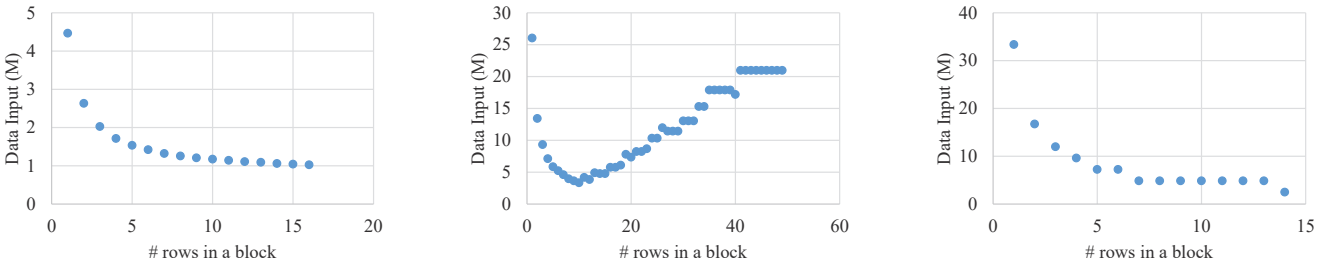
A compiler is proposed to map the network descriptor to the instructions. Optimization is done to deal with the high storage complexity of CNN. Some basic scheduling rules are followed in this compiler to fully utilize the data localization in CNN and reduce data I/O:

1. **Input channel first.** Sometimes, the input feature map needs to be cut into smaller blocks. We keep a set of loaded input feature map blocks in input buffer and generates as many output channels' intermediate results as possible. This means the convolution kernels are changing in this process. Usually, feature map is much larger than convolution kernels. So keeping the feature maps on-chip is better than keeping the convolution kernels.
2. **Output channel second.** When the feature maps are cut into blocks, we first calculate all the output blocks at the same position and then move on to the next position.
3. **No intermediate result out.** This means when the output buffer is full with intermediate results, we load a new set of input feature maps to input buffer and do accumulation on these output channels.
4. **Back and forth.** When a set of output buffer finishes the calculation, we have traversed all the input channels. The next round of traverse is done in the opposite direction. This reduces a redundant LOAD between two rounds of traverse.

Three steps are included in the compiling process:

Block partition. Since the on-chip memory is limited, especially for embedded platforms, not all the feature maps and network parameters for one layer can be cached on-chip. Thus we need to partition the calculation of one layer to fit each block into the hardware. Different partition strategies are analyzed, in order to achieve high efficiency, while almost any kind of partition can be implemented with the instruction set. The main problem of the partition is the bandwidth requirement. Reducing I/O can reduce power consumption and saves the bandwidth for other cooperative accelerators and the host in the system. To remain the data I/O burst length, we require that the feature map is cut horizontally for the row-major data format. Then the remained problem is to decide how many rows are in a single block.

Suppose a layer has M input feature maps of size $f \times f$ and N output feature maps of the same size. The convolution kernels are of size $K \times K$. The buffer size for input, output and convolution kernels are B_i, B_o and B_w. r rows are in each feature map block. Since we do not store intermediate result



(a) M=3, N=64, f=224, K=3, B_i=24K, B_o=32K, B_w=2K

(b) M=128, N=256, f=80, K=3, B_i=128K, B_o=32K, B_w=2K

(c) M=512, N=512, f=14, K=3, B_i=128K, B_o=32K, B_w=2K

Fig. 11: Examples of block partition. B_i, B_o and B_w are effective value.

to DDR, the output amount is a constant to a layer. We can generate the functions for the input amount of input feature maps and convolution kernels as D_i in equation (2) and D_w in equation (3).

$$R = \frac{B_i}{Mf} - K + 1 \tag{1}$$

$$D_i = \begin{cases} \frac{f}{r}(r+K-1)fM & r \le R \\ \frac{f}{r}\left\{[(r+K-1)fM - B_i]\frac{Nrf}{Bo} + B_i\right\} & r > R \end{cases} \tag{2}$$

$$D_w = \begin{cases} MNK^2 & MNK^2 \le B_w \\ \frac{f}{r}(MNK^2 - B_w) + B_w & MNK^2 > B_w \end{cases} \tag{3}$$

Equation (1) gives the boundary of the two branches for D_i. If r rows are in a block, we get f/r blocks of a feature map. $r + K - 1$ rows are loaded for each block considering padding and overlap between adjacent blocks.

If $r \le R$, the blocks at the same position of all the input channels can be buffered on-chip. Moving from one output channel to the next will cost no extra data exchange with external memory. So each block is loaded only once and the total amount of input is according to the first branch of equation (2). If $r > R$, extra data exchange is needed. Consider the computation for one output block, all the input blocks at the same position are needed. If the previous output block is at the same position, the input blocks can be reused. The maximum reuse size is B_i. So data input amount for each output block is $(r + K - 1)fM - B_i$, except for the first output channel. To utilize output buffer, B_o/rf output channels are grouped together. This means each group can be totally buffered on-chip. So getting the blocks at the same position of all the output channels needs Nrf/B_o rounds of calculation. This corresponds to the second branch of equation (2).

For convolution kernels, if the total amount of data is larger than weight buffer, then extra data exchange is needed when moving from the blocks at one position to the next. Similar to the input feature maps, B_w data can be reused and we get the second branch of equation (3). This is the common case for our design.

The above functions do not consider the non-divisible situations. In our compiler, a simulation is done to calculate all the input amount for each possible r. The r with the least input amount is selected. Three examples are shown in Figure 11.

As for case (a), only the first branch of D_i is satisfied. So the total input amount can be expressed as equation (4). r should be as large as possible in this case.

$$D_i + D_w = \frac{f}{r}\left[Mf(K-1) + MNK^2 - B_w\right] + const. \tag{4}$$

Case (b) is a typical layer in the middle of a CNN model where the number of channels is large and the feature maps are of middle size. The split condition R lies in the domain of r so both of the branches should be considered. For the second branch, the total input amount can be expressed as equation (5). In this case, a local minimum solution can be found.

$$D_i + D_w = (B_i - B_w + MNK^2)\frac{f}{r} + \frac{MNf^3}{B_o}r + const. \tag{5}$$

Case (c) is a typical layer at the end of a CNN model where the number of channels is large and the feature maps are small. Only the first branch in equation (2) is satisfied. So the solution is the same to case (a).

Note that B_i and B_o in case (a) are different from that in case (b) and case (c). Only three input channels are used in this layer while we have 16 input channels in hardware design. So B_i is only $3/16$ of the total input buffer size.

Memory Mapping. External memory space is allocated for the communication between host CPU and the CNN accelerator. First, input feature map memory space should be allocated. The feature maps should be in the row-major format with each channel stored continuously. Then, the memory space for the result of each layer should be allocated. The data format will be automatically handled by hardware. Only two blocks of memory are needed during the calculation of one layer, one for input and one for output. Thus the memory space for non-adjacent layer's result can overlap. The compiler supports the case if an intermediate layer's result is needed and preserves the space from rewritten by other layers.

Then, memory space for convolution kernels and bias is allocated. This space is preserved during the whole process of CNN acceleration. Usually this space is only initialized once before the first time for CNN acceleration. With the block

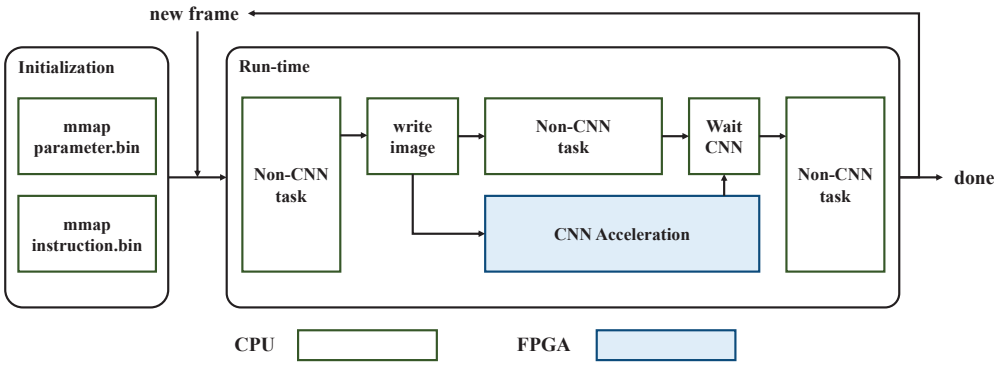


Fig. 12: Run-time work flow of the proposed system on embedded FPGA

partition result, the order of how the convolution kernels and bias are used is determined. A *parameter.bin* file filling the parameter memory space is generated according to this order.

On-chip memory is also allocated for input and output feature map blocks and also the convolution kernels according to the block partition result. After all the memory allocation, the corresponding address fields in the instruction sequence are filled.

Dependency Check. After memory mapping step, the instruction set can already finish the CNN calculation process. But data dependency check can find potential parallelism between calculation and data I/O. This step checks the data dependency among instructions and sets the flag bits in instructions to let the hardware explore the parallelism. The order of the instructions is also adjusted to make the most use of hardware parallelism.

D. Run-time Work Flow

The run-time work flow of the proposed system is shown in Figure12. In the initialization phase, the *parameter.bin* file generated by data quantization should be loaded into the memory according to the address given by compiler. Instructions should be prepared in the memory as well. At run-time, non-CNN tasks are run on the ARM core in the system. When CNN is to be called, the input image is first copied to the physical memory space allocated by the compiler, then the instructions are sent down to the accelerator. While the accelerator is working, other tasks can be executed with the host CPU. The host checks the state register of the accelerator to see if it is done. Then the algorithm goes on. Note that multiple CNN can be done within each frame while the graph is an example of one inference per frame.

V. Experiment

In this section, the proposed data quantization strategy is analyzed on different state-of-the-art CNNs. The hardware performance is then evaluated with the quantized networks.

A. Data Quantization Result

The proposed data quantization strategy is evaluated on four networks: GoogLeNet [23], VGG-16 network [2], SqueezeNet [13], and VGG-CNN-F model which is available from the model zoo in Caffe [24]. ImageNet classification dataset [25] is used for quantization and verification. 50 images are used to optimize the radix position of each layer. 5000 images are used to test the classification accuracy of the network. After data quantization, fine-tune is done on all the bit-width configurations. 32-bit floating point result is used as the baseline. Experimental results are shown in Table II.

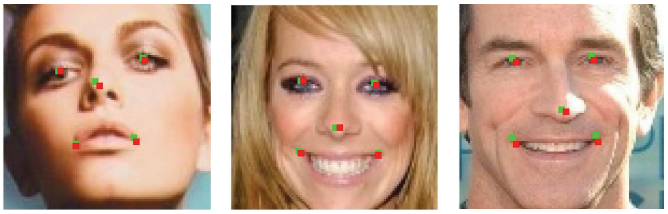


Fig. 13: Five point face alignment result. Red points: floating-point network result. Green points: 8-bit fixed point network result.

For all the networks, 16-bit data quantization brings within 1% accuracy loss on ImageNet dataset except for the fine-tune result on VGG-CNN-F. This is consistent with previous work. Going down to 8bit, VGG-16 and VGG-CNN-F model remains a similar performance as 16-bit while GoogLeNet and SqueezeNet suffer further performance loss. Until 8-bit data quantization, the performance of all the models remains relatively high. With 6-bit data quantization, all the models crash, to some extent. Thus we choose 8-bit and 16-bit in our hardware implementation.

Fine tune is also done on all the models. It works well on VGG-16 and VGG-CNN-F model but is not helpful to GoogLeNet and SqueezeNet. Focusing on VGG-16 and VGG-CNN-F, we see that fine tune is important especially when the bit-width is narrow. It brings more than 13% top-1 accuracy improvement on VGG-16 model when using 6-bit fixed point data.

Besides image classification, we also tested this strategy on the face alignment network in Figure 2. Compared with classification, the network used in this task outputs the key point coordinates rather than a relative score and thus requires a higher data precision. Example alignment results are shown in Figure 13. 8-bit data quantization in this application still offers good performance. The coordinate error is within 2

TABLE II: Data quantization result on different CNN models. The two columns for each bit-width configuration indicate the model is applied fine tune or not.

		fp-32		16bit		8bit		6bit	
		raw	fine tune	raw	fine tune	raw	fine tune	raw	fine tune
GoogLeNet	Top-1	68.60%	68.75%	68.70%	68.70%	62.75%	62.75%	16.65%	16.15%
	Top-5	88.65%	88.90%	88.45%	88.45%	85.70%	85.70%	31.55%	31.90%
VGG-16	Top-1	65.77%	67.93%	65.78%	67.84%	65.58%	67.72%	45.55%	58.86%
	Top-5	86.64%	88.14%	86.65%	88.19%	86.38%	88.06%	70.39%	81.56%
SqueezeNet	Top-1	58.69%	58.69%	58.69%	58.69%	57.27%	57.27%	30.56%	30.90%
	Top-5	81.37%	81.37%	81.36%	81.36%	80.32%	80.35%	53.94%	54.29%
VGG-CNN-F	Top-1	55.06%	57.78%	55.55%	57.55%	55.30%	57.55%	27.00%	32.50%
	Top-5	78.38%	80.61%	78.00%	79.50%	78.20%	79.40%	49.80%	54.60%

TABLE III: Hardware parameter and resource utilization

design	data	#PE	#Conv	FF	LUT	BRAM	DSP	Clock	Conv Performance (GOPS)
XC7Z045	16-bit	2	64	127653(29%)	182616(84%)	486(89%)	780(87%)	150MHz	187.8
XC7Z020	8-bit	2	16	35489(33%)	29867(56%)	85.5(61%)	190(86.4%)	214MHz	84.3
XC7Z030	8-bit	4	16	34097(22%)	43118(55%)	203(77%)	400(100%)	150MHz	105.2*
XC7Z045	8-bit	12	16	85172(19%)	139385(64%)	390.5(72%)	900(100%)	150MHz	292.0*

* The performance is estimated by simulation

pixels.

Another application of CNN is object detection. Recent work is using CNN to generate proposals from an image and give each one a classification result. In our test, we choose YOLO [26] detection algorithm with the YOLO tiny model for pedestrian detection task. This algorithm is applied to the video recorded from drones. 8-bit data quantization is also applied to the convolution layers of the network. One sample result is shown in Figure 14. These two examples show that 8-bit data quantization can support common applications.

Fig. 14: Pedestrian detection result using YOLO. The purple box shows the detected target.

B. Hardware Performance

Two FPGA-based designs of the hardware architecture are carried out. A 16-bit version of the design is implemented on the Xilinx XC7Z045 chip which targets at high-performance applications. An 8-bit version is implemented on the Xilinx XC7Z020 chip which targets at low power applications.

The hardware parameters and resource utilization of our design are shown in Table III. All the results are generated by Vivado 2015.4 version after synthesis and implementation. By choosing the design parameters properly, we can fully utilize the on-chip resource. Note that we are not using all the resource on XC7Z020 because the design coexists with an HDMI display logic for our demo. Comparing the 8-bit and 16-bit version result on XC7Z045, we see that 8-bit version offers 50% more parallelism while consuming less resource. This shows the importance of data quantization.

The VGG16 network is used to test the performance and energy efficiency of our design on XC7Z045 and XC7Z020 FPGAs. The result together with that of other FPGA designs for CNN and GPU is shown in Table IV. Some conclusions can be drawn from this comparison.

First, **precision greatly affects the energy efficiency.** Early designs [27] [7] using 48-bit fixed-point data or 32-bit floating point data are with much lower energy efficiency. Comparing the estimated 8-bit design on XC7Z045 with the 16-bit version also gives this conclusion. These two designs utilize similar resource and run with the same clock frequency, thus should consume similar power. But the 8-bit design offers more than 50% performance improvement, which means the energy efficiency is better.

Second, **the utilization of the hardware is important**. The reported performance in [6] is 200GOPs when the network perfectly matches the 10×10 convolver design. But for the 5×5 and 7×7 kernels, the performance is down to 23 GOPs. As discussed in section II, most of the computation in state-of-the-art neural networks is from 3×3 convolution. So the proposed design in this work should be better.

Third, **memory I/O affects the energy efficiency**. The

TABLE IV: Performance comparison of Angel-Eye on XC7Z045 and XC7Z020 with other FPGA designs and GPU

	[27]	[6]	[7]	[28]	Ours		GPU	
Platform	Virtex 5 SX240t	Zynq XC7Z045	Virtex 7 VX485t	Virtex 7 VX690T	Zynq XC7Z045	Zynq XC7Z020	Nvidia Titan X (maxwell)	
Clock (MHz)	120	150	100	156	150	214	1000	
Bandwidth (GB/s)	-	4.2	12.8	29.9	4.2	4.2	336	
Data format	48-bit fixed	16-bit fixed	32-bit float	16-bit fixed	16-bit fixed	8-bit fixed	float(batch 1)	float(batch 32)
Power(W)	14	8	18.61	30.2	9.63	3.5	174	210
Performance (GOP/s)	16	23.18*	61.62	565.9	137	84.3(CONV)	3544	5991
Energy Efficiency (GOP/s/W)	1.14	2.9	3.31	22.15	14.2	24.1	20.4	28.5

* The performance is of the face detector application in [6] where the 552M-op network is running at 42 frames per second.

TABLE V: Performance comparison of Angel-Eye on XC7Z020 with TK1 and TX1 on different tasks

	VGG		YOLO		Face Alignment	
	time/frame (ms)	performance (GOP/s)	time/frame (ms)	performance (GOP/s)	time/frame (ms)	performance (GOP/s)
7Z020	364	84.3	88.0	62.9	2.54	41.1
TK1(fp32)	347	88.4	150	36.9	14.3	7.3
TX1(fp32)	153	200	59.4	93.2	9.04	11.5
TX1(fp16)	96.5	318	42.4	131	2.18	47.9

energy cost of reading/writing data from/to memory is high. The design in [7] only implements channel parallelism to simplify the design of data path. But this strategy does not utilize the data locality in the convolution operations and leads to more data I/O. The design in [28] implements the whole AlexNet the large VX690T chip, where the intermediate result of each layer is not written back to memory. This further reduces data I/O and thus achieves higher energy efficiency compared with our 16-bit design. But this kind of design is hard to be scaled down to be deployed on embedded platforms with limited BRAM resources.

We also compared our design with desktop GPU using the VGG-16 network. Both batch one mode and batch 32 mode are tested. The batch one mode suffers about 41% performance loss compared with batch 32 mode. Our 8-bit design achieves even higher energy efficiency with the batch 1 mode on the large network. But the scale of GPU is too large for embedded platforms.

For the 8-bit version implementation on XC7Z020, two more tasks, YOLO, and face alignment are used for evaluation besides the VGG-16 network. We compare the performance of our design with the 28nm NVIDIA TK1 SoC and the latest NVIDIA TX1 SoC platforms. For YOLO and face alignment, CNN part is implemented on FPGA. The rest of the algorithms are handled by the integrated CPU in the SoC. Although a batched way of processing can fully utilize the parallelism of GPU on TK1 or TX1, it is not a good choice for real-time video processing because it increases latency. For some applications like tracking, the result of one frame is used for the computation on the next frame. This requires the frames to be processed one by one. So we do not use batch in our experiment. Performance comparison is shown in Table V.

All the three platforms perform better on larger CNN models. But the proposed design offers a more stable performance. On YOLO and face alignment tasks, Angel-Eye even offers better performance than TK1 and achieves similar performance as TX1. This is because the parallelism pattern of GPU does not fit into small network well. The running power of TK1 and TX1 are 10W while that of Angel-Eye on XC7Z020 is only 3.5W. So our design can achieve up to $16\times$ better energy efficiency than TK1 and $10\times$ better than TX1.

Performance of the 8-bit version on XC7Z030 and XC7Z045 is estimated with simulation. On XC7Z020, we measured the actual I/O bandwidth to be about 500MB/s. The estimation is based on this. XC7Z030 is with the same bandwidth and XC7Z045 doubles the bandwidth with an extra independent DDR port for FPGA. About 1.25x and 3.46x performance can be achieved by these two platforms compared with XC7Z020 with the help of more resource even with a conservative 150MHz estimated clock frequency.

VI. Related Work

Though many regions in machine learning benefit from neural network like algorithms, one of the main drawbacks is the high computation complexity, especially for CNNs. Various ways of accelerating CNN algorithms have been proposed, in hardware level, with dedicated designed accelerators, or in software level, aiming at compressing the network.

A. CNN Accelerator

It is common to accelerate the original version of CNN with 32-bit floating point data on GPUs since Caffe [24] and many other neural network frameworks are offering convenient

TABLE VI: Design character of state-of-the-art CNN accelerators

	platform	on-chip memory	external memory	loop unroll strategy			multi-layer
				kernel	feature map	channel	
CVPR 2011 [29]	Xilinx Virtex6 240t	N/A	yes	yes	N/A	N/A	no
CVPR 2014 [6]	Xinlinx XC7Z045	N/A	yes	yes	no	yes	no
ISCA 2015 [20]	65nm ASIC	288KB	no	no	yes	no	no
FPGA 2015 [7]	Xilinx Virtex7 485t	4.6MB	yes	no	no	yes	no
FPGA 2016 [8]	Xilinx XC7Z045	2.2MB	yes	yes	no	yes	no
ISSCC 2016 [30]	65nm ASIC	108KB	yes	dynamic	no	dynamic	no
ISSCC 2016 [31]	65nm ASIC	36KB	no	no	yes	yes	no
FPL 2016 [28]	Xilinx VC709	7.6MB	yes	yes	no	yes	yes
FPL 2016 [32]	Xilinx Virtex7 485t	9.8MB	yes	no	no	yes	yes

GPU interface. But the energy efficiency is not good and the high power of GPUs limits the application range. Thus various architectures have been proposed to accelerate CNNs, including both ASIC and FPGA designs. As discussed in [7], one Conv Layer can be expressed as six nested loops on input channel, output channel, two dimensions on feature map, and two dimensions on convolution kernel. The key point in CNN accelerator design is the unrolling strategy of the loops for each layer.

Fixed loop unrolling strategy is commonly applied in CNN accelerator designs. Zhang et al. [7] analyzed the data sharing relation of different iterations of a loop to evaluate the cost of unrolling. Calculation on different input channels and that for different output channels are of the lowest cost to be parallelized. But feature map level and kernel level parallelization are not fully explored. On these two levels, data locality is obvious. Utilizing this character can further reduce the data movement between different memory hierarchies and thus reduce energy cost. nn-X [6] adopted 2-D convolver design of size 10×10, which achieves kernel level parallelization. Our previous work [8] uses a 3×3 convolver design targeting at VGG network. Smaller convolver fits better with the trend of reducing convolution kernel size in CNN design. ShiDian-Nao [20] implements a mesh grid style structure to achieve parallelization on feature map level. A similar strategy is also adopted by [31].

Since the size of each layer is different, it is hard to use a fixed loop unrolling strategy to fit into all the layers. This means the calculation logic is not fully utilized. Configurable loop unrolling costs much in data routing but can fit into different network topologies better. Chen et al. [30] proposed a 2-D PE array design optimized for CNN. The global bus is used to broadcast and collect data from PEs. The connections are configurable to group different PEs together as convolvers of different sizes. The overhead is the routing cost and extra bits to identify the target PE of the data.

All the designs above achieves intra layer parallelization. Some other works focus on inter layer parallelization. Li, et al. [28] uses a pipeline design and accelerates all the layers concurrently on a single chip. By implementing each layer independently, calculation resource can be evenly allocated among different layers to achieve the highest efficiency for all the layers. This kind of solution is easily scaled up to a larger platform but hard to be scaled down. Also, state of the art CNN model involves up to 100 layers [14] which is also hard to be supported by this solution. Another work by Shen, et al. [32] implement a similar design but group some of the adjacent layers, making it less resource consuming.

Besides computation, the high storage complexity is another challenge for CNN accelerator designs. For real applications, totally using on-chip memory is not feasible, especially on embedded systems. Du, et al. [20] discussed data management in on-chip cache to fully utilize the hardware parallelization strategy. All the data is assumed on-chip in this work, so no external memory is used. Qiu, et al. [8] discussed the data arrangement in external memory to maximize the burst length of data access. This raises the bandwidth utility factor. Multi-layer implementations [28] [32] reduce the communication with the external memory for intermediate results but requires large on-chip memory.

A comparison of these designs is in Table VI. As discussed above, memory system and parallel strategy for each work are listed in this table. It is common to use a single layer implementation with static loop unroll strategy, which is the same as this work. The latest research on CNN explores the sparsity to further reduce the computation complexity. In this situation, more dedicated hardware should be designed to utilize sparsity. Latest accelerator designs [33] [34] [35] is focusing on sparsity to achieve higher energy efficiency.

B. Network Compression

Convolutional Neural Network offers a high performance against traditional CV algorithms but brings with it high computation complexity. Besides hardware acceleration, reducing the model complexity is also a cutting edge topic. On CPU and GPU platforms, usually 32-bit floating point data is used for computing. [6] and [20] use 16-bit fixed-point data with 8 bit for integer and 8bit for fractional in their hardware design, which proves to bring negligible accuracy loss. Our previous work [8] shows that 8-bit for CONV layers and 4-bit for FC layers is a promising solution for the VGG model. Han, et al. [9] compress the data to 4-bit by clustering. But the data are converted back to 32-bit floating point format for computation. Some of the recent work [10] [36] is trying 1-bit or 2-bit data.

This requires more techniques in training the network. More experiments are needed to validate these techniques.

Besides reducing the bit-width, reducing the number of connections is another way to compress the network. Singular value decomposition (SVD) is a common way for matrix approximation and has been applied to compress FC layers [9]. Han, et al. used iterative pruning in their work which reduces the number of connections of the FC layers of VGG-16 model to 1/13.

C. CNN Acceleration Design Flow

Besides single accelerator design, some work focuses on automatic tool mapping CNN onto hardware, which is similar to this work. Zhang, et al. [11] proposed a CNN acceleration framework to automatically choose the best hardware parameters given the model files from Caffe. Dedicated design space exploration is done based on roofline model. Data organization in DRAM is also handled in the framework. In [12], an ISA is proposed to describe the network as a data flow graph(DFG). With the DFG, software compiler can statically schedule the whole process of computing one network. But the graph is also converted to a finite state machine and is not run-time configurable. Another framework by [37] partitions deep learning algorithms into basic blocks and generates a combination of the blocks targeting at a certain network.

On the server side, targeting at a single network is a good choice to achieve extreme hardware performance. For real-time mobile applications, more than one network may be needed. The overhead of programming FPGA at run-time to switch network is too large. In this paper, the proposed CNN acceleration flow isolates the design of hardware and software. The hardware parameter can be chosen based on a certain network structure but it supports different networks by simply changing the software at run-time. This makes it more suitable for complex applications. Also, this kind of design usually requires more resource than single-layer implementations and thus is not suitable for embedded FPGA platforms.

VII. Conclusion

In this paper, we propose a complete flow for mapping CNN onto customized hardware. A data quantization strategy is proposed to compress the bit-width used in CNN. Evaluated on state-of-the-art CNN models, this strategy brings negligible performance loss with 16-bit and 8-bit configuration. A compiler is also implemented to map different CNN models to instruction sequences. Optimization is done on compilation to fully utilize the on-chip cache and the parallelism between calculation and data I/O. For the hardware, we extend our previous work [8] with a flexible instruction interface to support this work. Experimental results show that 16-bit Angel-Eye on XC7Z045 is 6× faster and 5× better in power efficiency than peer FPGA implementation on the same platform. The 8-bit version on XC7Z020 achieves up to 16× better energy efficiency than NVIDIA TK1 and 10× better than TX1. More importantly, we show that **data bitwidth**, **computation resource utilization**, and **memory I/O amount** are the three aspects that should be focused to design efficient hardware for CNN acceleration.

Some aspects of this work still need improvement. For CNN acceleration, better performance can be achieved. As mentioned in section VI-B, the latest network compression work is adopting 1-bit design. Focusing on hardware accelerator with narrower bit-width is one direction of future work. The sparsity of CNN offers more chance of acceleration. Also, fast algorithm on convolution has been proposed and proved to work well on CNN [38], integrating this algorithm into the accelerator is also a good choice to further improve the hardware performance.

For the whole system, simply accelerate CNN may not be the best choice. Though CNN is powerful in many regions, it can not cover every corner of an application. Optimization on integration with other accelerators to explore the best system level design should be done in the future.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," 2016.

[4] J. Zbontar and Y. LeCun, "Stereo matching by training a convolutional neural network to compare image patches," *Journal of Machine Learning Research*, vol. 17, pp. 1–32, 2016.

[5] O. Abdel-Hamid, L. Deng, and D. Yu, "Exploring convolutional neural network structures and optimization techniques for speech recognition." in *Interspeech*, 2013, pp. 3366–3370.

[6] V. Gokhale, J. Jin, A. Dundar *et al.*, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *CVPRW*, 2014, pp. 682–687.

[7] C. Zhang, P. Li, G. Sun *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*. ACM, 2015, pp. 161–170.

[8] J. Qiu, J. Wang, S. Yao *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*. ACM, 2016, pp. 26–35.

[9] S. Han, J. Pool, J. Tran *et al.*, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015, pp. 1135–1143.

[10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," 2016.

[11] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks."

[12] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J. K. Kim, A. Mishra, and H. Esmaeilzadeh, "Dnnweaver: From high-level deep network models to fpga acceleration," 2016.

[13] F. N. Iandola, M. W. Moskewicz, K. Ashraf *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 1mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[15] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015, pp. 3431–3440.

[16] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[17] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1440–1448.

[18] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *ECCV*, 2014, pp. 818–833.

[19] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, "A convolutional neural network cascade for face detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5325–5334.

[20] Z. Du, R. Fasthuber, T. Chen *et al.*, "Shidiannao: shifting vision processing closer to the sensor," in *ISCA*. ACM, 2015, pp. 92–104.

[21] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 199–204.

[22] B. Bosi, G. Bois, and Y. Savaria, "Reconfigurable pipelined 2-d convolvers for fast digital signal processing," *VLSI*, vol. 7, no. 3, pp. 299–308, 1999.

[23] C. Szegedy, W. Liu, Y. Jia *et al.*, "Going deeper with convolutions," *arXiv preprint arXiv:1409.4842*, 2014.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *Eprint Arxiv*, pp. 675–678, 2014.

[25] "Imagenet," http://www.image-net.org/.

[26] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *arXiv preprint arXiv:1506.02640*, 2015.

[27] S. Chakradhar, M. Sankaradas, V. Jakkula *et al.*, "A dynamically configurable coprocessor for conv olutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[28] L. J. Hunmin Li, Xitian Fan *et al.*, "A high performance fpga-based accelerator for large-scale convolutional neural networks," 2016.

[29] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Cvpr 2011 Workshops*. IEEE, 2011, pp. 109–116.

[30] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *ISSCC*. IEEE, 2016.

[31] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "A 1.42tops/w deep convolutional neural network recognition processor for intelligent ioe systems," in *ISSCC*. IEEE, 2016.

[32] M. F. Yongming Shen and P. Milder, "Overcoming resource underutilization in spatial cnn accelerators," 2016.

[33] Han, Song and Liu, Xingyu and Mao, Huizi and Pu, Jing and Pedram, Ardavan and Horowitz, Mark A. and Dally, William J., "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, 2016, pp. 243–254.

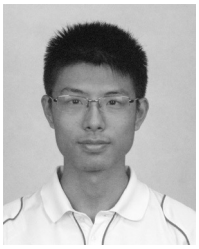
[34] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 1–13.

[35] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

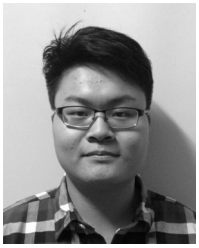
[36] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[37] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: automatic generation of fpga-based learning accelerators for the neural network family," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 110.

[38] A. Lavin, "Fast algorithms for convolutional neural networks," *arXiv preprint arXiv:1509.09308*, 2015.

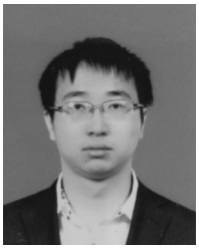


Kaiyuan Guo received his B.S. degree in 2015 from Tsinghua University, Beijing. He is currently pursuing the Ph.D. degree with the Department of Electronic Engineering, Tsinghua University, Beijing. His current research interests include hardware acceleration of deep learning and SLAM.



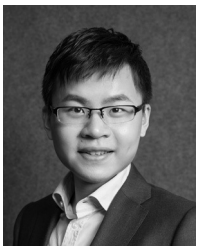
Lingzhi Sui received his B.S. degree in 2016 from Tsinghua University, Beijing. He is currently a senior engineer in DeePhi Technology, Beijing. His current research interests include network optimization and scheduling for hardware neural network acceleration.

Jiantao Qiu received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2015. He is currently pursuing the Ph.D. degree with the Center for Brain Inspired Computing Research, Tsinghua University, Beijing. His current research interests include computing architecture, brain inspired computing and system scheduling.



Jincheng Yu received his B.S. degree in 2016 from Tsinghua University, Beijing. He is currently a Ph.D. student with the Department of Electronic Engineering, Tsinghua University, Beijing. His current research interests include software optimization and hardware architecture for deep learning acceleration.

Junbin Wang received the B.S. degree in electronic engineering from Chongqing University, Chongqing, China in 2013 and the Master degree in the Institute of Microelectronics from Tsinghua University, Beijing, China in 2016. He now works as a CNN development engineer in Deephi Technology Co. Ltd., Beijing, China. His current research interests include SoC design, deep learning and reconfigurable computing.



Song Yao received his B.S. Degree in Tsinghua University in 2015. He is currently the CEO and Co-Founder of DeePhi Tech, a startup that is devoted to provide the world with more efficient deep learning platform. He is a well-recognized researcher in hardware acceleration of deep learning. He has received many awards including FPGA 2017 Best Paper, Top 30 AI Entrepreneurs in China, and Forbes 30 Under 30 Asia.

Song Han received his B.S. degree in 2012 from Tsinghua University, Beijing and M.S. degree in 2014 from Stanford University. He is now a fifth year PhD candidate advised by Prof. Bill Dally at Stanford University. His research focuses on energy-efficient deep learning, at the intersection between machine learning and computer architecture. His work won the Best Paper Award at ICLR16 and the Best Paper Award at FPGA17.

Yu Wang received his B.S. degree in 2002 and Ph.D. degree (with honor) in 2007 from Tsinghua University, Beijing. He is currently an associate professor with the Department of Electronic Engineering, Tsinghua University, Beijing. His research interests include application specific hardware computing (especially on the brain related problems), and parallel circuit analysis, power/reliability aware system design methodology.

Huazhong Yang received the B.S. degree in microelectronics and the M.S. and Ph.D. degrees in electronic engineering from Tsinghua University, Beijing, China, in 1989, 1993, and 1998, respectively. In 1993, he joined the Department of Electronic Engineering, Tsinghua University, where he is currently a Specially Appointed Professor of the Cheung Kong Scholars Program. He has authored and co-authored over 200 technical papers and holds 70 granted patents. His current research interests include wireless sensor networks, data converters, parallel circuit simulation algorithms, nonvolatile processors, and energy-harvesting circuits.