# A Streaming Deep Learning Accelerator with Selective Binarization

**Xuecan Yang**
Telecom ParisTech
46, Rue Barrault
xuecan.yang@telecom-paristech.fr

**Sumanta Chaudhuri**
Telecom ParisTech
46, Rue Barrault
sumanta.chaudhuri@telecom-paristech.fr

**Laurence Likforman-Sulem**
Telecom ParisTech
46, Rue Barrault
laurence.likforman@telecom-paristech.fr

**Lirida Naviner**
Telecom ParisTech
46, Rue Barrault
lirida.naviner@telecom-paristech.fr


This work was granted access to the HPC resources of [TGCC/CINES/IDRIS] under the allocation 2018-Accés prèparatoire OUESSANT N AP010610330 attributed by GENCI (Grand Equipement National de Calcul Intensif)

## Abstract

In this article we present a streaming deep learning accelerator capable of implementing convolutional layers of different precision. We can choose and combine between 16-bit floating point and binary values for features maps and weights. We present the architecture of our streaming accelerator and compare it with standard systolic array based architectures for TPUs. Taking the well known CNN YOLOv2 as an example we present exploration studies to determine the optimum precision for each layer. We train the tiny YOLO CNN with a drone object detection data-set (DAC-SDC). We present and compare the implementation results on the Xilinx PYNQ-Z1 board, and we show that it is possible to achieve 1.68x improvement in performance incurring a 10.5% loss in precision measured by IOU(Intersection over Union).

## Introduction

Unprecedented success rates of Deep Neural Networks (DNNs) for image recognition and similar tasks have ushered us into an era of deep learning. Although *neural networks* exist since the 1940s [23], the major breakthrough dates back to 2012 when Alexnet [13] surpassed the human programmed approaches for image classification. The major enabling factors behind this success are supposed to be the appearance of large datasets in the cloud, and the availability of enormous computing power (e.g GPUs).

Since then, there has been a steady growth of customized DNN accelerators research and several architectures have been proposed [11, 6, 4, 5], and more recently TPU (Tensor Processing Units) [9] have been commercialized by Google.

There are two phases associated with a given architecture (e.g. Alexnet, VGG16 [22], YOLO [20], SSD [17] etc.) and a given dataset (COCO [16], PASCAL-VOC [10] etc.), namely training and inference. For training, Stochastic Gradient Descent (SGD) [3] is a popular approach which requires floating point precision and often done in GPU servers. The inference phase can either be carried out in the cloud, or it can be carried out in an embedded device close to the data source.

Moreover, it has been shown that for the inference phase a lower precision is enough. [4] proposes half precision floating point, [9] uses 8 bit quantized weights, and [24, 7] propose the use of binarized weights and feature maps.

*Motivation & Scope*
Although DNNs can have several different variations such as multi-layer perceptrons, Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNNs) and their variants such as RNNS with Long Short Term Memories (LSTM). In this article we limit our scope to accelerating Convolutional Neural Networks (CNNs).

Although fully binarized (both binary feature maps and weights) networks work well with smaller networks such as Alexnet and Lenet as described in [24, 7], they tend to loose precision with deeper networks, e.g. YOLOv2, SSD.

To use binarized computing without losing too much precision, in this article, we propose a deep learning accelerator with selective binarization. This accelerator is targeted for the inference phase of CNN. And we don't optimize the training phase. Although the architecture proposed is generic, in our experiments we mainly target embedded system applications such as object detection/tracking. These applications often have real time constraints and a very tight power budget. We conduct our experiments with architectures and data-sets used in this domain, and we use the PYNQ-Z1 FPGA board for embedded systems.

*Contribution*
The major contribution of this paper is an end-to-end streaming architecture for a deep learning accelerator different from the systolic array architectures [9] or NoC based architectures [6]. Our architecture is capable of implementing mixed precision CNNs where we can choose from three different precision levels (half precision floating for feature maps and weights, half for feature maps and binary weights, and binary feature maps and weights) for each convolutional layer. Several works[12][15][8] related to quantization also provide architectures with mixed precision. But we provide the choice of binarized layer, which converts multiplication into addition/subtraction or XNOR logic operations, and we also made special architecture for binarized layer. To our knowledge this is the first architecture which proposes such a choice.

We propose a method of architecture exploration to find the optimum use of binary layers, which we call selective binarization. The goal is to use selective binarization to increase performance and decrease power consumption, within a tolerable precision loss.

*Organization*
The rest of the paper is organized as follows: Firstly, we describe our training and architecture exploration method. Then the details of the proposed architecture for the accelerator are presented. Subsequently we focus on the experimental results based on YOLO CNN architecture and

the PYNQ FPGA board. Finally, we compare the proposed architecture with previously well-known accelerators and conclusions are discussed.

## Training with selective binarization

According to different binary methods, we can build three different kinds of layers for CNNs:

- Float layer: Both input **F**eature maps and **W**eights are **H**alf (FHWH) precision floating point.
- Binary layer: Input **F**eature maps are in **H**alf precision and **W**eights are **B**inary (FHWB). In binary layer, the multiplication operation is reduced to an addition/subtraction. Although the computation rate is the same as the full precision channel, 16x weights can be stored in on-chip memory.
- XNOR layer: Both input **F**eatures maps and **W**eights are **B**inary (FBWB). Since all multipliers in this convolution belongs to $\{1, -1\}$, the multiplication operations are reduced to a XNOR operation . For this channel, we rearrange the data in input feature stream in such a way that 16 XNOR can be performed in parallel, so leading to a 16x speedup compared to other channels.

The compute speeds of the layers are different. Similarly, the accuracies are different. Depending on the usage scenario, we use the three kinds of layers to build mixed CNNs, so that performance increases and power consumption decreases, within a tolerable precision loss.

Tiny YOLO shown in Table 1 is an object detection network that is much faster but less accurate than the normal YOLO model. In this section, taking tiny YOLO as an example, we present our method to train mixed CNNs. Then, we compare the detection accuracies of the mixed tiny YOLO.

**Table 1:** Tiny YOLO network

| **Tiny-YOLO** | Filters | Size / Stride |
|---|---|---|
| Convolutional | 16 | 3 x 3 / 1 |
| Maxpool | | 2 x 2 / 2 |
| Convolutional | 32 | 3 x 3 / 1 |
| Maxpool | | 2 x 2 / 2 |
| Convolutional | 64 | 3 x 3 / 1 |
| Maxpool | | 2 x 2 / 2 |
| Convolutional | 128 | 3 x 3 / 1 |
| Maxpool | | 2 x 2 / 2 |
| Convolutional | 256 | 3 x 3 / 1 |
| Maxpool | | 2 x 2 / 2 |
| Convolutional | 512 | 3 x 3 / 1 |
| Maxpool | | 2 x 2 / 1 |
| Convolutional | 1024 | 3 x 3 / 1 |
| Convolutional | 1024 | 3 x 3 / 1 |
| Convolutional | 90 | 1 x 1 / 1 |
| Detection | | |

*Binarization method*

We use the method presented in [19] to binarize the feature maps and weights. The Darknet [19] framework which supports binarization, is used to train the network.

In order to constrain a convolution operation $X \otimes W$ to have binary weights, we replace the real weights $W \in R^{w \times h \times c}$ by binary weights $W^b = sign(W)$ with $W^b \in \{+1, 1\}^{w \times h \times c}$. A real scaling factor $\alpha = \frac{1}{n}||W||_{l1}$ where $n = w \times h \times c$ is used to approximate the convolution operation as :

$$X \otimes W \approx \alpha(X \oplus W^b)$$

where, $\oplus$ indicates a convolution without any multiplication. Since the weight values are binary, we can implement the convolution with additions and subtractions.

As well, if we use this method to binarize the feature maps, a convolutional layer can be approximated by:

$$X \otimes W \approx \alpha\beta(X^b \odot W^b)$$

where $X^b = sign(X)$, $\beta = \frac{1}{n}||X||_{l1}$, and $\odot$ indicates a convolution without any algebraic operation, which are replaced by logical operations XNOR.

We use the training method presented in [21], except that the binary and XNOR layers are computed by the method presented in [19]. Algorithm 1 demonstrates the procedure used in Darknet for training a mixed CNN. Lines 1-16 show the forward propagation phase, where $Forward$ means convolution within the $l^{th}$ layer, followed by max-pooling and batch normalization layers if necessary. The result of forward propagation $\hat{Y}$ encodes the prediction information. Lines 17-19 show the backward propagation

**Figure 1:** The results are for architectures the baseline, architecture 8 and 12 respectively.

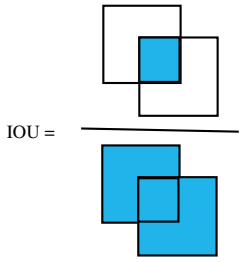



**Figure 2:** Illustration of the IOU (Intersection Over Union) detection metric:
$IOU = \frac{Area\ of\ Intersection}{Area\ of\ Union}$

that computes the gradients by using $\widetilde{W}$ instead of $W^t$ (in the case of a float layer, the two are equal), then applies these gradients to update the weights $W^t$. The function $UpdateParamaters$ can be realized by any update rules (e.g. SGD or ADAM). At the end of this Algorithm, in Line 20, the learning rate is updated according to the strategy used by Darknet.(We used fixed strategy in our training).

**Algorithm 1** Training an L-layers mixed CNN as used in Darknet

**Input**: A batch of inputs and targets $(I, Y)$, cost function $C(Y, \hat{Y})$, current weight $W^t$ and current learning rate $\eta^t$.

**Output**: updated weight $W^{t+1}$ and updated learning rate $\eta^{t+1}$.

1: //**Forward propagation**
2: $X_1 = I$
3: **for** $l = 1$ **to** $L$ **do**
4: **if** $l^{th}$ layer is XNOR layer
5: $X_l^b = sign(X_l)$
6: $\beta_{lk} = \frac{1}{n}||X_{lk}^t||_{l1}$
7: $X_l = \beta_{lk} X_l^b$
8: **if** $l^{th}$ layer is XNOR layer or binary layer
9: **for** $k^{th}$ filter in $l^{th}$ layer **do**
10: $\alpha_{lk} = \frac{1}{n}||W_{lk}^t||_{l1}$
11: $W_{lk}^B = sign(W_{lk}^t)$
12: $\widetilde{W}_{lk} = \alpha_{lk} W_{lk}^B$
13: **else**
14: $\widetilde{W}_{lk} = W_{lk}$
15: $X_{l+1}$ =**Forward**$(X_l, \widetilde{W}_l, l)$
16: $\hat{Y} = X_{L+1}$
17: //**Backward propagation**
18: $\frac{\partial C}{\partial \widetilde{W}}$ =**Backward**$(\frac{\partial C}{\partial \hat{Y}}, \widetilde{W})$
19: $W^{t+1}$ =**UpdateParamters**$(W^t, \frac{\partial C}{\partial \widetilde{W}}, \eta^t)$
20: $n^{t+1}$ =**UpdateLearningrate**$(\eta^t, t)$

**Table 2:** Mixed Tiny YOLO network

| Conv Layer | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Baseline | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch1 | $X$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch2 | $B$ | $X$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch3 | $B$ | $B$ | $X$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch4 | $B$ | $B$ | $B$ | $X$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch5 | $B$ | $B$ | $B$ | $B$ | $X$ | $B$ | $B$ | $B$ | $B$ |
| Arch6 | $B$ | $B$ | $B$ | $B$ | $B$ | $X$ | $B$ | $B$ | $B$ |
| Arch7 | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $X$ | $B$ | $B$ |
| Arch8 | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $X$ | $B$ |
| Arch9 | $X$ | $X$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch10 | $B$ | $B$ | $X$ | $X$ | $B$ | $B$ | $B$ | $B$ | $B$ |
| Arch11 | $B$ | $B$ | $B$ | $B$ | $X$ | $X$ | $B$ | $B$ | $B$ |
| Arch12 | $B$ | $B$ | $B$ | $B$ | $B$ | $B$ | $X$ | $X$ | $B$ |

*Architecture exploration*

Through the above binary method, we can binarize all layers to build binary tiny YOLO network. Taking this architecture as baseline architecture, we use grid search method, which replaces the convolutional binary layers by XNOR layers for each layer or each two layers. The $9^{th}$ layer which does not take up lots of resources encodes the final result (detection), so we keep the input feature maps of $9^{th}$ as floating point and binarize weights only, that is as a binary layer. Then 12 mixed CNN architectures composed of binary and XNOR layers are generated, as shown in Table 2, where $B$ means binary layer and $X$ means XNOR layer. They are divided into two groups based on number of XNOR layers they have (one or two).

*Training context and platform*

A drone object detection dataset DAC-SDC [1] is used for training these 12 networks. It includes 13 class of objects (car, building, person, etc.). Only one object is presented in each image. Images with 416x416 resolution is used to train the network. Same as in YOLO [20], we use Darknet

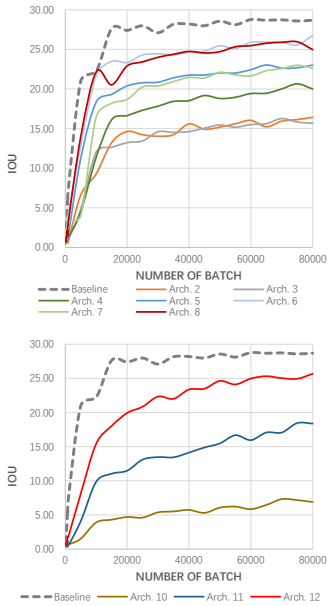



**Figure 3:** IOU along the training batches: The value plotted in the Y axis is the IOU value over a test dataset with 416x416 resolution, and X-axis plots the number of batches

framework for training.

We use convolutional weights that are pre-trained on Darknet Reference Model [2] as the initialization of weights. The size of the image is varied during the training by Darknet [2]. Darknet also randomly adjusts the exposure and saturation of the image by up to a factor of 1.5 in the HSV color space. Throughput training we use a batch size of 128, a momentum of 0.9 and a decay of 0.0005. The learning rate is fixed to $1e-5$. We have trained the networks upto 80K batches.

The hardware platform for training is composed of 12 IBM Power server compute nodes. Each node has 4 Tesla P100 GPU, with 3584 CUDA cores inside. We use 3 nodes which include 12 GPU to training 12 mixed tiny YOLO at the same time. Each training task is affined to one exclusive GPU (GPU not shared by others). It takes about 180 hours to train 80000 batches of images. The main evaluation criteria in object detection problem is IOU (Intersection Over Union), shown as Figure 2.

We can see examples of IOUs of the objects detected in Figure 1. The IOU along the training batches are shown in Figure 3. The figure does not contain architecture 1 and architecture 9 which binarize the input image in first layer. Since their IOUs are close to 0 until 35000 batches, we abandon the training. The IOU of architecture 12 is close to baseline. In fact, it is reduced by 10.5% compared to baseline. Moreover, it can be seen in Figure 3 that this loss may be reduced if we continue to train. At the same time, because of using XNOR layers in $7^th$ and $8^th$ layers, which are the most computationally intensive, it greatly reduces computation time. The architecture 6,7,8 that use XNOR in later layer also get relatively good IOUs.

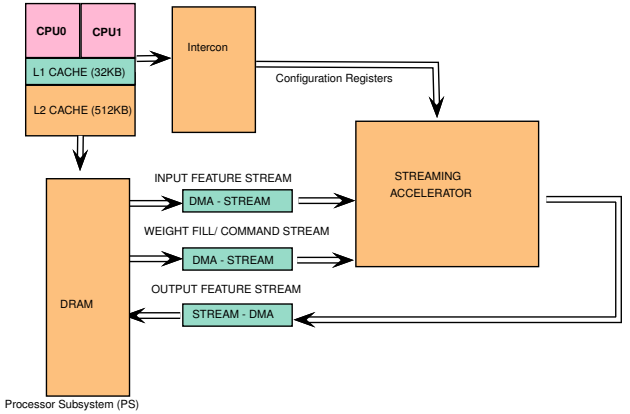



**Figure 4:** Overall Hardware Architecture

## Accelerator Architecture

Our accelerator architecture is suitable for computing one convolutional layer followed by a maxpooling layer at a time. Figure 4 depicts the overall architecture. As shown in Figure 5 the convolutions are not directly computed, but are converted to a multiplication of the matrix. This is a standard method in all GPU implementations and some accelerators [9]. The equivalent tensor of an input feature map or image with *C* channels of dimension $(H \times W \times C)$ is converted to a 2D matrix with rows of size $(f_h \times f_w \times C)$, and $H \times W$ rows. $f_h$ and $f_w$ are the filter height and weight, respectively. Similarly, the $C_{out}$ weights for a convolutional layer are arranged into a $(f_h \times f_w \times C) \times C_{out}$ matrix where each column contains distinct weights of filter. The matrix multiplication results in an image/feature map of same width & height as the input but with $C_{out}$ channels.

*Hardware/Software Partitioning*

Figure 4 presents the overall system architecture. The device driver running on the processor handles the tasks of

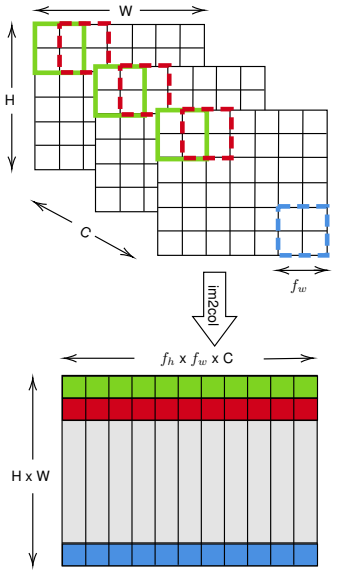



**Figure 5:** Image to Matrix Transformation

resizing, preprocessing images, pre-load weights into accelerator, and launch the computing for one convolutional layer. The image-to-matrix transformation can be done in both software and hardware. As we can see in Figure 8, for convolutional layers with large number of channels, the execution time on the processor is not very high. This is because with large row, the transformation is mainly done in the cache memory. For smaller rows, the processor has to fetch several non-contiguous lines and spends more time in to and fro main memory access.

*Hardware architecture*

Figure 6(a) presents the overall architecture of our accelerator. The input feature map matrix for a convolutional layer is streamed into the accelerator in row major format. Due to limited on-chip memory, only a part of the weights matrix can be loaded at one time (32 filters in the experiments) and it is necessary to use several iterations for convolutional layers with large number of filters. For example, for a convolutional layer with 256 filters 8 iterations are necessary. The output of the convolutional layer is fed into the maxpool layer which can then be streamed back to main memory.

The streaming accelerator in Figure 6(a) has $N$ convolutional lanes which perform an inner product between the incoming input feature and weights streams. There is a configurable image-to-matrix transformation stage which is only used for layers with very few channels. The input feature stream is then broadcasted to all the convolutional lanes. As each convolutional lane corresponds to one filter, the $N$ filter weights are distributed to $N$ lanes from the on-chip weight RAM. As we can see, there is a lot of data reuse (often expressed in terms of MAC/data [23]). In this case, the input feature stream has a data reuse of $N$ MAC/data and the weight has a data reuse of $H \times W$ MAC/data. The maxpool layer can be skipped. Finally, the output in $HWC$ format is regrouped and streamed back to main memory. For convolutional layer with $C_{out}$ channels, $\frac{C_{out}}{N}$ iterations are necessary.

As shown in Figure 6(b), the stream computing element performs an inner product of the input streams $A$ and $B$. The input stream $A$ is redirected to one of the three lanes according to the chosen precision. As mentioned earlier, it supports three precision: FHWH, FHWB and FBWB.

## Implementation on the PYNQ-Z1 board

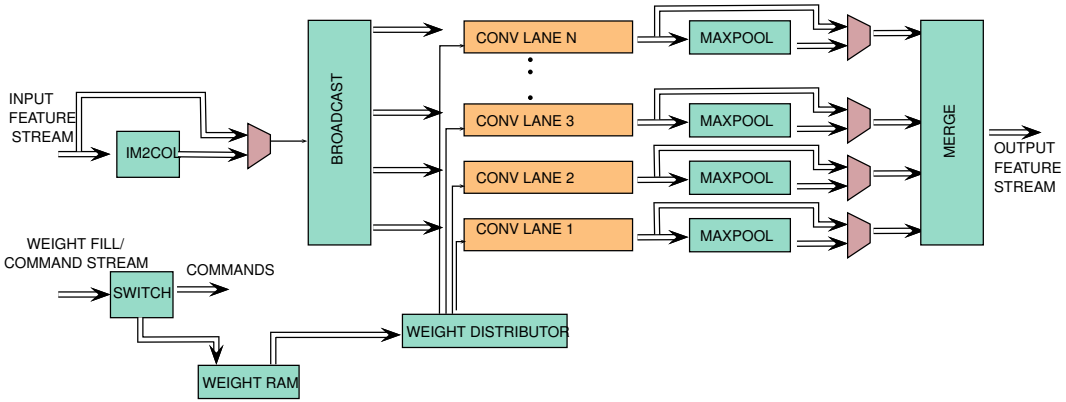
Table 3 shows the resource utilization of our implementation with the Z7020 FPGA. The main limiting factor is the on-chip memory. Because of this limitation we can only implement 32 lanes of convolution with half precision floating point running at 150 MHz. Thus the max. performance of this implementation is 9.6 Gflops/S. We consider a Multiply-accumulate operation as 2 flops. Based on this limitation we have explored other architectural choices:

**Table 3:** Resource Utilization in PYNQ-Z1 board with Zynq 7020 FPGA

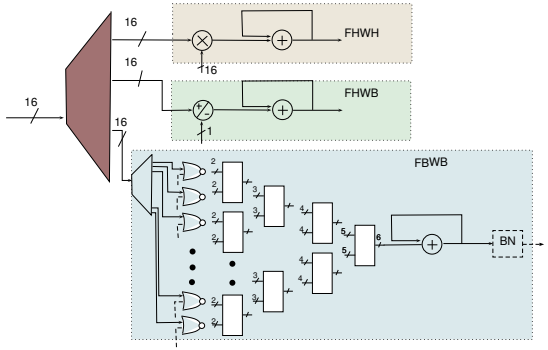
| Site Type | Used | Available | Util |
|---|---|---|---|
| Slice LUTs | 43167 | 53200 | 81.14 % |
| Block RAM Tile | 131.5 | 140 | 93.93 % |
| DSPs | 64 | 220 | 29.09 % |

- Choice 1: Feature Maps (FMs) and stored in main memory, and 32 filter weights are stored in FPGA on-chip memory.
- Choice 2: Both Feature Maps (FMs) and the weights are stored in main memory.

With the help of a roofline diagram, Figure 7, we show the performance limits of our choices. The roofline for FHWH and FHWB are the same, and the roofline for FBWB is




(a) Detailed architecture of the Streaming Accelerator




(b) Detailed architecture of the convolutional lanes

**Figure 6:** The accelerator architecture

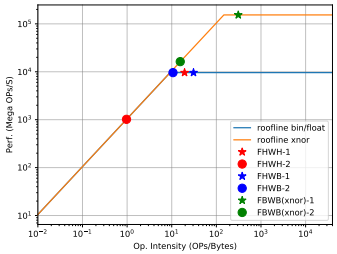



**Figure 7:** Architectural Choices and their explanation with roofline.

16x higher performance. We have plotted the points corresponding to the last layer (most compute intensive) on the roofline. The FHWH-2 architecture is limited by memory bandwidth as for each MAC operation we have to stream 66 bytes of data. For the FHWH-1 architecture the weights (2 bytes) are stored on chip, but in practice it is only possible for the first layer with few weights. Both FHWB-1 and FHWB-2 can achieve the performance limit but we prefer to store the weights on chip to reduce power consumption. Similarly FBWB-1 with on-chip storage can achieve the performance limit but FBWB-2 is limited by memory bandwidth.

## Experiments & Results

**Table 4:** Tiny YOLO parameters used in experiments

| # Conv. | H | W | $f_h \times f_w$ | C | $C_{out}$ | $N_{lanes}$ | $N_{iter}$ |
|---|---|---|---|---|---|---|---|
| 1 | 416 | 416 | 3x3 | 3 | 16 | 32 | 1 |
| 2 | 208 | 208 | 3x3 | 16 | 32 | 32 | 1 |
| 3 | 104 | 104 | 3x3 | 32 | 64 | 32 | 2 |
| 4 | 52 | 52 | 3x3 | 64 | 128 | 32 | 4 |
| 5 | 26 | 26 | 3x3 | 128 | 256 | 32 | 8 |
| 6 | 13 | 13 | 3x3 | 256 | 512 | 32 | 16 |
| 7 | 13 | 13 | 3x3 | 512 | 1024 | 32 | 32 |
| 8 | 13 | 13 | 3x3 | 1024 | 1024 | 32 | 32 |
| 9 | 13 | 13 | 1x1 | 1024 | 90 | 32 | 3 |

We conducted our experiments on the PYNQ-Z1 board. Due to resource constraint on the FPGA, the image-to-matrix transformation part is carried out in the processor. As the number of convolutional lanes are limited to 32, for each layer we need to do a few iterations. These are detailed in Table 4.

The 68.8% of the multiplication is in the $7^{th}$ and $8^{th}$ layers. Therefore, we analyze the architecture 8 and 12, which binarize the $7^{th}$ and $8^{th}$ layers. Figure 8 shows the detailed execution time for each layer of baseline and architecture 8 and 12 as well as the IOU with 80K batches images trained. Figure 8(d) shows the total execution time for the three architectures. We can see that the IOU of architecture 12 is close to baseline. In fact we can see that for architecture 12 a 1.68x speedup is achievable with 10.5% loss of IOU.

## Related Work

The two very well known deep learning accelerators are the tensor processing unit from Google [9] and Eyeriss [6, 5]. The TPU and a majority of deep learning accelerators use Systolic Arrays [14]. In Figure 9 we compare our architecture with a systolic array implementation. We assume

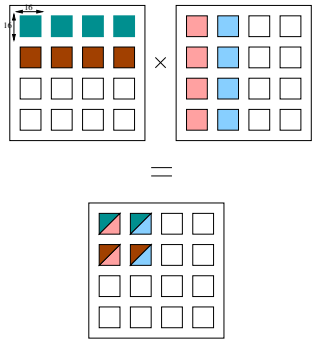


**Figure 9:** 64x64 matrix multiplication with Systolic Array and Tiling.

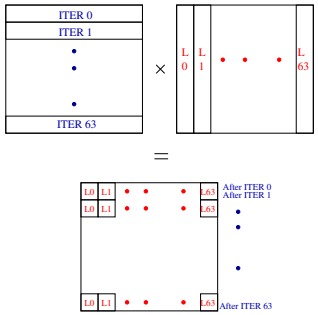
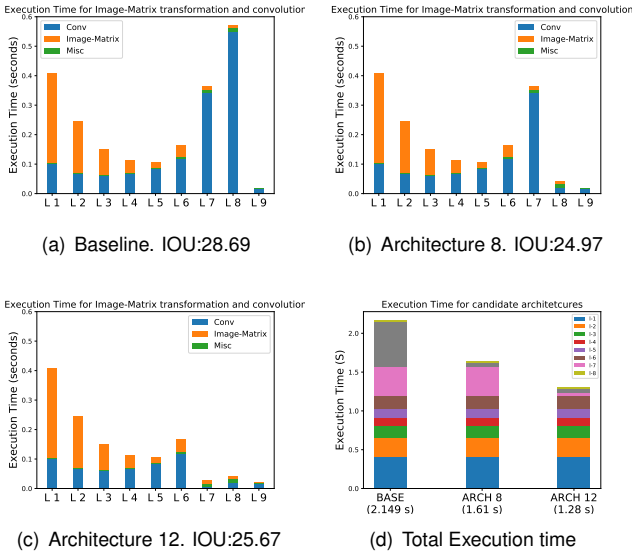



**Figure 10:** 64x64 matrix multiplication with our streaming architecture.




(a) Baseline. IOU:28.69 (b) Architecture 8. IOU:24.97

(c) Architecture 12. IOU:25.67 (d) Total Execution time

**Figure 8:** The execution time for candidate architectures.The architecture 8 and 12 archive 1.32x and 1.68x speedup respectively. If considering convolution time only a 2.5x speedup can be achieved.

that both architectures have the same number of processing elements (PEs). In Figure 10, the 64x64 square matrix is divided into 16 tiles of 16x16 matrices. Each 16x16 submatrix multiplication can be done with the help of a 16x16 systolic array in 16 pipelined cycles. To calculate each output sub-matrix we need 4 such multiplications described above. so in total we need 16x4x16 or 1024 cycles.

In our streaming architecture, each dot product between row and columns of a 64x64 matrix takes 64 cycles. If we assume the same number of PEs as the systolic array (256), the total computation can be completed in (64x64x64/256) or 1024 cycles. If the image-to-matrix transformation is done on chip, the memory transfer overhead should be similar for both. Our architecture can also be viewed as an one-dimensional systolic array.

Eyeriss [6] uses a square grid of processing elements but not as a systolic array. The convolution task is mapped to this square grid connected by a Network-on-Chip. [18] proposes the YOLOv2 architecture with floating point feature maps and binarized weights. It is difficult to compare the results as it is done on a different FPGA board, but it will be part of our future work.

## Conclusions & Future Work

In this article we presented methods and architectures to implement CNNs with varied precisions. We proposed an exhaustive search method to retrieve optimum configuration of layers, where the precisions can be chosen to be FHWH, FHWB, or FBWB. We also proposed a streaming deep learning architecture where the feature maps are directly streamed to the accelerator and the output stream is stored in main memory. The advantage of streaming architecture is the pipelined nature of operations where each stage is working in parallel.

This is a work in progress and various improvements are part of our future research. First of all the search method for selective binarization should be enhanced from a simple exhaustive search. Next, we plan to enhance our architecture by adding a image-to-matrix transformation on chip, which will considerably reduce the memory bandwidth. We also plan to add support for arbitrary precisions in the future.

## REFERENCES


1. 2017. DAC-HDC-2018. (26 September 2017). `http://www.cse.cuhk.edu.hk/~byu/2018-DAC-HDC/index.html`
2. Last accessed 23/11/2018. Darknet: Open Source Neural Networks in C. (Last accessed 23/11/2018).

https://pjreddie.com/darknet/

3. Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.
4. Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 269–284.
5. Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2018. Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. *CoRR* abs/1807.07928 (2018).
6. Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 367–379.
7. Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016).
8. Miguel de Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. 2018. QUENN: QUantization engine for low-power neural networks. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 36–44.
9. Norman P. Jouppi et. al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. 1–12.
10. M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. 2015. The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision* 111, 1 (Jan. 2015), 98–136.
11. C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*. 109–116.
12. Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).
13. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105.
14. H. T. Kung. 1982. Why Systolic Architectures? *Computer* 15, 1 (Jan. 1982), 37–46.
15. Liangzhen Lai, Naveen Suda, and Vikas Chandra. 2017. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073* (2017).
16. Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.
17. Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector. In *European conference on computer vision*. Springer, 21–37.
18. Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. 2018. A Lightweight YOLOv2: A Binarized CNN with A Parallel Support Vector Regression for an FPGA. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*. 31–40.
19. Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.
20. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Vol. 00. 779–788.
21. Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. *arXiv preprint* (2017).
22. Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
23. Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.
24. Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference *(FPGA '17)*. ACM, New York, NY, USA, 65–74.