Implemetation of Hardware Accelerator for MobileNets

`

Aditya Sharma
Electrical & Computer Engineering Dept.
State University of New York at Stony Brook
Stony Brook, Suffolk
aditya.sharma.1@stonybrook.edu

Mukul Avinash Javadekar
Electrical & Computer Engineering Dept.
State University of New York at Stony Brook
Stony Brook, Suffolk
mukulavinash.javadekar@stonybrook.edu

***Abstract*—This is a project report for implementing hardware accelerator for MobileNets. It contains brief description of the project which we have implemented. It gives the goal and motivation behind the project. This paper gives an overall description of our project, architecture used in implementing the project, tasks we did in implementing the project, results and the learnings we incurred while simulating the project.**

I. Introduction

The goal of the project is to implement an existing version of mobile net architecture on single-core Zynq 7Z007S development board, learn different design techniques that can be used to efficiently implement this model. To get a design with suitable memory utilization and latency with the hardware. To get suitable trade-off in design between latency and accuracy. The motivation behind selecting MobileNets as a topic is its strong performance compared to other popular models [1]. MobileNets are advantageous as they are low weight and efficient. MobileNets are most suitable for handheld embedded devices in computer vision. They are faster to train smaller networks with low computational cost, low latency and appreciable accuracy. It is easy to deploy them on embedded devices. The key idea behind MobileNets is Depthwise Separable convolution which is different from Standard convolution and can potentially reduce computation cost. The version of MobileNets architecture in the paper we referred has 28 layers, each layer is followed by a batchnorm [2] and ReLU nonlinearity activation function with the exception of the very first layer which is standard convolution layer and final fully connected layer which has no nonlinearity and feeds into a softmax layer for classification [1]. It is presented in paper on “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications” (04/17/17) by researchers at Google Inc.

We implemented a reduced model of actual MobileNets architecture on PyTorch and observed the accuracy of the design which was found to be approximately 74%. We successfully implemented CLP-Nano architecture on hardware which can be used do depth-wise and point-wise convolution with help of software. In hardware simulation we were able to simulate the 3 layers.

II. Background

A. *Convolutional Neural Networks in Deep Learning*

To start with the project, we studied Convolutional Neural Networks (CNN) which is the basic building block of the project as part of deep learning and how DNN layers can be implemented through PyTorch. We understood different hardware architecture for CNNs, such as CLP, CLP-Nano, CLP-Lite and what are the drawbacks and advantages of each.

B. *Depthwise and Pointwise Cnvolution*

The MobileNets architecture is based on Depthwise Separable Convolutions. It is a form of factorized convolutions which factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution. It involved understanding of the two separable convolutions and how to implement them on PyTorch as well as hardware.

a) Depthwise Convolution:

Depthwise convolution applies a single filter to each input Channel. Depthwise convolution with one filter per input channel (input depth) can be written as:

$$\hat{G}_{k,l,m} = \sum_{ij} \hat{k}_{I,j,m} . F_{k+i-1,l+j-1,m}$$

where, $\hat{k}$ is the depthwise convolutional kernel of size K × K × M

Depthwise Convolution has computational cost of:

$$K \text{ x } K \text{ x } M \text{ x } Rp \text{ x } Cp$$

Depthwise convolution is extremely efficient relative to standard convolution. However, it only filters input channels, it does not combine them to create new features.

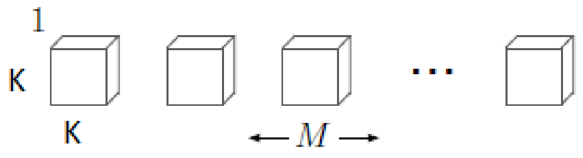


Fig. Depthwise Convolution

b) Pointwise Convolution:

It is a simple convolution with kernel size of 1 × 1. Since, Depthwise only filters depthwise input channels, pointwise is used that computes a linear combination of the output of depthwise convolution via 1 × 1 convolution is needed in order to generate the new features. The computational costs:

$$N \text{ x } M \text{ x } Rprime \text{ x } Cprime$$

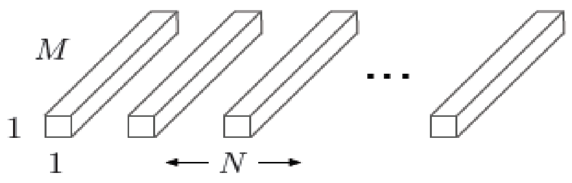


Fig: Pointwise Convolution

Therefore, by expressing convolution as a two-step process, we get a reduction in computation of:

$$\frac{K \times K \times M \times Rp \times Cp + M \times N \times Rp \times Cp}{K \times K \times M \times N \times Rp \times Cp}$$

$= 1/N + 1/Dk^2$

C. Architectural Idea

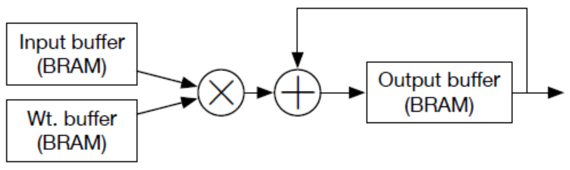
a) Software

The idea of building the MobileNets architecture came from the research paper [1] which we referred during the duration of the project. In PyTorch Our MobileNets architecture includes a standard convolutional layer followed by BatchNorm and ReLU, Depthwise Separable convolution again followed by BatchNorm and ReLU and finally passing through Avg Pooling and Fully connected layer.

b) Hardware

For this project, we are using a CNN with CLP-Nano i.e. Convolutional Layer Processor on hardware which is another architectural design essential to build the hardware design.

CLP-Nano is a simplified version of a CLP that can be used for relatively small CNN layers. As the convolution is the key computation we are performing for both pointwise and depthwise convolution it is a fair choice when we have limited memory.



It consists of:

- Input buffer: one input feature map: R'C'
- Weight buffer: one filter: K^2
- Output buffer: one output feature map: RC

At a time one input feature map and one weight feature map is fed to the design and result is stored to output feature map, final output feature map is generated once computation for all N dimensions of input feature map is completed. Software pseudo code is given below, here M is number of output feature map and is equal to number of set of weights we gave, N dimension of input feature map e.g. for a coloured image N = 3:

```
I[N][R’] [C’] // N input feature maps
W[M][N][K][K] // M x N x K x K weights
O[M][R][C] // M output feature maps
for(m=0; m<M; m++)
    Load Bias, C and K value
    for(n=0; n<N; n++)
        Load N value
        copy W[m][n][*][*] to HW weight
buffer
        copy I[n][*][*] to HW input
buffer
        tell HW to start and wait for it
to finish
    O[M][*][*] = [read values from HW
output buffer]
```

Here R', C' and K decide what should be size of our BRAMs. Below is our hardware pseudo code, counters are used in place of for loops to determine memory addresses.

```
Ibuf[R’][C’] // 1 input feature map
Wbuf[K][K] // 1 kernel of weights
Obuf[R][C] // 1 output feature map
//(R’ = R + K – 1 and C’ = C + K – 1)
wait for SW to send start signal
for(i=0; i<K; i++)
    for(j=0; j<K; j++)
        for(r=0; r<R; r++)
            for(c=0; c<C; c++)
            Obuf[r][c]+=Wbuf[i][j]*Ibuf[
            r+i][c+j]
send done signal to hardware
```

III. IMPLEMENTATION

A. PyTorch Implementation

Initially we started with implementing the MobileNets architecture on PyTorch. Our PyTorch implementation consists of a standard convolution, followed by depthwise and pointwise convolution. The standard convolution consists of 3 input nodes, 32 output nodes, stride as 1 with a filter of 3×3. This is followed by a Batch normalization and ReLU activation function. The second implemented is Depthwise layer (combination of Depthwise separable and Pointwise) which has 32 input nodes and 512 output nodes. Depthwise separable convolution uses filter of 3×3×32×1 and the Pointwise uses the filter of 1×1×32×512. Each of this layer is followed by a Batchnorm and a ReLU. There is an Average Pool layer of 7×7, finally concluding with the fully connected layer of 8192×10 feeding in to the Softmax layer for classification. Please refer the Table1. The hyper-parameters used are:
batchSize – 64, learning_rate – 0.01, momentum – 0.9, epochs – 12.

After implementing all these layers and using these hyper-parameters we observed the accuracy of our design to be approximately 74% on CIFAR-10 dataset. We successfully got the trained weights from which would be used as the weights to be fed in the hardware implementation.

Firstly, we saved the model using torch.save function and then loaded the full model when we required the weights. Since the weights were in tensor flow, we converted ndarray using numpy and accessed the weights of that particular layer.

MobileNet Body architecture

Type/Stride	Filter
Conv/s1	3×3×3×32
Conv dw/s1	3×3×32×1
Conv/s1	1×1×32×512
Avg Pool/s1	7×7
FC/s1	8192×10

Table 1

B. *Hardware Implementation*

For hardware implementation, we are using CLP-Nano as core engine for all the three layers and doing all the convolutions, full connected, depthwise and pointwise convolution through the C code. The size of each BRAM is kept to 4K and latency is kept to 6.

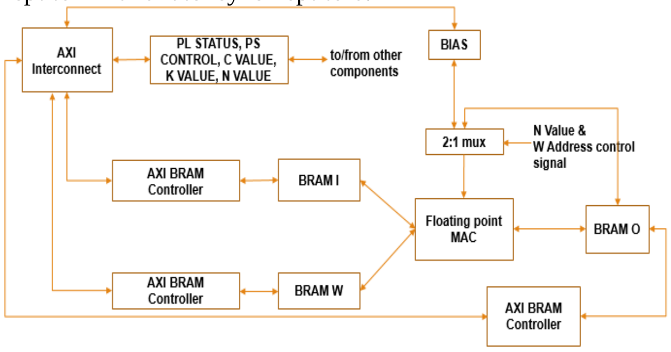


Fig: Block architecture

Firstly, for standard convolution we have an input matrix of 32×32 and 3 input feature maps. Weights (filter) of dimensions 3×3 would be convoluted with the input matrix with 3 biases (number of output feature maps) to give the output of 30×30 with 3 output feature maps.

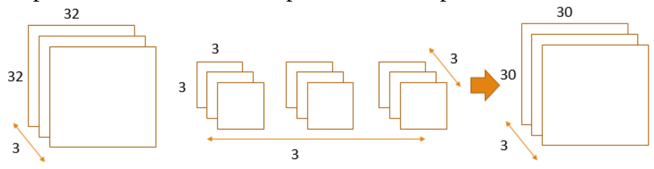


Fig: Layer1

This output of standard convolution is fed as input to depthwise convolution which would be of dimension 30×30 with filter or weights of 3×3×1×3 which will give an output of dimension 28×28.

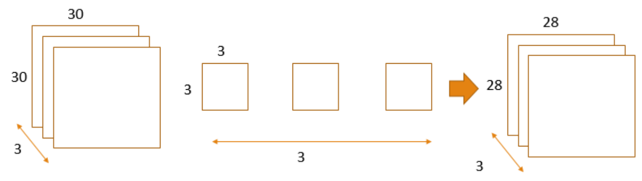
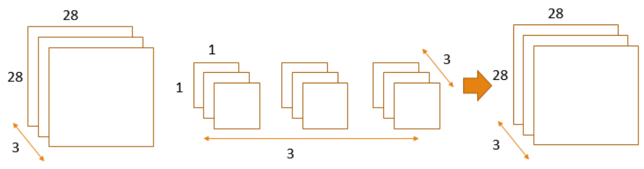


Fig Layer 2

This is given to a pointwise layer.



Since we are using CLP-Nano, there would be only 1 input feature map in the input buffer, first set of weights in the weight buffer and the first output feature map.

Description of control signals and counters:

- *counter1* – for setting read input BRAM addresses (0 to C-1)
- *counter2* – for setting read input BRAM addresses (0 to C2 – 1)
- *counter3* – for setting write output BRAM addresses (0 to C2 – 1)
- *counter4* – for knowing when final output generated (0 to C2 x K2 -1)
- *row*- for row of K x K kernel (1 to K)
- *col* – for column of K x K kernel (1 to K x K)
- *r_done* – Read addresses of one row of input matrix
- *m_done* – Read addresses of one matrix for a weight value
- *k_done* – Read all the addresses of input matrix for final output
- *c_done* – When final convolution complete
- *i_valid, w_valid, a_valid* – input to floating point for valid input data
- *w_add_g* – When W address greater than 0
- *bram_oa_we* – generated when output valid from floating point is one
- *pl_staus* – output from PL to PS
- *ps_control* – input to PL

IV. EVALUATION

A. *Method*

1) Comparison with other CLP-Nano versions

	Weights	Inputs	Outputs
V1 overhead	L1 – 3600 L2 – 3136 L3 - 3136	L1 – 108 L2 – 108 L3 – 108	none
V1 on-chip storage	0	0	0
V2 overhead	none	none	none
V2 on-chip storage	L1 – 324 L2 – 108 L3 – 36	L1 – 12288 L2 – 10800 L3 – 9408	L1 – 10800 L2 – 9408 L3 – 9408
V3 overhead	none	L1 – 12 L2 – 12 L3 – 12	none
V3 on-chip storage	L1 – 36 L2 – 36 L3 – 36	L1 – 4096 L2 – 3600 L3 - 3136	L1 – 3600 L2 – 3136 L3 – 3136

Table 2

2) We are planning to evaluate our design by checking the computational costs for Standard, Depthwise and Pointwise layer. As stated earlier, by expressing convolution as a two-step process, we get a reduction in computation costs.

Standard Convolution:

Computational costs = K*K*N*M*Rp*Cp

= 3*3*3*32*32*32

= 884736

Depthwise Convolution = K*K*N*Rp*Cp
= 3*3*32*30*30
= 259200

Pointwise Convolution = N*M*Rp*Cp
= 32*512*28*28
= 12845056

Reduced Computational cost

$$= \frac{K*K*M*Rp*Cp + M*N*Rp*Cp}{K*K*M*N*Rp*Cp}$$

= 14.811

3) The maximum frequency observed for our design was found around 12 MHz when latency is 8. The criticsl path for the design:
Source:
design_1_i/dnn_0/inst/dnn_v1_0_S00_AXI_inst/slv_reg4_reg[0]/C (rising edge-triggered cell FDRE clocked by clk_fpga_0 {rise@0.000ns fall@38.571ns period=77.142ns}

Destination:
design_1_i/dnn_0/inst/nn/lay1/bram_i_addr_reg[10]/D (rising edge-triggered cell FDRE clocked by clk_fpga_0 {rise@0.000ns fall@38.571ns period=77.142ns})

B. Contributions:

Aditya Sharma: Hardware Design, Debugging, Testing, Block Design, Presentation, Report
Mukul Javadekar: Debugging, Testing, PyTorch Implementation, Presentation, Report

References

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

[2] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167, 2015

[3] Slides referred from the class