**FPGA Acceleration for MobileNet Applications**

Kenneth Ng Bryan Moy

**Abstract**

MobileNet is a variation of convolutional neural network (CNN) aimed at low power applications. MobileNet requires less computation than standard CCNs like AlexNet and does not abstract away information to the extent that binarized networks do. With these tradeoffs, MobileNets are the middle ground and can hopefully be used to lower the cost of small applications that can fit on the MiniZed.

The goals for implementing MobileNet is to determine the network that gives us decent performance on the CIFAR10 dataset. Once the model has been established, the MobileNet structure can be split among the group members and implement parts of it for the milestone. By the draft deadline, we should have a working model and think about optimizing the network.

Keywords: input feature-map(iFM), output feature-map(oFM), configuration register (ConfigReg)

**1. Introduction**

MobileNet in contrast to standard CNNs, takes advantage of depthwise convolution and pointwise convolution to reduce the computation cost by using a different structure to reduce the number of operations required for convolution.

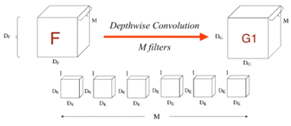
Refer to Figure 1, depthwise convolution has size and that filter is applied to all channels: for an input, F, with size an output, G1, with size is generated. The resulting computational cost will be .

Figure 1. Depthwise Convolution [1]

Refer to Figure 2, pointwise convolution has size and that filter is applied to all points in G1. The resulting output is of size . The computational cost for pointwise convolution is .

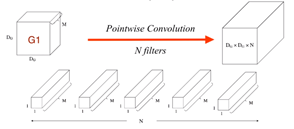


Figure 2. Pointwise Convolution [1]

The total cost for the depthwise convolution and pointwise convolution is . The resulting depthwise separable convolution is described in Figure 3.

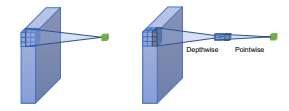


Figure 3. Depthwise Separable Convolution [3]

Compared to a standard convolution that has a computational cost , see Figure 4, the depthwise separable convolution according to the work of Howard et.al., is 9.4x less computation than AlexNet.[1]

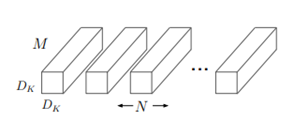


Figure 4. Standard Convolution [1]

In MobileNet V2, they introduce separable with linear bottleneck convolution and bottleneck with expansion layer which is a 1x1 pointwise convolution that maps 1:1 to another layer.

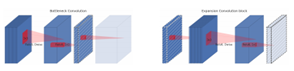


Figure 5. Bottleneck Convolution [2]

**2. Design Proposal**

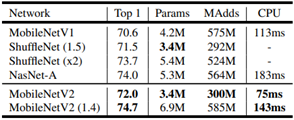
This MobileNet has two novel components: depthwise convolution and pointwise convolution The workload can be distributed so that one person implements each method. The two parts can be consolidated for the full depthwise separable convolution structure described in Howard’s paper. [1] There are multiple versions of MobileNets: for the limited time allotted, MobileNet V2 includes an extra bottleneck layer that is another 1x1 convolution after the pointwise convolution. The addition of this layer overall makes the network shallower for comparable performance; thus, decreasing the number of parameters, as seen in Table 1.

Table 1. MobileNet Performance Comparison [2]

Given that the MiniZed only has 14,400 LUTs and 1.8 Mb of BRAM, there will not be enough resources. The networks described in Howard’s work need 3.4 million parameters. MobileNet has some results for 8bit operating systems but requires many more multiply accumulators, See Table 2. We would most likely run out of LUTs and we must split the image into different regions and analyze each region.

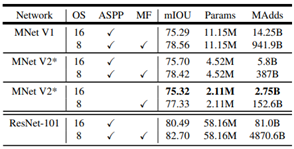


Table 2. MobileNet Resource Comparison [2]

First, we will build a network using the pytorch framework and principles from MobileNet. The CIFAR-10 dataset will be used to train and test the network before exporting the weights and implementing the network in hardware. A sample network used in Sandler’s paper is in Table 3, but as stated before, there might be a lack of resources implementing this network.

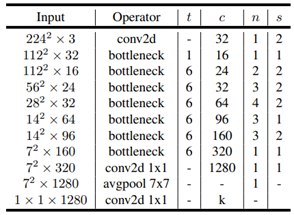


Table 3. MobileNet Sample Network

Next, once the model has been established and verified, the MobileNet structure is implemented through hardware and modules split among the group members to implement the milestone. By the draft deadline in May, we should have a working model and think about optimizing the network.

Careful consideration for FPGA resources will require potential usage of DMA, and/or other peripherals to fit the network on the board. The image will need to be serialized into a format recognizable by the network to test its’ capability.

Results will be quantified strictly through accuracy against similar sized networks.

**3. Progress**

As of **May 4th** the designs implemented and verified in both simulation and hardware include the traditional convolution, depthwise convolution, pointwise convolution, and batchnorm layers. The verification was done by cross referencing the result from the simulation with a python script that does the different convolutions in software. The test cases were relatively small with counters as initialized inputs for predictable results. Further functionality will be evaluated through the same process when we can copy and paste the result from the console to Excel and do a difference analysis. Therefore, we have all the basic components to implement the whole MobileNet system, which has already been started ahead of schedule The different layers will be controlled with a multiplexer. The remaining task is to adjust the size of the different layers and generate the C code that controls the different layers. We will then further test MobileNet by inserting an image, and use the result for the fully connected layer and measuring our system’s accuracy.

As of **May 15th** all forms of convolution work in both simulation and hardware and are organized using embedded ‘C’ memory structures. A batchnorm layer is available and the system can operate correctly on 16-bit input layers. Actual assembly is limited due oversighted quantization issues. Timing analysis was performed to see overheads due to PS operation and memory management. Convolution was performed on a 32 x 32 x 8 x 8 input and produced expected outputs. The system as is, could operate on 16-bit inputs with rounding performed for 32-bit results but it’d be better to modify hardware to an appropriate output value than patch the existing issue.

**4. Architecture**

The system has the assumption that DRAM will hold an input image at run time. For purposes of this system, it will just be stored in a ‘C’ int[] array.

The design is partitioned into 4 major modules:

* Normal Convolution
* Point-wise/Expansion convolution
* Depth-wise convolution
* Batchnorm (ReLU integration)

These modules are repeated in MobileNet’s 26 layers (Actually 58 when unfolding each bottleneck layer). Each module reads and writes from a single BRAM port. The read (iFM, weights) and write locations (oFM) are stacked on one another in memory. The system requires a PS to regulate interactions and will configure each layer at runtime from a predetermined lookup table in the PS memory. This table of *configReg* objects determines the operating dimensions for a convolution core when *calc()* is called. The PL will implement the config protocol through a multiplexer in the PL to select which layer is reading and writing to memory and what input layer sizes it should expect to work with. The pointwise and expansion layers are integrated into standard convolution and the depthwise can be implemented through that standard convolution but currently remains as a standalone module.

Each convolution layer was designed with seven states:

State 0: Wait for ps\_control == 1 (PS start signal)

State 1: Activate datapath; stay until we are issuing  
 last read request for the next weight

State 2: Load weight, W, from BRAM if it is not  
 the first weight, go to buffer at state 5

State 3: Load element from feature map, x, from  
 BRAM

State 4: Write to memory

State 5: Serves as a buffer when a new weight is  
 read

State 6: Write status = 1 (done); wait for  
 acknowledgment from PS   
 (ps\_control == 0)

The architecture was designed to hold a weight and use the weight for all calculations before using the next weight. We will further discuss how to tile the different feature maps for each layer and determine the best approach to fit the entire system on the PL.

The control will be done through a 32 bit AXI-lite. The lowest nibble will be dedicated to selecting an operating layer. A single bit for “ready/busy” is allocated for control synchronization. The upper 27 bits will be used to configure expected input sizes, channels sizes and kernel sizes with bit sizes determined by the max layer dimensions of mobilenet V2 specifications. With information from that, the block should understand the expected output and calculate appropriate addressing offsets to avoid overflow/overwriting (Fig Appendix - 2).

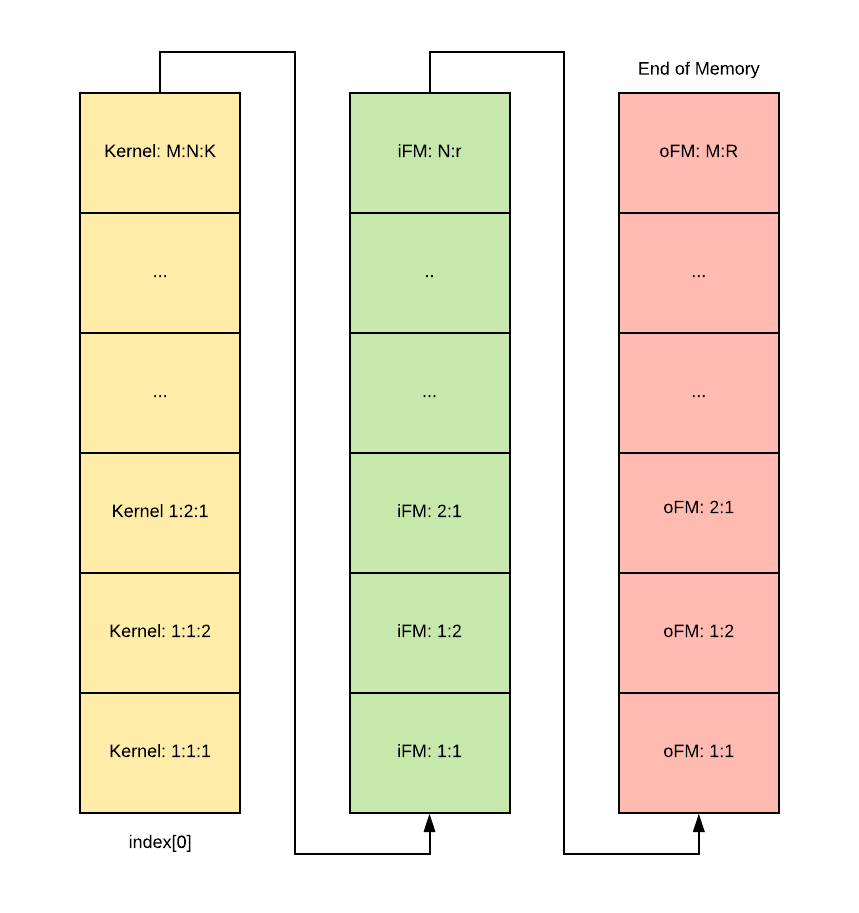


Figure 6: Single port Memory Organization

Memory is stacked on top of one another (Fig 6) The Kernel sits at the bottom (index[0]), iFM on top of that and oFM accumulates at the top. The control path for convolutional core pulls the operands serially through a single port.

The *averagePool()* and *batchnorm()* can be done in software since the runtime complexit is always going to be O(N) (our memory elements are adjacent and single-ported).

Minimum size 3x3 tiling is used to actually fit large layers like the 144 x 32 x 32 layer. Memory organization and overhead contribute significantly to run time when operating on layers this large.

**5. Quantization**

The pytorch framework used on desktop workstations uses IEEE-754 floating point numbers. Due to the complexity of floating point arithmetic, the system was modified to support 32 bit inputs and perform fixed point arithmetic. As a result of this change, there will be a quantization error when compared against the original system but it is not expected to cause substantial change in system operation.

Since the minimum value of a 32-bit signed integer with normalized to maxima/minima of +1/-1, with a max resolution of approx ~10^-10, floating point values below 10^-11 are remapped to zero.

Due to oversights in operation in ***section 6*** the current quantization poses some issues: either software overhead to convert 32-bit results back to 16 or overflow errors for results greater than 16 bits being fed into the input of a next layer.

**6. Bugs and Issues**

There was an oversight in the remapping of input values to quantized 32-bit fixed point integers. Multiplication of floating point values will yield 32-bit floating point values unless operating at order of extreme conditions (10^38) but 32-bit multiplication can yield a 64-bit product. As a result, our system faces overflow issues. This can be mitigated in a variety of data-type modifications.

* Switch to 32-bit floating point to match pytorch implementation exactly but risk performance losses due to inherent complexity of floating point
* Internally compute a 64-bit result and round back to 32-bit for next layer input

The batchnorm layers were moved to software due to lacking resources for single cycle square root arithmetic. Alternate solutions to this:

* If implemented on a larger FPGA, this layer might be feasible to accelerate
* Use a CORDIC-IP core
* Split the algorithm up among several clock cycles

**6. Considerations for optimizations**

The first iteration of this design will be used for proof of concept that the MiniZED resources can reasonably support MobileNet applications. Once the design is verified in its first configuration optimizations, it can be improved in parallelism and resource utilization.

Each layer as is, accesses all data through a single BRAM interface, limiting the bandwidth. Allocating three separate BRAM locations for input feature-maps, weights and output feature-maps will allow simultaneous read/write and reduce the transfer overheads for all layers. Since BRAM can only be set up in powers of two, the current configuration is capped at 128KB since the miniZED only supports ~229KB of BRAM. Splitting the memory could mean larger input feature maps support. oFM and kernel sizes from a given tile are often not very large and can make use of the ~100KB that is unused.

Since batchnorm is a common layer ( occurs after every single convolution) The integration of batchnorm into the convolution units could parallel compute the running average for a given output at the same time it computes its final output values for the convolution. This eliminates the need for ‘average-calculation’ in the batchnorm module (⅓ of transfers for this layer in both software and hardware). The improvement could provide at most a ~33% improvement in transfer overhead since the synthesizable operating frequency could change.

With intelligent usage of tiling, the design could be accelerated by a factor of P, for however many P parallel convolutional cores will fit with the remaining LUTs. The system will see the greatest speedups if the “bottleneck” layers can be parallelized efficiently (particularly depthwise convolution which works with expanded planes). A single convolutional core uses ~3800 of the available 14400 LUTs. This makes it possible to perform parallelization of P(2) tiling of a given iFM and possibly more if the convolutional core is optimized.

Double buffering/pipelining inputs cannot be implemented as easily since the system is processing 58 layers from a single input and reusing certain layers. A layer would only be allowed to take that 2nd input from this double buffer if the layer will not be used again for the previous input. If it were implemented, the memory sizes might become prohibitively large compared to the slight improvements in overhead since the bottleneck can only be double-buffered for certain layers. Bottleneck layers cannot be double buffered effectively because they are reused 17 times.

The point-wise convolutions makeup +90% of MAdds and +70% of all weights according to [2]. Parallelizing just this portion of the bottleneck layer should provide substantial increase in throughput.

Operating speed was limited to about ~9MHz due to heavy arithmetic that was implemented for calculating addressing. Simplification of these counters to reduce fanout and computational complexity should provide the system with a 5 - 6x improvement in operating frequency based on non-configurable synthesized convolutional layers (parameterized using *parameter* rather than AXI-Lite control interface).

**7. Evaluation**

The design’s performance can be evaluated through :

* Resource usage
* Throughput/Latency

With emphasis on resource usage and throughput/latency since mobile applications need small, low power solutions. The purpose of this implementation is to display the potential for FPGA-accelerated neural networks in the mobile market, so operations performed through the ARM-core will be evaluated separately. When compared against a desktop CPU using floating point data-types, the current state of the system is much slower in performance. The reasoning being that the frequency is ⅕ the anticipated frequency and other overheads and inefficiencies cause throughput bottlenecks. This can be resolved through some redesign targeted at synthesis and restructuring of memory. Given the operational speeds of FPGA as is, it could still be comparable in performance to desktop application but future testing, especially on larger input sets is needed.

The convolutional core in this design uses about ⅓ of the given LUTs and is primarily limited by this resource. Another convolutional core could be fit onto the FPGA or the design could be implemented on a slightly smaller FPGA.

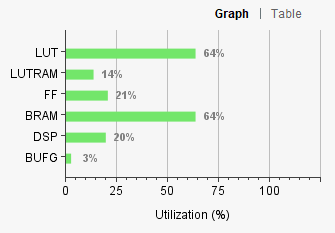
All design modules worked in both simulated hardware and on actual hardware.

Figure 7: Relative Resource utilization

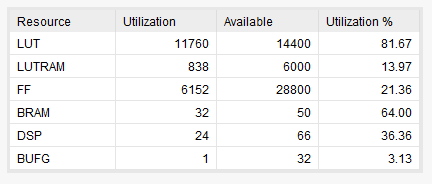


Figure 8: Absolute resource utilization

**8. Future Work**

Design foundations were established for mobileNet V2 to be implemented on an FPGA. Timing metrics make clear that overheads in transfers are substantial portions of the actual computing problem and it is not yet fully clear if this implementation on the MiniZED is a viable accelerator for Mobilenet V2. The assembly of the CNN can be done in the state the system is in but will likely perform poorly. In order for the system to be viable for actual mobile application, optimizations such as pipelining, memory restructuring, tile parallelism and memory access parallelism will help speed operation substantially.

Improvements to the system that could be tested would be adjusting the model in pytorch to support a wider set of input classes beyond the CIFAR-10 data set. Taking on larger training/testing sets will make the design more viable as a commercial solution. This requires only changing the weights.

Memory management and organization schemes for memory have massive impacts on system operation. Rudimentary/primitive memory organization structures can cause as much as ~50% run time latency.

Interesting alternate implementations that would require few changes in HDL code would be varying data-type modifications: How does the system work with 64-bit arithmetic, 32-bit floating point, 64-bit double floating point, and a much lower resolution implementation of maybe 24 or 16 bit?

Given that most weights and computations exist within the bottleneck layers, these layers should be targeted for maximizing efficiency.

**Contributions**

Kenneth Ng

* Pytorch implementation of mobileNetV2
* Pytorch layer timing analysis
* MobileNet Layer analysis
* Depthwise convolution layer
* Pointwise convolution layer
* Normal convolution layer
  + Testbenches for all above
* AXI- timer implementation
* Test image shrinking and quantization

Bryan Moy

* SDK assembly, memory organization and controls systems for convolutional cores
* Batchnorm layer
* Multi-memory convolution ( Simulation only)
* Single memory convolution (lite-weight loops; Simulation only)
* Weight remapping to 32-bit fixed point
* Implementation of configurable AXI-Lite registers for controlling convolutional layers
* AXI-timer analysis of total system vs FPGA acceleration

**4. Works Cited**

[1] A. G. Howard, M. Zhu, B. Chen, W. Wang, T. Weyland, M. Andreetto, H. Adam, and D. Kalenichenko, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.”

<https://arxiv.org/pdf/1704.04861.pdf>

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks.”

<https://arxiv.org/pdf/1801.04381.pdf>

[3] Y. Guo, Y. Li, L. Wang, and T. Rosing, “Depthwise Convolution Is All You Need for Learning Multiple Visual Domains,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 8368–8375, 2019.

<https://www.aaai.org/ojs/index.php/AAAI/article/view/4851/4724>

[4] Open Source Verilog Square root operation <https://verilogcodes.blogspot.com/2017/11/a-verilog-function-for-finding-square-root.html>