Automated flow for compressing convolution neural networks for efficient edge-computation with FPGA

Farhan Shafiq, **Takato Yamada**, **Antonio T. Vilchez**, and **Sakyasingha Dasgupta**

LeapMind, Inc.
Tokyo, Japan
{farhan, yamada, antonio, sakya}@leapmind.io

Abstract

Deep convolutional neural networks (CNN) based solutions are the current state-of-the-art for computer vision tasks. Due to the large size of these models, they are typically run on clusters of CPUs or GPUs. However, power requirements and cost budgets can be a major hindrance in adoption of CNN for IoT applications. Recent research highlights that CNN contain significant redundancy in their structure and can be quantized to lower bit-width parameters and activations, while maintaining acceptable accuracy. Low bit-width and especially single bit-width (binary) CNN are particularly suitable for mobile applications based on FPGA implementation, due to the bitwise logic operations involved in binarized CNN. Moreover, the transition to lower bit-widths opens new avenues for performance optimizations and model improvement. In this paper, we present an automatic flow from trained TensorFlow models to FPGA system on chip implementation of binarized CNN. This flow involves quantization of model parameters and activations, generation of network and model in embedded-C, followed by automatic generation of the FPGA accelerator for binary convolutions. The automated flow is demonstrated through implementation of binarized "YOLOV2" on the low cost, low power Cyclone-V FPGA device. Experiments on object detection using binarized YOLOV2 demonstrate significant performance benefit in terms of model size and inference speed on FPGA as compared to CPU and mobile CPU platforms. Furthermore, the entire automated flow from trained models to FPGA synthesis can be completed within one hour.

1 Introduction

Deep Convolutional Neural Networks (CNN) have achieved significant results in computer vision, speech recognition and language translation. However the computation and memory demands of recent CNN architectures require powerful GPUs, distributed CPU servers, Specialized ASIC or DSP processors. The size and power requirements of such platforms restrict the wide-spread adoption of CNN models for efficient edge computing, mobile devices and the Internet of Things in general. Interestingly, recent results on compression of these models using a mix of techniques like pruning[3], slimmed down architectures[4], [5] and quantization to low bit-width, especially ternarized [6] and binarized neural networks [2] has shown that model size can be reduced dramatically while maintaining reasonable levels of accuracy. Furthermore, low bit-width CNN are particularly suitable for FPGA based acceleration due to bitwise operations involved in such models. As such quantized and compressed models can be effectively deployed on mobile devices based on such FPGA or SoC acceleration. In this paper, we present an automatic flow from TensorFlow [1] trained CNN models to quantized CNN implementation on FPGA SoC, enabling efficient inference. Initially a full precision model is quantized to 1 bit weights and 2 bit activations, followed by retraining on the original dataset.

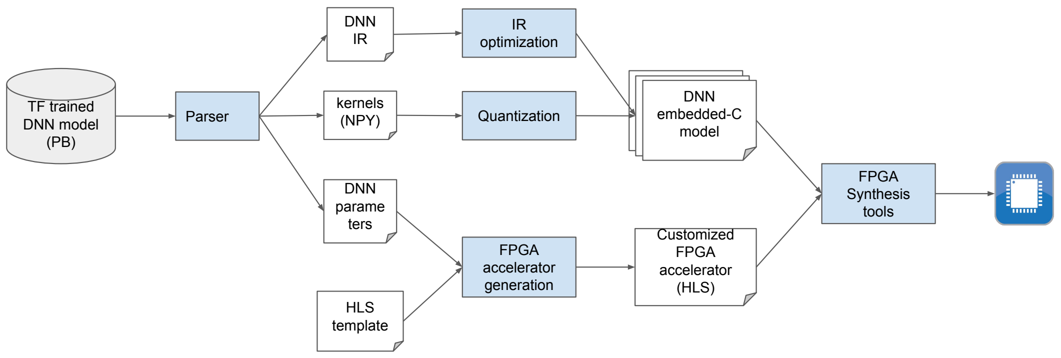


Figure 1: TensorFlow trained CNN model to FPGA implementation flow

Following this, the flow starts with the fully trained quantized model exported as Tensorflow protocol buffer format. This is followed by, the model being parsed and relevant graph transformations being applied. Embedded-C code for the quantized network is then generated and high level synthesis (HLS) implementation of the FPGA accelerator is customized for the quantized network. This is done using automated scripts that consider the model's memory requirements and computation complexity in order to choose the suitable level of parallelization and local memory usage. Figure 1 depicts the block diagram of this flow. Currently the FPGA accelerator design is done with some input from humans rather than a full design space exploration. There have been a few other works in this area complementary to our approach, namely, Yaman et al. [8] presented automatic generation of FPGA accelerators from customizable Xilinx HLS templates.

2 Model parsing

The starting point of our framework is a fully trained model obtained from a specific deep-learning framework (e.g. TensorFlow, PyTorch, Caffe). In the case of TensorFlow, the model is first serialized into a binary protocol-buffer file. This contains both the computational graph and the model parameters. The key aspect of our framework is the automatic and transparent management of quantized activations and weights. If quantization is used during training the corresponding subgraphs will be pruned and replaced by their bit-wise counterparts, which are much more efficient for inference. Figure 2. shows an example of two consecutive convolutions with a subgraph that describes a linear operation between them. In this case the kernel quantization subgraph can be deleted. These subgraphs contain constant tensors of real numbers in 32-bit floating-point precision. However, as the training can be performed on the binary quantized CNN model, the weights in each layer can be packed efficiently. This allows a single 4 byte word to contain up to 32 weight values. This drastically improves the model size, enabling the weights to be represented as standard arrays in C-code. Furthermore, this effectively eliminates the additional time previously required for loading weights and enables the generation of compact, self-contained inference units.

3 FPGA based acceleration

Post embedded C code generation, potential parts for FPGA optimization are mapped to an FPGA accelerator. In general, CNN convolutional layers, especially binary convolutional layers are inherently suitable for parallelization and FPGA acceleration. The accelerator design space is limited, by the amount of computation resource on as well as the maximum data bandwidth available. Recent FPGA

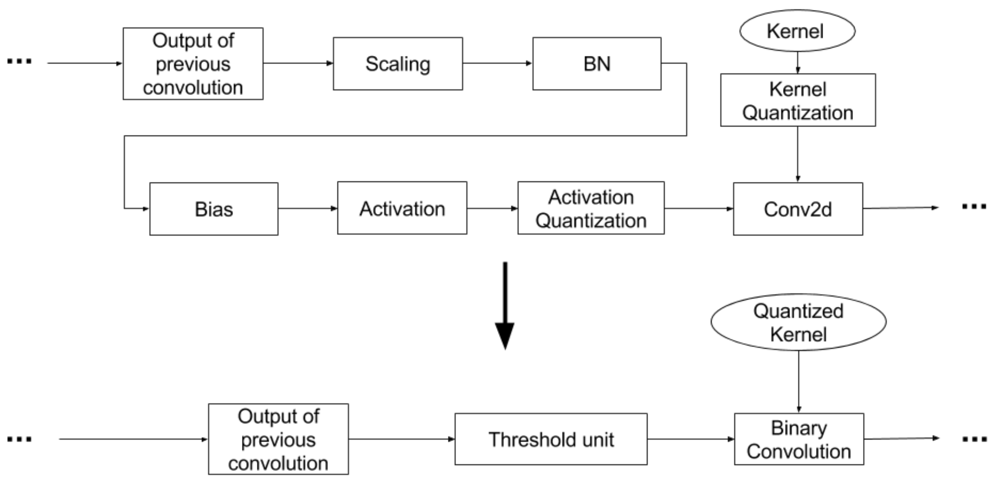


Figure 2: Subgraph between consecutive convolutions, replaced by a threshold unit.

implementations of CNN accelerators address the memory bandwidth problem by allocating on-chip memory space for all kernels, inputs and outputs [8], thus minimizing off-chip communications. Such an implementation, although effective, requires large on-FPGA RAM blocks and consequently, expensive FPGAs. Hence accelerator parallelization approaches are heavily dependent on the target FPGA device's computation resource, size of FPGA RAM blocks and the memory bandwidth.

3.1 Strategy for parallelization

We employ a scalable parallelization technique where the accelerator consists of the following building blocks.

- **Processing Element (PE)** Multiple kernel elements are packed into a single word (32bit). In case of 1-bit kernel elements a single processing element (PE) can process 32 kernel elements in parallel. Each PE is followed by a 32-bit accumulator and it exploits intra-kernel parallelization.
- **Processing Engine (PEN)** Multiple kernels can be processed in parallel. This can be achieved by employing a matrix of PEs processing the same input element and an element of different kernels in parallel. This exploits inter-kernel parallelism and increases input reuse.[1]

3.2 Design assumption

The FPGA accelerator is customized for the given network architecture and target FPGA device, in order to exploit the right parallelism. We assume that (1) Number of output feature maps is a multiple of 8. (2) Number of input feature maps is a multiple of 16 and (3) On-chip memory is limited. Assumptions (1) and (2) are in accordance with most of the popular network architectures in use while (3) is in accordance with the cost sensitive edge computing application. No assumptions are imposed on input/activation size.

3.3 Accelerator generation

The accelerator generation involves the following three steps: (1) Customize the basic building block i.e. Processing Element (PE) depending on bits per element for kernel and input. (2) Customize the Processing Engine (PEN) in the form of a matrix of (PEs). Number of (PEs) can be from 16 up to min($depth_i$) where $depth_i$ is the depth dimension of any layer's input (3) Automatically calculate other related parameters and control code. Figure 3 shows a system block diagram.

[1]It is also possible to exploit intra-input parallelization, however we do not discuss it here.

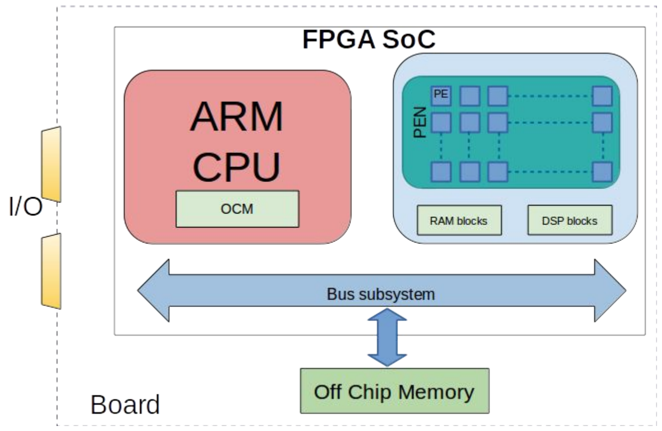
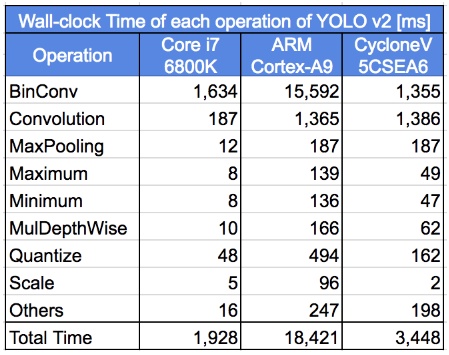


Figure 3: FPGA SoC platform block diagram with parallel processing elements



Wall-clock Time of each operation of YOLO v2 [ms]			
Operation	Core i7 6800K	ARM Cortex-A9	CycloneV 5CSEA6
BinConv	1,634	15,592	1,355
Convolution	187	1,365	1,386
MaxPooling	12	187	187
Maximum	8	139	49
Minimum	8	136	47
MulDepthWise	10	166	62
Quantize	48	494	162
Scale	5	96	2
Others	16	247	198
Total Time	1,928	18,421	3,448

Figure 4: Wall-clock time (ms) for each operation of YOLO v2

3.4 Data order optimization

Input, output and kernels for CNN can be visualized as 3-dimensional data arrays. In typical frameworks these arrays are stored in the order of $Depth \times Width \times Height$ (height dimension is updated first) or $Depth \times Height \times Width$. We propose to order the input, output and kernels in $Height \times Width \times Depth$ or $Width \times Height \times Depth$ (depth dimension is updated first) dimensions. This Depth-first ordering has a few merits as discussed below.

3.5 Memory bandwidth optimization

Memory bandwidth from FPGA device to off-chip DRAM is a usual performance bottleneck. This bottleneck can be relaxed by using burst transfers. However, the burst size is limited by the continuity of the memory addresses being accessed. The proposed ordering maximizes the burst size and improves effective memory bandwidth as a result. Figures below highlight the merit of depth first ordering.

W-bar and D-bar:

Figure 5 shows the element-wise overlap of input and kernel and respective sizes of W-bar and D-bar. Assuming that input and kernel consist of $Ih \times Id$ and $Kh \times Kd$ different width-wise arrays with Iw and Kw elements each respectively. These arrays are termed as “W-bar”. In contrast, "D-bar" signifies the $Ih \times Iw$ and $Kh \times Kw$ different depth-wise arrays with Id and Kd elements each respectively.

External and Local Memory access:

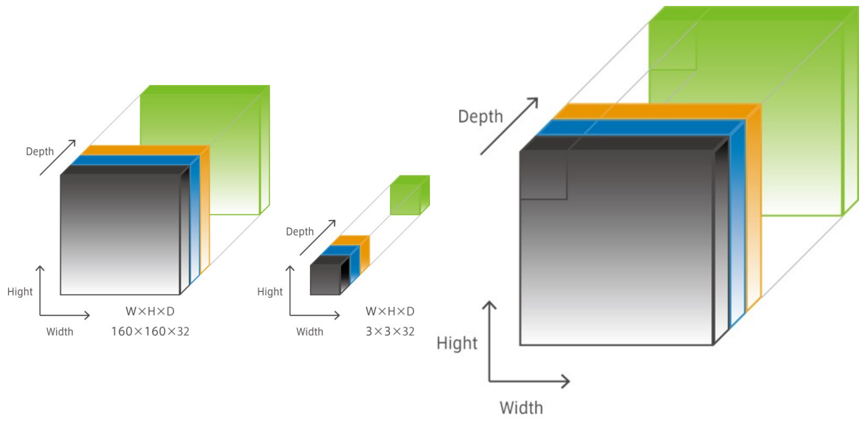


Figure 5: Element-wise overlap of input and kernel

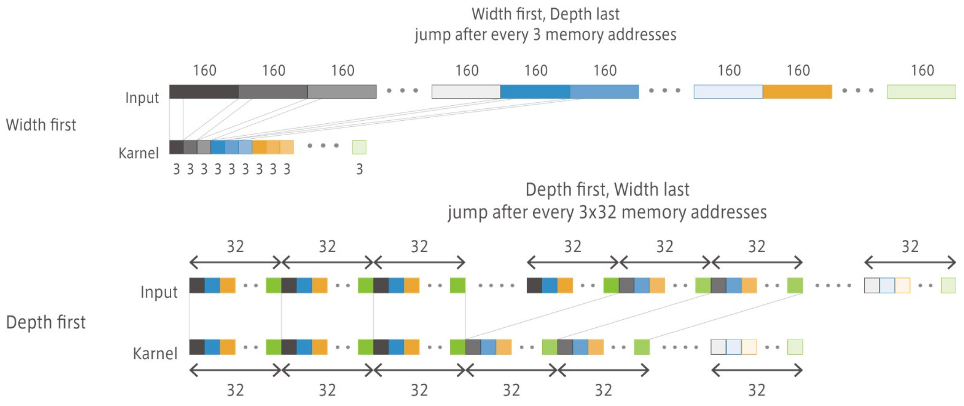


Figure 6: Differences in memory access based on data-ordering

Figure 6 depicts the difference of memory access continuity for width-first ordering as opposed to depth-first ordering.

- **Input from External memory:** The kernel W-bar overlaps with only Kw elements of the input W-bar at a time, which results in a jump in the memory addresses of the input when the next kernel W-bar is processed. In total over the course of a $Kw \times Kh \times Kd$ kernel, there are $Kh \times Kd$ jumps in the memory addresses of the input. On the other hand, the kernel D-bar overlaps with $Kd \times Kw$ elements of the input W-bar at a time, which results in only Kh jumps over the course of a $Kh \times Kw \times Kd$ kernel resulting in a longer continuity in memory addresses and better burst performance.
- **Local memory bit-packing and Processing Elements (PE):** As shown in Figure 7, The depth-wise ordering also results in a cleaner and efficient implementation of bit-packing for quantized values on the local RAM blocks, eliminating the need for multiple local memory accesses and masking operations. For example, assume that a single processing Element (PE) can process 32 binary inputs at a time. Proposed ordering helps avoid multiple memory access and bit-masking operations that would be necessary with traditional ordering. Moreover, the proposed ordering enables coarse-grain access of the local RAM (1 D-bar at a time as opposed to the fine-grain 1 W-bar or 1 byte at a time). This results in optimization of the local RAM access circuitry.
- **Inter-kernel parallelism:** Processing multiple kernels in parallel (inter-kernel parallelism) is an effective way to increase processing parallelism as well as increase input reuse. The same input is convolved with multiple kernels, getting output elements (at the same index) for multiple feature maps (output channels). With inter-kernel parallel processing, the order output elements are calculated, is naturally in the proposed depth first order. If the output is being saved on the local memory, proposed order enables a cleaner memory write circuitry.

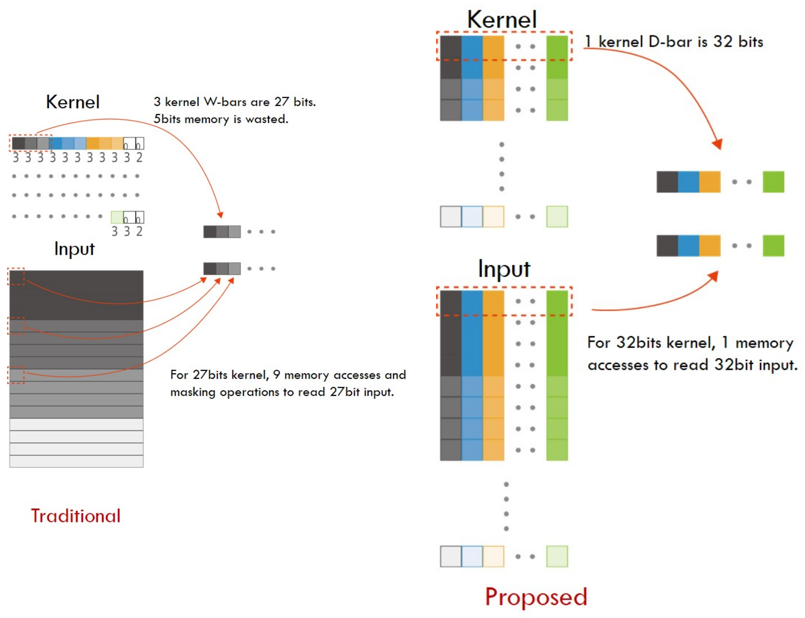

Kernel
3 kernel W-bars are 27 bits.
5bits memory is wasted.
3 3 3 3 3 3 3 3 3 2
3 3 2
Input
For 27bits kernel, 9 memory accesses and masking operations to read 27bit input.
Traditional
Kernel
1 kernel D-bar is 32 bits
Input
For 32bits kernel, 1 memory accesses to read 32bit input.
Proposed

Figure 7: Efficient implementation of bit-packing on local RAM blocks with proposed data-order optimization.

If these outputs are written directly to off-chip RAM, the proposed ordering enables better burst performance.

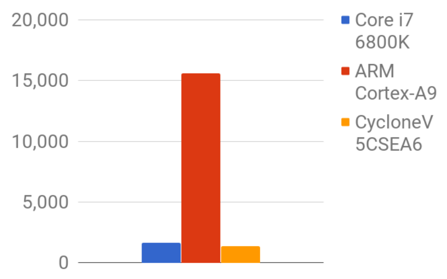


Figure 8: Wall-clock time (ms) for binary convolution

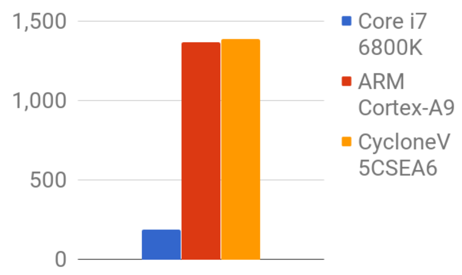


Figure 9: Wall-clock time (ms) for float convolution

4 Evaluation

This section describes experimental environment setup and performance results from a benchmark CNN architecture.

We perform evaluation of our framework, by implementing YOLO v2 [7]. The network model mostly follows original Darknet-19 design but the input size is 320x320. The network was trained on PASCAL VOC 2012 using Tensorflow and converted into C program using our generation flow. "Weights" and "Activations" are quantized to 1bit and 2bit respectively(first and last layer are not quantized) resulting in a 32x smaller model (original Yolo v2 at 255.82 MB, compressed Yolo v2 at 8.26 MB). The generated C program is built for three separate cases: CPU(Core i7-6800K), Mobile CPU (ARM Cortex-A9), Mobile CPU with FPGA (Cyclone-V 5CSEA6 SoC containing ARM Contex-A9). Every runtime was compiled by g++ with –O3 option.

In the FPGA-SoC case, binary convolution accelerator was generated using our customization and generation flow using Altera HLS compiler and synthesized as a hardware accelerator on the Programmable Logic. A comparison of performance on the three devices are reported. Figures 4, 8 and 9 show the wall-clock time of each operation running on each device. In terms of binary convolution operation (BinConv-Core) which is accelerated by FPGA, Mobile CPU with FPGA case achieves up to a x11.50 and x1.21 speed up over only Mobile CPU and normal CPU case respectively. In total, the FPGA results 5.34 times faster than the case using only Mobile CPU but it's 1.78x slower than corei7 CPU. The flow from a trained TF model to FPGA synthesis takes roughly around an hour.

5 Conclusion

This paper presented an automated flow from trained Tensorflow DNN models to Binarized (quantized) FPGA SoC implementation targeting size, cost and power constrained edge computing applications. An FPGA accelerator for quantized convolution is generated with accordance to specific constraints. A data-ordering optimization is proposed for improved memory performance. The proposed automated flow was evaluated based on implementation of the state of the art YOLO-V2 object detection framework. The resulting implementation achieves comparable performance to corei7 CPU. However, further improvements are needed for real time applications. In future works, further improvements of FPGA acceleration and SoC platforms will be explored.

References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *OSDI* (2016), vol. 16, pp. 265–283.

[2] COURBARIAUX, M., HUBARA, I., SOUDRY, D., EL-YANIV, R., AND BENGIO, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[3] HAN, S., MAO, H., AND DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[4] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[5] IANDOLA, F. N., HAN, S., MOSKEWICZ, M. W., ASHRAF, K., DALLY, W. J., AND KEUTZER, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[6] LI, F., ZHANG, B., AND LIU, B. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[7] REDMON, J., AND FARHADI, A. Yolo9000: better, faster, stronger. *arXiv preprint arXiv:1612.08242* (2016).

[8] UMUROGLU, Y., FRASER, N. J., GAMBARDELLA, G., BLOTT, M., LEONG, P., JAHRE, M., AND VISSERS, K. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2017), ACM, pp. 65–74.