

On the Design of an Object Detection System using Embedded Reconfigurable Logic: Lessons Learned

Daniel Granhão, Guilherme Carvalho, João Canas Ferreira and José Carlos Alves
INESC TEC and Faculty of Engineering of the University of Porto
Porto, Portugal 4200-465
Email: {granhao, gcarvalho, jcf, jca}@fe.up.pt

Abstract—Due to the need to move deep learning computation to the edge, the organizers of the Design Automation Conference (DAC) have launched several editions of the System Design Contest (SDC), focusing on the implementation of neural networks on embedded hardware featuring an FPGA. We describe our approach and gained experience participating in the 2020 edition of the DAC SDC contest, which targets a neural network based object detection algorithm.

I. INTRODUCTION

Convolutional Neural Networks (CNNs), like many other deep learning models, have been able to achieve astonishing results in many applications, but remain computationally complex algorithms that generally require significant computing and energy resources. More complex models are being pursued everyday as they allow harder problems to be tackled. These circumstances are in direct opposition to the need to have neural networks running on the edge. Computing deep learning algorithms such as CNNs directly on embedded devices is required in many potential applications in which access to the internet is not possible. Reconfigurable hardware platforms, such as FPGAs, allow CNNs and other deep learning algorithms to be computed more efficiently and therefor more easily deployed on embedded devices.

In this forum article, we describe our approach, as well as the learning experience we benefited from by participating in the DAC SDC 2020 contest [1]. In this contest, the goal was to implement a neural network based object detection algorithm on an embedded system and submissions were evaluated on detection accuracy and power, while having to meet a throughput requirement. The target embedded platform is an Ultra96-v2 board, featuring a Xilinx Zynq UltraScale+ MPSoC ZU3EG. **Spoilers:** we ended up not being able to implement a working solution in time for the contest and have since moved on to other projects, but we think the process we went through is worth sharing. We went into this challenge without any significant prior experience with neural network compression and adaptation for FPGA implementation neither with Vivado High-level Synthesis (HLS) development flow, which we set out to employ due to its faster development cycle.

II. OUR APPROACH

Our starting point was the DeepZ neural network [2], an adaptation of YOLOv3-tiny to our problem specifications. Namely, the original YOLOv3-tiny is able to simultaneously detect multiple objects and also classify them. In our case, we only needed to detect one object in each frame and classification was unnecessary.

A. An even tinier YOLOv3-tiny

Given that access to data on external memory is very energy intensive, we focused on reducing the size of the model with the goal of storing all parameters in on-chip memory. The original model's size is 44.5 MB while total on-chip memory on the FPGA is only 1.175 MB so we needed to apply a series of hardware-friendly changes to the model to drastically reduce its size.

a) Architecture Change: The first change applied to the model was the removal of 3 of its convolutional layers. This did come at the cost of 5.73% in accuracy, but it allowed the parameters to be reduced from 44.5 MB to 11.4 MB.

b) Network Slimming: Next, we employed the slimming approach described in [3]. Using this method, we were able to prune entire channels in each convolutional layer, reducing total parameter size from 11.4 MB to 2.6 MB at the cost of a small decrease in accuracy (2%).

c) K-Means Clustering of Parameters: The k-means clustering algorithm can be applied to encode parameters, therefor effectively reducing the number of bits of each parameter to \sqrt{K} , where K represents the number of quantization levels. We were able to reduce the number of bits per parameter from 32 bits (floating point representation) to 4, at the cost of making the model another 2% less accurate. The final size of the parameters became 0.325 MB but this does not include the need for codebooks that are used for the decoding of the 4-bit weight representations to their original size.

d) Accelerator-aware Pruning: Finally, we also explored the use of accelerator-aware pruning [4]. This compression approach manages to remove many individual parameters without negatively impacting the hardware implementation, which usually happens when pruning is employed. This is achieved by grouping weights and always leave a constant number of non-zero weights in each group, avoiding the load-imbalance problem. Pruning the largest convolutional layer by 50% reduced overall accuracy by 0.3%. We were not able to use HLS to design hardware that allowed the simultaneous use of this method with K-Means.

B. Adaptation Strategies

Besides the model compression techniques introduced in section II-A, a few other hardware-oriented strategies were pursued in order to more efficiently utilize the reconfigurable hardware resources. As a rule of thumb it is desirable to reduce the number of resources as much as possible, particularly the number of DSPs since they are scarce and power hungry.

a) *Offline Batch Normalization*: The original network employs batch normalization layers to accelerate learning and improve accuracy by normalizing and re-centering the input of each layer. Batch normalization layers involve division and squared root operations, which are not hardware friendly. We hid these operations by absorbing them into the parameters of the previous convolutional layer.

b) *Fixed-Point Representation*: Floating-point precision is the widely favored numeric representation in state of the art microcontrollers and desktop CPUs, however the same does not apply to FPGA systems, where most of the times fixed-point is the sole arithmetic that is natively supported by DSP slices. It is thus usually advisable to use fixed-point representation in FPGA-based systems. Naturally, an accuracy loss is to be expected when converting floating-point to fixed-point operation but neural networks are naturally error-tolerant.

c) *Activation Function*: We changed the original activation function used in the network, namely the Leaky Rectified Linear Unit (ReLU). The leaky parameter of the ReLU was adjusted from 0.1 to 0.125, allowing it to be implemented using a simple bit-shift, and the output was capped at the value of 6, effectively limiting the value range and decreasing precision errors when using a fixed-point representation.

d) *Winograd Fast Convolution*: The Winograd fast convolution is a method for computing the convolution between two vectors and is particularly hardware friendly when using small filters and unitary strides, allowing a significant reduction in DSPs versus the regular naive convolution. Furthermore, max pooling with stride and dimension 2 can be performed at virtually no additional cost, saving both execution time and resources.

C. Hardware Implementation

As previously said, we did not manage to implement the complete network. Despite that, we have partial results from the implementation of individual layers. In table I we present the resource usage for both a Winograd-based and a naive-based implementation of the largest layer in the network.

TABLE I. HLS SYNTHESIS RESULTS FOR THE LARGEST LAYER (N=208) FOR THE WINOGRAD AND NAIVE ARCHITECTURE

Resource Type	Winograd		Naive	
	Total Used	Usage (%)	Total Used	Usage (%)
DSP48E	18	5	167	46
FF	2235	1	23986	16
LUT	4619	6	29481	41
BRAM_18K	96	22	222	51

The Winograd-based implementation has the upper hand over the naive when it comes to resource usage, however, comparing throughput, the naive approach managed to achieve a speedup of around $1.39\times$ over the Winograd.

To compute the different layers in the network, we planned on using the dataflow HLS directive, which allows task-level pipelining, proving its viability with 2 layers. We found that without a deep restructuring of our approach, we would not have sufficient FPGA resources to finalize the implementation.

III. LESSONS LEARNED AND UNANSWERED QUESTIONS

Learning to use Vivado HLS was overall rewarding and we found the general development flow to be intuitive. Despite that, several interesting quirks stopped us from moving faster.

a) *Loop Unrolling Issues*: It is not possible to unroll a loop without unrolling all loops contained within that loop. From our point of view, only having experience with RTL development, we thought it would be possible to have a loop unrolled into several parallel processing elements (PEs) that implemented rolled loops. When this is tried, no errors are thrown but the unrolling is not implemented.

b) *Simulation Time*: Despite Vivado HLS being compatible with both C and C++, we found simulation to be much faster when using C++. We took some time to discover this as our initial code was in C. We only moved to C++ as fixed-point data types are not supported in C. Even then, when using fixed-point data types, simulation can be very slow.

We found that our approach would be significantly easier to implement if the original algorithm was designed from scratch to better fit this implementation platform. For example, we could have down-sized images before feeding them to the neural network, drastically reducing the size of the network and number of needed resources. From other participants in the challenge we know that it can be done without a significant negative impact on accuracy. Also, it would be advantageous to have every convolutional layer with a number of filters that is a multiple of a predefined number. That way, hardware resources could more easily be shared between different layers.

As previously mentioned, we were not able to design an architecture that allowed K-means to be combined with accelerator-aware pruning. We suspect HLS might be limited and RTL based design would have to be used but it could also be due to our lack of experience with the synthesis tool.

IV. CONCLUSION

We sure did not produce a working solution to the problem, but we think we are now much better prepared with regards to deep learning model compression and adaptation techniques for FPGA implementation, as well as HLS based hardware development. We found HLS design to be in fact a useful tool that provides fast development for complex designs and projects that would be otherwise be nearly impossible to complete within the same time frame using traditional development tools. However, not everything is as smooth-sailing as we would wish it to be and the tool feels lackluster in some aspects.

REFERENCES

- [1] "Design automation conference system design contest 2020," accessed on 20-05-2021. [Online]. Available: <https://dac-sdc-2020.groups.et.byu.net/doku.php>
- [2] DeepZ, accessed on 20-05-2021. [Online]. Available: <https://github.com/jndeng/DACSDC-DeepZ>
- [3] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2736–2744.
- [4] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*, 2019.