Inference of Quantized Neural Networks on Heterogeneous All-Programmable Devices

Thomas B. Preußer
Marie Skłodowska-Curie Fellow
Xilinx Research Labs
Dublin, Ireland
thomas.preusser@utexas.edu

Giulio Gambardella
Xilinx Research Labs
Dublin, Ireland
giulio.gambardella@xilinx.com

Nicholas Fraser
Xilinx Research Labs
Dublin, Ireland
nicholas.fraser@xilinx.com

Michaela Blott
Xilinx Research Labs
Dublin, Ireland
michaela.blott@xilinx.com

***Abstract*—Neural networks have established as a generic and powerful means to approach challenging problems such as image classification, object detection or decision making. Their successful employment foots on an enormous demand of compute. The quantization of network parameters and the processed data has proven a valuable measure to reduce the challenges of network inference so effectively that the feasible scope of applications is expanded even into the embedded domain.**

This paper describes the making of a real-time object detection in a live video stream processed on an embedded all-programmable device. The presented case illustrates how the required processing is tamed and parallelized across both the CPU cores and the programmable logic and how the most suitable resources and powerful extensions, such as NEON vectorization, are leveraged for the individual processing steps. The crafted result is an extended Darknet framework implementing a fully integrated, end-to-end solution from video capture over object annotation to video output applying neural network inference at different quantization levels running at 16 frames per second on an embedded Zynq UltraScale+ (XCZU3EG) platform.

***Index Terms*—all-programmable, quantized neural networks, object detection**

I. Introduction

Neural networks have proven to be a capable and generic machine-learning means to address hard or even otherwise intractable problems. They have been especially successful in image recognition, object detection and decision making. Standard benchmarks for the evaluation of network implementations therefore include, less surprisingly, challenges like MNIST [1] for the recognition of handwritten digits and ImageNet [2], [3] for the classification of whole images. Another visual but more demanding task is the object detection, which aims at classifying and localizing individual objects within images. Standard reference datasets for this challenge are the Pascal Visual Object Classes (Pascal VOC) [4], [5]. A scientific breakthrough and prominent show case of competitive decision making was the AlphaGo vs. Lee Sedol challenge match in the game of Go.

This work focuses on the object detection based on the Pascal VOC using a convolutional neural network (CNN) drafted after the example of Tiny YOLO [6]. We focus on the acceleration of the network inference, i.e. the employment of the network performing its designated task, aiming at the online processing of live video within an embedded all-programmable platform. While our network must be trained and, indeed, re-trained to recuperate loss of accuracy through quantization, we perform this important but single-time effort without any exceptional resource constraints on standard GPU hardware.

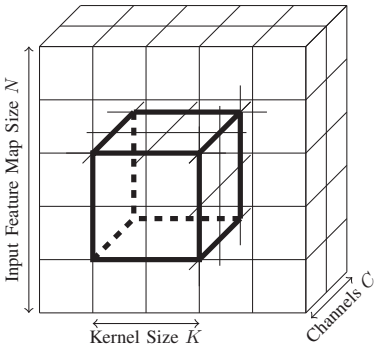


Fig. 1. Feature Map Convolution

The inference performed by a CNN is computationally dominated by a sequence of convolutions over 3-dimensional data volumes called feature maps. A convolutional layer is often immediately followed by a normalization, a non-linear activation and a pooling operation. While this computation is very well structured, it is also very intense. Particularly the convolution requires the computation of large dot products between the network parameters, i.e. the kernel weights, and the feature map elements. Assuming a convolutional kernel of size $K \times K$ and a feature map depth of C channels, these dot products comprise $K^2 \cdot C$ numeric multiplications for each application of a kernel over the width and height dimensions of the input feature map. This process is illustrated by Fig. 1. It is further duplicated for the same input feature map for each of the C' channels of the output feature map using the corresponding set of kernel parameters.

A typical way to approach the convolution is its reduction to a matrix multiplication. The rows of the multiplier matrix

This project has received funding from the European Union's Framework Programme for Research and Innovation Horizon 2020 (2014-2020) under the Marie Skłodowska-Curie Grant Agreement No. 751339.

are constructed by linearizing the weight parameters of the individual convolution kernels. The number of rows equals the count of output channels to produce. The columns of the multiplicand are correspondingly linearized kernel application footprints so that the result matrix will contain one convolution result in each of its elements. The multiplicand is generated by a procedure referred to as `im2col`. It regularly inflates the data of the input feature map significantly. Particularly, when the kernel size is small compared to the size of the input feature map and the stride of its application is one, the overlap of the kernel footprints causes `im2col` to essentially inflate the data volume by a factor of K^2. On the other extreme, a convolutional kernel of the same size of the input feature map degenerates into a single application and, thus, a fully connected layer with no input inflation at all.

The challenges that must be addressed by a CNN inference engine are the storage of and timely access to the network parameters as well as the enormous dot-product compute. Both challenges can be defused by quantization. Eliminating unnecessary precision from the network parameters reduces their memory footprint accordingly. Also, the multiply-accumulate backing the dot product computation benefits when the weights, and ideally also the feature map data, go from floating point to fixed point arithmetic and from wider to narrower data types. Programmable hardware as offered on all-programmable devices is able to effectively exploit such benefits even below an 8-bit quantization. We will refer to such aggressively quantized CNNs as QNNs.

In the remainder of this paper, we will give a brief overview on relevant related work with a strong emphasis on QNNs before describing how we gradually enabled a quantized derivation of the Tiny YOLO network to perform online Pascal VOC object detection on a live video stream in a small embedded Zynq UltraScale+ platform by leveraging the various compute capabilities of this heterogeneous SoC. Repeatedly identifying the most severe bottleneck, we individually describe our countermeasures and report the achieved performance gains. Sec. IV summarizes the undergone development.

II. Related Work

The presented work is an integration effort aiming at the optimal exploitation of a heterogeneous embedded platform. It relies on an hardware accelerator for the inference of quantized neural networks produced by our FINN framework [7]. The general idea of aggressive quantization, going as far as the full binarization proposed and pioneered by Hubara et al. [8] as well as by Rastegari et al. [9], has adopted significant momentum in the FPGA community. Besides FINN, also Zhao et al. have proposed a binarized neural network accelerator using Vivado HLS targeting Zynq devices [10]. Recently, Moss et al. have reported on a binary neural network implementation [11] within a hybrid data center environment comprising a Xeon CPU and an Arria 10 FPGA solves a similar integration task as our work. Their solution aims at saving power in the data center by offering an alternative to GPU accelerators. They do not at all target the ambitious resource limitations of embedded applications.

TABLE I
The Challenge Posed by Tiny YOLO versus Tincy YOLO

Layer #	Type	Tiny YOLO **Operations per Frame**	Tincy YOLO **Operations per Frame**	**Note**
1	conv	149520384	37380096	quant. sensitive
2	pool	173056	-	
3	conv	398721024	797442048	
4	pool	43264	43264	
5	conv	398721024	797442048	
6	pool	10816	10816	$>97\%$ of Compute
7	conv	398721024	398721024	
8	pool	2704	2704	Addressable by
9	conv	398721024	398721024	Offloaded
10	pool	676	676	HW QNN
11	conv	398721024	398721024	Accelerator
12	pool	676	676	
13	conv	1594884096	797442048	
14	conv	3189768192	797442048	
15	conv	43264000	21632000	quant. sensitive
Σ		6,971,272,984	4,445,001,496	

While full binarization has been shown to work for quite a few applications, it also fails regularly to maintain the desired degree of accuracy. This degradation can be countered by a slightly more moderate network quantization. The smallest possible retreat is ternary quantization. Suggested by Li et al. [12], this approach has been adopted for an FPGA implementation by Alemdar, Prost-Boucle et al. [13], [14]. The use of a wider 8-bit quantization in CNN inference can already be considered conservative with no relevant performance degradation. It is considered a safe enough choice to be used for ASIC implementations of inference engines or backing matrix multiplies as it has been done for the TPU by Google [15].

Our work was driven by a permanent analysis of the system performance, the identification of the limiting bottleneck and its mitigation. While the quantization of the network inference was a key technique to tame both the memory requirements of network parameters and the compute demand, we have also exploited other works in the course of this progress. First of all, we rely on Darknet [16] to provide us with an open-source neural network application environment available in customizable C code. We have used its show case network topologies YOLO and Tiny YOLO [6] as the starting point of our development making them fit to perform the object detection in a live video stream on an embedded all-programmable device. From the hardware point of view, we particularly exploit the programmable fabric of a Xilinx Zynq UltraScale+ device [17] and the Arm NEON technology [18].

III. Building TincyYolo

A. The Challenge

The computational challenge of object detection as posed by Tiny YOLO is best appreciated by studying Tab. I. It takes close to 7 billion floating-point operations to process a single

TABLE II
DOT-PRODUCT WORKLOADS OF QNN APPLICATIONS

	Ops / Frame			Primary Target
	Reduced	**8-Bit**	**Total**	**Application**
MLP-4	6.0 M $[W^1A^1]$	–	6.0 M	MNIST, NIST
CNV-6	115.8 M $[W^1A^1]$	3.1 M	118.9 M	CIFAR-10, Road Signs, ...
Tincy YOLO	4385.9 M $[W^1A^3]$	59.0 M	4444.9 M	Object Detection

TABLE III
INFERENCE PROCESSING TIME OF VIDEO FRAMES BROKEN INTO STAGES

Image Acquisition	40 ms
Input Layer	620 ms
Max Pool	140 ms
Hidden Layers	9160 ms
Output Layer	30 ms
Box Drawing	$\geq$15 ms
Image Output	$\geq$25 ms
Total	**10,030 ms**

frame. The vast majority of these operations can clearly be attributed to the convolutions within the hidden layers of this network. Whereas the input and output layers of the network have proven sensitive to quantization, the penalty paid for an aggressive quantization of the hidden layers in terms of detection accuracy could be contained within 3% by successful retraining. While still accounting for over 97% of the overall operations, these operations were simplified enormously by using binary weights $(-1, 1)$ and 3-bit feature map data. These are ideal circumstances for a successful acceleration by programmable hardware. For the input and output layers, this path was less attractive as they were best left using floating point or, at most, quantized to 8-bit fixed-point data to avoid harsh accuracy reductions.

To put the computational effort in relation to previous applications of FINN, refer to Tab. II. In anticipation of the further operational reductions that we have performed in the process of deriving our Tincy YOLO network, it already reports the reduced operational costs of our ultimately achieved solution. These are still greater than the previous FINN show cases by orders of magnitude just in terms of the plain operation counts. Note that also the individual operations are more complex as we were not able to produce sensible results with a complete binarization of Tincy YOLO. While the network weights are, indeed, binarized, we maintain a quantization of 3 bits for all feature map values. The input layer as well as the output layer must even process, at least, 8-bit quantities.

The significantly computational demands have a direct and severe impact on the available implementation options. While the fully binarized 4-layer MLP and 6-layer CNN lent themselves to an implementation of the inference engine with all layers residing one after the other in a dataflow pipeline, this option quickly fails on resource constraints for Tincy YOLO. Targeting a rather small XCZU3EG chip, only a single generalized convolutional layer together with its subsequent pooling layer would fit into the available fabric. The layers of the network must be run one after the other on the same accelerator. Note that this precludes concurrency across layers and implies a higher latency compared to a pipeline as the feature maps between layers are computed in full before the computation of the next layer can be triggered.

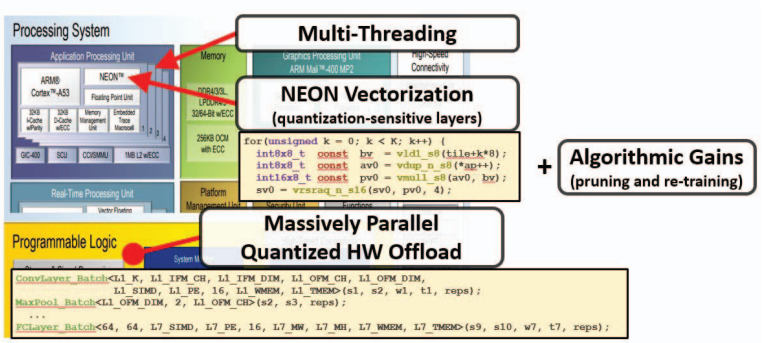
In summary, the desired object detection poses a significantly bigger computational challenge that is also less susceptible to quantization.

B. The Vision

Zynq UltraScale+ platforms are truly heterogeneous and offer more than just an FPGA accelerator. Specifically, these platforms include a powerful ARM multicore processor, which does not only allow to implement applications out of a convenient OS environment but also offers additional compute power through thread-based concurrency. In the context of intense linear algebra, also the NEON vector extension available in these processors is of utmost interest as it enables the parallel SIMD operation, e.g. in four single-precision floating-point lanes or in eight 16-bit integer lanes. These opportunities are illustrated in Fig. 2. All of them are easily exploitable through a C/C++-based application development. This is even true for the hardware accelerator, which is implemented through the HLS library of FINN. With this toolbox, we set out for taming the embedded live object detection. Note, indeed, that the Zynq UltraScale+ also incorporates a Mali GPU. The exploration of its specific workflow, toolchain and potential benefits has, however, not been part of our work so far.

C. Darknet Integration

Building on the Tiny YOLO topology, using Darknet as the training and inference framework is the natural choice. It is open-sourced, amendable and avoids the struggle of porting and reproducing available results within a different framework. Darknet provides basic generic training and inference implementations in C along with highly-optimized processing paths using CUDA-programmable GPUs, which are its primary execution targets. Having no such GPU available on the Zynq UltraScale+ platform, we are starting out with the generic inference. This delivers an disenchanting frame rate of **0.1 fps**. Live video processing is more than two orders of magnitude beyond reach. As shown in Tab. III, it is the inference in the hidden network layers which contributes the



Block Diagram [Xilinx, Inc.]: https://www.xilinx.com/content/dam/xilinx/imgs/products/zynq/zynq-eg-block.PNG

Fig. 2. Compute Opportunities Offered by the Zynq Platform Resources

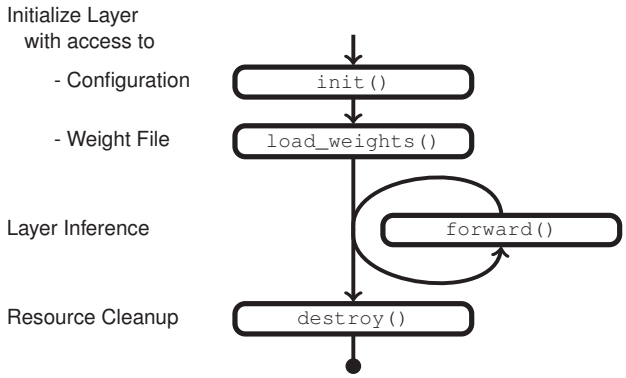


Fig. 3. Layer Life Cycle and Function Hooks Used by Offload Implementation

```
[convolutional]
filters=64
size=3
stride=1
activation=relu
binary=1

[maxpool]
size=2
stride=2
       ⋮
```

```
[offload]
# HW Interface Library
library=fabric.so
# Subtopology & Trained Weights
network=tincy-yolo-offload.json
weights=binparam-tincy-yolo/
# Output Geometry
height=13
width=13
channel=125
```

Fig. 4. Generic Offload Mechanism Built for Darknet

highest processing costs among all required processing stages from image acquisition all the way to video output.

As noted above, the inference of the hidden layers can be quantized and assigned to a FINN-based QNN implementation. For the integration of such an external accelerator, we implemented a generic offload mechanism that enables Darknet to pull a particular implementation from an arbitrary user-defined shared library. The offload mechanism builds upon the fact that Darknet is already virtualizing much of the layer functionality through function pointers. Essentially, the implementation of our new offload layer redirects those pointers to the library specified in the layer description. Thereby the life cycle and functionality of the layer, which is illustrated in Fig. 3, can be customized completely. Note that the abstraction of such an offload layer is solely Darknet's perspective. The backing custom implementation is only required to compute an output feature map from a given input feature map. Internally, it may, for instance, subsume the computation of multiple layers of various kinds. This is practiced by our fabric offload. The corresponding manipulation of Darknet's network configuration is shown in Fig. 4.

Using this added offload mechanism, the QNN hardware accelerator within the PL was integrated into the inference path of Darknet. Although the accelerator must process one hidden layer at a time and cannot benefit from pipelining gains due to resource constraints, it reduces the processing time of all hidden layers together to 30 ms, which corresponds to a speedup of more than 300× for this particular processing stage. Taking into account the surrounding processing, the net effect reduces to a 11× speedup allowing a frame rate of just above 1 fps. It is the input layer, which now defines the bottleneck of the computation.

D. NEON Vectorization

The generic implementation of the convolutional layers is not optimized since Darknet targets GPU accelerators for high-performance processing. It rather is a straightforward C implementation split into an explicit `im2col` followed by a matrix multiplication. While clearly being a valuable reference implementation, it must naturally ignore platform-specific capabilities and limitations. This is the lever available to us knowing that we target a set of ARM Cortex-A53 cores.

An obvious way to increase the number of arithmetic operations per cycle is vectorization as offered by the NEON extension of the platform processor. Using 128-bit registers, equivalent parallel computations can be performed in four 32-bit lanes up to sixteen 8-bit lanes. Also knowing that we could safely quantize the computation of the critical first convolutional layer down to eight bits, the employment of a NEON-optimized low-precision library appeared to be a promising approach. Using the already developed offload mechanism, we thus implemented a custom layer with an

`im2col` implementation that quantized the image data while arranging the multiplicand matrix and a matrix multiplication performed through the gemmlowp library [19]. The achieved $2.2\times$ speedup still left this layer as the key bottleneck of the computation.

A further significant gain by optimizing the individual operations appeared unlikely so that a fused implementation of the overall layer was aimed at. The rationale behind this step is a significantly increased data locality, which is especially beneficial on embedded platforms with rather small cache sizes. So, we have sliced the `im2col` transformation to produce the multiplicand matrix in vertical slices. The width of these slices is matched with the number of vector lanes that can be processed in parallel so that the corresponding slice of the result matrix can be produced row by row computing parallel dot products. The following input slices can subsequently re-use the same storage over and over until the matrix computation is complete. A generic convolutional layer implementation following this idea achieved a $2.1\times$ speedup albeit still operating on the original single-precision floating-point data. So, exploiting the capabilities of NEON is itself a benefit even without quantization.

The weight matrix of the first convolutional layer has a rather small dimension of 16×27. The 16 divides nicely by all lane counts that a NEON implementation might use, and 27 is small enough to be unrolled explicitly. Of course, such a fully customized implementation is no longer generic but the results are convincing. The floating-point computation can be reduced from 620 ms to 160 ms, a $3.8\times$ speedup. Re-introducing 8-bit quantization even yields 140 ms when using a 32-bit accumulator, and 120 ms when using a 16-bit accumulator. The 32-bit integer accumulation can actually not utilize more vector lanes than the floating-point implementation. However, the data locality of the 8-bit input data is increased. The 16-bit accumulation requires a careful management of the accumulator scale so as to avoid destructive numeric overflow in adding up the 27 products. Therefore, a rounding right shift by 4 bit positions must be performed before accumulation. This, in fact, introduces some small loss of detection accuracy so that the floating-point implementation is kept available as drop in reference for case-to-case evaluation.

The speedup of up to $5\times$ for the convolution of the first layer reduces the overall frame processing to 400 ms. The implied 2.5 fps are still not convincing. The major bottleneck remains within the input and its subsequent maxpool layer.

E. Algorithmic Simplification

Further improvements required more daring maneuvers on the algorithmic side. Besides the reduction of precision itself, several other changes were applied to Tiny YOLO to derive Tincy YOLO. Specifically, the following modifications were made: (a) leaky ReLU is replaced by ReLU; (b) the number of output channels of layer 3 is *increased* from 32 to 64; (c) the number of output channels of layers 13 & 14 is decreased from 1024 to 512; and (d) the first maxpool layer is removed along with increasing the stride of the first convolutional layer from 1 to 2. The surprising but most welcome result was that after retraining this modified network, the detection accuracy was practically maintained. (d) alone was able to replace the two biggest remaining bottlenecks with a lean convolution needing just 35 ms. With this additional speedup, a frame rate of more than 5 fps was at hand.

These algorithmic transformations are topological changes and turn the original Tiny YOLO network into our Tincy YOLO derivative. Accuracy scores for the modified networks are shown in Table IV.

TABLE IV
ACCURACY OF TINY YOLO VARIANTS

	Tiny YOLO	Tiny YOLO + (a)	Tiny YOLO + (a,b,c)	Tincy YOLO
Precision	Float	$[W^1A^3]$	$[W^1A^3]$	$[W^1A^3]$
Accuracy mAP(%)	57.1	47.8	47.2	48.5

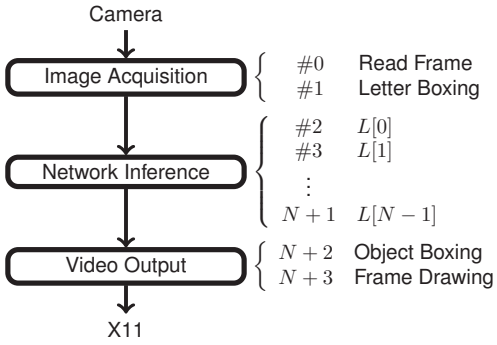


Fig. 5. Pipeline Stages of the New `demo` Mode

F. Parallelization

The steps taken so far have produced a sequence of frame processing steps that are all similarly complex. Only one of them requires the hardware accelerator as special resource, and the most complex stage takes 40 ms. With a total of six stages and four available processor cores, the theoretical maximum of a fourfold increase of the frame rate by turning these stages into a proper processing pipeline should only be diluted by parallelization and synchronization overhead.

Implementing the desired processing pipeline required a complete re-implementation of Darknet's `demo` mode, which had served well and delivered the complete end-to-end flow up to this point. In fact, even the network inference (forward) pass had to be disintegrated to gain access to the invocations of the individual layers.

The biggest chunks of the overall computation were further split into smaller pieces for a smoother pipeline operation. Such a move is not sensible in a sequential frame-by-frame processing scenario as it would only add overhead. However, in a pipelined parallel execution that requires synchronization at the stage boundaries, the competition over locks can be

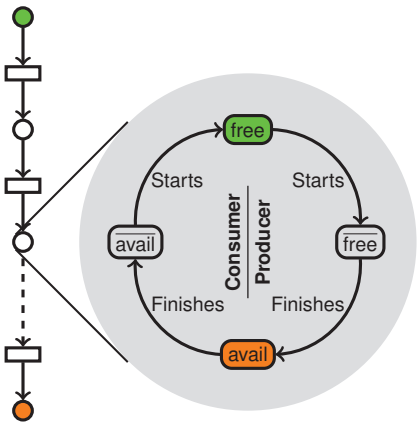


Fig. 6. Synchronization of Pipelined Frame Processing

reduced beneficially by a more fine-grained division into pipeline stages. In particular, the image acquisition was split into the camera access and the internal scaling of the captured frame. As illustrated in Fig. 5, the new `demo` mode derivative, thus, implements a pipeline that is four stages longer than the user-specified underlying network.

For our concrete Tincy YOLO application, also the implementation of the hardware offload layer was stripped of all pre- and post-processing of its input and output data, which were therefore moved into their own custom layer abstractions. This ensures that the blocking of the hardware is not unduly inflated to an overgrown containing layer abstraction but is rather limited to a tight wrapper around the accelerated computation.

The actual processing within the pipeline is performed by a pool of worker threads. One worker thread is allocated for each available core and tied to it. The pipeline breaks the overall computation in individual jobs, each of which advances the processed frame one step further. The worker threads process one such transaction at a time. If their current job is completed, the computed frame stays pending in the output buffer of the corresponding pipeline stage. A new job is selected for execution by finding the most mature one whose output buffer is free and whose input buffer has data pending. The video source and sink are always available and free, respectively. Note that this scheme of job scheduling prevents that one frame overtakes another so that the correct video sequence is maintained throughout the processing pipeline.

The re-implemented pipelined video processing `demo` mode achieved almost a threefold speedup resulting in a frame rate of 16 fps. This actually allows to play live video in a way that it is practically perceived as smooth.

IV. Conclusions

This paper has demonstrated how the individual heterogeneous compute resources of a modern Zynq Ultrascale+ platform can be systematically exploited for implementing an Pascal VOC object detection in a live video stream on an embedded platform. The presented measures can serve as a blueprint to enable other machine learning applications in resource-constrained environments. Key measures were the exploitation of quantization, hardware acceleration, NEON vectorization, algorithmic simplification and multi-threading for an overall speedup of 160×.

References

[1] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, Nov 2012.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *CVPR09*, 2009.

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[4] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (VOC) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, 2010.

[5] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, 2015.

[6] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *arXiv preprint arXiv:1612.08242*, 2016.

[7] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2017)*, ser. FPGA. New York, NY, USA: ACM, Feb 2017, pp. 65–74.

[8] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems 29 (NIPS 2016)*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., Dec 2016, pp. 4107–4115.

[9] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Computer Vision – ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 525–542.

[10] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2017)*. New York, NY, USA: ACM, Feb 2017, pp. 15–24.

[11] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High-performance binary neural networks on the Xeon+FPGA platform," in *27th International Conference on Field Programmable Logic and Applications (FPL 2017)*, Sep 2017.

[12] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," in *CoRR*, Nov 2016, vol. abs/1605.0.

[13] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Ptrot, "Ternary neural networks for resource-efficient AI applications," in *International Joint Conference on Neural Networks (IJCNN 2017)*, May 2017, pp. 2547–2554.

[14] A. Prost-Boucle, A. Bourge, F. Ptrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA," in *27th International Conference on Field Programmable Logic and Applications (FPL 2017)*, Sep 2017.

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, and Jerem, "In-datacenter performance analysis of a tensor processing unit," in *CoRR*, Jun 2017, vol. abs/1704.0.

[16] J. Redmon. (2013–2016) Darknet: Open source neural networks in C. http://pjreddie.com/darknet/.

[17] Zynq UltraScale+ MPSoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html. Xilinx Inc.

[18] Technologies NEON – ARM developer. https://developer.arm.com/technologies/neon. Arm Limited.

[19] gemmlowp: A small self-contained low-precision GEMM library. https://github.com/google/gemmlowp. Google.