A Case Study for an Accelerated DCNN on FPGA-based Embedded Distributed System

Anna Maria Nestorov Alberto Scolari Enrico Reggiani Luca Stornaiuolo Marco D. Santambrogio
Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
{annamaria.nestorov, enrico2.reggiani}@mail.polimi.it,
{alberto.scolari, luca.stornaiuolo, marco.santambrogio}@polimi.it

***Abstract*—Face Detection (FD) recently became the base of multiple applications requiring low latency but also with limited resources and energy budgets. Deep Convolutional Neural Networks (DCNNs) are especially accurate in FD, but latency requirements and energy budgets call for Field Programmable Gate Arrays (FPGAs)-based solutions, trading flexibility and efficiency. Nonetheless, the offer of FPGAs solutions is limited and different chips often require expensive re-design phases, while developers desire solutions whose resources can scale proportionally to the demands. Therefore, this work presents an FD solution based on a DCNN on a distributed, embedded system with FPGAs, proposing a general approach to reduce the DCNN size and to design its FPGA cores and investigating its accuracy, performance, and energy efficiency.**

***Index Terms*—DCNN, CNN, Quantization, Embedded, FPGA, Distributed, Face detection, PYNQ-Z1, HDL, HLS**

I. Introduction

FD consists in detecting the image regions that contain a human face with a given likelihood, and may be performed in embedded devices, close to the users, to avoid data transmission bottlenecks towards cloud infrastructures. While initial solutions had two different steps of feature extraction and classification, DCNN models unified them and also provide robustness to geometrical transformations and noise.

Yet, DCNNs are computationally heavy, in contrast with the latency requirements and power/energy constraints of embedded solutions. While the research focused on specialized accelerators Application Specific Integrated Circuits (ASICs) [1] or on FPGAs designs [2], these solutions do not "scale" to the application demands in that resources cannot be sized proportionally to the design size.

This "scalability" is instead typical of distributed systems, with resources added proportionally to the problem size. To investigate these guidelines, we propose an embedded, distributed system of FPGA-based processing elements tailored to FD. To achieve these goals, we make the following contributions:

1) porting a state-of-the-art FD DCNN [3] to such a system
2) prototyping this system via low-power embedded devices such as Xilinx PYNQ-Z1
3) evaluating this prototype against a fully-optimized pure software implementation and a server-class Graphic Processing Unit (GPU) implementation

This paper is organized as follows. Section II summarizes the relevant works in the literature. Section III shows the FD approach of HyperFace, the DCNN classifier at the base of our work, and our adaptations. Section IV explains the designs of the proposed solution, whose implementation is discussed in section V. Section VI evaluates the proposed approach, while section VII discusses limitations and possible future work.

II. Related Works

Deformable Parts Models (DPMs) models, such as [4], describe a face by its main components and consider their potential deformations, with specialized filters for capturing either the entire face or its components. Another solution for FD adopts local Edge Orientation Histogramss (EOHs) [5] as features, which are largely invariant to global illumination changes and capture geometric properties of faces better than linear edge filters. Instead, the Histogram of Oriented Gradient (HOG) method [6] is based on evaluating well-normalized local histograms of image gradient orientations in a dense grid.

To generalize over different illumination conditions or poses, different Convolutional Neural Network (CNN) models recently spread, which can automatically derive problem-specific feature extractors from the training data: examples are [7], [8] and [9], producing state-of-the-art results on many challenging and publicly available FD datasets.

III. Face Detection DCNN

This work starts from HyperFace [3], a state-of-the-art DCNN solution for simultaneous FD, landmark localization, pose estimation and gender classification that exploits the feature information hierarchically spread through the network [10]. HyperFace starts by identifying *candidate regions* that likely contain faces via the Selective Search (SS) algorithm [11] and resizes them to fit the model input size, evaluating them via the HyperFace DCNN. As in the original implementation [3], we trained our model instance from the AFLW dataset [12] via TensorFlow, using 32-bit floating-point values, with 21,997 real-world images. The validation classification accuracy over the ground truth obtained after two epochs of training is 97.04%, which is sufficient for our purposes.

By removing the layers for landmark localization, pose estimation and gender classification we tailored the modular

DOI 10.1109/IPDPSW.2019.00025

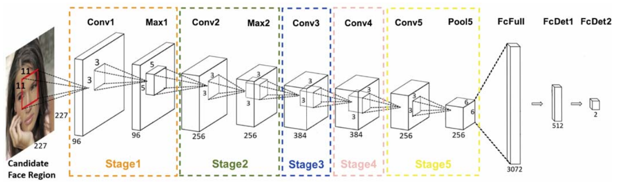

Fig. 1. Our FD architecture and the five final stages after implementation.

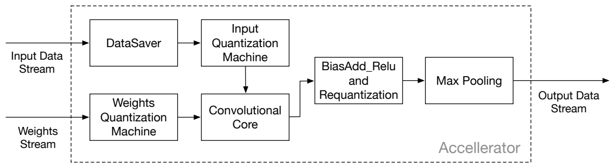

Fig. 2. A convolutional layer followed by a max-pooling layer architecture.

HyperFace architecture to a FD-only task, whose structure is shown in Fig.1: our FD model is composed by five convolutional layers, some followed by subsampling layers, and three fully-connected layers. Response normalization steps have been removed since they are assumed to have a minimal impact when limiting the number of tasks [13], while the removed tasks can be considered as extensions to this work.

IV. System Design

This section explains the design strategies to build our systems, starting from the quantization strategy to the design of the various cores.

a) Quantization Process: We quantized the FD model in TensorFlow by empirically observing the ranges of inputs, outputs, weights, biases, and intermediate Multiply and Accumulate (MAC) values through the various layers while running on a representative subset of input images, selecting 8-bit precision for our model. We implemented the quantized matrix multiplication within the convolutional cores based on the open source *gemmlowp* library [14], which internally uses more thnan 8 bits to accumulate and avoid overflows.

b) Node System Architecture: The node architecture should be flexible, to run one or more stages in a single bitstream. The Central Processing Unit (CPU) controls the computation by sending input data and weights to the accelerator via separate Direct Memory Access (DMA) cores, caches the biases in the on-chip Block Random-Access Memory (BRAM), reads the results back and sends them to the next node in the cluster. Fig.2 shows the architecture to compute a convolutional layer followed by a max-pooling layer, with input quantization, bias adding, Rectified Linear Unit (ReLU) application and to outputs requantization to 8 bits; these components communicate via First-In, First-Out (FIFO) queues, with a purely *dataflow* design.

The *Convolutional Core* performs the input convolution with the weights via MACs units. For any DCNN, two sources of parallelism exist: *intra-layer* is derived from the independence of output feature maps, as different filters can be applied simultaneously to the same input, while *intra-feature-maps* is based on the fact that multiple input features maps can be multiplied by the weight in parallel and then accumulated via an accumulation tree to compute the output feature map. In our design, Digital Signal Processors (DSPs) are properly time-multiplexed among independent operations on the same input feature map or among different output feature maps, computing each convolution with one DSP, which acts as a MAC unit by performing an element-wise multiplication between the input feature map and the corresponding weight-set, internally accumulating the partial results. The input data-path works at a sub-multiple of the clock frequency using a clock-enable signal, whose value is strictly dependent on the filter size and is multiplexed so that every element is consumed by the same MAC, thus mapping different convolutional layers with the same filter size on the same hardware.

To leverage data reuse and increase performance, the *DataSaver* reads the inputs from main memory through a DMA just once, re-orders the data based on current *intra-layer* parallelism and stores them in BRAM, acting as a pre-fetching and caching core for the following computational core.

The *Input and Weights Quantization Machines* take data coming from the *DataSaver*, or weights from main memory and normalize them by subtracting their zero values.

If the *intra-feature-maps* parallelism does not match with the number of input feature maps, partial results are accumulated in the *BiasAdd_Relu and Requantization Core* to obtain the final results, which also adds biases by first subtracting the bias zero point to the output of the convolution and then rescaling the result to the target range to finally add the bias; finally, it applies the ReLU operator and performs a second requantization to scale this value to the 8-bit output.

The *Max-Pooling Core* is a simplified version of the *Convolutional Core*, with the same data access pattern but applying the simpler *MAX* operator, with little DSPs usage. Adopting the same parallelism as the *Convolutional Core*, thus analyzing the same number of output feature maps produced in parallel, allows sustaining the same throughput, as the dataflow style prescribes.

V. Implementation

As a reference, we use a fully-optimized software implementation of the whole FD network based on state-of-the-art, floating-point Basic Linear Algebra Subprograms (BLAS) routines [15] optimzed for ARM CPUs from OpenBLAS [16], which are natively multi-threaded.

For the FPGA implementation, we split the FD model layers into *stages*, based on the most constraining resource required, in all cases DSPs. Table I shows the characteristics of the three stages designed to run on a single PYNQ-Z1 node, while Table II shows those of the six stages that run on the cluster of PYNQ-Z1 nodes, highlighted with colored boxes in Fig. 1, with *Iterations* being the number of calls to the stage logic to compute a single candidate region.

To run all stages in Table I on a single node, the application runs one bitstream at a time and re-configures the FPGA,

TABLE I
SINGLE PYNQ-Z1 SYSTEM STAGES

Stage	FMs		Parallelisms		Iterations	Conv. Core
	Input	Output	Intra-Feature-Maps	Intra-Layer		Frequency [MHz]
Stage1	3	96	3	48	2	140
Stage2	96	256	96	3	86	120
Stage3_4_5	256, 384, 384	384, 384, 256	128	1	2688	100

TABLE II
DISTRIBUTED SYSTEM STAGES

Stage	FMs		Parallelisms		Iterations	Conv. Core
	Input	Output	Intra-Feature-Maps	Intra-Layer		Frequency [MHz]
Stage1	3	96	3	48	2	140
Stage2_1	96	126	96	3	42	120
Stage2_2	96	130	96	3	44	120
Stage3	256	384	128	2	384	150
Stage4	384	384	128	2	576	150
Stage5	384	256	128	2	384	150

which takes two orders of magnitude more than a stage; multiple inputs are batched to run against each stage before reconfiguration, which is possible as an image contains hundreds of candidate regions, and the PYNQ-Z1 memory allowed storing all of them in memory during our tests. To further decrease reconfiguration costs in this scenario (despite this choice reduces the available *intra-layer* parallelism), Stage3, Stage4, and Stage5 are mapped to the same bitstream as shown in Table I.

Instead, the distributed system, composed by six PYNQ-Z1 nodes running the stages in Table II, achieves higher performance at the cost of a proportionally high resources usage, where nodes communicate via 100Mb/s Ethernet channels connected in a star topology to achieve a fully-pipelined execution, mimicking the structure of Fig. 1. To balance this pipeline, the second stage has been split into two substages (Stage2_1 and Stage2_2) since its latency is twice the other stages latencies. The communication between the nodes is implemented with Message Passing Interface (MPI) [17], using synchronous and non-blocking primitives with double buffering in order to parallelize input receiving, FPGA computation, and output transmission.

In all our PYNQ-Z1 implementations, the software part is written in C/C++ to minimize overheads and runs on the ARM Ubuntu Linux distribution natively installed on PYNQ-Z1, while the computational cores on FPGA are written in SystemVerilog and the cores for data movements and quantization are written in High Level Synthesis (HLS) to be easily modifiable for varying requirements of access patterns and quantization strategies.

VI. Experimental Results

Our goal is to achieve similar accuracies in FD scores with respect to the TensorFlow results with floating-point and a speedup in terms of latency with respect to the software implementation. In the following, we will refer to the software-only solution as *ARM* and to the FPGA-based implementation as *ARM-FPGA*; both can run in a single node or distributed settings, as from section V. Additionally, the original implementation in TensorFlow is run with the same input on an NVIDIA GeForce GTX 960 GPU with an Intel Core i7-6700 CPU at 3.40GHz, and is referred to as *GPU*, representing a server-class equipment with a high power budget that is unavailable in embedded scenarios (120W); here, the experiments are run with a batch size of 137, limited by the memory on the GPU card, in order to fully exploit its parallelism.

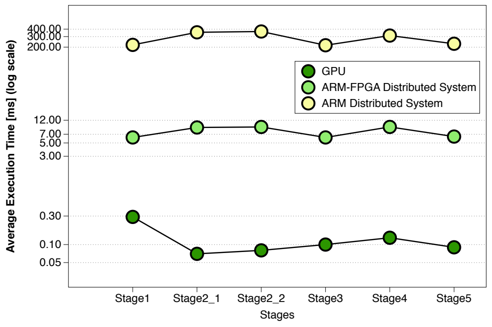

Fig. 3. Trend of the per-stage average execution times.

All the experiments use 100 input images for both *ARM* and *ARM-FPGA* implementations, with an average of 650 candidate regions per image generated through the SS algorithm. In the following sections, several aspects are evaluated, and the time results are referred to the pure convolutional part since it is the portion of the DCNN that it has been ported on hardware, being it the most computationally intensive part of a DCNN model.

A. Accuracy Loss with Quantization

With respect to the reference TensorFlow implementation, the probabilities computed by both *ARM* and *ARM-FPGA* implementations show negligible relative errors, 10^{-6} and 10^{-3} respectively. This translates to 100% and 99% respectively of predictions being equal to TensorFlow's, although our implementations use 8-bit integers instead of 32-bit floating point.

B. Per-Stage Execution Times

Fig. 3 shows the per-stage average execution times for the best representatives of each solution, with the *ARM* implementation being two orders of magnitude slower than *ARM-FPGA*, while the ratio between the *ARM-FPGA* and the *GPU* is around one order of magnitude, despite the different classes and power/energy characteristics of these systems.

On average, our *ARM-FPGA* implementation achieves a $37\times$ per layer speedup compared to *ARM*, while it remains on average three times slower than *GPU*.

C. Hardware Resource Usage

Table III shows that the most critical FPGA resource are DSPs, mainly used within the *Convolutional Core* to perform the computation. While most components run at 100MHz frequency, the *Convolutional Core* has a higher frequency because of the time-multiplexing optimization: using DSPs as

TABLE III
HARDWARE RESOURCE USAGE

Nodes	LUTs (53200)		FFs (106400)		DSPs (220)		BRAMs (240)	
	Usage	Percentage	Usage	Percentage	Usage	Percentage	Usage	Percentage
Stage1	37,029	69.60%	55,938	52.57%	196	89.09%	74.50	53.21%
Stage2	37,433	70.36%	64,991	61.08%	219	99.55%	122	87.14%
Stage2_1	36,246	68.13%	64,966	61.06%	219	99.55%	121	86.43%
Stage2_2	37,453	70.40%	64,957	61.05%	219	99.55%	136	97.14%
Stage3	35,466	66.67%	60,477	56.84%	200	100%	119	85%
Stage4	33,071	62.16%	62,746	58.97%	219	99.55%	133	95%
Stage5	34,053	64.01%	61,240	57.56%	219	99.55%	119.50	85.36%
Stage3_4_5	36,845	69.26%	63,583	69.26%	135	61.36%	121.50	86.79%

TABLE IV
POWER AND ENERGY RESULTS

	GPU	*ARM*		*ARM-FPGA*	
		Single	Distributed	Single	Distributed
Run Time [s]	1.104	1,403	304.606	98.159	7.722
Power Max [W]	205	4.2	24.8	3.7	22.2
Energy Per Image [J]	226	5,891	7,554	363	171

MAC units makes this core the design bottleneck since data flow every clock enable cycles.

Note that Stage3_4_5 also runs at 100MHz frequency because of timing issues due to the congested design, as from Table I and Table II. The *Max-Pooling Core* does not need to be clocked at higher frequencies if its filter size is smaller than the convolutional one; its frequency can indeed be tuned to match the convolutional core maximum throughput.

D. End-to-End Time

As the *ARM* implementation is sequential, the end-to-end time to analyze an entire image (assuming 650 candidate regions) is more than 18m, which is the sum of all stages execution times in Fig. 3 multiplied by the number of candidate regions. Instead, the single node *ARM-FPGA* implementation takes 1m18s to perform the same computation, which accounts for the computation, the reconfiguration, and related overheads; here, reconfiguration occurs after all candidate regions are computed in a stage and intermediate results are buffered in memory, with a $14.7\times$ speedup over the single node *ARM*.

The distributed *ARM-FPGA* system analyzes a candidate region every 9.36ms, while the *ARM* every 369.22ms; since the total time to analyze an image is 6.08 seconds and 4 minutes, respectively, we achieve a $39.5\times$ speedup over distributed *ARM*, while we are slower than *GPU* by a factor of $12.2\times$.

E. Power Consumption and Energy Efficiency

Table IV shows power and energy results from the analysis of a single image. The *ARM* implementations of both the single node and the distributed system have higher execution times and power consumption than the corresponding *ARM-FPGA* implementation, which has $16.2\times$ higher energy efficiency. Similarly, in the distributed setting the *ARM-FPGA* implementation achieves $44\times$ higher efficiency than *ARM*. The *GPU* implementation is $1.3\times$ less energy efficient than the distributed *ARM-FPGA* system and it is characterized by a very high peak power. However, due to limitations of the PYNQ-Z1 platform, the results in Table IV include all the board components, while for the *GPU* setting they include GPU and CPU only.

VII. CONCLUSIONS AND FUTURE WORK

This work introduced design guidelines to accelerate a FD CNN to a distributed, embedded system and implemented it on the PYNQ-Z1 platform, with a final speedup of $39.5\times$ in terms of analysis time and $44\times$ higher energy efficiency with respect to a distributed, fully optimized software implementation and negligible accuracy loss despite quantization greatly helped increasing compute density on FPGA. As future steps, we are considering automated scheduling approches to cope with node failures and changing workloads.

REFERENCES

[1] A. Erdem, C. Silvano, T. Boesch, A. Ornstein, S.-P. Singh, and G. Desoli, “Design space exploration for orlando ultra low-power convolutional neural network soc,” Jul. 2018, pp. 1–7.

[2] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, “Going deeper with embedded fpga platform for convolutional neural network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’16, Monterey, California, USA: ACM, 2016, pp. 26–35.

[3] R. Ranjan, V. M. Patel, and R. Chellappa, “Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2018.

[4] M. Mathias, R. Benenson, M. Pedersoli, and L. Van Gool, “Face detection without bells and whistles,” in *European Conference on Computer Vision (ECCV)*, Sep. 2014.

[5] K. Levi and Y. Weiss, “Learning object detection from a small number of examples: The importance of good features,” in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 2, Jun. 2004, pp. II–II.

[6] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, vol. 1, Jun. 2005, 886–893 vol. 1.

[7] R. Ranjan, V. M. Patel, and R. Chellappa, “A deep pyramid deformable part model for face detection,” *CoRR*, vol. abs/1508.04389, 2015. arXiv: 1508.04389.

[8] S. Yang, P. Luo, C. C. Loy, and X. Tang, *From facial parts responses to face detection: A deep learning approach*, Dec. 2015.

[9] S. S. Farfade, M. J. Saberian, and L. Li, “Multi-view face detection using deep convolutional neural networks,” *CoRR*, vol. abs/1502.02766, 2015. arXiv: 1502.02766.

[10] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” *CoRR*, vol. abs/1311.2901, 2013. arXiv: 1311.2901.

[11] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013. arXiv: 1311.2524.

[12] M. Köstinger, P. Wohlhart, P. M. Roth, and H. Bischof, “Annotated facial landmarks in the wild: A large-scale, real-world database for facial landmark localization,” in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, Nov. 2011, pp. 2144–2151.

[13] (2018). Convolutional neural networks (CNNs / ConvNets), [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[14] (2018). gemmlowp - Low-precision matrix multiplication, [Online]. Available: https://opensource.google.com/projects/gemmlowp.

[15] (2018). BLAS (Basic Linear Algebra Subprograms), [Online]. Available: http://www.netlib.org/blas/.

[16] W. S. Zhang Xianyi Wang Qian. (2018). Openblas - an optimized blas library, [Online]. Available: https://www.openblas.net.

[17] L. L. N. L. Blaise Barney. (2018). Message passing interface (mpi), [Online]. Available: https://computing.llnl.gov/tutorials/mpi/.