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*—In recent years Face Detection (FD) became the base of multiple applications, which often require the computation to be performed locally with low latency but with limited resources and energy. The spreading of Deep Convolutional Neural Networks (DCNNs), which achieve high accuracy and generality, for FD solutions conflicts with these constraints, and calls for solutions based on Field Programmable Gate Arrays (FPGAs) for a good trade-off between flexibility and efficiency. While FPGAs are increasingly used, their offer is still limited in terms of configuration and different chips often require expensive re-design phases. Instead, developers ideally desire solutions whose resources can scale proportionally to the demands, as in distributed systems. This work investigates how to implement an accurate FD solution based on a DCNN on a distributed, embedded system equipped with FPGAs, proposing a general approach to reduce the DCNN size and to design its FPGA cores. This solution is compared to the original classifier in terms of accuracy, performance, and energy efficiency within an embedded and server-like scenario.**

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

I. Introduction

FD essentially consists in detecting the regions within an image that contain a human face with a given likelihood. It is the base of multiple services and products, like photo analysis in social networks, entertainment systems, movies production, and others. Particularly important are also face recognition systems, which start from FD to find the image region most likely to contain the face to be authenticated. A recent use case is marketing applications that count the number of users to measure the effectiveness of an advertisement and possibly customize it based on features like gender, race or age. These kinds of systems are often deployed close to the user, have soft-real-time constraints and cannot resort to cloud-like computing facilities due to limited connectivity and budget. For these reasons, designers adopt embedded solutions that perform the computation locally but are constrained in power consumption, latency, and design budget.

While initial solutions in the start of the 2000s had two different steps of feature extraction and classification, designed separately, in the latest years DCNN models unified them, being also robust to geometrical transformations and noise and highly generalizable, provided that a sufficiently large training set is available.

Yet, DCNNs' classification power comes at the cost of high compute demands, which contrasts with the real-time requirements and power/energy constraints of embedded applications for FD. To increase performance and energy efficiency, the research recently focused on specialized accelerators, either on FPGAs solutions [1] or custom Application Specific Integrated Circuits (ASICs) [2]. However, even in the case of FPGAs, designers are constrained by the market offer, so that changes in the DCNN features (like adding more layers) may also cause changing the target chip and in turn force a major re-design, especially if the new chip has a different resources distribution. Essentially, available hardware resources and design effort do not "scale" proportionally to the application demands.

This "scalability" can yet be achieved if resources are added proportionally to the design size, as in contemporary distributed systems, with design components and practices that should also be scalable. To investigate these guidelines, we propose an embedded, distributed system composed of FPGA-based processing elements that cooperate within a single application. To achieve these goals, we make the following contributions:

1) starting from a state-of-the-art DCNN for FD [3], we show how to map it to an embedded, distributed system of multiple FPGAs
2) we prototype these system via low-power, embedded devices with very limited resources such as the Xilinx PYNQ-Z1
3) we evaluate the prototype system against a pure fully optimized software implementation and a server-class Graphic Processing Unit (GPU) implementation, with respect to several metrics

This paper is organized as follows. Section II introduces the reader to the context of this work, reviewing the basic concepts and the relevant works in the literature. Section III explains the FD approach of HyperFace, with the DCNN classifier at the base of our work and how we adapted it to our use case. Then, section IV explains the designs of the proposed solution, from the quantization phase to the details of the FPGA cores, whose implementation is discussed in section V. Section VI evaluates the proposed approach, while section VII discusses limitations and possible future work.

II. Context definition

This sections introduces the context of this work by presenting background material in section II-A and related work in section II-B.

A. Background

A Convolutional Neural Network (CNN) is a sequence of layers which takes a generic "raw" input and returns scores for each class the input may belong to. The first part of a CNN performs *feature extraction*, where subsequent layers extract more and more complex features out of the initial input, and a second *classification* part that aggregates the features and provides the prediction scores. The *convolutional layer* is the core building block of a CNN and is the most computationally heavy part. Each convolutional layer has a set of filters, generally small in size, that run through the full input to extract a set of relevant features. When the filter slides over the width and height of the input volume a 2D activation map, called *feature map*, is produced, giving the responses of that filter at every spatial position. The *pooling layer*, also called *sub-sampling layer*, is often inserted between successive convolutional layers. It reduces the spatial size and extracts the most relevant features by applying operators such as *MAX* or *MEAN* on a small filter area, producing a single output pixel.

After the feature extraction, the *classification* occurs through a sequence of *fully-connected* layers, with the last layer having as many neurons as classification classes, with a normalization operator (e.g. *SoftMax*) usually applied on the output scores to interpret them as probabilities.

These three main types of layers are stacked to form the CNN architecture. When the number of stacked layers increases, the CNN is classified as DCNN and has even higher storage, compute and energy demands. To tailor DCNNs to embedded scenarios, they should be resized to the limited resources budget: assuming the structure and hyper-parameters are fixed, the main reduction technique is *quantization* (also called *discretization*), through which the designer can suitably trade the resources usage with the accuracy loss. In general, quantization consists in using integer types instead of floating point precision, often also significantly reducing the bit-width of integer values to 8-bit or even less. Quantization is especially effective when inputs, weights, and biases have narrow value ranges and the model is inherently resilient to noise, as in the case of CNNs. Here, the quantization process tracks both the weights and biases ranges for each layer by recording minimum and maximum, and then each floating point weight/bias is approximated to the closest integer value within the linear interval [0, 255]. Central Processing Units (CPUs) and GPUs usually have little to no benefit from quantization as they support a fixed set of operations and data types, while ASIC or FPGA solutions can achieve higher compute density and energy efficiency.

B. Related Works

Nowadays, FD techniques can be roughly categorized into two general groups: one with rigid templates, learned mainly via boosting-based methods with a boosted cascade of classifiers like DCNNs; and Deformable Parts Models (DPMs), which describes a face by its main parts and takes into consideration potential deformations between them.

DPMs-based models, such as [4], are typically divided into two parts: a coarse root filter that approximately covers an entire face, and higher resolution part filters that cover smaller details such as eyes, nose, and mouth. After discovering a root location with a high score, the corresponding part locations can be found by looking up the optimal displacements. Another solution for FD adopts local Edge Orientation Histograms (EOH) [5] as features. EOHs 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. Here, the image window is divided into small "cells", where a local 1-D histogram of gradient directions or edge orientations is accumulated. Then, considering a *block* of multiple cells on which the "energy" of all local histograms is accumulated, each *cell* is contrast-normalized with its *block* "energy", finally an Support Vector Machine (SVM) classifier it run to predicted the feature class [4], [7].

To generalize over different illumination conditions or poses, different CNN models have been developed, which can automatically derive problem-specific feature extractors from the training examples. Recent examples of CNN based architectures are [8], [9] and [10] producing state-of-the-art results on many challenging and publicly available FD datasets.

III. Face Detection DCNN

This work investigates how to port a state-of-the-art DCNN named HyperFace [3] to an embedded, distributed system for FD. Therefore, section III-A explains the HyperFace structure and training options, while section III-B explains how it was adapted to our use case and system.

A. The Hyperface model

In the literature, tasks like FD, landmark localization, pose estimation and gender classification have generally been solved separately, but recent research showed that correlations among them can speed up both training and classification [11]–[13]. HyperFace [3] is a solution based on a CNN for simultaneous FD, landmark localization, pose estimation and gender classification that exploits the feature information that was shown to be hierarchically spread through the network [14]: the first layers search for edges and corners and are more relevant to landmark localization and pose estimation, while the deeper layers are more complex and class-specific and are thus more specific to gender prediction and FD.

HyperFace first identifies *candidate regions*, also called *proposed regions*, that likely contain objects via the Selective Search (SS) algorithm [15]; this algorithm first over-segments the image based on the intensity of pixels, and then groups these segments in a bottom-up fashion based on a similarity

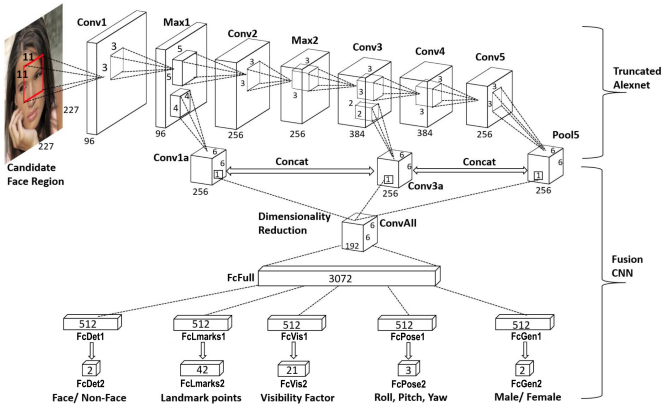

Fig. 1. The original HyperFace architecture

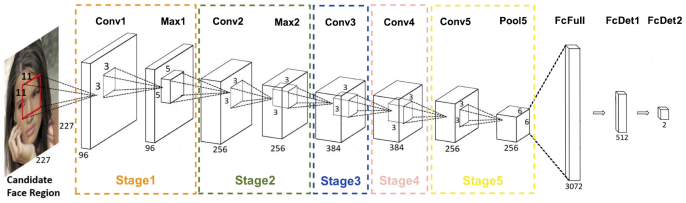

Fig. 2. Our FD architecture, adapted from HyperFace; both the adapted DCNN structure and the 5 final stages after implementation are shown

metric. The output candidate regions are then decreased in size to 227×227 pixels, in order to fit the model input dimensions. Based on the used SS parameters, the HyperFace authors generate an average of 2000 proposed regions per image.

HyperFace assigns to each candidate region a face/non-face score, and if the predicted class is "face" it also returns the other feature predictions. Its DCNN classifier, shown in Fig.1, is based on Alexnet [16] and consists of two main parts. The first part, the "Truncated Alexnet", is composed of five convolutional layers, where the first two ones and the last one are followed by a max pooling layer; the last three fully-connected layers of Alexnet are not present in HyperFace as they encode image-specific classification information, not needed for pose estimation and landmarks extraction. After each pooling layer, a local response normalization is applied to "adjacent" feature maps, in order to better generalize the network to the various tasks. For the same purpose, *hyperfeatures* from Max1, Conv3 and Pool5 are "concateneted" together into a high-dimensional $6 \times 6 \times 768$ stack of feature maps, with intermediate layers Conv1a and Conv3a that convert features from Max1 and Conv3 to the same size of $6 \times 6 \times 256$ Pool5 output.

The second part, the "Fusion CNN", contains no pooling layer in order to avoid local invariance, thus keeping precise location information for FD and landmarks localization tasks [1]. Here, the fused feature map from the Truncated Alexnet is reduced in size via the convolutional layer ConvAll and the fully-connected layer FcFull, from where the various tasks are learned via five fully-connected layers.

As in the original implementation [3], we trained HyperFace from the AFLW dataset [17] via TensorFlow, using standard 32-bit floating-point data types, with 21,997 real-world images with full pose, expression, ethnicity, age, and gender variations. Out of these 21,997 images, 19,000 images have been used for training, 2,000 for validation and 1,997 for testing using the HyperFace loss function. The validated classification accuracy over the ground truth obtained after two epochs of training is 97.04%, which is sufficient for our purposes.

[1]Pooling layers decrease spatial resolution, hence removing the information about the exact location of a feature; in particular, max pooling's strength consists in detecting the presence feature rather than its exact location.

B. Our FD model

Thanks to the modularity of HyperFace, we tailored its architecture to a FD-only task, which is the goal of our work and is sufficient to investigate how to efficiently implement a DCNN on a distributed, FPGA-based embedded system; indeed, the other tasks can be considered as extensions to this work. As Fig.2 shows, we removed all tasks other than FD and the related fully-connected layers, leaving FcDet1 and FcDet2 unmodified. Instead, since we removed ConvAll due to single task left, we modified FcFull so that it takes in input the $6 \times 6 \times 256$ tensor coming from Pool5, while still returning an output of size 3072. Another noticeable difference with respect to HyperFace is the removal of response normalization steps, which are assumed to have a minimal impact, especially when limiting the number of tasks [18].

IV. System Design

This section explains the design strategies to build our systems. The FD model is first quantized to reduce storage and increase the operations density, as section IV-A explains. Then, section IV-B explains the architecture of a single node of the distributed system, which computes part of the FD DCNN model. Which part can be offloaded to a single node highly depends on the CNN characteristics and on the available FPGA resources, and is discussed in detail in section V with respect to our use-case. However, the accelerator architecture of section IV-B and its cores, detailed in section IV-C, are designed to be modular and can be scaled to different use-cases.

A. Quantization Process

As from section II-A, we employed TensorFlow to empirically observe the ranges of inputs, outputs, and intermediate Multiply and Accumulate (MAC) values throughout the various FD modelgemm layers, by running it on a representative subset of the input images. Table I shows these ranges, typically very narrow, in terms of minimum and maximum values, and also shows the *scale* value, i.e. the minimum variation of the real value that causes a variation also in the quantized value; the scale is obtained as $(max - min)/256$ since we use 8 bits, which is sufficient to keep good accuracy as validated in section VI.

The quantized matrix multiplication implemented in the cores for the convolution is based on *gemmlowp* [19], an

TABLE I
MIN, MAX AND SCALE VALUES FOR THE CONVOLUTIONAL STAGES

Variable	Parameters	Conv1	Conv2	Conv3	Conv4	Conv5
Weights	max	0.035250	0.033975	0.035723	0.033043	0.036801
	min	-0.038022	-0.036101	-0.036986	0.032491	-0.036293
	scale	0.000287	0.000274	0.000285	0.000256	0.000286
Intermediate Accumulations	max	4.055379	3.00309	2.880889	2.857010	3.528848
	min	-4.023486	-1.932962	2.522948	-2.194070	-2.558575
	scale	0.031681	0.019357	0.021191	0.019808	0.023872
Biases	max	0.167030	0.109820	0.096142	0.093614	0.110384
	min	-0.170138	-0.105325	-0.093545	-0.090699	-0.106447
	scale	0.001322	0.000843	0.000743	0.000722	0.000850
Output	max	4.121626	3.037831	2.910869	2.785253	3.608217
	min	-4.002198	-2.012389	2.575947	-2.278581	-2.515303
	scale	0.031858	0.019804	0.021516	0.019858	0.024013

open source low-precision matrix multiplication library, which avoids overflows in MACs by internally using more than 8 bits (typically 16). In order to *requantize* the results from these intermediate MACs to the $0-255$ output domain, we adopted the same approach of *gemmlowp* based on a sequence of offset/muliply/shift operations [20].

B. Node System Architecture

The quantization process, as stated at the beginning of this chapter, significantly reduce the resources utilization. In particular, when dealing with FPGAs, integer arithmetic implies fewer Digital Signal Processors (DSPs), Look-Up Tables (LUTs) and Flip-Flops (FFs), allowing to make the most of the available resources. The embedded target Xilinx PYNQ-Z1 Platform has a limited amount of resources, and even if the network has been quantized, it would not be reasonable in terms of latency to port the whole network on a single PYNQ-Z1, more precisely with a single bitstream which configures the platform to execute the entire network. Depending on the target goal, the objective can vary from using a single PYNQ-Z1, which would increase the latency since the hardware needs to be reconfigured a certain number of times in order to cover the whole network computation, or using a cluster of PYNQ-Z1 and therefore computing the different stages in pipeline to achieve a competitive latency.

Heterogeneous embedded devices are typically designed as a System on Chip (SoC), where the accelerator, here the FPGA, resides on the same die of the CPU and also shares the same physical memory; Fig.3 shows this organization together with our designed architecture. This SoC architecture allows data sharing and communication between computing resources, which can easily cooperate. Indeed, in our reference PYNQ-Z1 platform, the CPU performs the initial data pre-processing, in particular the extraction of the candidate regions via the SS algorithm, and resizes them to the 227×227 DCNN input size. Then, the CPU sends the candidate regions and the weights to the FPGA logic, while the biases are sent only during initialization since they can fit in the FPGA on-chip Block Random-Access Memory (BRAM). As from Fig.3, two Direct Memory Accesss (DMAs) cores are deployed in the design to transfer candidate regions and weights in parallel. Note that in case the DCNN is split on multiple nodes, where each node runs only some layers, the FPGA sends intermediate

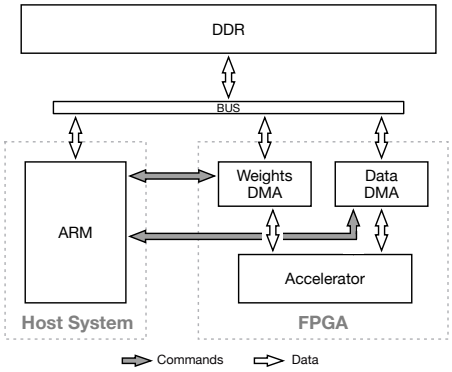

Fig. 3. High level block desing of the proposed architecture

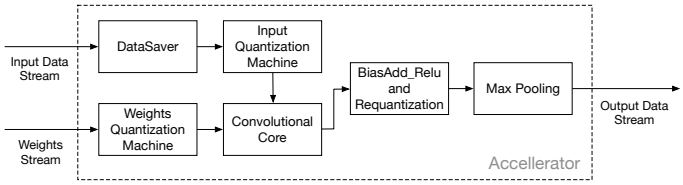

Fig. 4. The architecture of a convolutional layer followed by a max-pooling one

results back to main memory and the CPU sends them to the next node, where they are sent to the local FPGA logic for the following compute-heavy stage.

C. The Accelerator

Fig.4 shows the architecture to compute a single convolutional layer followed by a max-pooling layer, also considering the steps to quantize the input values and to requantize the outputs to the desired bit-width: in HyperFace, a convolutional layer is always followed by a Rectified Linear Unit (ReLU) layer, and sometimes by a max-pooling layer as in Fig.1; all these components communicate via Firt-In, First-Out (FIFO) queues and no centralized controller nor state is needed, thus achieving a purely *dataflow* design. In case larger parts of the DCNN are offloaded to a single FPGA, multiple cores sequences convolutional-ReLu-pooling can be chained together storing in main memory only the final results.

The cores in Fig.4 are designed as follows.

a) Convolutional Core: The *Convolutional Core* performs the input convolution with the weight via a sequence of MACs. For any DCNN, two sources of parallelism exist and are used in the design to boost performance. The first source is called *intra-layer* parallelism and is due to the independence of output feature maps, as different filters, i.e. different weight sets, can be applied simultaneously to the same input, which can then be re-used. The second source of parallelism, referred to as *intra-feature-map*, is the computation of each output feature map, where input features maps can be multiplied by the weight in parallel and then accumulated via an accumulation tree. However, the number of *intra-feature-map* parallel operations can be quite high for our FD DCNN, where, for example, Conv4 reaches $13 \times 13 \times 384$, which can quickly

saturate the available DSPs. Therefore, DSPs must be properly time-multiplexed among independent operations on the same input feature map, or even among different output feature maps.

The proposed architecture is based on this approach, computing each convolution with one DSP: the DSP acts as a MAC unit, performing an element-wise multiplication between the input feature map and the corresponding weight-set, while internally accumulating the partial results of the filtering window. 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 can be consumed by the same MAC, one at a time. Thanks to this design it is possible to map different convolutional layers on the same hardware, provided they have the same filter size. This solution offers a substantial reduction in the amount of DSPs needed for the convolution, leaving room to leverage *intra-feature-maps* and *intra-layer* parallelism degrees and leveraging data-sharing for less off-chip data movement.

b) DataSaver: In a naïve convolutional core design, the number of input transfers from main memory would be equal to the number of output features divided by the *intra-layer* parallelism. To leverage data reuse and increase performance, the *DataSaver* reads the input feature maps from main memory through the DMA just once and stores them in the BRAM, thus acting as a pre-fetching and caching core for the following computational cores. Furthermore, the *DataSaver* may re-order the input data, coming from the previous layer in the DCNN, in case data have a different ordering, enforced by the previous *intra-layer* parallelism, then the current layer requires in input.

c) Input and Weights Quantization Machines: Input data coming from the *DataSaver* and weights coming directly from main memory are normalized by subtracting their zero values.

d) BiasAdd_Relu and Requantization Core: If the *intra-feature-maps* parallelism does not match with the number of input feature maps, partial results have to be first accumulated in order to obtain the actual final results. Then, biases, not considered so far, are cached in BRAM during initialization and should be added to the final output of the *Convolutional Core*. To perform this operation, the *BiasAdd_Relu and Requantization Core* first subtracts the bias zero point to the output of the convolution, then rescales this value to the target range and finally adds the bias. Then, it performs a second requantization to scale this value to the final 8-bit output The special case in which the output value is correctly scaled but exceeds 255 should be considered, which could happen in the case in which the corresponding floating point value is outside the observed fixed ranges. In this situation, the output value is automatically set to 255. Along with this outlier situation, the ReLU operator, based on the current output zero point, is also applied.

e) Max-Pooling Core: It is a substantially simplified version of the *Convolutional Core*, sharing the same data access pattern but applying the simpler *MAX* operator, which can be synthesized with common FPGA logic 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 implementation, 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 [21] implemented and optimzed for the PYNQ-Z1 ARM-based SoC within OpenBLAS [22], which are natively multi-threaded. In particular, we used the *sgemm* routine for matrix-matrix multiplications and *sgemv* for matrix-vector multiplications, employed throughout all convolutional layers and all fully-connected layers respectively.

For the FPGA implementation, we split the FD model layers in different parts, called *stages*, based on their FPGA requirements of the most constraining resource, in all cases DSPs. Table II shows the characteristics of the three stages designed to run on a single PYNQ-Z1 node, while Table III shows those of the six stages highlighted with colored boxes in Fig.2. In all cases, the number of times the logic must be called from CPU to compute an entire layer, called *Iterations* in the tables, is computes as

$$Iterations = \frac{OutputFMs}{Intra - Layer} \times \frac{InputFMs}{Intra - Feature - Maps}$$

The two implementations shown in the tables represent the extreme possibilities available with a distributed system. On one hand, all FD DCNN layers are run in a single node, but cannot be simultaneously deployed on the scarce FPGA resources; hence, the application has to run one bitstream at a time and re-configure, which takes 300ms, around two orders of magnitude more than a stage run. Here, the only possible optimization is batching multiple inputs for each reconfiguration, which is possible as an image contains hundreds of proposed regions with the model approach but may be limited by the available memory or in different scenarios (e.g. in case an entire photo is analyzed at once). In our tests, for example, we adopted this strategy as each image contains on average 650 candidate regions, which occupy around 146MB of main memory out of 512MB on the PYNQ-Z1. As from the last row of Table II, Stage3, Stage4, and Stage5 are mapped into the same stage with proper multiplexing logic for inputs and outputs between the respective *Convolutional* and *BiasAdd_Relu and Requantization Cores* and the other cores for data movement and quantization; despite this choice reduces the available *intra-layer* parallelism to only one due to DSPs, it avoids two intermediate re-configurations. Once the last stage has been computed, all the candidate regions pass through the classification stage implemented in software, which returns the face/non-face for each candidate region.

On the other hand, the distributed system of Table III with six PYNQ-Z1 nodes achieves the best performance, at the cost of a proportionally high resources usage. Here, the application implements a fully pipelined communication between

TABLE II
SINGLE PYNQ-Z1 SYSTEM STAGES

Stage	FMs		Parallelisms		Iterations	Conv. Core Frequency [MHz]
	Input	Output	Intra-Feature-Maps	Intra-Layer		
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 III
DISTRIBUTED SYSTEM STAGES

Stage	FMs		Parallelisms		Iterations	Conv. Core Frequency [MHz]
	Input	Output	Intra-Feature-Maps	Intra-Layer		
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

nodes, mimicking the structure of Fig.2 via 100Mb/s Ethernet channels connected in a star topology. As from Table III, the second convolutional layer is split to two different stages (Stage2_1 and Stage2_2) to balance the latency with respect to all other stages in this design: indeed, the latency of the second stage in Fig.2 is twice the latency of the other stages, and splitting allows balancing the time each node spends computing; furthermore, as the FPGA computation is still the bottleneck with respect to the data transmission, the latency of the first node (sending data twice) is roughly balanced anyway. The communication between the nodes is implemented with Message Passing Interface (MPI) [23], using synchronous and not blocking primitives together with double buffering in order to parallelize input receiving, FPGA computation and output forwarding to the next node.

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. The computational cores on FPGA are written in SystemVerilog to efficiently use available resources, while 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. Finally, the DMA cores are those available in Xilinx Vivado 2017.2, which are directly controlled by the software via memory mapping, thus avoiding expensive user-kernel context switches.

VI. EXPERIMENTAL RESULTS

Our goal is to achieve similar accuracies in terms of FD scores with respect to the reference 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. This platform, referred to as *GPU*, represents a server-class equipment that can achieve the best performance among commonly

TABLE IV
AVERAGE ERRORS AND STANDARD DEVIATIONS.

Implementation	KPI	Face	Non-Face
ARM	Average Error	9.194×10^{-8}	1.094×10^{-7}
	Average Relative Error	9.439×10^{-7}	2.053×10^{-7}
	Standard Deviation	712.735×10^{-6}	712.741×10^{-6}
ARM-FPGA	Average Error	3.497×10^{-3}	3.497×10^{-3}
	Average Relative Error	38.754×10^{-3}	6.749×10^{-3}
	Standard Deviation	225.705×10^{-3}	224.021×10^{-3}

available solutions, but has a much higher power budget that is unavailable in embedded scenarios (120W). The experiments on the target GPU are run with a batch size of 137, in order to fully exploit its parallelism; the batch size is limited by the memory on the GPU card. All the experiments have been conducted on 100 input images for both *ARM* and *ARM-FPGA* implementations, with each image having an average 650 candidate regions generated through the SS algorithm, for a total number of input candidate regions of 65,978. 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 computational intensive part of a DCNN model.

A. Accuracy Loss with Quantization

In order to evaluate the loss of accuracy due to quantization, we compute the average error, the average relative error and the standard deviation on the computed scores expressed as real numbers as shown in Table IV. We also take into account the percentage of the total number of correct predictions defined as binary values, thus mapping all the scores greater than 0.5 as 1 or as 0 otherwise. Analysing the pure total number of correct predictions with respect to the reference, we achieved 100% and 99% (65,778 over 65,978). The difference between the two is motivated mainly by the used data types: the *ARM* implementation uses a single precision floating point data type, as TensorFlow does, while the *ARM-FPGA* implementation uses for the convolutional part unsigned integer 8-bit data type and floating-point data type for the fully-connected computation (that runs in software). Looking at the different computed metrics in Table IV, both implementations show almost irrelevant errors with respect to the reference TensorFlow implementation.

B. Per-Stage Execution Times

The per-stage average execution times are shown in Fig.5 and refer to the computation of a single candidate region; this figure shows only the best representatives for each solution, i.e. the distributed implementation for both *ARM* and *ARM-FPGA* and the *GPU* implementation. For *GPU*, the execution times are obtained by dividing the time on the whole batch by the number of images in the batch. As it can be seen from Fig.5, the *ARM* implementation remains two orders of magnitude slower compared to *ARM-FPGA*, while the ratio between this implementation and the *GPU* implementation

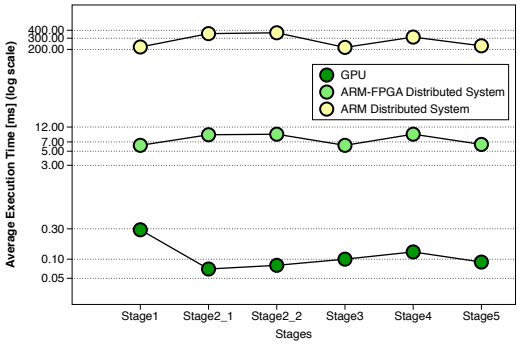

Fig. 5. Trend of the average per-stage execution times

TABLE V
PER-STAGE AVERAGE EXECUTION TIMES [MS]

Stage	*GPU*	*ARM-FPGA*	*ARM*
Stage1	0.29	6.18	218.12
Stage2_1	0.07	9.07	353.85
Stage2_2	0.08	9.25	365.08
Stage3	0.10	6.18	214.95
Stage4	0.13	9.26	321.16
Stage5	0.09	6.39	227.70

remains around one order of magnitude, despite the different classes and power/energy characteristics of these systems.

Analysing the obtained speed-ups, on average our *ARM-FPGA* implementation obtains a $60\times$ per layer speed-up compared with the *ARM* implementation, while it remains on average three times slower than the *GPU* one. Indeed, the *GPU* implementation consistently outperforms our distributed *ARM-FPGA* system, as expected since the used GPU has a higher power consumption and is by design well suited to these tasks and highly parallel. Comparing the *ARM* and the *ARM-FPGA* implementations, the *ARM-FPGA* is capable to obtain a per-layer speed-up of $37\times$ on average.

C. Per Stage Hardware Resource Usage

Analyzing the resources usage, as section V already suggested, the most critical FPGA resource are DSPs. In this specific work, the *Convolutional Core* is the one which takes the most number of DSPs, out of the *BiasAdd_Relu and Requantization Core* which uses some of them. As can be noted by Table VI, reporting the different resource utilization

TABLE VI
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%

of all the implemented hardware stages, apart from Stage1 and Stage3_4_5, all the other implementations use almost 100% of the available DSPs. The frequency at which the system runs is always 100MHz, with the exception of the *Convolutinoal Core*., which is set to run at a higher frequency, because, in the other case, due to the time-sharing mechanism, the *Convolutional Core* would become the slowest element since it takes, by design, a new input every clock enable clock cycles. In all the stages the *Convolutional Core* frequency was indeed increased, apart from Stage3_4_5 in which timing was not met due to the high number of components placed in the design, as shown in Table II and Table III. Theoretically, also the *Max-Pooling Core* should be clocked at higher frequencies since it is a simplified version of the *Convolutional Core*. However, in the first two convolutional stages the filter sizes of the *Convolutional Core* is bigger than in pooling and in the remaining stages, for each set of concurrent feature map in output the *BiasAdd_Relu and Requantization Core* has to wait for three complete iterations to get the final results; hence, there is no need to increase the pooling frequency.

D. End-to-End Time

On a single PYNQ-Z1 node, since the different convolutional stages of the network are executed in a sequential manner, the execution time is given by summing up the single execution times since there is no data transfer. The *ARM* implementation computes every candidate region layer by layer, therefore, it would take 18 minutes and 26 seconds on average, computed as $(218.12ms + 353.85ms + 365.08ms + 214.95ms + 321.16ms + 227.70ms) \times 650$. On the other hand, the *ARM-FPGA* implementation average time necessary to compute all the candidate regions is obtained as $(14.89ms + 41.44ms + 61.56ms) \times 650 + 300ms \times 3$, which corresponds to 1 minute and 18 seconds. The first three times represent the actual hardware computation time, additional latency is added due to the fact that hardware buffers have to be retrieved from DDR since they are not cached, while 300ms represent the average reconfiguration time. Therefore, the obtained FPGA speed-up on the single PYNQ-Z1 implementation amounts to $14.7\times$.

Looking to the distributed *ARM-FPGA* system, thanks to the double buffering technique, to the synchronous non-blocking communication and to the pipelined structure, every 9.36ms a candidate region output is ready. Differently, the *ARM* computation produces an output every 369.22ms. This means that every 6.08 seconds and 4 minutes, respectively, an image is fully analyzed. In this configuration, the *ARM-FPGA* implementation achieves a $39.5\times$ of speed-up with respect to the *ARM* one. When comparing the *GPU* implementation, in which a single candidate region takes 0.76ms, being it the sum of its stage execution, to our distributed *ARM-FPGA* system, *GPU* reaches a $12.2\times$ speedup.

E. Power Consumption and Energy Efficiency

Taking as reference the computation of one input image from which 825 candidate regions have been extracted, the ex-

TABLE VII
POWER CONSUMPTIONS

	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

perimental results regarding power consumption are presented in Table VII. As can be noted, the *ARM* implementations of both single PYNQ-Z1 and the distributed system are not efficient since they have greater execution times as well as higher power consumption with respect to the corresponding *ARM-FPGA* implementation. Indeed, the single PYNQ-Z1 *ARM-FPGA* implementation obtains $16.2\times$ of energy efficiency with respect to the *ARM* counterpart, while, on the distributed design, the *ARM-FPGA* implementation guarantees a $44\times$ of energy efficiency. If real-time execution is not a constraint, for example, if the images have to be analyzed for an advertisement analysis, the distributed *ARM-FPGA* system would be the best choice. When comparing our best implementation, represented by the embedded *ARM-FPGA* distributed system, to the *GPU* implementation, only $1.3\times$ is obtained for energy efficiency; however, the energy measured for the distributed *ARM-FPGA* system includes all the components on the PYNQ-Z1 board (which could be excluded when realizing a custom board for the market), while the *GPU* energy consumption accounts for the GPU and CPU only.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we introduced design guidelines to accelerate a FD CNN to a distributed, embedded system and implemented a working prototype on the PYNQ-Z1 platform. The distributed system reaches a speed-up of $39.5\times$ in terms of a single image while guaranteeing to be $44\times$ energy-efficient with respect to the concurrent fully optimized software implementation. On the single PYNQ-Z1 system, a $14.3\times$ of speed-up has been obtained, along with an energy efficiency of $16.2\times$. We obtained a 99% of accuracy on face/non-face class binary scores average errors on the scale of 10^{-3}, thus they can be considered irrelevant when comparing with floating point results.

As a continuation of this work, several aspects can be improved. The first aspect is the mapping of the FPGA stages to the nodes, which can be automated via a scheduler that would also cope with failures and changing workloads, like the feature classifiers of the original HyperFace. Another aspect is the node power consumption, which can be better controlled by turning off unneeded resources and by better integration with Linux, in order to decrease the CPU frequency when the FPGA is active. Finally, the quantization phase can be automated and possibly fully integrated with available tools like TensorFlow.

REFERENCES

[1] 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.

[2] 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.

[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. Benenson, M. Mathias, T. Tuytelaars, and L. V. Gool, "Seeking the strongest rigid detector," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2013, pp. 3666–3673.

[8] 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.

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

[10] 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.

[11] X. Zhu and D. Ramanan, "Face detection, pose estimation, and landmark localization in the wild," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2012, pp. 2879–2886.

[12] D. Ramanan, "Face detection, pose estimation, and landmark localization in the wild," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, ser. CVPR '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 2879–2886.

[13] D. Chen, S. Ren, Y. Wei, X. Cao, and J. Sun, "Joint cascade face detection and alignment," in *ECCV*, 2014.

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

[15] 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.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105.

[17] 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.

[18] (2018). Convolutional neural networks (CNNs / ConvNets), [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[19] (2018). gemmlowp - Low-precision matrix multiplication, [Online]. Available: https://opensource.google.com/projects/gemmlowp.

[20] (2018). Building a quantization paradigm from first principles, [Online]. Available: https://github.com/google/gemmlowp/blob/master/doc/quantization.md.

[21] (2018). BLAS (Basic Linear Algebra Subprograms), [Online]. Available: http://www.netlib.org/blas/.

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

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