An FPGA-based architecture for embedded systems performance acceleration applied to Optimum-Path Forest classifier

Wendell F.S. Diniz, Vincent Frémont, Isabelle Fantoni, Eurípedes G.O. Nóbrega

An FPGA-based architecture for embedded systems performance acceleration applied to Optimum-Path Forest classifier

Wendell F. S. Diniz[a,b,*], Vincent Fremont[b], Isabelle Fantoni[b], Eurípedes G. O. Nóbrega[a]

[a] *Department of Computational Mechanics, Faculty of Mechanical Engineering, University of Campinas. Rua Mendeleyev, 200 - CEP 13083-860 Cidade Universitária Zeferino Vaz - Campinas - SP*

[b] *Sorbonne universités, Université de technologie de Compiègne, CNRS 7253 UMR/Heudiasyc. CS 60319, 60203 Compiègne cedex*

Abstract

Classification techniques development constitutes a foundation for machine learning evolution, which has become a major part of the current mainstream of Artificial Intelligence research lines. However, the computational cost associated with these techniques limits their use in resource constrained embedded platforms. As the classification task is often combined with other high computational cost functions, efficient performance of the main modules is fundamental requirements to achieve hard real-time speed for the whole system. Graph-based machine learning techniques offer a powerful framework for building classifiers. Optimum-Path Forest (OPF) is a graph-based classifier presenting the interesting ability to provide nonlinear classes separation surfaces. This work proposes a SoC/FPGA based design and implementation of an architecture for embedded applications, presenting a hardware converted algorithm for an OPF classifier. Comparison of the achieved results with an embedded processor software implementation shows accelerations of the OPF classification from 2.18 to 9 times, which permits to expect real-time performance to embedded applications.

Keywords: FPGA implementation, Machine Learning, Classification, Optimum-path Forest

1. Introduction

Machine learning methods for object detection, recognition, and classification are key features for autonomous robotic systems. Current development approaches have been mainly relying on efficient parallel computation based on Graphics Processing Units (GPUs), which are generally intensive power consumers. On the other hand, embedded applications have been turning into the more common computational application nowadays, which may greatly benefit from embedded parallel computation solutions. FPGA (Field Programmable Gate-Array) systems present hardware concurrency and high modularity, being a natural candidate to implement low power parallel programming concepts. FPGA-based solutions can face the complexity of advanced machine learning approaches, enabling these applications in small robots with limited power resources.

FPGAs present a level of flexibility that general purpose CPUs can not reach. Also, their computational power by energy consumption rate surpasses GPUs', with the newest models offering even more computational power than late GPU models [1, 2]. In the last years, the main producers introduced SoC/FPGA boards, incorporating in the same encapsulation a regular processor, an FPGA and a high-speed communication bus between them. This type of SoC (System on a Chip) is a very convenient one to implement parallel digital processing, with extended flexibility due to the heterogeneous processors configuration.

Graph-based pattern recognition provides an interesting and powerful framework for classification problems. These methods formulate the classification problem as a graph, where the nodes represent the samples' feature vectors and the edges represent the relation between nodes. Application of heuristic rules leads to identify groups of similar samples as sets of connected components, which are then used to classify unknown samples.

Recently, the Optimum-path Forest (OPF) has been proposed as a new framework for graph-based classification [3, 4]. The method addresses a graph partition task as a competitive process, in which a series of connected components, rooted on key samples called prototypes, try to conquer the unknown samples according to a path-cost function. The prototype that offers the best reward propagates its label to the unknown sample. The method has shown good performance in several applications, presenting similar accuracy to known methods such as Support Vector Machines (SVM) and Artificial Neural Networks (ANN) [5, 6, 7, 8]. The set of operations used by the OPF algorithms consists of vector-based operations with great potential for fine-grain parallelism. This characteristic can be explored to build a classification framework dedicated to the embedded systems. All these ideas point to FPGA-based solutions as an attractive option to provide hardware acceleration by parallelism to OPF algorithms, while keep-

*Corresponding author

Email address: wdiniz@fem.unicamp.br, wfiorava@hds.utc.fr (Wendell F. S. Diniz)

ing small power consumption. Some recent works present parallel implementations of the OPF training stage using the OpenMP specification [9] and GPUs [10, 11].

The main contribution of this work is the design, implementation, and test of an FPGA-based parallel processing architecture for the OPF classifier, aiming embedded machine learning systems. The main goal is to achieve hardware acceleration provided by adopting SoC/FPGA flexible devices. The designed system is intended to fit into hard real-time signal processing applications, however implying a low power system. The processing speed gain will be evaluated by comparison between the parallel hardware implementation and the algorithm software version running on the SoC embedded processor. The main architectural challenge is to balance parallelism with memory management, considering latency efficiency and bandwidth.

Recent launching of an OpenCL (Open Computing Language) compiler aiming FPGA development, implying shorter development times through an efficient workflow proposition, makes it the today preferred language to adopt for parallel programming [12, 13]. Regarding OPF algorithm development and performance, some OPF published classifiers can greatly benefit from an effective embedded architectural framework. The versatility of the classifier and its potential for parallelism and application for embedded systems is the main motivation to adopt the OPF to exemplify the proposed architecture. To the better knowledge of the authors, there are no published works presenting an FPGA-based parallel architecture focusing on the classification stage or the respective implementation of the OPF algorithm adopting OpenCL.

The paper structure is presented in the following: Section 2 presents a literature review of OPF applications and hardware-based approaches for machine learning using FPGA. Section 3 presents an overview of the OPF classifier followed by the guidelines for its implementation in FPGA hardware in Section 4. Section 5 proceeds with the performance evaluation of the proposed implementation. Finally, Section 6 presents the paper conclusions and directions for future developments.

2. Related works

2.1. Optimum-path Forest theory and applications

An OPF based classifier can be viewed as a generalization of the Image Forest Transform (IFT) [14], which is a design tool for image processing operators. Both algorithms share the principle of a clustering of similar nodes offering a reward measured by the path-cost function, in order to conquer a new individual one. The OPF classifier extends the concepts of IFT from images to general graphs [3]. The pixels and intensity/color values of the images are respectively equivalent to samples and feature vectors in a classification task. Those features can be extracted in many different application-specific ways. For example, the feature vectors may be just an intensity vector for RGB levels, or, to design a face recognition system, the feature vectors may be a description of salience points using a well-known technique, as SIFT [15] or Haar-like feature [16], among many others. Moreover, OPF presents some interesting characteristics: it is non-parametric, intrinsically multi-class, can deal with non-linear feature spaces without requiring a transformation or change in domain, does not make any assumption about the shapes of the classes and can handle some degree of overlapping between classes. Both supervised and unsupervised learning approaches are possible to be used for training an OPF system, however, for the moment, the focus is on the supervised learning version.

The method has been successfully applied to several classification tasks, including spoken emotion recognition [6], oropharyngeal dysphagia identification [5], infrared face recognition [7], and satellite image processing for land use classification [8]. Considering these publications, OPF showed to be at least at a similar level of accuracy and computational efficiency of the well-known Support Vectors Machine (SVM) classifier. The implementation complexity of the OPF is also smaller than SVM's, which, allied to its faster execution time, indicate its suitability for embedded applications.

2.2. FPGA accelerated machine learning

FPGAs have been consistently used as an alternative for building high-performance embedded systems. Their employment on classification tasks has several examples. Artificial neural networks were the first methods to take advantage of the massive parallelism capacity offered by FPGA implementations. Reference [17] shows an early effort in this field, presenting an architecture suitable for applications in image processing, pattern recognition, and neural networks. It uses a Single Instruction Multiple Data (SIMD) approach, with a network of Processing Elements (PE) and a custom memory controller for distributing data among the PEs. Recently, new developments in this area can be found in [18, 19].

FPGA promises have attracted implementation of several classification methods. An SVM application for single digit recognition using an FPGA implementation can be found in [20]. Parallel to improvements in the SVM algorithm and evolution of the FPGA devices themselves, new propositions for SVM implementations appeared in [21, 22, 23]. In the recent years, CV applications, such as pedestrian detection, have received some FPGA implementations [24, 25]. Convolutional Neural Networks, which has been the basis for Deep Learning applications, are also often implemented in FPGA [26, 27, 28, 29].

3. Optimum-path Forest Classifier

The following subsections present the adopted OPF supervised training and classification stage details.

3.1. Training stage

The basic training routine presented in [3] is done in two stages, called fitting and learning phases. The fitting phase starts with the selection of a set of samples from the data universe to act as the training set. Considering that this set must represent all possible classes, the algorithm fits the problem by constructing a complete graph using the training set samples as the nodes. The weights of the edges are obtained by calculating the dissimilarity between the nodes, using a previously defined dissimilarity function. The Euclidean distance (L^2 norm) is the most used dissimilarity function, but it can be substituted by any distance function that suits a particular requirement of an application. From this complete graph, a Minimum Spanning Tree (MST) is extracted. The algorithm associates a cost to each node and also mark special nodes called prototypes. The prototypes are defined as nodes of different classes that share an edge. The edges between prototypes are then removed because they are the leaves of a classification tree, consequently, the associated cost for all prototypes is set to 0. The algorithm then assigns to each node their respective cost, which corresponds to the value of the maximum edge weight found in the path from the node to its respective prototype. The fitting process is shown on Figure 1.

The resulting forest can be directly used to classify the unknown samples, but some methods were proposed to increase the accuracy of the classifier including another set of samples called the evaluation set [3, 30]. The classifier is built from the training set, and its accuracy is evaluated by classifying the samples in the evaluation set. A learning procedure is applied, consisting in switching misclassified nodes in the evaluation set with samples of the training set and restarting the training and evaluation again with the new sets. After a determined number of restarting iterations, depending on an adopted convergence criterion, the instance with the best accuracy is selected as the classifier. This procedure is meant to find the most informative samples in the universe set for being used for the final classifier.

3.2. Classification stage

For a given unknown sample s, its resulting classification label $\lambda(s)$ is given by:

$$\lambda(s) = \lambda(t) \mid t = \min_{\forall t \in T}\{\max\{C(t), d(s,t)\}\}, \qquad (1)$$

where T is the classifier set, $C(t)$ is the associated cost of the classifier nodes and $d(s,t)$ is the value returned by the dissimilarity function. The tree that offers the minimum cost to connect to the unknown sample, that is, the one that holds the lesser dissimilarity, propagates its class to the sample. So, the classification is a process of minimization of dissimilarity. The classification process is exemplified in Figure 2.

Papa et al. [30] present a variation of the classifying algorithm exploring a theoretical property of the OPF that can accelerate the classification process. As the classifier finds the node that offers the minimum cost to connect to the unknown sample, one can assume that the winning one will have a small cost (not necessarily the lowest). So, if we present the classifier's nodes in an ascending order of their costs, the probability of the winning one to be at the beginning of the list is high. Then, one must search through the list until it finishes (the worst case) or finds a node whose cost is greater than the calculated cost for the unknown sample in the previous iteration. That node is the winner node. Algorithm 1 shows the process.

Algorithm 1 OPF classification algorithm.

Require: Ordered classifier set T, label map $\lambda(T)$, connectivity cost map $C(T)$, dissimilarity function $d(s,t)$ and test set S.
Output: Test set label map $\lambda(S)$.
Auxiliary: Variables *tmp* and *mincost*, counter i.

```
1: function OPF_Classifying(T)
2:     for all s ∈ S do
3:         i ← 1
4:         mincost ← max{C(t_i), d(s, t_i)}
5:         λ(s) ← λ(t_i)
6:         while i < |T| and mincost > C(t_{i+1}) do
7:             tmp ← max{C(t_{i+1}), d(s, C(ti + i)}
8:             if tmp < mincost then
9:                 mincost ← tmp
10:                λ(s) ← λ(t_{i+1})
11:            end if
12:            i ← i + 1
13:        end while
14:    end for
15: end function
```

4. Proposed FPGA architecture for OPF classification

4.1. High level system design

The proposed architecture was conceived to use an SIMD (Single Instruction, Multiple Data) based auxiliary Parallel Processor associated with a Host Processor (HP) and a double-access global memory module. Figure 3 shows an overview of the main system. In the adopted configuration, the host processor controls the main application execution flow, managing the data access, and dispatching commands to the parallel processor (PP). HP processed data written on the Global Memory (GM) is distributed by PP to be processed in parallel by the several processors that are inside the Elementary Processor Array (EPA). As the processing task finishes, it signals the host back.

The host processor can be based on any kind of all-purpose device. It is usually programmed using a high-level programming language, like C/C++.

The auxiliary parallel processor is designed to execute the computationally intensive tasks of the application. It

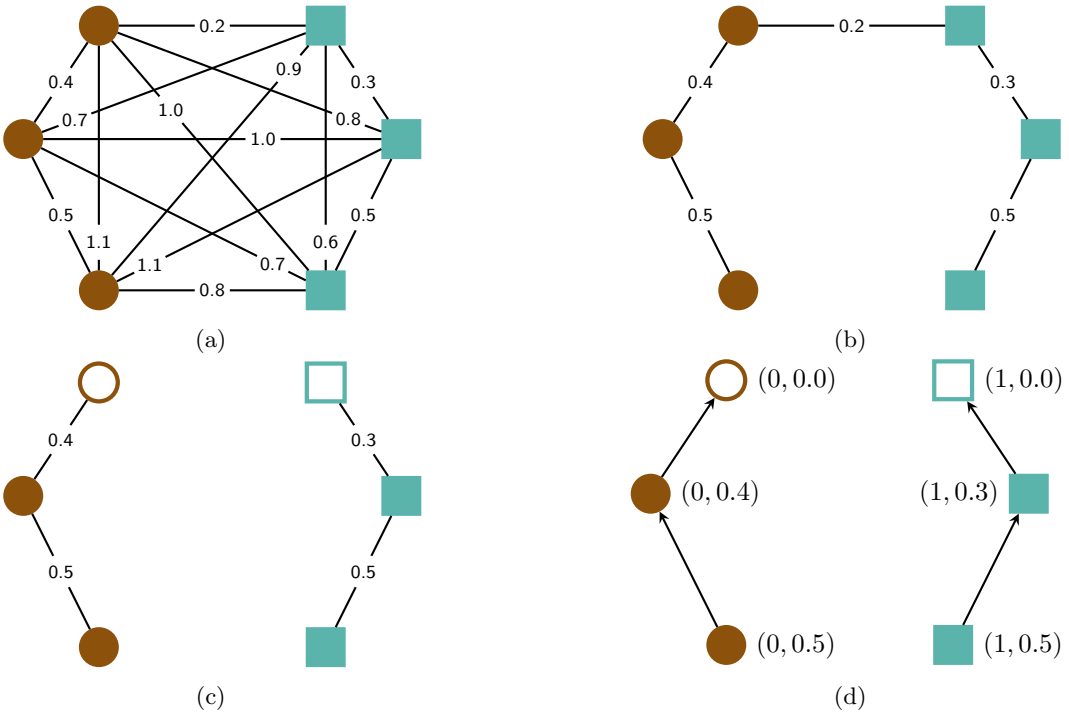

Figure 1: Training sequence for the OPF classifier. In (a), the complete graph with edges weighted by dissimilarity. In (b), the Minimum Spanning Tree is found. Following, in (c), the prototypes are marked and their connections undone. The last step, shown in (d), is to assign the labels and the costs for each node. The prototypes are assigned a cost of 0 and propagate their labels to the nodes in their trees. The cost of the nodes are the maximum value of the edges in the path from them to their respective prototypes.

consists of a Control Logic Module (CLM) and one or more EPA. Communication with HP is done via an interconnected bridge, which also grants access to GM, which is the main interface for data exchange between the processors. The bridge implements a shared access policy which contributes to reducing data access latency and a Direct Memory Access (DMA) channel (not shown Figure 3) grants the parallel processor the ability to read and write data directly, making better use of the bandwidth. The host processor controls the DMA channels, avoiding racing conditions that could corrupt the data, through an arbiter that orders access requests to the shared region, and also protecting the access to reserved areas. This policy also allows the parallel processor to run asynchronously, leaving the host processor free to run other tasks.

Control Logic Module provides an interface with the host processor for receiving commands and the memory addresses to read and writing back. It also distributes the data among the EPAs, control their execution and coordinates the memory access for writing back the results.

Figure 4 details the EPA configuration. Each EPA is composed of a number of Elementary Processors (EP), internal memories and a control module.

Each EPA has its own Local Shared Memory (LSM) with an associated Memory Management (MM) module. This memory is used for fast data exchange between the EPs. The MM module implements the same protection policy to avoid memory corruption that the host processor uses. It is also responsible for coordinating the access to the global memory. There is also a Local Private Memory (LPM) module for each EP. The LPM access is managed by the EP itself, which uses it to store intermediary data.

The Elementary Processors provide the core functionality to the application. They are responsible to effectively execute the computationally intensive task to be accelerated. All the EPs are identical, executing the same operation in different blocks of data, complying with a hardwired SIMD architecture. Figure 5 shows how it is organized.

In this work, the EPs implement the OPF classification algorithm shown in Algorithm 1. The dissimilarity function is the most computationally expensive step in the algorithm, therefore a specific hardware module implements the respective process. Alongside the natural speed gain by implementing the function in a dedicated hardware, the architecture explores parallelism, enabling several data chunks to be processed simultaneously. Nonetheless, the parallel architecture was designed to be flexible enough to be adapted to different applications simply by redesigning the EP.

4.2. System implementation details

Considering that OpenCL is available since 2013 for Altera® products, representing a considerable advance to FPGA development [31], it was adopted for the implementation of the proposed architecture. Because OpenCL is an open standard for parallel programming on heterogeneous

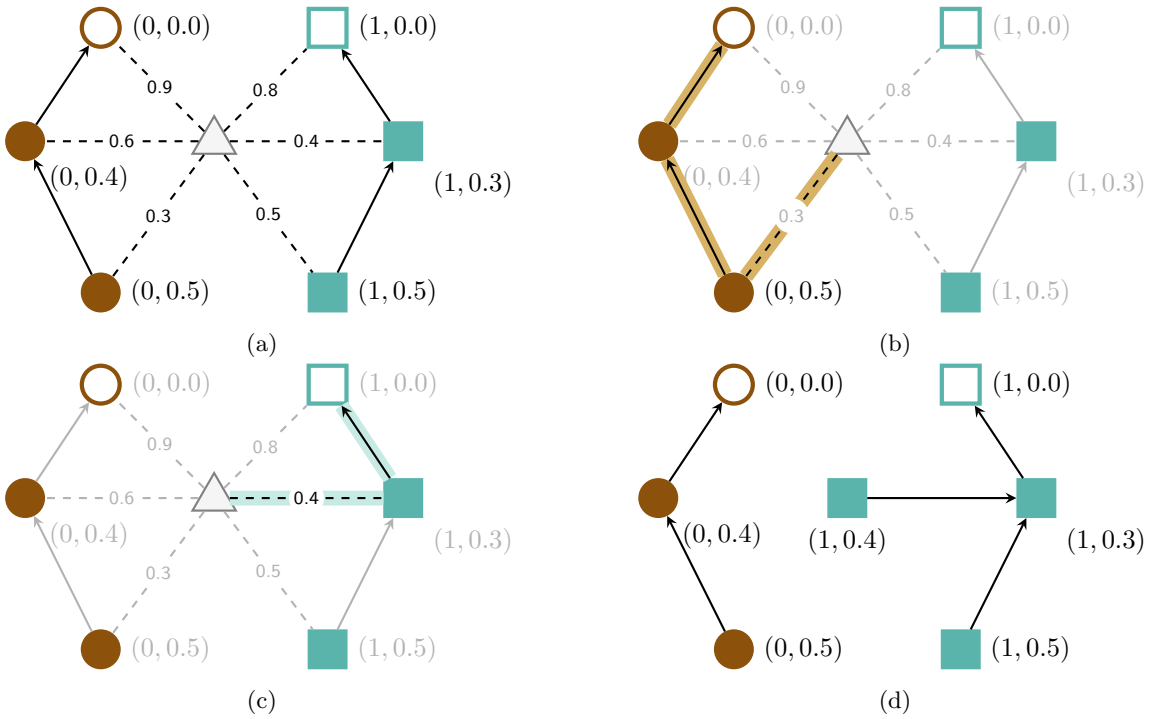

Figure 2: Classification sequence for the Optimum-Path Forest classifier. In (a), the unknown sample is presented to the classifier nodes. Then, they compete to connect to the sample, offering their costs. The cost assumed by the sample is calculated according to OPF connectivity function (Eq. 1). The values in parenthesis next to the nodes are, respectively, the label and the path cost from the node to its respective prototype. In (b), notice in the highlighted path that although the node that generates this path is the closest to the sample, with a distance of (0.3), the final path cost is given according to the OPF connectivity function, that is, the highest edge value in the path to the prototype, resulting in a final path cost of (0.5). Then, in (c), applying the connectivity function to the highlighted path, results in a cost of (0.4). Finally, in (d), after repeating the process to all possible paths, the one with the minimum path cost wins the unknown sample and gives its label. Thus, the sample is classified as one of the square class.

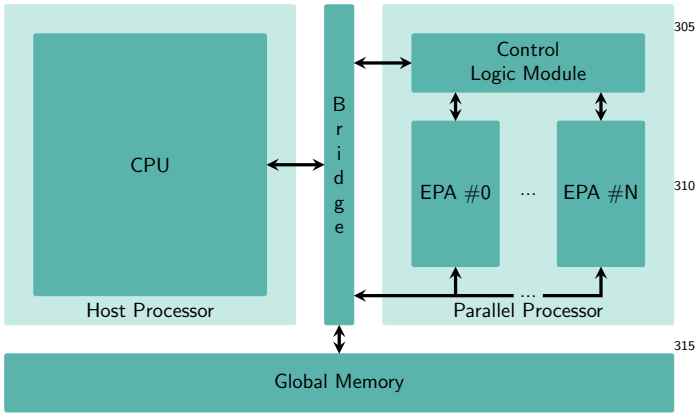

Figure 3: Proposed architecture general overview. The two processing elements communicate through a bridge that also grants memory access. The parallel processor can receive parameters directly from the host processor and seek the data out through a DMA channel accessed from the bridge.

platforms, it may be seen as more than a language, implying a specific workflow, which provides solutions for FPGA design and implementation. Therefore, it complies exactly with the HP/PP framework, which, adopting a SoC/FPGA board, conducts to a natural function distribution between the embedded microprocessor and the FPGA chip. This lead to adopting the Arrow SoCKit as the development platform. It is built around a Cyclone V SoC FPGA chip featuring, in a single chip, a dual-core A9-Cortex ARM processor, an FPGA device and an internal high-speed bridge connecting the processor to the FPGA fabric, as well as a second low-speed bridge. The ARM processor enables the implementation of stand-alone embedded systems, eliminating the need for an external host computer.

Some OpenCL key concepts that organize the parallel code are necessary to be introduced. The first one is the division of the code into parts: host code and device code. The host code is the sequential part of the application, comprising mostly algorithm configuration and execution flow control. The device code is the computationally intensive part that will run on the specific parallel hardware.

Parallel functions in the OpenCL context are called kernels, which are implemented by the programmer taking into account the parts of the application that require parallel execution and using the respective available API (Application Programming Interface). The OpenCL compiler converts it into the platform specific machine code according to the programmable device, in this case the Cyclone V. This means that an OpenCL development is comprised off a sequential part, written in C/C++ and executed in the host, which prepares data to be processed in

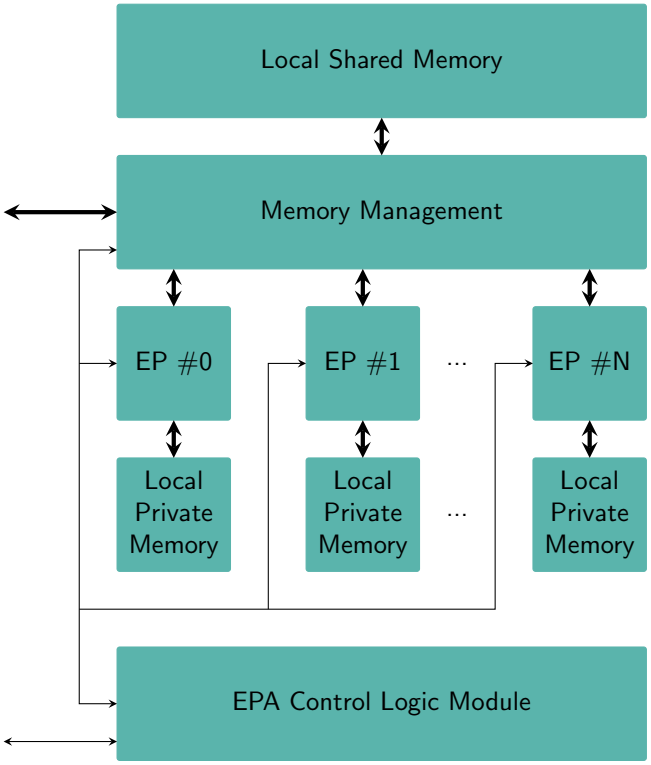

Figure 4: Elementary Processor Array (EPA) block diagram. Each EPA has a Memory Management (MM) module that controls the access to the external global memory and the local shared memory. Each Elementary Processor (EP) can access these memories through the MM. Individual local private memory blocks are accessed and controlled by their respective EPs. A Control Logic Module manages the application flow through parameters received by the external controller.

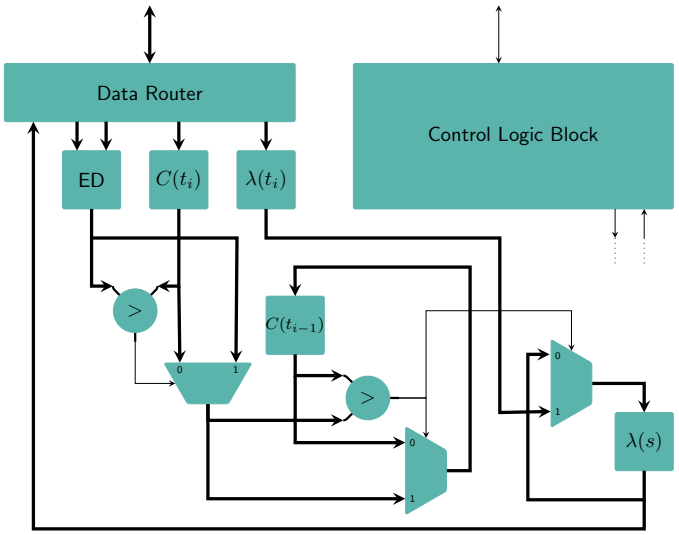

Figure 5: Elementary Processor showing the hardwired OPF algorithm. The control block receives the parameters from the external controller to command the data flow between the processing components. The Euclidean Distance (ED) block perform the calculation using floating point hardware in the FPGA device. The comparisons are them processed in combinational logic, with the controller generating a synchronization signal to update the registers and then writing the result back to the previously assigned memory address.

parallel, and a sequence of kernel calls, written in OpenCL, to process the data in specialized parallel hardware.

Two other OpenCL important concepts are work-items and work-groups. A work-item is the elementary processing element in an OpenCL kernel, meaning the part of the hardware designed to effectively execute parallel code and hardware implemented. A work-group is the user-defined association of a given number or work-items into an abstract encapsulation, in order to organize the parallel execution of the desired kernel function.

4.2.1. OpenCL workflow for FPGA programming

Traditionally, the Register Transfer Level (RTL) workflow is used for FPGA development when a Hardware Description Language (HDL, e.g. Verilog or VHDL) is the programmer option. These languages can always be used to design parallel systems. This process requires a deep knowledge of digital circuit architectures design and involves time-consuming simulation and verification stages. However, the OpenCL workflow has a higher abstraction level, which facilitates the complete development in general, and it has to be forcefully adopted when OpenCL is used. The main advantage of the OpenCL workflow is to incorporate the RTL procedures in a transparent way, which brings efficiency to the development. Figure 6 shows the relation between both workflows.

It may be seen in Figure 6 the RTL workflow comprising some fundamental steps, taken after the code development, which are: functional simulation of the compiled code, synthesis of the approved simulated code, routing of the viable synthesis result considering all power and timing constraints, to finally upload to the target board the bitstream hardware configuration file. All these steps are manually controlled by the programmer, and eventually, the process has to return to the initial step if any intermediary test fails.

The OpenCL workflow, also shown in Figure 6, rests over the RTL one. It means that the RTL steps are yet necessary, but now it proceeds in an automatic and transparent way. After compilation of the OpenCL code, the functional verification is done through an emulation of the system in the host developing computer. After functionally approved, the system is optimized and checked against the resource constraints, and the procedure follows up with the automatic synthesis and routing steps. This avoids the always present respective feedback to the initial step imposed to the manual procedure.

The compiler also takes care of handling the interfacing between the host processor and device processor through predefined interconnect Intellectual Properties (IP) that map the data exchange to the actual hardware. This is traditionally the most time-consuming task in the system's design using RTL workflow, as the developer must tune his solution to work in very specific protocols to be able to manage the data transporting between the system com-

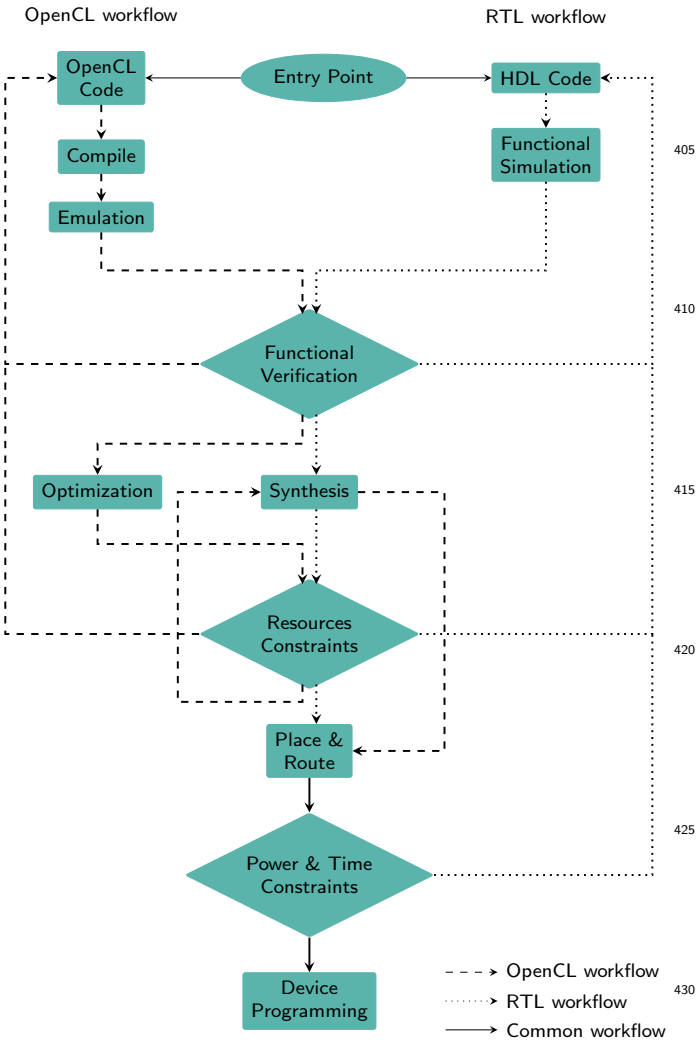

Figure 6: The FPGA OpenCL programming workflow. It incorporates the RTL workflow as an automatic process, based on verified components. This introduces a higher level of abstraction to the development, as the user will concentrate on the algorithm design.

ponents and extract the best performance, and linking the design with the actual hardware in use.

4.2.2. Host Processor code organization

This section details the HP design transcription for implementation on the board. The ARM processor SoC board represents the HP and as such it executes the host code, which is written in C/C++. The code is organized into sub-tasks: input file reading, data preparation, buffer preparation, kernel configuration, kernel launching, presentation of results and resources deallocation. Figure 7 presents the execution flow of each sub-task, which is explained in the following:

- Input file reading: the data sets is organized as two input files, one containing the classifier itself and another with the test data to be classified. Both the files are in the OPF binary format provided by the library.
- Buffer preparation: the OpenCL API uses a specific data structure as buffers to communicate data between Host and Device memory spaces. Therefore, it is necessary to prepare these buffers before moving data around them. As the adopted board uses a shared memory space between the ARM processor and FPGA fabric, there is no need to make an explicit call to write and read functions. Once the buffers are defined, both the Host and Device can access them. The host writes the input data into the corresponding buffers and the device will be responsible for writing its processing results into the output buffers.
- Kernel configuration: Once all the buffers are correctly set, the kernel interface is read and configured to run. During the compiling process, the kernel code is stored into a binary file that holds the image to be configured into the FPGA fabric and its interface description. Each buffer is associated with its corresponding argument in the kernel interface. These steps prepare the kernel to execute.
- Kernel launching: At this point, the execution is transferred from the host code running on the ARM processor to the device synthesized in the FPGA fabric. The FPGA execution is asynchronous, that is, the host code will continue to run independently of the parallel hardware. It is possible, in the case of very complex parallel code, that the host finishes its execution before the kernel finishes. The API provides barrier function calls to prevent this behavior. Once the kernel completes its execution, the results are written into the output buffers and are ready to be accessed by the host.
- Presentation of results: The host code can finally present the results in the manner the user chooses to do so.
- Resources deallocation: Once the application finishes, the buffers must be freed to let the device ready for a new task, if it is the case.

The efficiency of the classifier was evaluated using offline-trained OPF data stored in libOPF [32] format. The final system is flexible enough to permit the use of different datasets, with diverse feature vector dimensions and multiple classes. Therefore, it can be adapted for different classification tasks just changing the data acquisition and feature extraction methods to a more suitable one to the application in question.

4.2.3. Parallel Processor code organization

Algorithm 1 is implemented as a hardwired SIMD architecture in the EP, as shown in Figure 5. The algorithm executes in two loops, in order to classify each sample, which corresponds to the outer loop. For each sample, the inner loop will iterate over the classifier nodes to identify

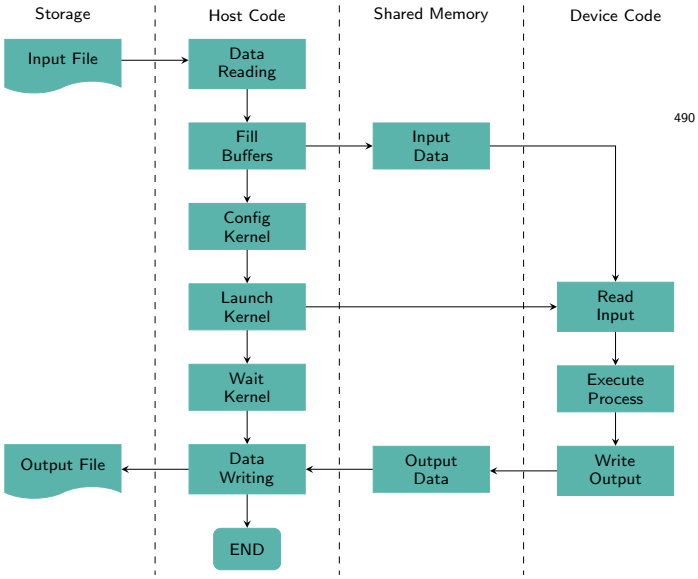

Figure 7: Host code sub-task execution flow, showing the interactions between the processing elements through the shared memory space.

the respective minimum cost prototype, taking into account the euclidean distance between the sample and each node of the forest. Each EP corresponds to an OpenCL work-item and the EPAs correspond to work-groups. The EPs hardwired SIMD code is implemented as an OpenCL kernel. Following these directives, the kernel was organized such that each EP/work-item will load one sample from the test set and perform the inner loop over the classifier nodes. Notice that the number of samples to be classified in parallel is the total number of EPs considering all EPAs. The currently available compiler does not support more than one kernel simultaneous instances, nor to call a kernel inside another kernel, thus restricting the inner loop to run sequentially inside each EP/work-item.

Figure 8 shows how the kernel is organized. The classifier data is shared among the EPs/work-items that belong to the same EPA/work-group, while the input data is divided among all EPs/work-items. The most computationally expensive operation in the OPF classification algorithm is the Euclidean distance calculation. The compiler builds the internal configuration of EPs/work-items as a pipelined structure, finely tuned with the memory access timing, which contributes to increasing the system's throughput by a better employment of the memory bandwidth. The compiler directive `num_compute_units(N)` can be used to replicate compute units N times. Each compute unit corresponds in the architecture to an EPA, and each one is capable of simultaneous execution of different work-groups, thus increasing the parallel processing capacity. However, this replication comes with a penalty in global memory bandwidth, as the access to distribute the data between the units will be shared. To effectively profit from having multiple EPAs, the problem to be processed must have a favorable computation-to-memory access ratio, when the global memory access time will be amortized by the computation time. Therefore, a case-to-case analysis must be performed to find the best configuration for each specific problem.

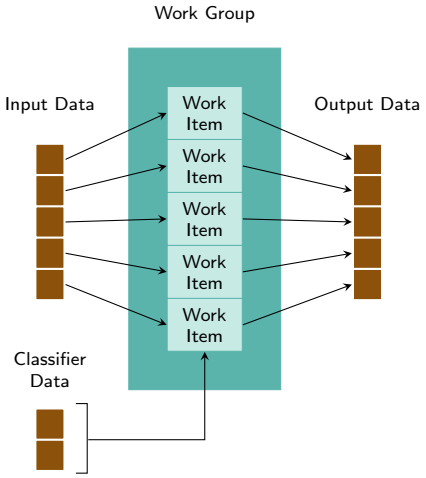

Figure 8: Data distribution for execution by the OpenCL kernel converted in parallel hardware.

This work evaluates two configurations for the PP synthesis, using one or two EPAs per PP. Considering the number of logic elements necessary to synthesize the architecture, two EPAs was the maximum possible on the current hardware. The compiler tries to optimize the shared access, resulting in a rearrangement of the kernel maximum clock frequency that can be achieved in the final synthesis. Aspects of the board implementation also affect the optimizations that the compiler implements, as they are highly coupled with the speed parameters of the memory chips. This results are detailed in the next section.

There are some freedom and some restrictions to configure the parallel execution. Initially, the number of work-items in a work-group was defined as 512, as a design choice to be assessed after testing. Memory access is 32 bit-aligned, using the shared memory controller on the HP, which implements a direct interface between the FPGA fabric and the Hard Processor System (HPS) DDR controller, which in the current configuration implements a data port 256 bits wide. For the adopted board, there is 1 GB of DDR memory. The first 512 MB are reserved for the operational system and the remaining 512 MB are set as the shared memory space. Each kernel is launched with a configuration given by the number of elements to classify divided by the maximum number of work-items. This gives us the required number of work-groups for execution. Uneven results are padded with zeros and the appropriate range verification is done in the EPs. PP configurations with more than one EPA can execute one work-group per EPA, however, as mentioned before, global memory access is serialized between the EPAs. Therefore, to avoid sub-optimal utilization of the hardware capabilities, the task to be processed must have a favorable computation-

to-memory ratio. This relation is well known in parallel code design and serves as a guideline to the appropriate domain division for parallel execution, accounting both for the problem and the parallel hardware characteristics.

5. Results and performance analysis

5.1. Hardware and software specifications

The development board host processor was set up to run at 800 MHz. The operational system used was the image provided by the manufacturer, consisting of a Linux distribution based on the Yocto Project. The distribution was installed to the board with the addition of the OpenCL runtimes libraries that expose the FPGA side interfaces to the Linux Hardware Abstraction Layer. A micro-SD card hosts the operational system and acts as mass storage device.

The offline training of the different datasets used a PC equipped with an Intel® Core™ 2 Quad Q8400 CPU 2 GHz with 8GB DDR2 RAM memory running Ubuntu 14.04 Trusty Tahr operating system. The resulting classifiers were saved in a file and then transferred to the development board micro-SD card.

The comparison was made with the software-only classification running in the ARM-based processor and its accelerated implementation on the FPGA. The libOPF version used was 2.1, the latest one available at the time of this work.

5.2. Metrics and key performance indicators

The focus of the comparison is the processing speed gain obtained by measuring the acceleration provided by the FPGA parallel hardware against its software-only counterpart. The execution times for classifying the whole dataset was measured and the average duration to classify an individual sample was calculated.

The chosen performance metric for evaluating the quality of the classification was the Average Accuracy. As the classifier function used for the software and hardware versions are same, there is no reason to perform a complete qualitative analysis of the results. This metric gives us a general idea of the classification's quality and can be used to assess the divergence (if any) in the results of the two versions caused by design decisions.

As defined in [33], the Average Accuracy measures the average per class effectiveness of a multi-class classifier and is calculated as:

$$Acc = \frac{\sum_{i=1}^{l} \frac{tp_i + tn_i}{tp_i + fn_i + fp_i + tn_i}}{l}, \qquad (2)$$

where l is the number of different classes of the problem and tp, fp, tn and fn stand for, respectively, true positives, false positives, true negatives and false negatives for the $i\text{-}th$ sample in the testing set.

5.3. Dataset descriptions

The system was tested by running the classification in six different datasets. The first five were picked from the publicly available Machine Learning repository of University of California Irvine [34]. The last one is composed of HOG descriptors taken from a compilation of several popular pedestrian detection datasets generated from an authors' previous work [35]. The data sets D1, D3, D5, and D6 are originally from CV applications, with different descriptors used to generate the feature vectors. Using these data sets will permit to analyze the performance of the OPF in diverse CV scenarios. Table 1 compiles each dataset characteristics. Datasets with a different number of classes and attributes are used to assess how this variation affects both the software and hardware versions.

Table 1: Dataset descriptions

Id	Name	# attr.	# classes	# samples
D1	Brest Cancer Winsc. (Diag.)	9	2	569
D2	Glass Identification	9	6	214
D3	Image Segmentation	19	7	2,310
D4	Iris	4	3	150
D5	Parkinsons	22	2	197
D6	Pedestrian	3,780	2	12,160

The classifiers were generated by Repeated Random Sub-sampling, choosing the best instance of 100 different randomly generated collections of training/evaluation/testing sets. The training set used 40% of the total samples, and the evaluation set 20%. The remaining 40% constitutes the testing set.

5.4. Performance analysis

Table 2 presents the accuracy observed for every data set running on each version of the classifier, compiling the kernel to result in only one compute unit (EPA). It presents also the number of samples in the test set, the total time spent in the classification in milliseconds, the average classification time per sample and the speed-up obtained by using the hardware implementation against its corresponding software version. Table 3 shows the same metrics, but now for adopting two compute units (EPAs).

Table 2: Accuracy and classification times for software (S) and hardware (H) versions of the OPF classifier, kernel compiled to generate 1 compute unit

Id/Version	# of samples	Accuracy	Total time (ms)	Avg. time per sample (ms)	Speed-up
D1/SW	276	0.902174	5.66221	0.02051	-
D1/HW	276	0.902174	0.74398	0.00269	7.62
D2/SW	91	0.882784	3.75272	0.04123	-
D2/HW	91	0.882784	0.57631	0.00633	6.51
D3/SW	924	0.927644	220.702	0.23885	-
D3/HW	924	0.927644	40.516	0.04384	5.44
D4/SW	60	0.955556	0.34493	0.00574	-
D4/HW	60	0.955556	0.12171	0.00202	2.84
D5/SW	80	0.8125	4.40499	0.05506	-
D5/HW	80	0.8125	0.77412	0.00967	5.69
D6/SW	4,864	0.801809	1,531,710.0	314.90748	-
D6/HW	4,864	0.801809	389,658.0	80.11060	3.93

Table 3: Accuracy and classification times for software (S) and hardware (H) versions of the OPF classifier, kernel compiled to generate 2 compute units

Id/Version	# of samples	Accuracy	Total time (ms)	Avg. time per sample (ms)	Speed-up
D1/SW	276	0.902174	5.39210	0.01953	-
D1/HW	276	0.902174	0.59944	0.00217	8.99
D2/SW	91	0.882784	3.82270	0.04200	-
D2/HW	91	0.882784	0.60408	0.00663	6.32
D3/SW	924	0.927644	220.7240	0.23887	-
D3/HW	924	0.927644	40.3604	0.04368	5.46
D4/SW	60	0.955556	0.377469	0.00629	-
D4/HW	60	0.955556	0.172921	0.00288	2.18
D5/SW	80	0.812500	4.40748	0.05509	-
D5/HW	80	0.812500	0.80762	0.01009	5.45
D6/SW	4,864	0.801809	1,532,010.0	314.96916	-
D6/HW	4,864	0.801809	379,957.0	78.11615	4.03

The two compute units configuration is the maximum number possible to synthesize, given the available resources on the adopted FPGA model. It can be perceived that changing the number of EPAs does not significantly affect the processing times for the OPF classification algorithm.

Table 4 shows the resource utilization for each kernel configuration. This information can be obtained using the manufacturer's provided profiling tools that accompany the OpenCL SDK.

Table 4: FPGA resource utilization for each kernel configuration

Resource	1 compute unit	2 compute units
ALUTs	14,112	23,710
Registers	22,402	40,056
Logic utilization	11,153	19,364
DSP blocks	8	16
Memory bits	1,450,336	2,640,320

Table 5 shows the bandwidth achieved for each dataset in each kernel configuration also obtained with the profiling tools. The datasets D3 and D6 achieved the highest bandwidth, what is expected, given that they have a bigger number of testing elements, what causes the data transfer to computation ratio to be better. This ratio is also affected by the feature vector length, so it is expected to vary.

Table 6 shows the power estimation for each kernel configuration, also obtained using the corresponding manufacturer's provided tool.

The difference in area between the two configurations is 73% while the difference in power consumption is 34%. However, as the difference in performance between the two configurations was negligible, it is recommended to use in this application the configuration with one compute unit, as it reduces the power consumption and leaves some FPGA fabric free for other uses.

The clock frequency is determined by the compiler as 112.5 MHz for the first configuration with one compute unit and 92.6 MHz for the second one, which are approximately eight times smaller than the HP clock frequency (800 MHz). Nonetheless, It is important to remark that the hardware accelerated processing times were up to 9 times faster than the pure software counterpart running in the HP alone. The

Table 5: Global memory bandwidth for each dataset and kernel configuration

Dataset	Global Bandwidth 1 comp. unit (MB/s)	Global Bandwidth 2 comp. units (MHz)
D1	1,037	926.4
D2	75.1	71.5
D3	2,183.6	2,000.2
D4	133.6	126.7
D5	109.3	100.4
D6	2,080.4	1,885.4

Table 6: Power estimation for each kernel configuration

Component	Thermal Power (W) 1 compute unit	Thermal Power (W) 2 compute units
Logic	0.131	0.239
RAM	0.241	0.446
DSP	0.023	0.046
I/O	0.005	0.005
PLL	0.031	0.031
Clocks	0.122	0.197
HSDI	0.000	0.000
Hard Memory Controller (HMC)	0.000	0.000
XCVR	0.000	0.000
PCS HIP	0.000	0.000
Static	0.315	0.325
Total FPGA	**0.868**	**1.289**
Hard Processor System (HPS)	0.217	0.217
HPS Static	0.139	0.141
Total SoC Power	**1.224**	**1.647**

variation in speed-up values is expected, given the nature of the OPF classification algorithm and the variation in dimensionality and number of classes of each dataset.

6. Conclusions

This work proposes an architecture for embedded system parallel processing comprising a host processor and a parallel multiprocessor array. Its implementation of a computer vision application algorithm in a SoC/FPGA board using the OpenCL language and workflow is also presented. Adopting OpenCL yields, in general, a shorter development time, considering that it implies the use of higher level abstraction and verified IPs and consequently less programming error correction effort.

A software version running on the dual-core ARM host processor is used to assess the acceleration provided by the hardware implementation. The comparison shows that the hardware implementation was able to execute 2.18 to 9 times faster than the software version. Also, two different configurations, using one or two compute units were compared, exposing the differences in global memory bandwidth and energy consumption of each configuration, giving support to decide which configuration to use in specific applications. The achieved acceleration was sufficient to justify the use of the implementation. There is space for improvement though, as some parameters defined at runtime can be fixed at compilation time, aligned to a particular task. This modification contributes to decrease global memory accesses, thus improving the computation-to-memory access ratio and consequently, improving the

processing acceleration and is a good candidate for further investigation.

Future work will consider testing the substitution of the floating point operations by fixed point ones, which generally grants performance improvement against a compromise in precision. For many applications, this precision reduction does not represent a significant loss, considering that the achieved acceleration can be very attractive, or even the viable solution for a hard real-time embedded system. The parallel implementation shall be studied using alternative devices in a future work, in order to validate the proposed architecture to different application demands.

Acknowledgment

This work is supported by the PDSE program of Coordination for Improvement of High Education Personnel (CAPES) of Brazilian Ministry of Education, process nº13077/2013-09 and carried out in the framework of the Labex MS2T, funded by the French Government through the program "Investments for the future" managed by the National Agency for Research (Reference ANR-11-IDEX-0004-02).

References

[1] M. Parker, June 2014 Altera Corporation Understanding Peak Floating-Point Performance Claims, Tech. Rep., 2014.

[2] A. Corporation, Leveraging HyperFlex Architecture in Stratix 10 Devices to Achieve Maximum Power Reduction, Tech. Rep., , 2015.

[3] J. P. Papa, A. X. Falcão, C. T. N. Suzuki, Supervised pattern classification based on optimum-path forest, Int. J. Imaging Syst. Technol. 19 (2) (2009) 120–131, ISSN 08999457, doi:, .

[4] J. P. Papa, A. X. Falcão, V. H. C. de Albuquerque, J. M. R. Tavares, Efficient supervised optimum-path forest classification for large datasets, Pattern Recognit. 45 (1) (2012) 512–520, ISSN 00313203, doi:, .

[5] A. A. Spadotto, J. C. Pereira, R. C. Guido, J. P. Papa, A. X. Falcao, A. R. Gatto, P. C. Cola, A. O. Schelp, Oropharyngeal dysphagia identification using wavelets and optimum path forest, in: 2008 3rd Int. Symp. Commun. Control Signal Process., IEEE, ISBN 978-1-4244-1687-5, 735–740, doi:, , 2008.

[6] R. C. Guido, J. C. Pereira, J. F. W. Slaets, A. I. Iliev, M. S. Scordilis, J. P. Papa, A. X. Falcão, Spoken emotion recognition through optimum-path forest classification using glottal features, Comput. Speech Lang. 24 (3) (2010) 445–460, .

[7] G. Chiachia, A. A. N. A. Marana, J. P. Papa, A. X. Falcao, Infrared Face Recognition by Optimum-Path Forest, in: 2009 16th Int. Conf. Syst. Signals Image Process., IEEE, ISBN 978-1-4244-4530-1, 1–4, doi:, , 2009.

[8] R. Pisani, P. Riedel, M. Ferreira, M. Marques, R. Mizobe, J. Papa, Land use image classification through Optimum-Path Forest Clustering, in: 2011 IEEE Int. Geosci. Remote Sens. Symp., IEEE, ISBN 978-1-4577-1003-2, 826–829, doi:, , 2011.

[9] A. Culquicondor, C. Castelo-Fernandez, J. P. Papa, A New Parallel Training Algorithm for Optimum-Path Forest-based Learning, in: XXI Iberoam. Congr. Pattern Recognit. CIARP' 2016, November, IAPR, Lima, , 2016.

[10] M. V. T. Romero, A. S. Iwashita, L. P. Papa, A. N. Souza, J. P. Papa, Fast optimum-path forest classification on graphics processors, VISAPP 2014 - Proceedings of the 9th International Conference on Computer Vision Theory and Applications 2 (2014) 627–631, .

[11] A. S. Iwashita, M. V. T. Romero, A. Baldassin, K. A. P. Costa, J. P. Papa, Training Optimum-Path Forest on Graphics Processing Units, in: 2014 International Conference on Computer Vision Theory and Applications (VISAPP), 581 – 588, , 2014.

[12] Khronos Group, OpenCL Specification, , 2009.

[13] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, D. P. Singh, From opencl to high-performance hardware on FPGAS, in: 22nd Int. Conf. F. Program. Log. Appl., IEEE, ISBN 978-1-4673-2256-0, 531–534, doi:, , 2012.

[14] A. X. Falcão, J. Stolfi, R. De Alencar Lotufo, The Image Foresting Transform: Theory, Algorithms, and Applications, IEEE Trans. Pattern Anal. Mach. Intell. 26 (1) (2004) 19–29, ISSN 01628828, doi:, .

[15] D. Lowe, Object recognition from local scale-invariant features, in: Proc. Seventh IEEE Int. Conf. Comput. Vis., vol. 2, IEEE, ISBN 0-7695-0164-8, ISSN 0-7695-0164-8, 1150–1157 vol.2, doi:, , 1999.

[16] P. Viola, M. Jones, Rapid object detection using a boosted cascade of simple features, in: Proc. 2001 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognition. CVPR 2001, vol. 1, IEEE Comput. Soc, ISBN 0-7695-1272-0, ISSN 1063-6919, I–511–I–518, doi:, , 2001.

[17] J. Cloutier, E. Cosatto, S. Pigeon, F. Boyer, P. Simard, VIP: an FPGA-based processor for image processing and neural networks, in: Proc. Fifth Int. Conf. Microelectron. Neural Networks, IEEE Comput. Soc. Press, ISBN 0-8186-7373-7, 330–336, doi:, , 1996.

[18] C. Farabet, C. Poulet, Y. LeCun, An FPGA-based stream processor for embedded real-time vision with Convolutional Networks, 2009 IEEE 12th Int. Conf. Comput. Vis. Work. ICCV Work. (2009) 878–885doi:, .

[19] J. Javier Martínez, J. Garrigós, J. Toledo, J. Manuel Ferrández, An efficient and expandable hardware implementation of multilayer cellular neural networks, Neurocomputing 114 (2013) 54–62, ISSN 09252312, doi:.

[20] J. Manikandan, B. Venkataramani, V. Avanthi, FPGA Implementation of Support Vector Machine Based Isolated Digit Recognition System, in: 2009 22nd Int. Conf. VLSI Des., IEEE, ISBN 978-0-7695-3506-7, ISSN 1063-9667, 347–352, doi:, , 2009.

[21] J. Gimeno Sarciada, H. Lamel Rivera, M. Jiménez, CORDIC algorithms for SVM FPGA implementation, in: H. H. Szu, F. J. Agee (Eds.), SPIE Defense, Secur. Sens., International Society for Optics and Photonics, 77030G–77030G–8, doi:, , 2010.

[22] S. Bauer, S. Kohler, K. Doll, U. Brunsmann, FPGA-GPU architecture for kernel SVM pedestrian detection, in: 2010 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. - Work., IEEE, ISBN 978-1-4244-7029-7, 61–68, doi:, , 2010.

[23] M. Papadonikolakis, C. Bouganis, Novel Cascade FPGA Accelerator for Support Vector Machines Classification, IEEE Trans. Neural Networks Learn. Syst. 23 (7) (2012) 1040–1052, ISSN 2162-237X, doi:, .

[24] D. Mulligan, Implementation of real-time pedestrian detection on FPGA, in: 2008 23rd Int. Conf. Image Vis. Comput. New Zeal., IEEE, ISBN 978-1-4244-2582-2, 1–6, doi:, , 2008.

[25] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, K. Doll, FPGA-Based Real-Time Pedestrian Detection on High-Resolution Images, in: 2013 IEEE Conf. Comput. Vis. Pattern Recognit. Work., IEEE, ISBN 978-0-7695-4990-3, 629–635, doi:, , 2013.

[26] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, E. Culurciello, Hardware accelerated convolutional neural networks for synthetic vision systems, Proc. 2010 IEEE Int. Symp. Circuits Syst. (2010) 257–260doi:, .

[27] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, E. Culurciello, An efficient implementation of deep convolutional neural networks on a mobile coprocessor, in: 2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS), IEEE, ISBN 978-1-4799-4132-2, 133–136, doi:, , 2014.

[28] V. Gokhale, J. Jin, A. Dundar, B. Martini, E. Culurciello, A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks, in: 2014 IEEE Conference on Computer Vision and Pattern Recognition

Workshops, IEEE, ISBN 978-1-4799-4308-1, 696–701, doi:, , 2014.

[29] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444, ISSN 0028-0836, doi:, .

[30] J. P. Papa, F. a.M. Cappabianco, A. X. Falcao, Optimizing Optimum-Path Forest Classification for Huge Datasets, in: 2010 20th Int. Conf. Pattern Recognit., Ieee, ISBN 978-1-4244-7542-1, ISSN 1051-4651, 4162–4165, doi:, , 2010.

[31] K. Hill, S. Craciun, A. George, H. Lam, Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA, in: 2015 IEEE 26th Int. Conf. Appl. Syst. Archit. Process., vol. 2015-Septe, IEEE, ISBN 978-1-4799-1925-3, ISSN 10636862, 189–193, doi:, , 2015.

[32] J. Papa, C. Suzuki, A. Falcao, LibOPF A library for the design of optimum path forest classifiers, , 2014.

[33] M. Sokolova, G. Lapalme, A systematic analysis of performance measures for classification tasks, Inf. Process. Manag. 45 (4) (2009) 427–437, ISSN 03064573, doi:, .

[34] M. Lichman, {UCI} Machine Learning Repository, , 2013.

[35] W. F. S. Diniz, V. Fremont, I. Fantoni, E. G. O. Nóbrega, Evaluation of optimum path forest classifier for pedestrian detection, in: 2015 IEEE Int. Conf. Robot. Biomimetics, Ml, IEEE, Zhuhai, ISBN 978-1-4673-9675-2, 899–904, doi:, , 2015.