HLS code refactoring using SDSoC applied to multiclass SVM classification of Hyperspectral Images

Abelardo Báez Quevedo, Himar Fabelo, Samuel Ortega, Gustavo M. Callicó, Roberto Sarmiento
Research Institute for Applied Microelectronics
(IUMA), University of Las Palmas de Gran Canaria
(ULPGC), Las Palmas de Gran Canaria (Spain)
{abaez, hfabelo, sortega, gustavo, roberto}@iuma.ulpgc.es

***Abstract*— Nowadays, High-Level Synthesis (HLS) methods and tools are a highly relevant area in the strategy of several leading companies in the field of System on Chips (SoCs) and Field Programmable Gate Arrays (FPGAs). HLS allows FPGA manufactures to widen the target market, smoothing the existing barriers that prevented potential users from adopting reconfigurable hardware technologies and easing the work of system developers, who benefit from integrated and automated design workflows, considerably reducing the "time to market" constrain. On the other hand, although many advances have been made in this research field, there are some uncertainties about the quality and performance of the designs that results from the use of HLS processes. Since HLS tools increase the level of abstraction, it is necessary to evaluate if the possible performance losses compensate the design time reduction. For these reasons, it is highly important to know how to write efficient code for HLS tools, being mandatory a good understanding of the HLS methods to achieve the best results. In this paper, an optimization of the HLS methodology by code refactoring using SDSoC™ (Software-Defined System-On-Chip) is presented. Several options were analyzed for each alternative through the code refactoring of a multiclass Support Vector Machine (SVM) classifier written in C, using the programmable logic of a Zynq®-7000 SoC device by Xilinx. Thus, a quantitative evaluation of the results achieved is presented order to provide designers with a methodology that will speed up their implementations.**

Keywords—High-Level Synthesis, HLS, SDSoC, Support vector mcahines, SVM, hardware friendly code, Zynq

I. INTRODUCTION

High-level synthesis (HLS) methodologies allow hardware (HW) designers to increase the abstraction level and accelerate the automation for the synthesis and verification of the design process. The current raise in the complexity of the applications and the increment of the capabilities of silicon technologies, as well as the so called *time to market* constrain, will make HLS methodologies and tools of mandatory use in the near future [1]. Due to the multiple solutions that multiprocessor system-on-chip (MPSoC) manufacturers are commercializing nowadays, it is strictly necessary to provide improvements in the development of techniques and methodologies that can deal with the multiple implementations possibilities by using a high-level design [2].

Some implementations of SVM in FPGAs have been released in different applications, like imaging processing, automotive, medical applications and data signal processing among others. These implementations use different platforms depend on the application and the desired accuracy and timing. For readers who are interested in different implementations using different devices and including not only a training implementation but also a classification implementation, we recommend [3]. Another interesting research from the same authors is a SVM classifier for melanoma detection using a Zynq device (ZC7020) and HLS methodology, but in this case the dataset are not HSIs (HyperSpectral Images), the model used in that research generates an output of the class as 1 (melanoma), -1 (non-melanoma) [4] and the implementation depend on directives used directly in Vivado HLS. Finally, it is relevant taking into account that in every implementation the communication between the software and the hardware part in an embedded system represents the bottleneck to save. For example, in [5] the different stages of a LS-SVM implementation using a Zynq device is approached separating the code of the algorithm is different parts, depend on the communications necessary for each part, then some parts are suitable to compute using the ARMs than implementing them in the PL part.

In this paper, an evaluation of code refactoring and SDSoC design methodology and implementation is performed. To test the implementation design flow, the SVM codes are modified to increase the speedup. In Section 2, the most relevant specifications of the research work are described, like the device specifications (Zynq), SDSoC tool, and the basic SVM main features. In Section 3, a summary of the images and their features are detailed. In Section 4, a detailed explanation of code refactoring of the binary and the multiclass SVM classification algorithms is provided, together with an explanation of the methodology. This paper concludes including some results in Section 5 and exposing the conclusions in Section 6.

II. TOOLS AND PLATFORMS

A. *Zynq SoC* by *Xilinx*

Zynq is a SoC provided by Xilinx [6]. All versions have the same PS features, a dual-core ARM Cortex A9 (ARMv7-A architecture), 32KB Level 1 cache for instructions, and 32KB Level 1 cache for data. The two cores share a 512KB L2 cache, and a 256KB on-chip memory (OCM). The PL part can

access the DDR memory, the OCM memory, and the L2 cache in the PS via AXI interfaces, with coherency behaviour through the Accelerated Coherency Port (ACP). The resources of the PL part depend on the version selected, in this paper two Zynq versions were selected, a ZC7020 in a Zedboard Evaluation Kit [7] and the ZC7045 in a Xilinx Zynq-7000 SoC ZC706 Evaluation Kit [8]. These devices prevent the designer from wasting too much hardware or software design time, increasing the communication performance between the two parts by using the provided communication interfaces, but sometimes some modifications are required to get a suitable HLS implementation. The transactions between the PL and the PS part suppose a relevant challenge for the designer and dramatically affect the final system performance.

B. *SDSoC* by *Xilinx*

SDSoC is a tool developed by Xilinx that provides the designer with the possibility of creating complete embedded systems from C or C++ code using Zynq devices as the target system. This kind of tools mean a new level over the traditional HLS tools and it is of interest in the research community [9], [10]. SDSoC includes a system compiler that analyses the code in order to determine the data flow between the PS and PL part and provides the designer the complete system; invokes Vivado to create the system and Vivado HLS to create the IPs for the desired accelerated functions, which includes the accelerated functions and the Data Movers IP for transaction data. In order to provide an efficient time implementation, the tool generates a thread for each accelerated function ensuring synchronization between software and hardware threads. The designer can configure the communication between PL and PS part in the code with SDSoC pragma directives to meet the application and solution constrains and adds Vivado HLS directives to create the desired accelerated IP.

The methodology applied in this paper includes the methodology proposed by Xilinx [11] with some modifications creating a well-defined six-step design flow as shown in Fig. 1. After the code verified in ARM checking the results, the first step in the design flow is the profiling stage, in this step a profiling tool is needed to detect the functions that must be accelerated. This step can be carried out with different profiling tools, like Valgrind for memory usage and gprof for timing.

Because SDSoC uses Vivado HLS, the next step includes the optimization suggested by the Vivado HLS and SDSoC guidelines. The next step consists of code refactoring, restructuring the source code for an improvement of the latency. In some cases this step is mandatory if an speedup is desired, and without this code refactoring step, the accelerator could not be affordable. This step is the main contribution of this paper, and it is explained in detail in section IV.

The next step is to obtain the performance estimation provided by the tool, and check if the result is the desired one. In this step a detailed report of resources and speedup of accelerated functions is provided and an iteration can be done to improve the expected implementation. The final iteration of this step depends on the resources of the PL part, and the resources used will be shown in the HLS report obtained in the next step.

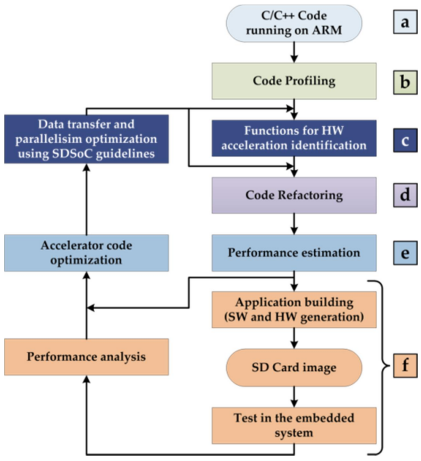


Fig. 1. SDSoC Design Flow.

This step is driven by constraints, SDSoC compiler directives and code refactoring, this step have a high impact in the quality of the final implementation.

The designer can use also Vivado HLS directives with SDSoC directives. The directives give instructions to the compiler to meet the characteristics of the hardware architecture and the desired timing constrains (e.g. use of pipelines to implement loops, type of communication channels for data-flow implementations (Data Movers), FPGA resources to be used for variable storage, etc.). To improve the results is necessary to take into account the inferred implementation of the compiler tool.

The final step lets the designer check the estimated performance in the board selected. SDSoC invokes Vivado HLS in order to generate the HDL implementation files for the accelerated functions in HDL (VHDL or Verilog), and several comprehensive synthesis reports.

The information provided in the synthesis reports helps the designer to meet the performance and resource-usage requirements for a specific application. SDSoC also generates all the files needed to run the application in the embedded system, the bitstream for the PL part, the connection between PS and PL part (Data Movers), and the files of the OS in Linux or FreeRTOS with the executable binary (ELF) for running the application. This final step is mandatory due to the difference between the real and the estimated performance because real performance usually is lower than the estimated one.

C. *Support Vector Machines Classifier*

Support Vector Machines (SVMs) algorithm is a binary classification approach proposed by Vapnik in 1979 [12]. The main goal of this algorithm is to find a hyperplane that separates two classes according to its features with maximum margin. We are given a set of data x_i ($x_i \in \mathbb{R}^d$) and labels

associated to this data ($y_i \in \mathbb{R}$). Each label provides information about data x_i, if $y_i = 1$ the class is positive and if $y_i = -1$ the class is assumed to be negative. For example, if we are dealing with a diagnostic test, a positive class could mean 'disease' while a negative can represent 'non disease'. According to the input data x_i, we can write the following equation:

$$\hat{y} = x_i \cdot w + b \quad (1)$$

Where $\hat{y}$ is the predicted class for the instance x_i, and the parameters w and b define the maximum margin hyperplane ($w \in \mathbb{R}^d$ and $b \in \mathbb{R}$). These parameters, w and b, are learned from a training set, consisting of tuples of data and labels (x_i, y_i). One of the main features of the SVM algorithm is that it can be easily generalized for non-linear data [13], which is useful for complex data where a linear separation hyperplane is not capable to separate data accurately. Similarly to other binary classifiers, SVM can be extended to a multiclass classifier by combining several binary classifiers [14].

SVMs are kernel-based supervised classifiers that have been widely used in the classification of HS (HyperSpectral) images [15]. In the literature, SVMs achieve good performance for classifying HS data, even when a limited number of training samples are available [16]. Due to its strong theoretical foundation, good generalization capabilities, low sensitivity to the curse of dimensionality, and ability to find global classification solutions, many researchers usually prefer SVMs instead of other classification algorithms for classifying HS images [17].

In this paper, we cover the implementation of the SVM classification stage. The determination of the parameters from data is out of the scope of this work. In this sense, we employed an implementation of the basic SVM binary classifier to perform the experiments and optimizations. Then, a multiclass SVM classifier implementation based on the one-vs-one method was used apply and evaluate the optimizations proposed with the binary algorithm. The pseudocode of the multiclass SVM algorithm is presented in Algorithm 1. This algorithm was split in 4 different stages: 1) Variables declaration and initialization; 2) Binary probability computation; 3) Multiclass probability computation; 4) Output classification map generation.

III. In-Vivo HS Human Brain Cancer Database

In this work, the HS data employed to evaluate the performance of the implementations belong to an *in-vivo* HS human brain cancer database [18]. This database was generated intraoperatively using an HS acquisition system developed in [15] during the execution of the HELICoiD project [19]. Particularly, three HS images that belonged to three adult patients undergoing craniotomy for resection of intra-axial brain tumor at the University Hospital Doctor Negrin of Las Palmas de Gran Canaria (Spain) were employed. The patients had a confirmed grade IV glioblastoma tumor by histopathology. The study protocol and consent procedures were approved by the *Comité Ético de Investigación Clínica-Comité de Ética en la Investigación* (CEIC/CEI) of the University Hospital Doctor Negrin and written informed consent was obtained from all subjects. HS data from these images were labeled as tumor and normal tissue following the method explained in [20]. The HS dataset are formed by 128 (Fig. 2.a) spectral bands, respectively, covering the spectral range between 450 and 900 nm. Fig. 2.b shows the synthetic RGB representations of the HS cubes selected for this study.

1) Variables declaration and initialization

$x \rightarrow sample\ to\ classify$
$W = [w_{12}, w_{13}, \dots, w_{nm}] \rightarrow w - vectors$
$B = [b_{12}, b_{13}, \dots, b_{nm}] \rightarrow bias$
$Sa = [Sa_{12}, Sa_{13}, \dots, Sa_{nm}] \rightarrow sigmoid\ parameters\ a$
$Sb = [Sb_{12}, Sb_{13}, \dots, Sb_{nm}] \rightarrow sigmoid\ parameters\ b$

2) Binary probability computation:

for $i = 1$ **to** $n_{class} - 1$ **do**
  for $j = i + 1$ to $n_{class} - 1$ **do**
    $d_{ij} = w_{ij} \cdot x + b\_ij$
    $P_{ij} = \frac{1}{1+e^{(d_{ij} \cdot Sa_{ij} + Sb_{ij})}}$
    $P_{ji} = 1 - P\_ij$
  end
end

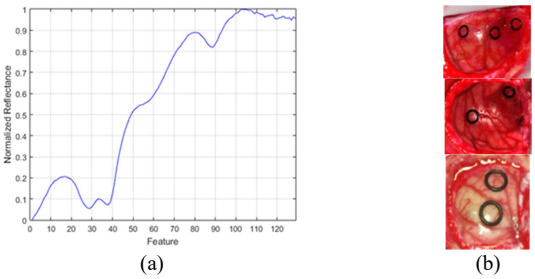
3) Multiclass probability computation:

$Pc_1 = \cdots = Pc_n = \frac{1}{n_{class}}$
for $i = 1$ **to** n_{class} **do**
  $Q_{ii} = \sum_{j \neq i}^{n_{class}} P_{ji}^2$
  $Q_{ij} = Q_{ji} = -P_{ij} * P_{ji}$
  $Qp_j = \sum_{j=1}^{n_{class}} Q_{ij} * p_{cj}$
end
$p^T Qp = \sum_{i=j}^{n_{class}} Qp_i * p_{ci}$

4) Output classification map generation:

for $iterations = 1$ **to** 100 **do**
  if $\forall\ i\ \in n_{class} | Qp_i - p^T Qp < epsilon;$ ***break***
  for $i = 1$ **to** n_{class} **do**
    $diff = \frac{-Qp_i + p^t Qp}{Q_{ii}}$
    $P_{ci} = P_{ci} + diff$
    $p^T Qp = \frac{p^T Qp + diff * (diff * Q_{ii} + 2 * Qp_i)}{(1+diff)^2}$
    for $j = 1$ **to** n_{class} **do**
      $Qp_j = \frac{Qp_j + diff * Q_{ij}}{1+diff}$
      $Pc_j = \frac{Pc_j}{1+diff}$
    end
  end
End
$Pc = [Pc_1, Pc_2, \dots, Pc_n] \rightarrow class\ probability$

Algorithm 1: Multiclass SVM Classification algorithm



(a) (b)

Fig. 2. HS in-vivo brain human database. (a) Example of the spectral signatures of the HS dataset. (b) Synthetic RGB representations of the HS images, where the data were labelled and extracted to conform the HS database employed in this work. From up to bottom: OP8C1, OP12C1 and OP20C1.

IV. CODE REFACTORING

The starting point of this work are two different software implementations of the SVM classifier. The first one is an own implementation of a binary SVM classifier, and the second one is a one-vs-one multiclass SVM classifier. The first version of the binary classifier was written in C++ language and modified in plain C following a hardware friendly way, however, the one-vs-one classifier was written in plain C but in a non-hardware friendly way. One of the goals in this research is to take advantage of the methodology used for the binary classifier in the one-vs-one classifier.

The main goal of this work is to avoid an excessive number of modifications that are dependent on the tool, in order to reuse the code in the future with different HLS tools. The reference code has been modified until the final implementation showed clear indications of reaching the performance objectives. After each change or restructuration in the code, a serial verification was performed in order to check the results. These modifications were applied to the binary classifier code. Once the optimal modification was reached, the same methodology to the one-vs-one classifier code was applied.

A. Improving on transferring data

If the accelerated function only process one pixel in each iteration, not speedup is obtained even with the pragmas of the tool. In order to improve the acceleration of the classification function, several pixels are transferred between PS and PL part in the same clock cycle. Due to the 533 MHz DDR3 SODIMM bandwidth constrain, an optimal amount of data must be selected in order to avoid wasted data cycles. Because the implemented system not always can achieve the total bandwidth, it is necessary to find the highest data transfer near to the bandwidth constrain. It is necessary to take into account that the amount of pixels is not always an integer multiple of the optimal amount of pixels for a data cycle, so zero padding is a good option to avoid calculating non-existent values. Fig. 3 shows the re-factored code applied in order to improve the transferred data using the proposed modification, where `BLOCKSIZE` is the amount of pixels in each data transfer, `BANDS` is the number of bands values for each pixel, `PIXELS` is the number of pixels in the image and `inputInter/outputInter` are the arrays for intermediate input/output data transfer.

B. Improving on processing data

The classification function has a temporal dependency because the actual value at each iteration depends on its value in the previous iteration. Each classification value for a pixel (*clValue*) is calculated adding the *bias* data and then accumulating the result of multiplying the weight of every band obtained in the training classification (*bandWeight*), by the value of the pixel in that band (*bandWeight*). So pipelining is not possible to be used in the function given in (2), and pipeline pragma do not improve the speedup.

$$clValue \mathrel{+}= bandValue \cdot bandWeight \qquad (2)$$

```
int nElemBlocks = BLOCKSIZE * BANDS;
int lastElement = BANDS * PIXELS;
int currentPixel = 0;

for (int currentElement = 0;
     currentElement < lastElement;
     currentElement += nElemBlocks){

  for (int element = 0;
       element < nElemBlocks;
       element++){

    if(currentElement+element<lastElement){
      inputInter[element] =
      inputData[currentElement + element];
    }else{
      inputInter[element] = 0;
    }
  }
  svmClassifyHW(inputInter, bias, weights,
                outputInter);
  for(int pixelIndex = 0;
      pixelIndex < BLOCKSIZE;
      pixelIndex++){

    if(currentPixel + pixelIndex < PIXELS){
       output[currentPixel + pixelIndex] =
              outputInter[pixelIndex];
    }
  }
  currentPixel += BLOCKSIZE;
}
```

Fig. 3. Modified code for transferring a block of pixels.

To improve the execution of this function, to calculate *clValue,* instead of using just one accumulator, several intermediate accumulators are implemented. At the end, the final value for *clValue* is the addition of the intermediate accumulators. Fig. 4 shows the modified code applied in order to improve the processing data using the proposed modification. In this figure lets the pipelining implementation to use 8 accumulators, where `BLOCKSIZE` is the number of pixels for each data transfer, `BANDS` is the amount of bands for each pixel, `intputData[n]` is the array with the pixel values, `outputVector[n]` is the array with the classification results, `weights[n]` is the array with the weights for the classification and `inter[m]` is the array for intermediate accumulators.

```
for( int i = 0; i < BLOCKSIZE; i++){
  for (int k = 0; k < 8; k++){
    #pragma HLS pipeline
    inter[k] = 0;
  }
  for (int m=0; m < BANDS/8; m++){
    for (int j=0; j<8; j++){
      #pragma HLS pipeline
      inter[j] += inputData[i*BANDS + m*8 +j]
                  * weights[m*8 + j];
    }
  }
  outputVector[i] = biasData + inter[0]  +
                    inter[1] + inter[2]  +
                    inter[3] + inter[4]  +
                    inter[5] + inter[6]  +
                    inter[7]
}
```

Fig. 4. Modified code to parallelize the data processing in groups of 8.

Fig. 5 shows a diagram of the improving on transferring and processing data where P is the number of pixels, P_n is the block of pixels processed in each data transfer, B_n is the block of bands in which is divided the total bands value for each pixel, A_n represent the intermediate accumulators, and A is the final accumulator for that pixel.

C. Including redundant data inside accelerated function

Every time the classification is called, bias and weights are transferred via the data-mover IP to the accelerated function in the PL part.

Classification data type is double (4 bytes, 32 bits), therefore every time (2) is called, bias and the corresponding weight needs to be transferred for computation for each pixel in the image. If SVM training is done before, weights will not change, hence weights and bias can be included in the IP reducing the data transfer and improving the speedup.

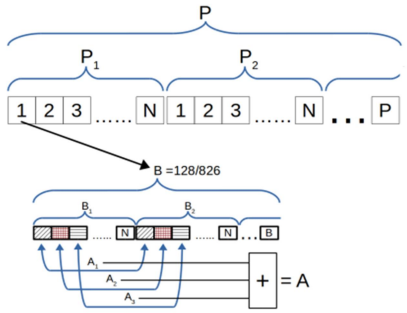


Fig. 5. Diagram of pixels and bands parallelized.

V. Results

All the results were obtained on board, no estimate performance was used in these results. In summary about 70 implementations were tested in order to obtain accurate results. Each implementation had an iteration of 100 classifications on board to obtain a reliable average. Linux was used as OS in all the implementations for controlling and verification purposes. The speedup was calculated calling the classification twice, the first one in software without any modification at all and the second one in hardware, with all the modifications incorporated.

The first result obtained without the code refactoring shows a speedup of 0.67×, this result was the main reason to change the code in order to find a better implementation.

Once the code was improved by changing the amount of pixels per clock cycle and improving on the processing data, and selecting 100 MHz for Data Movers IP and 100 MHz for accelerated function, it was shown the cycles between 1.15× and 1.41×, as can be seen in Fig. 6. It is worth noticing that the speedup decreases once the blocksize (number of pixel per clock cycle) increases above 128 pixels. In this picture a comparison with and without the modification can be se

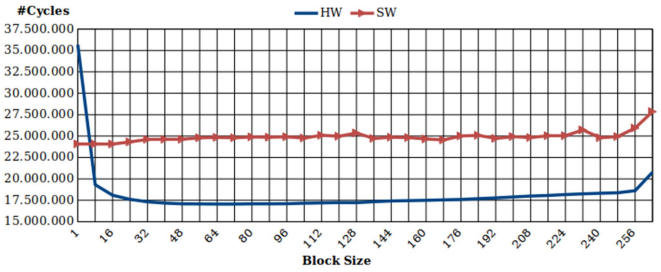


Fig. 6. Speedup changing the amount of pixels per clock cycle (100MHz for Data Movers Data Moversand accelerated function)

Increasing the frequency for Data Movers and for accelerated function to 200 MHz shows a speedup of 1.61×. Including weights and bias inside the accelerated function and keeping the 200 MHz for Data Movers and accelerated IP showed a speedup of 2.35×. Finally, keeping all the configurations, 200 MHz for Data Movers and accelerated function, including weights and bias in the accelerated function, and changing the data type from double to float showed a speedup of 2.89×.

Fig. 7 shows a speedup comparison applying all the above modifications, and using different pixels per data cycle and different partitions for bands value. In the best case, with the code refactoring and changing the data type, the highest speedup achieved is 2.89× with a blocksize of 64 pixels per data cycle and partitioning the bands value by 16.

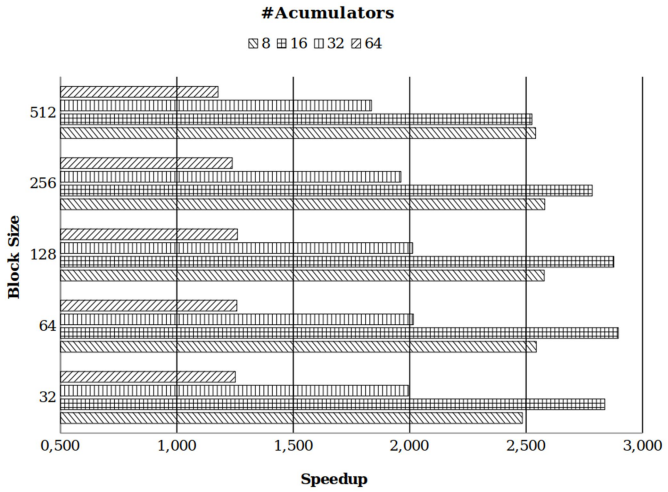


Fig. 7. Speedup changing the amount of pixel per clock cycle and partitions per bands value (200 MHz for Data Movers and accelerated function)

Finally the same methodology was applied to the multiclass SVM classifier. In this case the code was divided in 4 stages (see 0), and once the performance analysis was obtained two versions were implemented, the full one (including all the stages in the PL part) and the separated one implementing only the most intensive computational stage in the PL part. In the second case, the stage implemented on the PL part was number 1. This difference allows us to compare the speedup versus the resources occupied in the PL part and the power consumption.

Table I shows the time consumption and speed up using the Zedboard (ZC7020) for SW and HW implementation and in both cases the full implementation, and the separated one.

TABLE I. Cycles and SpeedUp for both implementations (F= Full, S=Separated)

Image	Pixels×bands	Type	SW Cycles	HW Cycles	Speedup
Op8C1	251,532×128	F	6,709,635,352	4,772,617,172	1.40×
		S	9,169,489,664	4,160,169,766	2.20×
Op12C1	219,232×128	F	5,883,668,448	4,186,056,302	1.40×
		S	7,996,559,609	3,623,266,697	2.20×
Op20C1	124,033×128	F	3,324,090,523	2,366,485,944	1.40×
		S	4,522,192,859	2,052,877,331	2.20×

Table II shows the resources occupied using the ZC706 (ZC7045) and the Zedboard (ZC7020) for both cases, full and separated implementation. In all cases the selected frequency for PL part is 100 MHz.

Finally, Table III shows the power consumption for ZC706 and Zedboard using both implementations. As can be seen comparing the results, it is a good idea separate the code, since consumes less power than the full one, use less resources and obtain better latency.

TABLE II. RESOURCES CONSUMPTION FOR BOTH IMPLEMENTATIONS (F= FULL, S=SEPARATED)

Board	**ZedBoard (ZC7020)**		**ZC706 (ZC7045)**	
Type	**F**	**S**	**F**	**S**
LUT (%)	67.22	20.22	16.68	4.84
LUTRAM (%)	5.32	4.30	1.27	1.00
FF (%)	33.72	14.18	7.45	2.76
BRAM (%)	4.64	6.07	1.19	1.56
DSP (%)	49.55	15.45	12.11	3.78
BUFG (%)	9.38	9.38	9.38	9.38
MMCM (%)	25.00	25.00	12.50	12.50

TABLE III. POWER CONSUMPTION (W) FOR BOTH IMPLEMENTATIONS (F= FULL, S=SEPARATED)

Board	**ZedBoard (ZC7020)**		**ZC706 (ZC7045)**	
Type	F	S	F	S
Dynamic Power	2.424	1.890	2.618	1.911
Static Power	0.171	0.152	0.224	0.219
Total	**2.595**	**2.042**	**2.843**	**2.13**

VI. CONCLUSIONS

The results obtained in this work demonstrates the major benefits of writing an efficient code for HLS tools, in this case SDSoC, to improve an accelerated version of a SVM classifier. This methodology can be replicated in other tools in order to validate the inferred system, as only few tool directives have been used. It is recommended to include in the accelerated function all the redundant data in order to decrease the interfaces between PS and PL parts, thus improving significantly the speedup of the system.

ACKNOWLEDGMENT

This work is supported by the Spanish Ministry of Science and Innovation as part of the I+D+I Plan support programs in the context of PLATINO: Plataforma HW/SW distribuida para el procesamiento inteligente de informacion sensorial heterogenea en aplicaciones de supervision de grandes espacios naturales (TEC2017-86722-C4-1-R) and ITHaCA: IdenTificacion Hiperespectral de tumores CerebrAles (ProID2017010164).

REFERENCES

[1] S. Lahti, P. Sjovall, J. Vanne, and T. D. Hamalainen, "Are We There Yet? A Study on the State of High-level Synthesis," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, pp. 1–1, 2018.

[2] R. Nane *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, 2016.

[3] S. M. Afifi, H. Gholamhosseini, and R. Sinha, "Hardware Implementations of SVM on FPGA : A State-of-the-Art Review of Current Practice," *Int. J. Innov. Sci. Eng. Technol.*, 2015.

[4] S. Afifi, H. GholamHosseini, and R. Sinha, "A low-cost FPGA-based SVM classifier for melanoma detection," in *2016 IEEE EMBS Conference on Biomedical Engineering and Sciences (IECBES)*, 2016, pp. 631–636.

[5] M. Ning, W. Shaojun, P. Yeyong, and P. Yu, "Implementation of LS-SVM with HLS on Zynq," in *Proceedings of the 2014 International Conference on Field-Programmable Technology, FPT 2014*, 2015.

[6] Xilinx, "Zynq-7000 All Programmable SoC Data Sheet: Overview," *Ds190*, 2017.

[7] "ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide," 2014.

[8] Xilinx, "ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/%0Aug954-zc706-eval-board-xc7z045-ap-soc.pdf. [Accessed: 03-Feb-2018].

[9] M. Cacciotti, V. Camus, J. Schlachter, A. Pezzotta, and C. Enz, "Hardware Acceleration of HDR-Image Tone Mapping on an FPGA-CPU Platform Through High-Level Synthesis," in *International System on Chip Conference*, 2019.

[10] M. Kowalczyk, D. Przewlocka, and T. Krvjak, "Real-Time Implementation of Contextual Image Processing Operations for 4K Video Stream in Zynq UltraScale+ MPSoC," in *Conference on Design and Architectures for Signal and Image Processing, DASIP*, 2018.

[11] Xilinx, "SDSoC Environment User Guide UG1027," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017%0A_4/ug1027-sdsoc-user-guide.pdf. [Accessed: 03-Feb-2018].

[12] V. N. Vapnik and S. Kotz, *Estimation of dependences based on empirical data*. Springer, 2006.

[13] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the fifth annual workshop on Computational learning theory - COLT '92*, 1992.

[14] C. Hsu and C. Lin, "A comparison of methods for multiclass support vector machines," *Neural Networks, IEEE Trans.*, 2002.

[15] G. Mountrakis, J. Im, and C. Ogole, "Support vector machines in remote sensing: A review," *ISPRS J. Photogramm. Remote Sens.*, vol. 66, no. 3, pp. 247–259, May 2011.

[16] G. Camps-Valls and L. Bruzzone, "Kernel-based methods for hyperspectral image classification," *IEEE Trans. Geosci. Remote Sens.*, vol. 43, no. 6, pp. 1351–1362, Jun. 2005.

[17] Q. Li, X. He, Y. Wang, H. Liu, D. Xu, and F. Guo, "Review of spectral imaging technology in biomedical engineering: achievements and challenges," *J. Biomed. Opt.*, vol. 18, no. 10, p. 100901, Oct. 2013.

[18] H. Fabelo *et al.*, "In-Vivo Hyperspectral Human Brain Image Database for Brain Cancer Detection," *IEEE Access*, pp. 1–1, 2019.

[19] H. Fabelo *et al.*, "HELICoiD project: a new use of hyperspectral imaging for brain cancer detection in real-time during neurosurgical operations," in *Hyperspectral Imaging Sensors: Innovative Applications and Sensor Standards 2016*, 2016.

[20] H. Fabelo *et al.*, "An intraoperative visualization system using hyperspectral imaging to aid in brain tumor delineation," *Sensors*, vol. 18, no. 2, p. 430, Feb. 2018.