Hardware Acceleration on Gaussian Naive Bayes Machine Learning Algorithm

Georgios Tzanos
Dep. of Electr. and Computer Engineering
NTUA
Athens, Greece
grg.tzan@gmail.com

Christoforos Kachris
ICCS-NTUA
& DUTH
Greece
kachris@microlab.ntua.gr

Dimitrios Soudris
Dep. of Electr. and Computer Engineering
NTUA
Athens, Greece
dsoudris@microlab.ntua.gr

***Abstract*—Naive Bayes is one of the most effective and efficient classification algorithms and its classifiers still tend to perform very well under unrealistic assumptions. Especially for small sample sizes, naive Bayes classifiers can outperform the more powerful classifiers. Therefore the acceleration of such an algorithm becomes a great asset in machine learning applications. The SDSoC environment provides a framework for developing and delivering hardware accelerated embedded processor applications using standard programming languages. A Hardware Acceleration project has been implemented on the all-programmable MPSoC and it has been evaluated on a typical machine learning application based on Naive Bayes. The Naive Bayes kernel has been developed as accelerator in both training and prediction part. The performance evaluation shows that the heterogeneous accelerator-based MPSoC can achieve up to 16.8x system speedup compared with an embedded ARM processor in the training part and 14x speedup in the prediction part. The accelerator had been also integrated with Python using the Pynq-Z1 device.**

I. Introduction

Most software today is written so that instructions are executed in sequence, and to speed up execution programmers have typically pushed the hardware designers to build processor with ever higher clock rates. That has given rise to heavily pipelined processors that operate at clock rates of 3 GHz and more. These processors also include architectural tricks such as large caches, and functions such as out-of-order execution to get the most out of every clock cycle. However, faster processors generate lots of heat and today, clock speeds have, for the most part, leveled off since the heat generated by the faster circuits ends up constraining the clock speeds. To continue the march towards ever-faster execution, hardware designers have switched from a single processor on a chip to dual, quad, and even more CPU cores on a single chip. The operating system can then allocate the processors to different applications, all running in parallel. The next level down from there is to find ways to parallelize the code running in each application and then run that parallelized code on multiple engines either within the CPU or in a companion co-processor that is optimized to execute that particular segment of parallelized code.

In the latest generation field programmable gate arrays (FPGAs), designers can accelerate computationally-complex algorithms such as encryption, compression, search and sort, up to 1000x times over a general-purpose processor. Also, DSP algorithms that need billions of integer or floating-point operations per second for image and audio processing, can readily be accelerated by an FPGA-based co-processor. So this issue has been identified in the literature as the dark silicon era in which some of the areas in the chip are kept powered down in order to comply with thermal constraints and one way to address this problem is through the utilization of hardware accelerators. Hardware accelerators can be used to offload the processor, increasing the total throughput.

II. Related Work

In the last few years, there are several efforts for the efficient deployment of hardware accelerators based on Naive Bayes.

In [1] was proposed and implemented a real-time face detection system on FPGA. Face detection was based on a Naive Bayes classifier that classified an edge-extracted representation of an image. The FPGA system used about six times less resources than a comparable FPGA face detection system.

In [2] was proposed a VLSI architecture of Naive Bayes classifier for multi-classification on FPGA. The objective of this work is to facilitate real time classification of the facial expressions into seven categories: happy, surprise, sad, disgust, fear, anger and neutral, which could be used in any monitoring system including lie detector.The implemented architecture can perform real time classification operating at a frequency of 241.55 MHz. Accuracy increased as the number of extracted features increased.

In [3] was proposed a real-time face detection system based on Naive Bayesian classifier using Field programmable gate array(FPGA). The detection system divided into three main parts, feature extraction, candidate face detection, and false elimination.As a result of the FPGA parallel processing capabilities, in 640x480 resolutions, the face detection of an image executes within 16.7 milliseconds.

In [4] was proposed a hardware architecture for Naive Bayes inference engine that was used to classify email contents for spam control. The inference engine design is synthesized targeting the Altera Stratix FPGA device and the Naive Bayes inference engine was found to have the capability to classify more than 117 million features per second.

In [5] was proposed a Naive Bayes design on FPGA using very limited hardware resources and runs quickly and efficiently in both training and testing phases. It was first tested on a handwriting digital number dataset, and then applied in the visual object recognition on a single FPGA based visual surveillance system. It was compared with a binary Self Organizing Map (bSOM) using tri-states operation on FPGA, and the experimental results demonstrated both its higher performance and lower resource usage on the FPGA chip.

In this paper we present a utilization of hardware accelerator that can be used in embedded systems. Compared to other implementations on FPGA based on Naive Bayes which are mainly use multinomial and Bernoulli distributions trying to increase the classification performance, our work is giving emphasis to optimize the performance of this machine learning algorithm on the FPGA using the Gaussian distribution. Our purpose was to exceed embedded ARM processors, turning our implementation into a good prospect for embedded systems that execution speed really matters, something that we eventually achieved based on our experimental results. While we also integrated our hardware accelerator in a Python environment, through CFFI and precompiled libraries, for an efficient and low-latency communication using Pynq.

III. Gaussian Naive Bayes

Naive Bayes Classifier (NBC) is a simple probabilistic classifier based on Bayess theorem[6]. It builds a probability model on the category description for all feature vectors in the training set. During the testing, makes classifications using the Maximum A Posteriori decision rule.

A. The Model

The goal of any probabilistic classifier is, with features x_1 through x_n and classes c_1 through c_k, to determine the probability of the features occurring in each class, and to return the most likely class. Therefore, for each class, we want to be able to calculate $P(c_i|x_0, \ldots, x_n)$.In order to do this, we use Bayes rule. Recall that Bayes rule is the following:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

and in our case is:

$$p(Y = C|\mathbf{x}) = \frac{p(\mathbf{x}|Y = C)p(Y = C)}{\sum_{k=1}^{|C|} p(\mathbf{x}|Y = C_k)p(Y = C_k)}$$

both the denominator and the numerator can become very small, typically because the $p(xi|Ck)$ can be close to 0 and we multiply many of them with each other. To prevent underflows, one can simply take the log of the numerator. Therefore, in order to prevent underflows: If we only care about knowing which class ($\hat{y}$) the input $(x = x1, \ldots, xn)$ most likely belongs to, with the maximum a posteriori (MAP) decision rule, we don't have to compute the denominator. For the numerator we can simply take the logarithm to prevent underflows: $log(p(x|Y = C)p(Y = C))$. More specifically:

$$\begin{aligned} \hat{y} &= \underset{k \in \{1,\ldots,|C|\}}{\operatorname{argmax}} \; p(C_k|x_1, \ldots, x_n) \\ &= \underset{k \in \{1,\ldots,|C|\}}{\operatorname{argmax}} \; p(C_k) \prod_{i=1}^{n} p(x_i|C_k) \end{aligned}$$

which becomes after taking the log:

$$\hat{y} = \underset{k \in \{1,\ldots,|C|\}}{\operatorname{argmax}} \; \log\left(p(C_k|x_1, \ldots, x_n)\right) \tag{1}$$

$$= \underset{k \in \{1,\ldots,|C|\}}{\operatorname{argmax}} \; \log\left(p(C_k) \prod_{i=1}^{n} p(x_i|C_k)\right) \tag{2}$$

$$= \underset{k \in \{1,\ldots,|C|\}}{\operatorname{argmax}} \left(\log\left(p(C_k)\right) + \sum_{i=1}^{n} \log\left(p(x_i|C_k)\right)\right) \tag{3}$$

where

$$P(x_i \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_{C_k}^2}} e^{-\frac{(x_i - \mu_{C_k})^2}{2\sigma_{C_k}^2}}$$

*σ = standard deviation, μ = mean

B. Training

Let assume he have a dataset in which every line contains the class and the features of an object that belongs to this class. In order to train our model based on this dataset we have to calculate the means of every feature of every class, the variance of every feature of every class and the prior terms.Thus we have all the necessary values in order to calculate Gaussian Bayes Probability.Additionally in order to avoid multiple accesses on data array, which is not an optimized option for a hardware implementation, we use the formula bellow in order to calculate variance.

$$\sigma^2 = \frac{\sum (X - \mu)^2}{N} = \frac{\sum X^2}{N} - \mu^2$$

C. Classification

Now that we have a way to estimate the probability of a given data point falling in a certain class, we need to be able to use this to produce classifications. Naive Bayes handles this in a very simple manner; simply pick the c_i that has the largest probability given the data points features.This is referred to as the Maximum A Posteriori decision rule. This is because, referring back to our formulation of Bayes rule, we only use the $P(B|A)$ and $P(A)$ terms, which are the likelihood and prior terms, respectively.

IV. Accelerator Architecture

SDSoC development Environment is a very powerful tool in the hands of designers making hard acceleration not only more designer-friendly but also more sufficient taking full advantage of the benefit of the device.

In this project we have used SDSoC in order to accelerate Naive Bayes machine learning algorithm both in Training and Prediction part. The fact that both functions are highly intensive depending on the given data, gives us the ability to take advantage of the device whenever it is more suitable for our application. In particular we had two completely different functions to accelerate. The training function is mainly communication intensive while it does not have very complex mathematical calculations while the prediction function is mainly computational intensive.The use of SDSoc gives us the capability to make use of the internal architecture of FPGA in a way to achieve maximum performance.

One of the main reasons that this heterogeneous system can achieve such a performance is the setup of communication between the PS(The embedded ARM processor) and the PL(Programmable Logic) which is the AXI(Advanced eXtensible Interface)-Interface(Fig.1). Therefore we can achieve the fastest data transferring for our application, firstly by defining the access pattern as sequential in order to generate the streaming interface. Secondly, by specifying the data-mover type, used to transfer the array arguments, as AXI_DMA simple which provides high-bandwidth direct memory access between memory and AXI4-Stream-type peripherals and finally by determining the memory port that provides a cache coherent interface between PL and external memory, for fast cache flushing/invalidation.

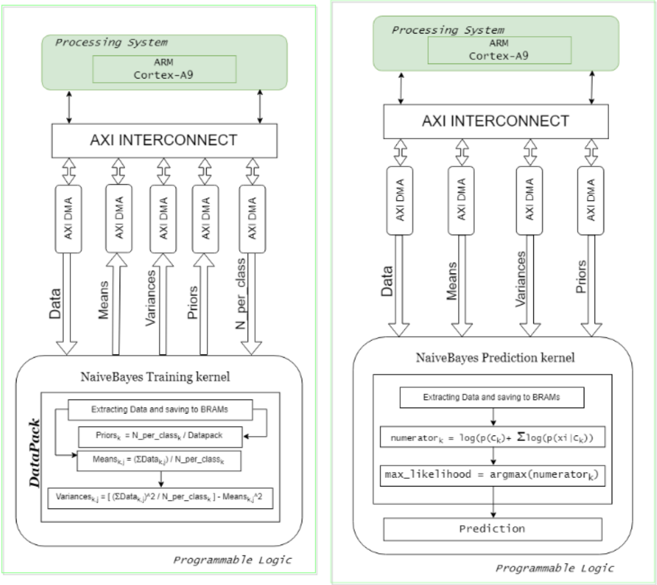
Processing System
ARM
Cortex-A9
AXI INTERCONNECT
AXI DMA
AXI DMA
AXI DMA
AXI DMA
AXI DMA
Data
Means
Variances
Priors
N_per_class
NaiveBayes Training kernel
DataPack
Extracting Data and saving to BRAMs
Priors$_k$ = N_per_class$_k$ / Datapack
Means$_{k,j}$ = (ΣData$_{k,j}$) / N_per_class$_k$
Variances$_{k,j}$ = [(ΣData$_{k,j}$)^2 / N_per_class$_k$] - Means$_{k,j}$^2
Programmable Logic
Processing System
ARM
Cortex-A9
AXI INTERCONNECT
AXI DMA
AXI DMA
AXI DMA
AXI DMA
Data
Means
Variances
Priors
NaiveBayes Prediction kernel
Extracting Data and saving to BRAMs
numerator$_k$ = log(p(C$_k$)+ Σlog(p(xi|C$_k$))
max_likelihood = argmax(numerator$_k$)
Prediction
Programmable Logic

Fig. 1. Simplified Block Designs

The second part of our hardware accelerator implementation is to perform a data-flow analysis to understand how data has to move between the different logic and computational elements and then perform a latency analysis to try and determine where potential bottlenecks may occur and find the best possible performance depending on the cost of the implementing design. Taking into consideration the fact that in our implementation has been used a ZedBoard device.

The main challenge was to find the balance between device-sources and theoretical analysis of the problem.

A. Training Implementation

Our Naive Bayes training implementation is based on three main points.

The first one is to try to store as many data is possible in local BRAMs in order to have direct access from the FPGA and reduce to the minimum the latency of transferring the data from the external memory. We accomplished that by copying the data per line to local arrays:

```
for (i = 0; i < lines; i++){
   for (j = 0; j < N_feat; j++){
   #pragma HLS pipeline II=1
      loc_data[j] = data[offset+i*N_feat+j];
   }
}
```

The second main point in our implementation is to try to avoid memory access bottlenecks. More specifically partitioning large arrays into multiple smaller arrays or into individual registers we were capable of improving access to data and removing block RAM bottlenecks:

```
#pragma HLS array_partition variable = var
    block factor=28
#pragma HLS array_partition variable =
    feature_means block factor=28
#pragma HLS array_partition variable =
    sq_feature_means block factor=28
for (class = 0; class < N_class ; class++){
   ...
   for ( k = 0; k < 28; k++){
      for (j = 0; j < N_feat; j+=28){
      #pragma HLS pipeline II=1
         feature_means[j+k]=(sums[j+k]/(float)
             N_per_class[class]);
         sq_feature_means[j+k]=(sq_sums[j+k]/(
             float)N_per_class[class]);
         var[k+j] = sq_feature_means[j+k] - ((
             feature_means[k+j])*(
             feature_means[k+j]));
      }
   }
}
```

Last but not least, and probably the most important modification in the algorithm is the pipelining of nested loops. By pipelining the loops in the design, through ***pragma HLS pipeline***, with the best possible Initiation Interval and Iteration Latency we have the maximum parallelism and therefore our design can now compete an optimized software implementation.

B. Prediction Implementation

Prediction implementation also follows the same principles but the nature of the algorithm allow us to have bigger acceleration. As we can see probability calculation formula is really intensive and this is something we tried to take advantage. In this case beside the extraction of the data to local BRAMs and the partition of large arrays into smaller into avoid Block Ram bottlenecks we also used two other algorithmic techniques.

The first one is that we separate the formula in independent calculations and save them in individual local registers.The purpose of this manual unroll of this calculation is to both avoid bottlenecks and use as much DSPs cores as we can simultaneously increasing the parallelization:

```
for (j = 0; j < N_feat; j++){
#pragma HLS pipeline II=1
   temp[0] = loc_data[j] - loc_mean[i][j];
   temp[1] = temp[0] * temp[0];
   temp[2] = (-2) * loc_var[i][j];
   temp[3] = temp[1] / temp[2];
   A[j] = temp[3];
   temp[4] = 2 * Pi * temp[2]/(-2);
   temp[5] = sqrt(temp[4]);
   B[j] = log(1/temp[5]);
}
```

The second technique we implement, is the use of a tree-adder(Fig.2) to the individual results we have from the previous stage. We create an array of eight elements and we add its values decreasing the stages the accumulator needs from (n) to log(n).

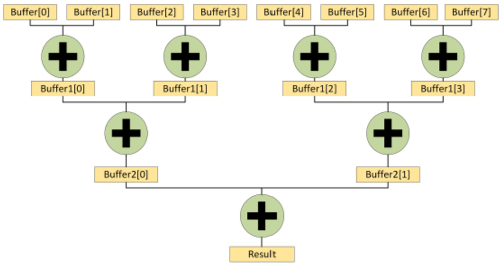

Fig. 2. Tree adder

Thus by using techniques which allows FPGA to run in the most optimized way, we can achieve the maximum parallelization. The C programming experience that SDSoC provides, alongside the directives to determine kernel operation, give designers the friendly development environment of a high level programming language with the simultaneously ability to interfere with the hardware making designing easier to be developed and surely easier to maintain.

V. Performance Evaluation

As a case study, we built a classification model with 784 features and 10 labels using 2k available training samples, for a handwritten digits recognition problem (MNIST)[7].

In cases that the communication of the processor and the accelerator is often and bidirectional, this latency can be a major overhead and may diminish the speedup of the accelerator. However, in applications where the processor sends a bulk amount of data (e.g. through the AXI streaming interface), the communication overhead is overlapped by the computation time. In the case of the Gaussian Naive Bayes, the processor needs to send a large amount of data for the training of the application (as the training part is not so computational intensive) and therefore the communication overhead is overlapped by the computation time being up to 16.8x times faster than an ARM processor .On the other hand, prediction part of the algorithm because of its high computational nature although it calculates each data line independently reaches up to 14x time faster than an ARM.

In terms of resource allocation, Table I shows the utilization of the hardware resources for the Zynq FPGA SoC in Training function and the utilization of the hardware resources for the Zynq FPGA SoC in Prediction function. (Fig.3) depicts the performance of both functions according to measurements performed in the Xilinx ZedBoard evaluation board (C implementation) using SDSoc.

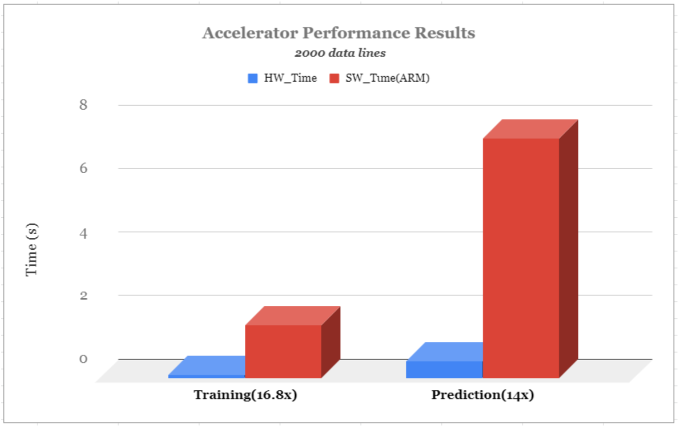

Fig. 3. Performance Results

Moreover, we observed that as the size of the buffer(packets), that is transferred through AXI stream to the accelerator, gets smaller (such as 10 or 40 data lines), the communication overhead limits the speedup. If the size of the packets sent to the training accelerator is over 400 data lines, then the communication overhead is overlapped by the computational saving in terms of execution times. The maximum kernel speedup (16.8x) is achieved when the packet size is almost 2000 data lines (8 Mbytes). This means that by splitting each partition in 2K (to make use only of simple DMAs) we can exploit our accelerator to the maximum.

TABLE I
Hardware Functions Recourses

Recourses	*Used*	*Available*	*Utilization*
Training			
DSP	197	220	89.5%
BRAM	56	140	40%
LUT	37929	53200	71.3%
FF	29271	106400	27.5%
Prediction			
DSP	93	220	42.3%
BRAM	112	140	80%
LUT	34977	53200	65.7%
FF	31779	106400	29.8%

VI. PYNQ: All Programmable Systems on Chip(APSoCs)

Xilinx released in 2016 the Pynq framework that allows the utilization of the heterogeneous all-programmable SoC based on Python[8].Using the Python language and libraries, designers can exploit the benefits of programmable logic and microprocessors in Zynq to build more capable embedded systems. Programmable logic circuits are presented as hardware libraries called overlays. These overlays are analogous to software libraries. Trying to set a cluster of nodes(Pynq-workers) as a future work, taking advantage of the benefits that Pynq offers, we first measured the performance of our application running on a single Pynq compared both to an ARM processor and an Intel i5(4th generation).

We achieved this implementation by firstly creating a pre-compiled library (.so) which mainly contains the hardware functions we previously implemented with SDSoC and Vivado and secondly by using CFFI which is a function interface for calling C code. The fact that, Python is one of the most suitable programming-languages for our cluster but at the same time is slow, gives us remarkable results in our application compared to common processors. We had a direct access to our Vivado/HLS driven implementation through CFFI that helped us achieve a competitive performance towards an Intel Xeon cluster. In particular, results showed that for 2000 data lines in both training and prediction can achieve a significant speedup as it is shown in the following table:

Training	Time (s)	Speed-up
Hw_accel(Pynq)	1.1s	-
Arm-A9(Pynq)	124s	112x
Intel i5	4.5s	4.1x

Prediction	Time (s)	Speed-up
Hw_accel(Pynq)	2.7s	-
Arm-A9(Pynq)	840s	311x
Intel i5	66s	24.4x

VII. Conclusion

Hardware accelerated data processing is still in an early stages, but is likely to become more widespread, as the accelerating technologies continue to make strides over todays CPU architecture as they can improve significantly the performance and the energy efficiency of data analytic applications.

We have implemented a hardware accelerator for Naive Bayes that is connected to processor through the AXI interface. The proposed system can reduce the execution time compared to an embedded processor up to (16.8x) times in the training part and up to (14x) times in the prediction part.

We also integrated our hardware accelerator with Python through CFFI we achieved execution speed, which resembles to C implementation, in Python.

VIII. Acknowledgment

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 687628 - VINEYARD, the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 2212-CloudAccel and it is supported by Xilinx University program.

References

[1] Nguyen, D., D. Halupka, P. Aarabi, and A. Sheikholeslami. Real-Time Face Detection and Lip Feature Extraction Using Field-Programmable Gate Arrays. IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics) 36, no. 4 (August 2006): 90212.

[2] P. Chaudhary and M. K. Sharma, VLSI Hardware Architecture of Real Time Pattern Classification using Nave Bayes Classifier, in Proceedings of the 2017 2nd International Conference on Multimedia Systems and Signal Processing - ICMSSP 2017, Taichung, Taiwan, 2017, pp. 6165.

[3] Y. P. Chen, C. H. Liu, K. Y. Chou and S. Y. Wang, "Real-time and low-memory multi-face detection system design based on naive Bayes classifier using FPGA," 2016 International Automatic Control Conference (CACS), Taichung, 2016, pp. 7-12.

[4] M. Marsono, M. W. El-Kharashi, and F. Gebali. Binary lnsbased na[i-umlaut]ve bayes inference engine for spam control: noise analysis and fpga implementation. IET Computers and Digital Techniques, 2(1):5662, 2008.

[5] H. Meng, K. Appiah, A. Hunter, and P. Dickinson, FPGA implementation of Naive Bayes classifier for visual object recognition, in CVPR 2011 WORKSHOPS, Colorado Springs, CO, USA, 2011, pp. 123128.

[6] M. Bayes and M. Price. An Essay towards Solving a Problem in the Doctrine of Chances. By the Late Rev. Mr.Bayes, F. R. S. Communicated by Mr. Price, in a Letter to John Canton, A. M. F. R. S. Philosophical Transactions, 53:370418, 1763.

[7] Y. LeCun and C. Cortes. The MNIST database of handwritten digits. 1998.

[8] Pynq: Python productivity for Zynq, http://www.pynq.io/