A survey of FPGA implementation of machine learning algorithms

Chandini Shetty
Department of Computer Engineering
University of California, Riverside
Riverside, CA
cshet001@ucr.edu

ABSTRACT

This paper looks at FPGA based implementation of popular machine learning algorithms- Support Vector Machines (SVM's) and clustering algorithms. One is a supervised learning algorithm while clustering is unsupervised algorithm. Machine learning algorithms usually operate on large datasets and have significant amount of computations that can be parallelized. They involve some form of Matrix-Vector or Matrix-Matrix multiplications. SVM's are considered to be one of the most efficient classifiers introduced by Vapnik et al in 1995 and are very popular. This paper looks at several different implementation of SVM that have been done on FPGA's and the various optimizations to the algorithm itself for suitable FPGA acceleration. Some of the earlier implementations try to optimize the DSP/LUT's ratio to achieve good area utilization along with power efficiency, while others look at better parallelization. Next part of the paper looks at FPGA implementations of clustering algorithm- k-means. This algorithm is widely used in varying domains such as bioinformatics, medical image applications, image segmentation etc. The goal is to look at the techniques used in efficiently mapping the algorithm to FPGA hardware and speedup's achieved. As a final part, the paper looks at two generic frameworks proposed in recent works that aim to generate FPGA implementations for a class of Machine learning algorithms without being specifically tune to one specific algorithm. The key ideas are of pattern based hardware generation and abstract programming concepts to model the algorithm in hardware circuits.

Keywords

FPGA; Machine learning; k-means; SVM classifier; Reconfigurable hardware

1. INTRODUCTION

Machine learning algorithms are being widely adopted in various applications that involve recognition and prediction related tasks. These algorithms are however very computationally intensive and usually involve large dense matrix based operations. As a result on-chip caches and caching based strategies, which are at the core of general purpose processors, are not suitable for these algorithms- especially in real world scenarios were the data involved is huge. Most of the algorithms operate in such a way that one iteration or one cycle of the computation will go over GB's of data without requiring to re-use them. So caching strategies would not be of much help. Hence several works have looked at accelerating these algorithms on GPU's and FPGA's. This paper looks at acceleration of one class of supervised ML algorithm -SVM's and one unsupervised- k-means clustering. SVM's have become popular for classification tasks because they can be used on varied types of datasets and are robust to errors. But a significant challenge is the training phase of SVM's which involves significant computations. One of the popular benchmarks, that several of the papers surveyed in this project have considered, is the MNIST dataset which consists of 2 million images of handwritten digits(0-9), each 28 x 28 and hence 784 dimensions. Typical number of the support vectors that would be required to be calculated is about 60,000 which will require 2M vector products(1.6 billion MAC's) [1]. Also any new data point(a new handwritten digit's image) that will need to be classified will also involve significant computations. All these numbers give a general idea of how computationally intensive machine learning algorithms are and why multi-core architecture will not be able to perform very well in these domains.

K-means is widely used in data-mining applications such as big data analysis, pattern recognition, image processing. It is ubiquitous mainly due to its algorithmic simplicity and is probably the most commonly used algorithm in data-analysis- especially to find patterns or to group objects into clusters based on how similar they are. It is however computationally intensive and is limited in terms of parallelization that can be achieved- because the intermediate steps has to be done sequentially. Only each iteration can proceed in parallel, since the next iteration requires calculated centroids from previous step to proceed. More background on the algorithm is provided in next section.

Section 2 provide brief explanation of the two algorithms- SVM and k-means. Section 3 looks at popular implementation of SVM on FPGA's. Section 4 looks at popular k-means implementation on FPGA's. Section 5 details a high level overview of generic approaches to hardware generation and

CS 223 Spring 2016, UC Riverside, CA, USA
© 2016 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

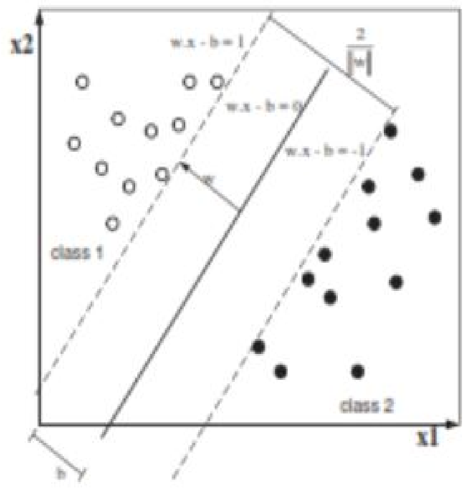

Figure 1: A two class SVM classifier with 2 attributes. Dotted line is the margin and bold line is the classifier's decision boundary

their performance. Section 6 provides some concluding observations

2. BRIEF BACKGROUND ON THE ALGORITHMS

2.1 SVM

For a two-class classification problem, where target classes are {-1.+1}, the SVM algorithms tries to find the best separating hyperplane and with a margin, such that the distance between opposite classes is minimized. The objective function for this classifier is usually written as follows:

$$min\frac{1}{2}||w||_2{}^2 \quad s.t \quad y_i(K(w,x_i)+b) \geq 1 \tag{1}$$

where K(), is called the kernel function. The idea is to find a plane that satisfies the optimization constraints stated above and this results in a quadratic programming problem.

There are several variations of this in the papers studied. Few FPGA implementations of the algorithms consider a simplified version of quadratic programming such as the SVM-SMO(sequential minimization optimization) which is a fast way to train the SVM classifier algorithmic-ally [7] especially when the datasets are huge and also ensures that the algorithm converges when the data presented is not stable. Also the SMO algorithm is gradient descent based.

Other variations include Gilbert's algorithm based computation of the SVM separating plane as provided in [6] which is more of geometric approach. However this appears to adapt well to FPGA implementation and gives better performance than the SVM-SMO algorithm. However SMO algorithm is more widely accepted in terms of robustness to large datasets and accuracy wise. Another variation to be noted is in the Kernel Function K(.,.). This can be linear or non-linear- amongst the non-linear ones Radial Basis Function(RBF) is the most widely used. Few of them evaluate RBF while others consider polynomial, sigmoid and Gaussian kernels non-linearities as well for evaluations

2.2 Clustering algorithms

K-means is one of the most popular clustering algorithms that has been used and is an example of unsupervised algorithm. It tries to obtain a pattern amongst a dataset. The algorithm involves iteratively calculating the k-centers of the clusters to be created. These clusters represent data that are more similar to each other versus data in another cluster. With the initial k-centers points are assigned to clusters based on their distances. A new set of cluster centers are are calculated from the average of the cluster member points and points are reassigned. The algorithm continues till the cluster centers stop changing at which the algorithm is said to have converged. In certain cases the algorithm may not converge based on the dataset characteristics and hence it could be NP-complete.

The papers studied in this project contain few variations of the k-means algorithm. Some use the Manhattan (or L1) distance as metric while the more commonly used is the Euclidean(L2) distance. L1 tends to be less computation because of no squaring and subsequent square root calculation. Most of them focus on optimizing the distance calculation portion of the algorithm which is most computationally intensive. One of the work uses Map-reduce framework type of approach with the CPU and FPGA working independently. Some implementations completely run the algorithm on the FPGA with host interaction being minimal- such as copying input/output data while others do only partial computations on the FPGA.This saves some resources in terms of control unit and it's complexity, dedicating more area to computation.

3. FPGA IMPLEMENTATIONS OF SVM

One of the earliest work that looked at hardware based SVM classifier was by D.Anguita et al[Anguita]. Here they come up with a new approach to the SVM algorithm which is less sensitive to quantification errors and the focus of the paper appears to be more on the accuracy of the algorithm in terms of mis-classification as compared to other approaches.The datasets used are quite small in terms of size as well as dimensions(208 samples of 60 features each). Also speed-up observed for FPGA over CPU is not very clearly stated and the focus is more on miscalculations rate between CPU's and FPGA's, whilst the hardware implementation is just to achieve a faster performance than CPU so as to prove the correctness of new algorithm with several trials. But one of the key ideas from this work is the utilization of variable bit precision advantage of FPGA's which has been carried over and modified as well as improved by several future papers and they reference this work[Anguita]. The implementation was fixed point arithmetic based(to achieve low power) and on a Xilinx Virtex-II platform with frequency ranging between 19-35 MHz.

Another popular hardware implementation that looks at the SVM-SMO algorithm is by Cadambi et al[1] with the FPGA as a co-processor to CPU. The approach presented here relies on taking advantage of low-precision arithmetic. Previous work by the same authors studied the impact of varying precision in the Kernel function evaluation. Here they develop differing precision implementation and for cer-

tain parts of the algorithm- specifically where the computations are dense, low-precision arithmetic is used since it gives better performance. For some benchmarks they show 16 bit precision accuracy is almost equal to that of Floating point, and for couple of other benchmarks they further reduce it to 4-bit thereby gaining in performance. The computations are also divided with the FPGA doing all the kernel-dot products and the CPU doing some of the final kernel evaluations(which requires more precision).
A third highly referenced implementation is by [6]. The authors have separate works that looks at just the SVM training phase implementation on FPGA and another which is just the classifier. The work on classifier [6] looks at generating a heterogeneous architecture that delivers good performance on both homogeneous and heterogeneous datasets. The authors point out that a previous work [1] maps well only to homogeneous datasets. In a homogeneous architecture the implementation only consist of uniform MAC based operations. Here they evaluate the kernel function from a geometric analysis point of view and conclude that the most time consuming part is the projection of all N dataset points onto a line which is the decision surface.However this can be very well parallelized by following the approach of breaking this down to smaller sub problems and parallel units to solve them. The training data is stored on FPGA board's RAM Banks and each sub-problem is mapped to a hypertile that does the projection. The hypertile is the core design of the heterogeneous architecture and is made up of parallel multiplier and adder tree. In the homogeneous implementation MAC is used which is good if precision of all dimensions are the same.

One of the core ideas that differentiates this work from others is the optimization done with respect to attributes. In most real world datasets, some attributes can be binary, continuous or categorical. Hence variable bit precision of FPGA's can be leveraged here. If only MAC were to be used it would result in a waste of resources. Hence a custom precision multiplier is devised and the architecture of heterogeneous hypertile datapath is split between fixed point and floating point. The algorithm first sorts the features by precision and produces the most cost effective adder by computing per node fixed point precision. Based on adder tree, resource usage by hypertile is estimated. DSP allocation per hypertile is estimated based on the datasets feature and target device's(Altera/Xilinx) available DSP/LUT Ratio. Parallel multipliers are initially instantiated using LUT's and then replaced with DSP's for larger precision features. DSP's are mostly reserved for floating point arithmetic

The proposed heterogeneous architecture scales well for homogeneous problems. Altera Stratix 3 is used to demonstrate the implementation but the authors state that this can be targeted to other devices as well. The projection computation would be the core task and influences the frequency of operation. The paper evaluates different kernel functions such as- linear, Gaussian, Polynomial, Sigmoid. Comparison is done between homogeneous and heterogeneous architectures for different datasets. In most cases, the heterogeneous architecture outperforms the homogeneous one by a 2-3 factor. The efficiency of the proposed design increases with the precision diversities of the dataset attributes which is good outcome.

In comparison to [1] SVM-SMO based implementation, where only the dot products are done on FPGA while the host evaluates the kernel, this work has the entire pipe lined kernel processor embedded on the FPGA. But it trades off precision in some parts of the algorithm. -This achieves higher operating frequency

4. K-MEANS CLUSTERING FPGA IMPLEMENTATIONS

A recent most work by [2] looks at a Map-reduce model for this algorithm. The implementation has a set of FPGA's setup as a cluster implementing the Reduce portion of the model while CPU controls the Mapping part. A special software framework is also designed to control and handle all inter-FPGA communication over a specialized network. The authors point out that their model could scale to increased number of FPGA's for larger datasets. Specifically the initial centroids are stored by the CPU in the mapper FPGA's BRAM and input is data streamed to FPGA's onchip buffer.Each FPGA calculates the distance (L2) to its set of centroid and returns the minimum as (x_i, C_i) pair and then picks the next point.The reduce set of FPGA's picks up these pair's and processes it to compute new set of centroids. The communication between FPGA's is over an ethernet switch and controlled by a Scheduler program.

The test setup included a cluster of 3 Xilinix KC705 FPGA board, with two acting a mappers and 1 reducer. Comparison is with a optimized software implementation of k-means in Hadoop's Mahout library. The authors use a dataset of 2M datapoints and 9 attributes. Only 2 attribute and 4 attributes are selected with clusters of k =3,6,12 sizes being tried. Speedup achieved is between 15.5x to 20.6x over software, and the lower speedup is observed in 4D data. No results are provided for higher dimension since it clearly starts reducing in performance when higher dimension(which is a realistic case) is data provided. There is also the overhead of data copying from memory to FPGA's which increases significantly when data set size increases.

Overall the implementation looks more focused on scaleability to process large amounts of data- without considering how to efficiently parallelize the individual functions of map /reduce. Also the mapper FPGA's would be idle during the reduce FPGA's working. Here some of the latency mitigation that were discussed in class could prove to be effective.

Another work that parallelizes k-means on FPGA is from the bioinformatics domains [4]. The application is in Microarray techniques used by biologists to study gene expression and regulation which produce large datasets from experiments . Multiple datasets will need to be analyzed at the same time and k-means is used frequently to analyze this.The output of the experiment is an image which is processed to a huge 2D matrix that becomes the input to k-means clustering algorithm. Here the others fall back on L1 distance as the metric although L2 is known to produce good results, in order to save the squaring which results in multiply operations.

The authors approach is to first pre-process the real data- they take a real dataset and study it's characteristics- to decide the word width for fixed point computation and precision requirements. This is done using Matlab on the PCU independently. This step appears to be very dataset specific and for new experiment results has to be redone on CPU.

For the example dataset after studying the dynamic ranges, 3 bits were chosen for integer representation of input data and 10 bits for the fractional part in floating point precision - hence a total of 13 bits. The algorithm computation is broken down into blocks. The first block computes 8 distances for each point simultaneously(8 cluster/centroids is hardcoded in design- in future work the authors suggest parameterizing this) in one clock cycle and absolute distances in aother one cycle. Minimum of these absolute distances is obtained by running through a comparator tree which takes 2 clock cycles. Overall pipeline latency is 4 clock cycle with one result of per cycle. The next block assigns the datapoint to one of the 8 clusters. The third block is a sequential divider that computes new centroids by averaging over the points. This block has latency of 60 cycles.

The first implementation on a Xlinix FPGA (Xilinx XC4VLX25-10SF363) shows that the design occupied 20% of the floor area. Hence the authors decide to replicate the design so as to make it 5 cores. Hence from from 10.3x speedup they reach 51.7x speedup on the same FPGA device. The same single-core implementation on another Xilinx ML403 (XC4VFX12 FPGA) board operating at 100 Mhz shows 8 x speedup over CPU and consumes only 10W while the CPU uses up 90W in terms of power. No reason is given on why two implementation on slightly differing hardware is done since the first one operates at 126 Mhz while other details remain same. Also to be noted is that no optimization is mentioned for memory bandwidth bottleneck except for the initial processing that decide word-width. Some of the other works that I briefly review here without delving into too much specifics are starting from early 2000's. [3] is referred by several later works and is on the k- means implementation that splits the workload between CPU and FPGA's. The distance calculations are all done on the FPGA's and in fixed point-arithmetic and all distances are calculated using Manhattan(L1) metric. This and smaller word-width of input vectors are key optimizations that result in a 50x speedup over CPU's. The implementation is on Xilinx Virtex FPGA operating at 50 Mhz with CPU being a Pentium III at 500 Mhz.

Another work by [9] primariliy looked at k-means implementation on FPGA for image segmentation based application where the input data is hyper-spectral image pixels. This data is however homogenous since the range of the attributes id fixed and known. The approach followed to start with different variations of the algorithm- for example just doing the outer loop processing on FPGA and gradually progress to offloading more computationally intensive parts of the algorithm to the FPGA co-processor. The implementation is on a Altera Excalibur processor.The implementation eventually finalizes to doing the distance computation on FPGA by using 32 Processing elements in parallel that gives the best result. The speed up achieved here is close to 11x with operating frequency of 33Mhz over a Pentium III processor which is at 1 GHz

5. GENERIC FPGA DESIGNS FOR MACHINE LEARNING

In this section I look at two proposed works from 2016 that look at providing reconfigurable hardware accelerators for wide variety of machine learning algorithms. First is the TABLA framework [5] that aims at providing a generalized acceleration for class of machine learning algorithms on FPGA's. It is mainly based on solving the stochastic optimization problem that is an integral part of several ML algorithms mainly in the form of gradient descent.Using the custom high level language designed the user needs to only specify the gradient objective function that the learning task constitutes of. It also looks at making programming FPGA's easier by providing programming abstraction for the actual hardware circuit generation. Implementation are shown for a wide range of algorithms- such as Logistic regression, SVM classifiers, Recommendation systems, back propagation (in neural networks) and linear regression which overall makes a good comparison. Evaluvation is done on Xilinx Zynq ZC702 FPGA platform and for various benchmark datasets from UCI repository. A notable fact is that number of lines of code for these implementations is very minimal therby reducing the complexity for the programmer. This is mainly due to several abstractions especially for the gradient iterator.

The paper also does extensive performance evaluvations for each class of algorithm over the CPU as well as a range of GPU's - from low power Tegra K1, GTX 650 and the high performance Tesla K40 GPU. This is one of the only paper where I was able to find such comprehensive comparison for GPU's. Also comparison is provided for an ARM CPU and Intel Xeon 2. In short, the FPGA implementation achieves 4x-115x range of speedup over ARM and an average of 2.9x over Xeon. A GPU-CPU comparison is also provided. For smaller size becnhmarks the FPGA implementations outperform the GPU's by about 18% but with increased dataset sizes the GPU gets better due to more workload avaialble for parallelization. However the FPGA is still the best performer in terms of power- an average of 33.4x higher power-per-watt performance over the Tesla K40.

A second work on similar lines is from the Prabhakar et al [8] that also looks at automatic generation of reconfigurable hardware by providing a framework and set of optimized parallel patterns library constructs. The work is mostly focused on dataflow amenable functional languages. The work mostly leverage on another compiler framework Delite by the same authors and looks at generating optimized hardware templates based on pattern matching in the code. It largely relies on compiler optimizations such as strip mining, loop interchange , code motion to first generate highly optimized code for an algorithm written in functional language such as Scala. The authors use k-means as an running example and show how the code is generated from Parallel pattern language. Apart from the tiling optimizations they do metapipelining which is basically similar to software pipelining- wherein the number of stages in the pipeline is dictated by the algorithm. Some of the benchmark implemented, perform very well with metapipelining because they are more amenable and generate balanced pipeline stages, while few algorithms do not show much improvement.

The setup utilized is a Altera 28nm Stratix V FPGA at a clock frequency of 150MHz T and 48GB of dedicated off-chip DDR3 DRAM with a peak bandwidth of 76.8GB/s. Memory bound benchamarks which are mainly consisted of Dot-product type of calculation do not achieve much speedup as the parallelization factor is increased. So metapipleining approach does not cause much benefit here. Benchmarks with good spatial and temporal locality do not show any improvement because are already highly optimal design. The highest improvement are observed on blackscholes and k-means that have streaming data with several floating point calculations required. On average the speed up observed is in the range of 4.5x - 39.4x over CPU.

6. CONCLUSIONS

A key observation across all the papers reviewed is that most of them use hand tuned and application specific designs. Be it [4] which is very highly tuned to bioinformatics data with no scope for how to handle higher dimensional data or different number of clusters or the SVM's that work well for homogeneous datasets. There is no clear path provided on how to generalize this to any other domains and the goal appears to be mostly acceleration with high area utilization, instead of generic techniques to achieve effective parallelism on FPGA's. Also the implementations are all done in Hardware description language and rarely consider adapting from existing software implementations that may have been written in High level languages and highly parallelized. Also most of the works involve some trade off- either in terms of precision in the input data representation or the metric calculations (such as L1 over L2 distance).

Few of the more recent works like [8] and Tabla[5] are different in their approach where they aim to provide a larger framework to generate FPGA based accelerators across a range of machine learning algorithms. These are not fine tuned to a particular algorithm or a particular benchmark. They rely on pattern matching and compiler optimization techniques that have been widely validated over time. hence produce somewhat lesser speedup. But considering how often machine learning algorithms are used and the wide variety of tasks that these are applied to, we may prefer to use different classifiers for different datasets. In that case the generalized framework is more suitable thereby shaping up as machine learning processors

7. ACKNOWLEDGMENTS

I would like to thank my Professor, Dr. Walid Najjar for introducing me to the area of Reconfigurable Computing with FPGA's. The concepts explained during the teaching of the course helped me write this project report in a better way that I would have thought. The key ideas provided by you greatly helped in studying the literature in this area and better understand them.

8. REFERENCES

[1] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf. A massively parallel fpga-based coprocessor for support vector machines. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 115–122, April 2009.

[2] Y.-M. Choi and H. K.-H. So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 9 – 16, June 2014.

[3] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski. Algorithmic transformations in the implementation of k- means clustering on reconfigurable hardware. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, pages 103–110, New York, NY, USA, 2001. ACM.

[4] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 248–255, June 2011.

[5] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 14–26, March 2016.

[6] M. Papadonikolakis and C. S. Bouganis. A heterogeneous fpga architecture for support vector machine training. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 211–214, May 2010.

[7] J. C. Platt. Advances in kernel methods. chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[8] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun. Generating configurable hardware from parallel patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 651–665, New York, NY, USA, 2016. ACM.

[9] M. Gokhale, J. Frigo, K. McCabe, J. Theiler, C. Wolinski, and D. Lavenier, âĂIJExperience with a hybrid processor: K-means clustering,âĂİ J. Supercomputing, vol. 26, no. 2, pp. 131-148, 2003.