JMLR: Workshop and Conference Proceedings 53:1–16, 2016 BIGMINE 2016

FPGASVM: A Framework for Accelerating Kernelized Support Vector Machine Training using FPGAs

Mudhar Bin Rabieah MOB10@IMPERIAL.AC.UK
Christos-Savvas Bouganis CCB98@IMPERIAL.AC.UK
Department of Electrical and Electronic Engineering
Imperial College London, UK

Editors: Wei Fan, Albert Bifet, Jesse Read, Qiang Yang and Philip Yu

Abstract

Support Vector Machines (SVM) are powerful supervised learnings method in machine learning. However, their applicability to large problems, where frequent retraining of the system is required, has been limited due to the time consuming training stage whose computational cost scales quadratically with the number of examples. In this work, a complete FPGA-based system for kernelized SVM training using ensemble learning is presented. The proposed framework builds on the FPGA architecture and utilises a cascaded multi-precision training flow, exploits the heterogeneity within the training problem by tuning the number representation used, and supports ensemble training tuned to each internal memory structure so to address very large datasets. Its performance evaluation shows that the proposed system achieves more than an order of magnitude better results compared to state-of-the-art CPU and GPU-based implementations, providing a stepping stone for researchers and practitioners to tackle large-scale SVM problems that require frequent retraining.

Keywords: SVM Training; FPGA; Large-scale training

1. Introduction

Support Vector Machines (SVMs) are powerful supervised learning methods in machine learning Vapnik (1998), and they have been deployed to tackle diverse problems. For example, they have been used successfully in many applications, such as: object recognition, e-mail filtering, DNA sequencing and speech recognition Wang (2005).

Training on large datasets is a very challenging and time consuming process as the computational complexity for many solvers is $O(n^2)$ where n is the number of training points Bottou and Lin (2007). This hinders the applicability of such algorithms in situations where the characteristics of the data change over time. As a result, the need for constant retraining is desirable. Evidence of this can be seen in the google flu trend (GFT) where the need for model retraining was emphasized Lazer et al. (2014).

To tackle the problem of high computational costs of SVM training, several algorithmic techniques have been proposed. For example, decomposition techniques transform the training problem into a sequence of sub-problems known as working sets Bottou and Lin (2007). At each iteration only the coefficients of the working set data are updated. As a result, the memory requirements are reduced. Moreover, Ensemble training can be used to transform the overall training dataset into smaller datasets that can be trained independently and in parallel.

Hardware improvements can contribute towards training acceleration. For example, GPUs are used to accelerate computationally intensive parts such as the dot product operations Athanasopoulos et al. (2011). Reconfigurable structures (e.g. FPGAs) are also viable options for speeding up SVM training. Their heterogeneous hardware resources allow for the exploitation of the different precision requirements of the training problem which will increase the parallelization factor Graf et al. (2008); Cadambi et al. (2009); Papadonikolakis and Bouganis (2008).

In this work, a complete FPGA-based system for accelerating nonlinear SVM training is presented. Ensemble learning is proposed to address large datasets and take advantage of available FPGA on-chip memory blocks. This approach allows each training subproblem to fully realize the parallelization potential. In addition to ensemble learning, cascaded multi-precision training flow is proposed by exploiting FPGA reconfigurability. This flow exploits the higher parallelization factor at the lower precision phase, which produces a reduced dataset for the higher precision phase to generate a more accurate model.

2. Background

2.1. Support Vector Machine

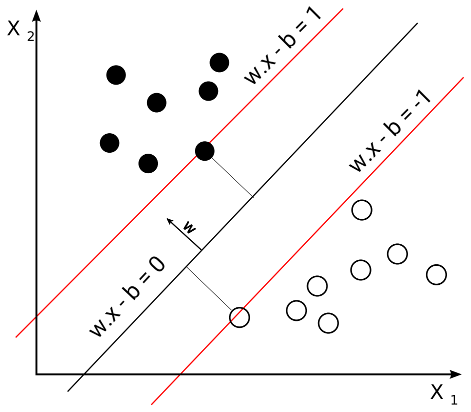


Figure 1: Support Vector Machines

Support Vector Machines (SVMs) are supervised learning methods that construct a hyperplane to classify data between two classes {-1,+1} as shown in fig. 1. Given a set of examples $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, where $y_i \in \{-1, +1\}$ and $x_i \in R^d$. For the separable case, the hyperplane w is found by solving the following optimization problem:

$$min \ \frac{1}{2}\|w\|^2, \ \textbf{s.t.} \ y_i(wx_i - b) \geq 1 \tag{1}$$

The classification function is:

$$y = sign(wx - b) \tag{2}$$

For the non-separable case, slack variables ξ are introduced to account for misclassified data. Now, the separating hyperplane w is found by solving the following:

$$min \ \frac{1}{2}\|w\|^2 + C\sum_i \xi_i, \ \textbf{s.t.} \ y_i(wx_i - b) \geq 1 - \xi_i, \ \xi_i \geq 0 \tag{3}$$

where C is a constant to control the trade-off between the error and the simplicity of the model w. The above optimization problem can be reformulated in the dual form as:

$$min \sum_i \alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j \tag{4}$$
$$\textbf{s.t. } w = \sum_i \alpha_i y_i x_i,\ \sum_i \alpha_i y_i = 0,\ 0 \le \alpha_i \le C$$

where α_i is the Lagrange multiplier. The dual formulation of SVM expresses the separating hyperplane w in terms of the training points. Also, kernel functions can be used instead of dot product operations, which will extend SVM to nonlinear cases.

Solving SVM training problem can be achieved through various methods Bottou and Lin (2007). A commonly adopted method is decomposition method. An example of this class is the Sequential Minimum Optimization (SMO) is a widely used decomposition algorithm. Here, the working set consists of only two points at each iteration. Other ways of solving the problem is by realizing the geometric interpretation of SVMs, which is also the basis of this work due to its simple control structure and overhead.

2.2. Gilbert Algorithm

In Keerthi et al. (2000), it was shown that the the separable case of SVM is equivalent to solving the nearest point problem between two convex hulls. As for the non-separable case, it was suggested in Friess (1998) to modify the objective function to include the sum of the squared errors. Thus, the objective function becomes:

$$min \frac{1}{2}\|w\|^2 + \frac{c}{2}\sum_i \xi_i^2 \tag{5}$$
$$s.t.\ y_i(wx_i - b) \ge 1 - \xi_i,$$

In Friess (1998), it was shown that the above formulation is equivalent to the separable case with the exception of the kernel definition. The kernel now is defined as: $\hat{k}(x_i, x_j) = k(x_i, x_j) + \delta_{ij}/C$, where $\delta_{ij} = 1$ when $i = j$, otherwise it is 0. This is equivalent to adding a constant to the diagonal of the kernel matrix which yields a valid kernel Shawe-Taylor and Cristianini (2004).

In Martin (2005), Gilbert's algorithm Gilbert (1966) was suggested to solve the nearest point problem. Gilbert's algorithm tries to find the point s^* on a given secant hull S which is the nearest to the origin. To describe the steps of Gilbert's algorithm, we need to define the following:

Definition 1 *Whenever there is a dot product operation, it can be replaced by a kernel function :*

$$\langle x, y\rangle \Leftrightarrow k(x, y)$$

Definition 2 *The nearest point on the line segment connecting points a and b:*

$$[a, b]^* = (1-\lambda)a + \lambda b, \tag{6}$$

$$\lambda = \begin{cases} \frac{-\langle a, b-a\rangle}{\|b-a\|^2} & \textit{if } 0 < -\langle a, b-a\rangle < \|b-a\|^2; \\ 0 & \textit{if } -\langle a, b-a\rangle \le 0 \\ 1 & \textit{if } \|b-a\|^2 \le -\langle a, b-a\rangle \end{cases} \tag{7}$$

Definition 3 *Support function:*

$$g_S(x) = max\langle x, s_m\rangle \ where \ s_m \in S \tag{8}$$

Definition 4 *Contact function:*

$$g_S^*(x) = s_{m0} \ where \ \langle x, s_{m0}\rangle = g_S(x) \tag{9}$$

Now, gilbert algorithm proceeds as follows:

```
Gilbert's algorithm
1. choose random point $w_0 \in S$
2. $k = 0$
3. **Repeat**
4. $k = k + 1$
5. find $g_S^*(-w_{k-1})$ as described in equation 9
6. $w_k = [w_{k-1}, g_S^*(w_{k-1})]^*$.
7. **Until** $\|w_k - w_{k-1}\| = 0$
```

Gilbert proved that as $k \to \infty$, $\|w_k - s^*\| \to 0$. In Martin (2005), it was observed that w_k converges to s^* in angle faster than it does in norm. So, it was suggested that the angle should be used as a stopping criteria: $\alpha = \frac{\langle w_k, w_{k-1}\rangle}{\|w_k\|\|w_{k-1}\|}$. Fig. 2 shows a visualisation of an iteration of Gilbert's algorithm.

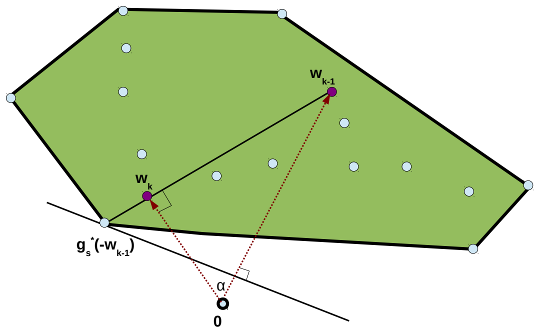


Figure 2: Visualisation of an iteration of Gilbert's Algorithm

In order to make Gilbert's algorithm applicable to SVM, the secant hull S is defined in terms of the two classes X and Y:

$$S = X - Y;$$

Now, $g_S^*(-w_{k-1})$ can be decomposed as:

$$g_S^*(-w_{k-1}) = g_X^*(-w_{k-1}) - g_Y^*(w_{k-1}) \tag{10}$$

2.3. Related Work

Most SVM implementations are targeting CPU platforms. This can be attributed to the ease of developing software on such platforms. This has resulted in a plethora of machine learning packages available freely to the community. Many of these packages target a very broad range of algorithms. Examples of these packages include LIBSVM Chang and Lin (2011) and SVM^{light} Joachims (1999). In these two packages, the original problem is decomposed into smaller chunks. This allows the algorithm to only store the kernel matrix for a manageable number of points at a time. In SVM^{light}, the algorithm is enhanced by allowing the dataset to shrink. This is done by realizing that SVM produces sparse solutions. So, SVM^{light} will shrink the dataset to the points which it suspects that they are support vectors.

Several implementations target graphical processing units (GPUs) to accelerate the training process. In Athanasopoulos et al. (2011), the whole kernel matrix is calculated on the GPU and passed on to the CPU. In Catanzaro et al. (2008), the MapReduce framework was applied to the GPU to implement the algorithm applied in LIBSVM. The advantage of using the MapReduce framework on a GPU instead of a cluster of computers is local synchronization between GPU processors. Another example is GTSVM Cotter et al. (2011). In this implementation, special attention was placed for sparse datasets (datasets with many zero features). Here, vectors with similar sparsity patterns, are grouped sequentially. This allows for a coalesced memory access.

Recently, FPGAs were also targeted as a means of acceleration. In Graf et al. (2008); Cadambi et al. (2009), kernel operations were done as fixed point operations. Both implementations demonstrated that for many datasets, the solution's accuracy was not affected. Both implementations are used as a co-processor. The FPGA handles kernel evaluations, whereas the rest of the learning algorithm is run on the CPU.

In addition to using fixed point operations, the kernel function could be sped up by approximation. This approach was applied in Nagarajan et al. (2011). Here, the exponential operation in the Gaussian kernel was replaced by the second-order Taylor series expansion. In Papadonikolakis and Bouganis (2008), the kernel operation is divided between fixed point and floating point operations. The fixed point domain handles the dot product between the attributes of the data point. This is fed to the kernel processor to complete the evaluation of the function in the floating point domain.

In the previous implementations, the fixed structure of CPUs as well as GPUs hinder their ability to exploit any heterogeneity within the training problem. As for the FPGA implementations mentioned above, some implementations as in Graf et al. (2008); Cadambi et al. (2009), only implement part of the training algorithm on the FPGA which reduces the parallelization potential. In Papadonikolakis and Bouganis (2008), only the datapath was mapped to the FPGA and no provision was made in the case when the dataset does not fit the block rams available. Also, all the previously mentioned FPGA implementations do not exploit the reconfigurability of such structures. These are the issues that our proposed framework addresses. It allows for custom precision per attribute. Also, for homogeneous datasets, a multi-stage multi-precision flow is proposed. Finally, ensemble learning is deployed to allow each training instance to fully fit the block rams.

3. Framework Overview

The proposed framework consists of both software and hardware modules. Fig. 3 shows an overall view of the framework architecture. Hardware module (FPGA) is responsible for running the gilbert algorithm to solve an SVM training instance. Software modules, which run on the CPU, are responsible for preprocessing the data (SVM Data), generating a customized hardware (SVM Configuration, Synth Tool), communication and setting up the training process (SVM Train) and running the classification process (SVM Classify).

Customized hardware is generated by providing high level description of the training problem. This description includes kernel function used (RBF, polynomial ...) and

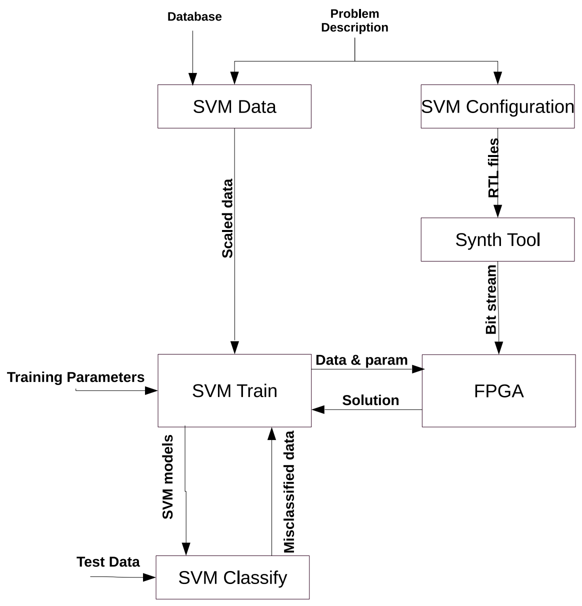


Figure 3: Framework Overview

number and types of dataset attributes. Various types are supported by the proposed framework including real-valued, categorical, boolean and integer. This allows for heterogeneous datasets where several data types coexist. Such high level description masks the underlying hardware from the user which opens up the framework to non hardware designers and provides a consistent front end regardless of the FPGA platform used.

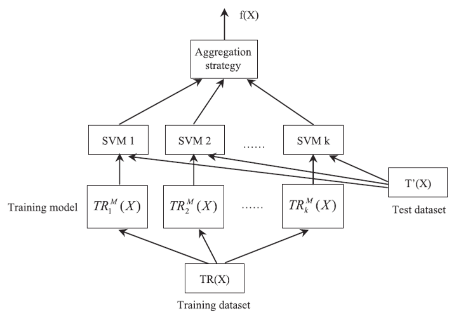
Training process is managed through "SVM Train" which is a multi-threaded program. One thread runs interactively giving the user the ability to define the training parameters at runtime. These parameters include: kernel parameters (e.g gamma for RBF kernels), stopping criteria and regularization parameter C, which is passed to the FPGA along with the training data. Another thread is responsible for receiving the solution from the FPGA. This multi-threaded approach is used so that the module is not blocked waiting for the FPGA to finish. This allows the module to prepare the next batch of data as soon as it sends the current one which is used in ensemble learning. "SVM Classify" module is coupled with "SVM Train" to allow tuning the parameters through performing validation tests.

3.1. SVM Ensemble

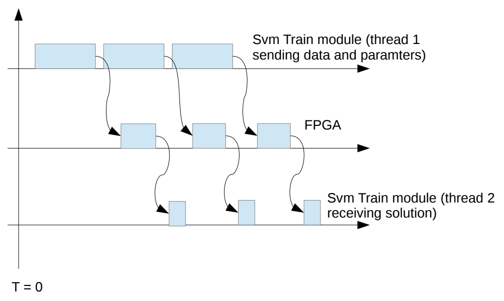
FPGAs provide low latency high throughput on-chip memory blocks. These blocks could be instantiated multiple times to accommodate multiple processing units, each with its own memory. However, for many training problems, the memory requirements exceed the on-chip memory available within the FPGA. A straightforward fix is to use external SDRAMs to store the data. However, the associated control logic as well as the access scheme limits the number of possible processing units instantiated which reduces the parallelization factor.

In the proposed framework, the solution is provided by creating an SVM ensemble of k problems where each problem can fit into available on chip memory. The main dataset is divided into k groups each of the size N/k, where N is the total number of points in the original dataset. This allows each training subproblem to fully realize the parallelization potential. Fig. 4(a) shows an abstraction of SVM ensemble. Each group is fed to the FPGA and a model for each group is created independently. These models are aggregated (majority voting, weighted sum ...) at the classification phase. The user has the freedom to invoke the ensemble mode even if the data fits the FPGA. This will give the user the ability to create several models and produce a more accurate model Kim et al. (2002).

Fig. 4(b) shows timing diagram showcasing the ensemble training flow. Here, "SVM Train" module will create two threads running on the CPU: one for sending the parameters and data while the other is for probing the FPGA for the solution. Thread 1 will create the data groups and send it to the FPGA one by one. In case the FPGA is still not ready to receive the data, these groups will wait in a queue until the FPGA signals that it is ready to receive the next batch. Once the FPGA is done from training a single batch, it will signal thread 2 that a solution is ready so it can read it and store it.



(a)


Svm Train module (thread 1 sending data and paramters)
FPGA
Svm Train module (thread 2 receiving solution)
T = 0

(b)

Figure 4: On the left: (a) Svm ensemble abstraction. (b) Ensemble training flow

3.2. Cascaded Multi-precision Training

By exploiting the reconfigurability properties of FPGAs, multiple hardware instances with different precisions are generated for the same problem. In this scenario, a cascaded training flow is created as shown in fig. 5. At the lower precision phase, the training runs faster since more processing units can be instantiated. The support vectors generated together with misclassified data from this phase is then passed to a higher precision phase. At the higher precision phase, less processing units is instantiated. This is mitigated by reducing the number of data points at this phase since not all the original training data is passed along. This scheme is guaranteed to converge to a feasible solution since the solution produced at the lower precision phase satisfies the condition $\sum_i \alpha_i y_i = 0$ which is a feasible solution.

This scheme has the potential of speeding up the training process if the reduction of data points is significant. It also has the potential of reducing the final number of support vectors as shown in the evaluation results.

4. Hardware Implementation

4.1. Top Level Architecture

In the proposed framework, the hardware module is responsible for executing Gilbert training algorithm. The top level compromises of several processing elements (PE), each with its

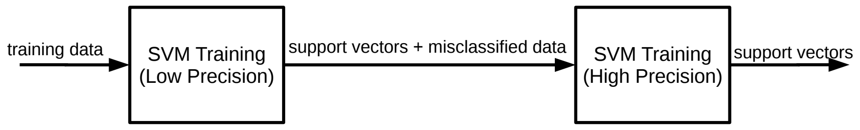

training data
SVM Training (Low Precision)
support vectors + misclassified data
SVM Training (High Precision)
support vectors

Figure 5: Low precision to high precision two stage training flow

own on-chip memory blocks. It, also, consists of "lambda calculation" and "angle monitor" blocks to manage the next iteration and monitor the stopping criteria. All this is managed through the main control unit.

Training SVMs using Gilbert algorithm constitutes three main tasks:

- **projection task:** This task involves finding $g^*_X(-w_{k-1})$ and $g^*_Y(w_{k-1})$ which are basically equivalent to finding $min\langle w_{k-1}, x_i\rangle$ and $max\langle w_{k-1}, x_j\rangle$, where $x_i \in X$ (class +1) and $x_j \in Y$ (class -1). The complexity of this task is $O(n)$ where n is the number of datapoints.
- **calculate lambda**: Here, lambda is calculated according to equation 7. The complexity of this task is $O(1)$.
- **check stopping criteria**: Here, the angle between w_k and w_{k-1} is calculated to check whether the algorithm has reached the stopping criteria. The complexity of this task is $O(1)$.

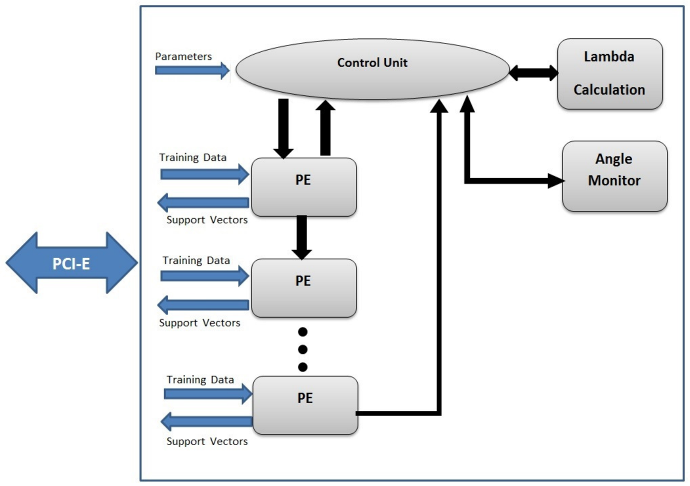


Figure 6: Top Level SVM Architecture

In all of the above operations, kernel operations are present; and in the case of projection task, kernel operations are performed between one single point and all the training data (hence the complexity of $O(n)$). At the top level, as shown in fig. 6, kernel operations are done within processing elements (PE). The instantiation of multiple processing elements, allows the system to distribute the data points between these elements. At each iteration each processing element will compute its local maximum and minimum and pass them to the next processing element. The last processing element will output the global maximum and minimum.

The processing elements are connected in a stacked manner to reduce the interconnection complexity as the control unit just needs to pass the parameter to the first processing elements and it will propagate from to next and so on. The drawback of such scheme, is that it will introduce a delay of N_{PE} cycles, where N_{PE} is the number of processing elements instantiated.

The (lambda calculation) block is responsible for calculating λ according to equation 7. Again, any dot product operation is done utilizing the processing elements, which leaves the (lambda calculation) block with just some simple floating point scalar operations. The (angle monitor) block is responsible for calculating $\frac{\langle w_k, w_{k-1}\rangle}{\|w_k\|\|w_{k-1}\|}$ which is used as a stopping criteria. Again, similar to (lambda calculation block), any dot product operations are performed using the processing elements.

The training time can be modelled as follows:

$$time = iter\left(\frac{N}{\#PE} + \#PE + t_{\lambda+angle}\right) * clk_{period} + t_{comm}, \tag{11}$$

where $iter$ is the number of times the loop body of the algorithm is executed, N is number of training points, $\#PE$ is the number of processing elements instantiated, $t_{\lambda+angle}$ is the time needed to evaluate λ and check whether the angle convergence has been met, clk_{period} is for clock period, and t_{comm} is the time needed to transfer the data to/from the board.

4.2. Processing Element

Fig. 7 shows the architecture of the Processing Element. As mentioned earlier, processing elements performs kernel operations required for the projection task as well as for calculating the norms. The kernel computation is divided between fixed and floating point domains in a similar fashion to what was proposed in Papadonikolakis and Bouganis (2010). The dot product operations that appear in the kernel function are performed in fixed point. The rest of the kernel operations are done in floating point. For example, the RBF kernel can be expressed as: $k(x_j, x_i) = e^{-\gamma\|x_j - x_i\|^2} = e^{-\gamma(x_j.x_j - 2x_j.x_i + x_i.x_i)}$. To achieve maximum throughput, each attribute within the dataset is fed directly to a dedicated multiplier. Also, each attribute can have its own precision. This will allow for a heterogeneous dataset. To optimize the hardware utilization of the dot-product tree, the multipliers are ordered with respect to their precisions in ascending order. Unlike Papadonikolakis and Bouganis (2010), each processing element updates the solution (SVs cache) corresponding to the training points it processes. This makes the solution update runs in parallel with the projection task.

Also, caching is used to speed up the overall algorithm (Kernel Cache). At each iteration, $k(w_k, x)$ is calculated to perform the projection task. This is a very expensive computation since from equation (4): $w_k = \sum_i \alpha_i y_i x_i \Rightarrow k(w_k, x) = \sum_i \alpha_i y_i k(x_i, x)$. As a result, as the algorithm progresses, $k(w_k, x)$ becomes more challenging due to the increased number of support vectors. This can be solved by observing the recursive nature in which w_k is computed:

$$\begin{aligned} w_k &= [w_{k-1}, g_S^*(w_{k-1})]^* \\ &= (1-\lambda)w_{k-1} + \lambda g_S^*(w_{k-1}) \\ \Rightarrow k(w_k, x) &= (1-\lambda)k(w_{k-1}, x) + \lambda k(g_S^*(w_{k-1}), x) \\ &= (1-\lambda)cache + \lambda k(g_S^*(w_{k-1}), x) \end{aligned}$$

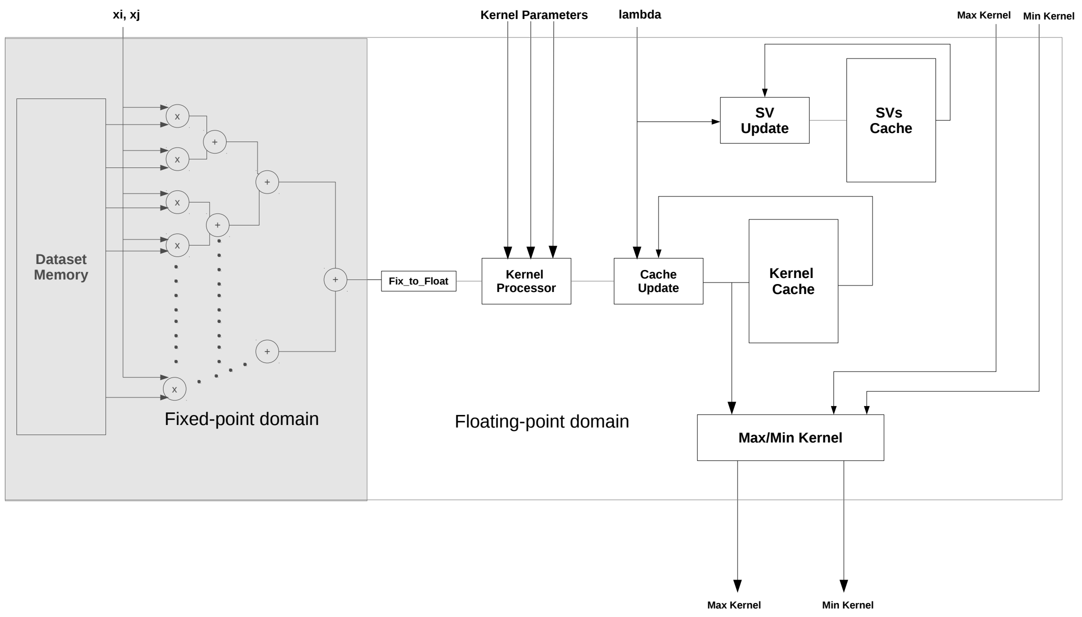


Figure 7: Processing Element Architecture

Now, only two kernel operations are required since

$$k(g_S^*(w_{k-1}), x) = k(g_X^*(-w_{k-1}), x) - k(g_Y^*(w_{k-1}), x)$$

In addition to performing the kernel operations, each PE contains comparators to perform the operations $min\langle w_{k-1}, x_i\rangle$ and $max\langle w_{k-1}, x_j\rangle$. Each PE compares its local minimum and maximum with the previous PE in the stack. The output is passed to the next PE. The last PE produces the global maximum and minimum.

5. Evaluation Results

The FPGA system is implemented on Xilinx board ML605. The maximum operating frequency of the core of the system is 150Mhz. However, the overall system is clocked down to 62.5Mhz to match the reference clock of the PCI port instantiated. The communication between the PC and FPGA board is carried through PCI port. RIFFA Jacobsen et al. (2012) framework is used to facilitate the communication between our IP core and PCI endpoint on the board.

5.1. Training and Accuracy

Three datasets were tested, namely: adult, forest covertype Bache and Lichman (2013)and MNIST data sets Lecun and Cortes. The tests were performed on our system as well as SVM^{light} Joachims (1999), GPUSVM Catanzaro et al. (2008) and GTSVM Cotter et al. (2011). SVM^{light} was run on an Intel Core i7-3770 machine with 16GB RAM on board. Both GPU implementations were run on Nvidia Quadro K4000 GPU.

The adult dataset contains 32K training points with 14 heterogeneous features; whereas, the forest covertype contains 522K training points with 54 heterogeneous features. Forest covertype was transformed into a binary classification problem by training class 2 versus all the others. The MNIST dataset contains 60K training points with 784 homogeneous features. It was converted into a binary problem by training the data for class odd versus even. For all the datasets, the kernel used was the Gaussian kernel.

The adult dataset fits completely in the FPGA; whereas covertype and MNIST does not fit which require the use of SVM ensemble mode. From the forest dataset, an ensemble of 13 SVMs was created; whereas, for the MNIST dataset an ensemble of 27 SVMs was created for the full precision case (8-bits per attribute). The aggregation scheme for both is majority voting.

Table 1, shows a summary of the training time results compared to the other implementations (data preparation and setup times are not included for all implementations). Test error is computed as the ratio of misclassified data to the overall number of testing points. Power consumption of the FPGA implementation was estimated using XPower from Xilinx with a switching probability of 0.5. The power consumption of the other CPU as well as GPU implementations is reported from the data sheets of Intel Core i7-3770 and Nvidia Quadro K4000 GPU. The deviation in the number of support vectors can be attributed to the different algorithms implemented for each implementation. Reducing the number of support vectors has the benefit of speeding up the classification process (classification function scales linearly with the number of support vectors). The FPGA implementation shows significant speed ups compared to the other implementations. For the adult dataset where it can fit the FPGA, the speed-up ranges from 8x to 160x. For forest covertype, the speed ups ranges from 80x to 2880x. As for MNIST dataset, the speed ups ranges from 11.6x to 343.8x.

For the MNIST dataset, several tests were performed. The first was for the full precision case (8 bits per attribute). The second was for a low precision case (4 bits per attribute). The Third was for two stage training scheme (Low precision = 4bits, High precision = 8bits). It is obvious that the low precision case achieves the best training time. However, it suffers slightly in accuracy. The cascaded scheme achieves similar results to the full precision case in terms of accuracy. However, it takes slightly more time in training. This is due to the fact that the transition from the low precision to high precision phase did not result in a significant reduction in the number of points (support vectors and misclassified data were 49444 points). However, the final tally of the number support vectors was reduced. For the full precision case, the power consumption estimation is less due to the fact that only one processing element was instantiated and it is not possible to fully utilize the FPGA resources.

5.2. Timing Analysis

If setup time is included, the the overall end to end training time increases. This setup time includes the processing of data into fixed point values, configuring the training parameters (kernel parameters, regularization parameter ...), preparing data buffers and configuring communication channels between PC and the FPGA. However, the cost of setup time is only considerable at the start of the training process . Fig. 8 shows the data preparation time with respect to communication overhead between the FPGA and the PC (communication between SVM train module and FPGA) and the FPGA training time. It is clear that for small datasets (e.g Adult) the setup time becomes more considerable. As for the communication overhead, it is negligible except for the case of MNIST. This can be attributed to the fact that the dataset has many attributes which greatly reduce the number of points in each ensemble.

Fig. 9 shows the speed ups with respect to the number of processing elements instantiated under the assumption that the underlying FPGA resources are large enough

Table 1: Summary of results

Data Set	Implementation	Test Error(%)	Training Time	speed up	SVs	Power
Adult ($C = 1, \gamma = 0.05$)	SVM^{light}	14.8	80s	1x	18152	77W
	GPUSVM	17.2	5s	16x	18344	80W
	GTSVM	15	4s	20x	19138	80W
	FPGA	16.8	0.5s	160x	17845	6W
Forest Covertype ($C = 10, \gamma = 0.125$)	SVM^{light}	13.9	43200s	1x	277102	77W
	GPUSVM	13.9	1850s	23.4x	277402	80W
	GTSVM	29.9	1200s	36x	278564	80W
	FPGA	14	15s	2880x	294490	6W
MNIST ($C = 10, \gamma = 0.125$)	SVM^{light}	4.6	2062.75s	1x	43733	77W
	GPUSVM	5	425.7s	4.8x	43731	80W
	GTSVM	4.7	70s	29.5x	43584	80W
	FPGA (8 bits)	4.8	6s	343.8x	46671	5.7W
	FPGA (4 bits)	5	3s	687.6x	47812	6W
	FPGA (4 + 8 bits)	4.8	8s	257.8x	43882	5.85W

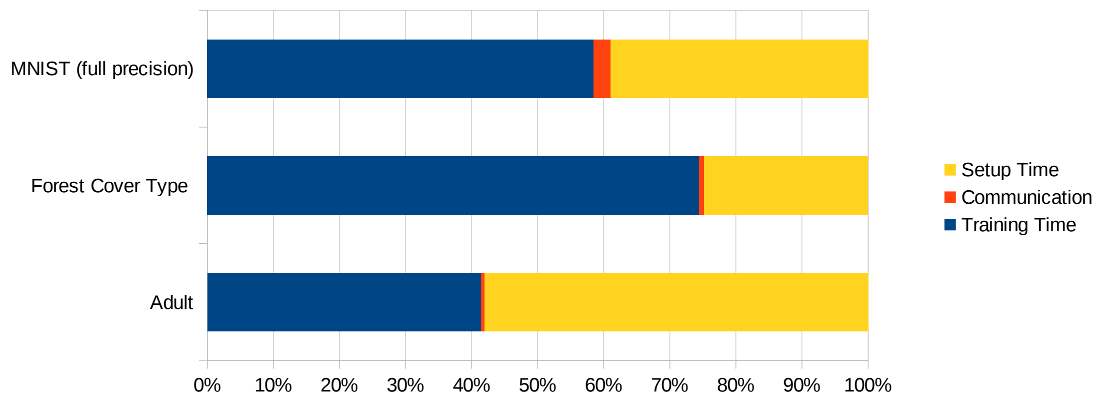


Figure 8: Setup and communication times overhead

(Equation 11 was used to estimate the training time when it is not possible to instantiate the number of processing elements required). Increasing the number of processing elements will reduce the time to complete the projection task within Gilbert algorithm. However, all other operations are not affected. As a result, speed ups saturates (or roll-off) when the number of processing elements is large compared to the number of training points. In such cases each processing element handles only a few data points, which leads other operations (lambda and angle calculations) to dominate the training time. Also, the stacked manner in which the processing elements are connected adds a delay in these situations.

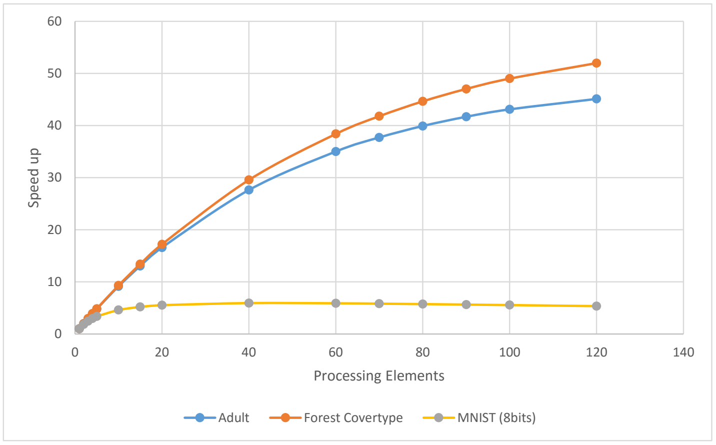


Figure 9: Speed ups with respect to number of processing elements instantiated

5.3. Hardware Utilization

Table 2 shows the FPGA resource utilization with respect to each dataset. For the MNIST dataset (8 bits per attribute), only a single processing element was instantiated due to the fact that the dataset has many attributes (a single processing element will cause the training module to occupy 46% of the slices). For the adult and forest covertype, more processing elements were instantiated because of the low number of attributes within the datasets.
For the MNIST dataset, the precision of the data was changed to study the effects on hardware utilization for a single processing element. For 6 and 4 bits cases, the percentage of occupied slices allows for the instantiation of a second processing element. Also, since the BRAM utilization is lower, more data points can be loaded. Furthermore, for both cases, the training accuracy was almost the same (test error 4.8% for 6-bits and 5% for 4-bits).

Fig. 10 shows the number of processing elements instantiated with respect to the number of attributes (8-bits per attribute). It is clear that dimensionality redections trechniques can be of great benefit in maximizing the number of processing elements instantiated.

Table 2: Hardware Utilization

Data Set	Processing Elements Instantiated	FPGA Slices occupied	FPGA Slice Registers	FPGA Slice LUTS	BRAMS
Adult	20	93%	41%	81%	70%
Forest Covertype	20	96%	36%	76%	87%
MNIST (8 bits per attribute)	1	46%	17%	29%	88%

Table 3: Effect of data precision on hardware utilization

Precision	FPGA Slices occupied	FPGA Slice Registers	FPGA Slice LUTS	BRAMS
8-bits	46%	17%	29%	88%
6-bits	38%	14%	24%	68%
4-bits	32%	11%	19%	46%

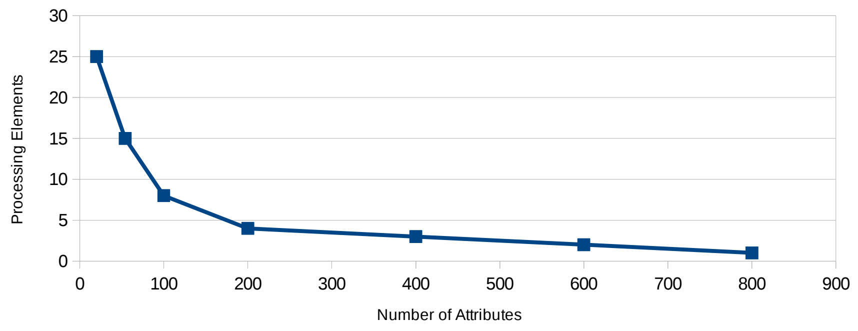


Figure 10: Number of processing elements instantiated with respect to number of attributes (8-bits per attribute)

6. Conclusion

In this paper, a complete FPGA-based system for accelerating nonlinear SVM training has been presented. The proposed framework utilises a cascaded multi-precision training flow, exploits the heterogeneity within the training problem, and supports ensemble learning. Performance evaluations shows that the proposed system outperforms other implementations across different datasets while still maintaining comparable accuracy.

References

Andreas Athanasopoulos, Anastasios Dimou, Vasileios Mezaris, and Ioannis Kompatsiaris. Gpu acceleration for support vector machines. In *Procs. 12th Inter. Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011), Delft, Netherlands*, 2011.

K. Bache and M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

Léon Bottou and Chih-Jen Lin. Support vector machine solvers. *Large scale kernel machines*, pages 301–320, 2007.

Srihari Cadambi, Igor Durdanovic, Venkata Jakkula, Murugan Sankaradass, Eric Cosatto, Srimat Chakradhar, and Hans Peter 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. IEEE, 2009.

Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A map reduce framework for programming graphics processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.

Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

Andrew Cotter, Nathan Srebro, and Joseph Keshet. A gpu-tailored approach for training kernelized svms. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 805–813. ACM, 2011.

T. Friess. Support Vector Neural Networks: The Kernel-Adatron with Bias and Soft-Margin. Technical report, UK, 1998. URL `http://www.google.com/search?client=safari&rls=en-us&q=Support+Vector+Neural+Networks:+The+Kernel-Adatron+with+Bias+and+Soft-Margin&ie=UTF-8&oe=UTF-8`.

Elmer G Gilbert. An iterative procedure for computing the minimum of a quadratic form on a convex set. *SIAM Journal on Control*, 4(1):61–80, 1966.

Hans P Graf, Srihari Cadambi, Venkata Jakkula, Murugan Sankaradass, Eric Cosatto, Srimat Chakradhar, and Igor Dourdanovic. A massively parallel digital learning processor. In *Advances in Neural Information Processing Systems*, pages 529–536, 2008.

Matthew Jacobsen, Yoav Freund, and Ryan Kastner. Riffa: A reusable integration framework for fpga accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 216–219. IEEE, 2012.

Thorsten Joachims. Making large scale svm learning practical. 1999.

S Sathiya Keerthi, Shirish Krishnaj Shevade, Chiranjib Bhattacharyya, and Krishna RK Murthy. A fast iterative nearest point algorithm for support vector machine classifier design. *Neural Networks, IEEE Transactions on*, 11(1):124–136, 2000.

Hyun-Chul Kim, Shaoning Pang, Hong-Mo Je, Daijin Kim, and Sung-Yang Bang. Support vector machine ensemble with bagging. In *Pattern recognition with support vector machines*, pages 397–408. Springer, 2002.

David M Lazer, Ryan Kennedy, Gary King, and Alessandro Vespignani. The parable of google flu: traps in big data analysis. 2014.

Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits. URL http://yann.lecun.com/exdb/mnist/.

Shawn Martin. Training support vector machines using gilbert's algorithm. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005.

Karthik Nagarajan, Brian Holland, AlanD. George, K.Clint Slatton, and Herman Lam. Accelerating machine-learning algorithms on fpgas using pattern-based decomposition. *Journal of Signal Processing Systems*, 62(1):43–63, 2011. ISSN 1939-8018. doi: 10.1007/s11265-008-0337-9. URL http://dx.doi.org/10.1007/s11265-008-0337-9.

Markos Papadonikolakis and C 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. IEEE, 2010.

Markos Papadonikolakis and C-S Bouganis. A scalable fpga architecture for non-linear svm training. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 337–340. IEEE, 2008.

John Shawe-Taylor and Nello Cristianini. *Kernel methods for pattern analysis*. Cambridge university press, 2004.

Vladimir N Vapnik. Statistical learning theory. 1998.

Lipo Wang. *Support Vector Machines: theory and applications*, volume 177. Springer Science & Business Media, 2005.