High-Level Synthesis of Support Vector Machines using Vivado HLS

Venkatesh Mahadevan
Department of Electrical and Computer
University of Toronto, Toronto, Canada
venkatesh.mahadevan@utoronto.ca

***Abstract*—This paper describes Support Vector Machines and how they are utilized in various data mining algorithms. A software implementation using the Sequential Minimal Optimization (SMO) methodology proposed by Platt is described in detail. Furthermore, literature on hardware implementations of the SMO algorithm are also described. Finally, a hardware implementation of the algorithm is elucidated, and the results with the software implementation are compared. The paper concludes by putting forth suggestions on how the SMO algorithm could be improved and how these can be achieved by changes in hardware implementation.**

***Keywords* — SMO, Lagrange, SVM, kernels, Zynq, Vivado** .

I. INTRODUCTION

Support Vector Machines are often used in machine learning algorithms for supervised learning. Given a set of training examples, which may belong to one or two categories, an SVM training algorithm builds a model which can be used for classifying new examples into one of the above classes. These models represent examples as points in space, separated into categories by the maximum margin possible. Newer examples are mapped into the same space, and are classified on the basis of the side of the margin they fall in. Thus, SVM's are primarily viewed as non-probabilistic binary linear classifiers. They can also be used in their non-linear form by employing different kernels and multi-dimensional input spaces.

The next sections go on to describe how the SVM algorithm works in the linear and non-linear cases, followed by a discussion of SMO and Platt's implementation of the same.

II. HOW DO SVM'S WORK?

SVM's work via the construction of one or more hyperplanes in high-dimensional plane, which can be used for classification and regression, amongst others [1]. A good classifier is often characterized by the maximum distance of the hyperplane to the nearest training data point of any class. This is often referred to as the *'functional margin*' of the classifier, and a higher value is often desired to lower the generalization error of the classifier. To keep the computational load reasonable, the mappings used by SVM schemes are designed to ensure that dot products may be computed easily in terms of the variables in the original space, by defining them in terms of a kernel function k(x,y) selected to suit the problem. The hyperplanes in the higher-dimensional space are defined as the set of points whose dot product with a vector in that space is constant. The vectors defining the hyperplanes can be chosen to be linear combinations with parameters α_i of images of feature vectors that occur in the data base. With this choice of a hyperplane, the points in the feature space that are mapped into the hyperplane are defined by the relation:

$\sum \alpha_i k(x_i, x) = \text{constant}$	Eqn 2.1

Note that if k(x, y) becomes small as y grows further away from x, each term in the sum measures the degree of closeness of the test point x to the corresponding data base point x_i. In this way, the sum of kernels above can be used to measure the relative nearness of each test point to the data points originating in one or the other of the sets to be discriminated.

:

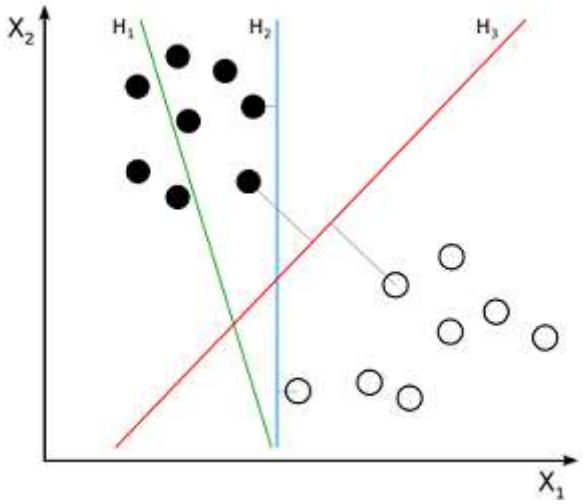


Fig 2.1 H1, H2 and H3 hyperplanes. H3 performs the best separation.

The following section describes the mathematical formulations behind linear and non- linear SVM's.

III. LINEAR AND NON-LINEAR SVM'S

Mathematically, linear SVM's can be formulated as follows:

Given some training data D, a set of n points of the form:

$D = \{(x_i, y_i)$ s.t x_i belongs to set of reals and y_i is either -1 or +1 for all i from 1 to n.	Eqn 3.1

where the y_i is either 1 or −1, indicating the class to which the point x_i belongs. Each x_i is a p-dimensional real vector. We want to find the maximum-margin hyperplane that divides the points having y_i=1 from those having y_i=-1. Any hyperplane can be written as the set of points x satisfying:

w*x-b= constant	Eqn 3.2

where * denotes the dot product and W the (not necessarily normalized) normal vector to the hyperplane. The parameter $\frac{b}{\|\mathbf{w}\|}$ determines the offset of the hyperplane from the origin along the normal vector w.

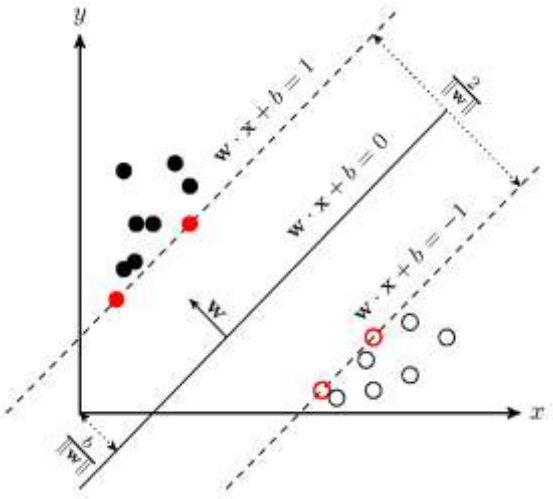


Fig 3.1 Maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors.

If the training data are linearly separable, we can select two hyperplanes in a way that they separate the data and there are no points between them, and then try to maximize their distance. The region bounded by them is called '*the margin*'. These hyperplanes can be described by the equations

$w*x-b= 1$	Eqn 3.3

and

$w*x-b= -1$	Eqn 3.4

By using geometry, we find the distance between these two hyperplanes is $\frac{2}{\|\mathbf{w}\|}$, so we want to minimize ||w||. As we also have to prevent data points from falling into the margin, we add the following constraint: for each i either

$w*x_i-b >= 1$	Eqn 3.5

for x_i of the first class or:

$w*x_i-b <= -1$	Eqn 3.6

for xi of the second class.
This can be rewritten as:

$y_i*(w*x_i-b) >= 1$	Eqn 3.7

for i in the range [1,n]. Putting this together we get the optimization problem:
Minimize (in w, b) ||w||, subject to Eqn 3.7 above.
Since the above equation involves the norm of w (which requires taking the square root), we transform it into a quadratic programming problem by replacing ||w|| with $1/2*\|w\|^2$. The problem can now be re-stated as follows:

$\arg\min 1/2*\|w\|^2$	Eqn 3.8

subject to the same constraints as in Eqn 3.7 above. Since multiple solutions can arise, leading to both local and global minima, we try to find scaling factors (also called Lagrange multipliers and denoted by the symbol α) such that a saddle point is reached. This constraint allows us to restate the problem in the following manner:

$\arg\min (w,b) \max (\alpha>=0) \{ 1/2*\|w\|^2 - \sum\alpha_i*(y_i*(w*x_i-b)) - 1)$	Eqn 3.9

This allows the problem to be solved via standard quadratic programming techniques and software. The solution to the problem can be expressed via the stationary *Karush-Kuhn-Tucker* (KKN) condition:

$w=\sum\alpha_i*y_i*x_i$	Eqn 3.10

Only a few of the Lagrange multipliers will be greater than zero, and the corresponding x_i are regarded as the support vectors. It is also possible to calculate the offset b for each point from the hyperplane, by averaging over all the support vectors as follows:

$b=(1/N_{SV})*\sum(w*x_i-y_i)$	Eqn 3.11

where N_{SV} is total number of support vectors for all examples from the training data. This formulation is often referred to as the '*primal*' form for linear SVM's.

In 1992, Boser, Guyon and Vapnik suggested a method of creating non-linear SVM's by using the kernel trick and multi-dimensional feature spaces. The mathematical proposition remains the same as above, with the only difference between that the dot products in the equations above are replaced by non-linear kernel functions. This allows the classifier to be a hyperplane in the high feature space, and remain non-linear in the input space. The only problem is that higher dimensions result in an increase in the generalization error, and the challenge is to bound the latter via suitable choice of the kernel functions. Examples of such kernels include, but are not limited to:

- *Polynomial (homogenous)*: $k(x_i, x_j)= (x_i*x_j)^d$
- *Polynomial (inhomogenous)*: $k(x_i, x_j)= (x_i*x_j+1)^d$
- *Gaussian radial basis function*: $k(x_i, x_j)= \exp(-\gamma\|x_i-x^j\|^2)$ for $\gamma > 0$.γ can also substituted as $1/(2*\sigma^2)$.
- *Hyperbolic tangent*: $k(x_i, x_j)= \tanh(\kappa*x_i+x_j+c)$ for some $\kappa > 0$ and $c < 0$.

The kernel trick can then be expressed as:

$w*\varphi(x)=\sum(\alpha_i*y_i*k(x_i , x)$	Eqn 3.12

Since the quadratic programming problem is an $O(n^3)$ algorithm, the time need to solve the problem increases dramatically as the number of inputs increases. Typical data mining problems often involve millions of examples with numerous features, which increases the runtime of the problem dramatically. This necessitates the need for using specialized algorithms for solving the QP problem by breaking down the main problem into smaller, manageable chunks and using heuristics for the same. A variety of such algorithms exist, that rely on the following four methodologies mainly:

- *Sequential minimal optimization*, which breaks down the problem into 2 dimensional sub-problems which are solved analytically,
- *Interior-point method*, which uses Newton-like methods to satisfy the KKN conditions for the primal form,
- *Sub-gradient descent*, which is an iterative method used for convex optimization problems,
- *Co-ordinate descent*, which seeks to minimize a multivariable function, one direction at a time.

The next sections describes the SMO algorithm and Platt's implementation of the same.

IV. SMO ALGORITHM

SMO is a simple algorithm that solves the SVM QP problem without any extra matrix storage and without invoking an iterative numerical routine for each sub-problem [2]. SMO decomposes the overall QP problem into QP sub-problems similar to Osuna's method. It does this by choosing the smallest optimization problem at each step. For the standard SVM QP problem, the smallest possible optimization problem involves two Lagrange multipliers because the Lagrange multipliers must obey a linear equality constraint. At every step, SMO chooses two Lagrange multipliers to jointly optimize, finds the optimal values for these multipliers, and updates the SVM to reflect the new optimal values.

The advantage of SMO lies in the fact that solving for two Lagrange multipliers can be done analytically. Thus, an entire inner iteration due to numerical QP optimization is avoided. The inner loop of the algorithm can be expressed in a small amount of C code, rather than invoking an entire iterative QP library routine. Even though more optimization sub-problems are solved in the course of the algorithm, each sub-problem is so fast that the overall QP problem can be solved quickly. In addition, SMO does not require extra matrix storage (ignoring the minor amounts of memory required to store any 2x2 matrices required by SMO). Thus, very large SVM training problems can fit inside the memory of an ordinary personal computer or workstation. Because manipulation of large matrices is avoided, SMO may be less susceptible to numerical precision problems.

There are three standard components of the SMO algorithm:

1. Solving for the two Lagrange multipliers
2. Heuristic to choose which multiplier to optimize
3. A method to compute b, the distance from the hyperplane.

To start, the QP problem to train an SVM can be stated as:

$$\max(\alpha) = W(\alpha) = \sum\alpha_i - (1/2)* \sum\sum y_{i*}y_{j*}k(x_i, x_j)*\alpha_{i*}\alpha_j, \quad 0<=\alpha_i<=C, \text{ (for all i)} \qquad \text{Eqn 4.1}$$

$$\sum y_{i*}\alpha_i = 0 \qquad \text{Eqn 4.2}$$

The solution found is considered to be optimal only if the KKT conditions are fulfilled and $Q_{ij}=y_i*y_j*\ k(x_i, x_j)$ is positive semi-definite. The KKT conditions are as follows:

$$\begin{aligned} &\alpha_i = 0 \rightarrow y_i*f(x_{i)} >=1, \\ &0 < \alpha_i < C \rightarrow y_i*f(x_{i)} =1, \\ &\alpha_i = C \rightarrow y_i*f(x_{i)} <=1 \end{aligned} \qquad \text{Eqn 4.3}$$

The above conditions can be evaluated one at a time, which is useful when constructing the SMO algorithm.

In order to solve for the two Lagrange multipliers, SMO first computes the constraints on these multipliers and then solves for the constrained maximum. For convenience, all quantities that refer to the first multiplier will have a subscript 1, while all quantities that refer to the second multiplier will have a subscript 2. Because there are only two multipliers, the constraints can easily be displayed in two dimensions. The bound constraints in (Eqn 4.3) cause the Lagrange multipliers to lie within a box, while the linear equality constraint in (Eqn 4.3) causes the Lagrange multipliers to lie on a diagonal line. Thus, the constrained maximum of the objective function must lie on a diagonal line segment. This constraint explains why two is the minimum number of Lagrange multipliers that can be optimized: if SMO optimized only one multiplier, it could not fulfill the linear equality constraint at every step

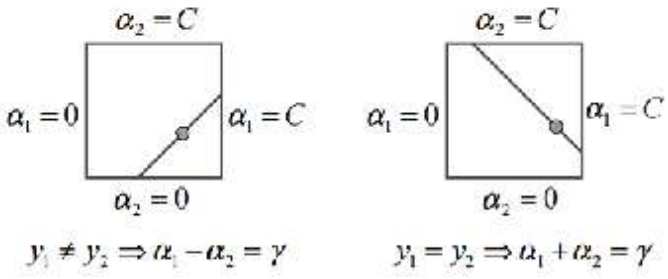


Fig 4.1 The two Lagrange multipliers must fulfill all of the constraints of the full problem. The inequality constraints force them to lie in a box, while the linear equality constraint forces them to lie on a diagonal line. Hence, one step of the SMO must find the optimum of the objective function on a diagonal line segment. In the above figure, $\gamma = \alpha_1^{old} + s* \alpha_2^{old}$, is a constant that depends on the previous values of α_1 and α_2, and $s=y_1*y_2$.

The ends of the diagonal segment are computed by computing the second Lagrange multiplier α_2, and computing the former in terms of the latter. If the target y_1 does not equal y_2, the following bounds apply to α_2:

$$L= \max(0, \alpha_2^{old} - \alpha_1^{old}),\ H = \min(C, C + \alpha_2^{old} - \alpha_1^{old}) \qquad \text{Eqn 4.4}$$

If the target y1 equals the target y2, then the following bounds apply to α_2 :

$$L= \max(0, \alpha_1^{old} + \alpha_2^{old} - C),\ H = \min(C, \alpha_1^{old} + \alpha_2^{old}) \qquad \text{Eqn 4.5}$$

The second derivative of the objective function along the diagonal line can be expressed as:

$$y=2*k(x_1, x_2) - k(x_1, x_1) - k(x_2, x_2) \qquad \text{Eqn 4.6}$$

The next step of SMO is to compute the location of the constrained maximum of the objective function in Eqn 4.3 while allowing only two Lagrange multipliers to change. Under normal circumstances, there will be a maximum along the direction of the linear equality constraint, and will be less than zero. In this case, SMO computes the maximum along the direction of the constraint:

$$\alpha_2^{new} = \alpha_2^{old} - ((y_2*(E_2-E_1))/\eta) \qquad \text{Eqn 4.7}$$

where $E_i = f^{old}(x_i)-y_i$ is the error on the i-th training example. Next, the constrained maximum is found by clipping the unconstrained maximum to the ends of the line segment:

$$\begin{aligned} \alpha_2^{new, clipped} &= \{ H, \text{ if } a_2^{new} >=H, \\ &= \{ a_2^{new}, \text{ if } L< a_2^{new} <H \\ &= \{ L, \text{ if } a_2^{new} <=L \end{aligned} \qquad \text{Eqn 4.8}$$

Now, let $s=y_1*y_2$. The value of α_i is computed from the new, clipped α_2:

$$\alpha_1^{new} = \alpha_1^{old} + s* (\alpha_2^{old} - \alpha_2^{new, clipped}) \qquad \text{Eqn 4.9}$$

Under unusual circumstances, η will not be negative. A zero η can occur if more than one training example has the same input vector

x. SMO will work even when η is not negative, in which case w is computed at the ends of the line segment. Only those terms in the objective function depending on α_2 need to be evaluated. SMO then moves the Lagrange multipliers to the end point with the highest value of the objective function. In order to allow for joint maximization at both ends, we need to make sure that the objective function is not the same at both ends, even when the kernel obeys Mercer's conditions.

SMO will always optimize two Lagrange multipliers at every step, with one of the Lagrange multipliers having previously violated the KKT conditions before the step. That is, SMO will always alter two Lagrange multipliers to move uphill in the objective function projected into the one-dimensional feasible subspace. SMO will also always maintain a feasible Lagrange multiplier vector. Therefore, the overall objective function will increase at every step and the algorithm will converge asymptotically. In order to speed convergence, SMO uses heuristics to choose which two Lagrange multipliers to jointly optimize.

There are two separate choice heuristics: one for the first Lagrange multiplier and one for the second. The choice of the first heuristic provides the outer loop of the SMO algorithm. The outer loop first iterates over the entire training set, determining whether each example violates the KKT conditions (Eqn 4.3). If an example violates the KKT conditions, it is then eligible for immediate optimization. Once a violated example is found, a second multiplier is chosen using the second choice heuristic, and the two multipliers are jointly optimized. The feasibility of the dual QP (Eqn. 4.3) is always maintained. The SVM is then updated using these two new multiplier values, and the outer loop resumes looking for KKT violators.

To speed training, the outer loop does not always iterate over the entire training set. After one pass through the training set, the outer loop iterates only over those examples whose Lagrange multipliers are neither 0 nor C (the non-bound examples). Each example is checked against the KKT conditions, and violating examples are eligible for immediate optimization and update. The outer loop iterates over all the non-bound examples until they obey the KKT conditions within ε. The outer loop then alternates in iterating over the entire training set (single passes) and the non-bound subsets (multiple passes) until the entire training set obeys the KKT conditions within ε. At this point , the algorithm terminates.

The first choice heuristic concentrates CPU time on examples most likely to violate the KKT conditions: the non-bound subset. As the algorithm progresses, the Lagrange multipliers that are within bounds are likely to stay so, while Lagrange multipliers that are not at the bounds will change as other examples are optimized. The SMO algorithm will thus iterate over the non-bound subset until that subset is self-consistent, then SMO will scan the entire data set to search for any bound examples that have become KKT-violated due to optimizing the non-bound subset.

Once a first Lagrange multiplier is chosen, SMO chooses the second multiplier so as to maximize the size of the step taken during joint optimization. Since evaluating the kernel function is time consuming, SMO approximates the step size by the absolute value of the numerator in Eqn 4.7. It keeps a cached error value E for every non-bound example in the training set, and chooses one that maximizes the step size. If E_1 is positive, SMO chooses an example with minimum error E_2 and vice versa. In case positive progress cannot be made (if some training examples share identical input vectors), a hierarchy of second choice heuristics are used to find Lagrange multipliers such that a non-zero step can be found on joint optimization of the former. As described previously, this iteration is performed over both the non-bound examples and the entire training set, and starts at random locations so as to not bias the algorithm towards the examples at the beginning of the training set. If a second example corresponding to a first example is not found, the latter is skipped and another suitable first example is searched for.

$$b_1 = E_1 + y_1*(\alpha_1^{new} - \alpha_1^{old})*k(x_1, x_1) + y_2*(\alpha_2^{new,clipped} - \alpha_2^{old})*k(x_1, x_2) + b^{old} \qquad \text{Eqn 4.10}$$

After each step, b is computed separately so that the KKT conditions are fulfilled for both optimized examples. Threshold b_1 for example (in Eqn 4.10) is valid only when the new α_1 is within bounds, because it forces the output to be y_1 when the input is x_1. The same can be said for b_2 with respect to α_2, y_2 and x_2.

$$b_2 = E_2 + y_1*(\alpha_1^{new} - \alpha_1^{old})*k(x_1, x_2) + y_2*(\alpha_2^{new,clipped} - \alpha_2^{old})*k(x_2, x_2) + b^{old} \qquad \text{Eqn 4.11}$$

When b_1 is equal to b_2, they are valid. When both Lagrange multipliers are bound and L is not equal to H, the interval between b_1 and b_2 are all thresholds consistent with the KKT conditions. SMO then chooses b to be halfway between b_1 and b_2, and subtracts b from the weighted sum of the kernels.

$$E_k^{new} = E_k^{old} + y_1*(\alpha_1^{new} - \alpha_1^{old})*k(x_1, x_k) + y_2*(\alpha_2^{new,clipped} - \alpha_2^{old})*k(x_2, x_k) + b^{old} - b^{new} \qquad \text{Eqn 4.12}$$

Regarding the cached error value E, the SMO algorithm computes it any time an example has a Lagrange multiplier that is neither 0 nor C, and sets it to 0 when joint optimization occurs and the multiplier is non-bound. The stored errors for all non-bound multipliers α_k that are not involved in the optimization are updated according to Eqn 4.12 above.

When an error E is required by the algorithm, it will look it up in the error cache if the corresponding Lagrange multiplier is not bound. Otherwise, it will evaluate the current SVM decision based on the current α vector.

V. PLATT'S IMPROVEMENTS TO THE SMO ALGORITHM

With regards to speeding up the SMO algorithm, it has been found that greater performance can be achieved in the case when the input data is sparse. Since most of the kernels use dot products, which can be computationally intensive, it is often better to store the id and the value of those features which are non-zero and only compute the dot-product for them, as shown in the Fig 5.1 (num1 and num2 are sizes of the two vectors). This can be used to compute linear and polynomial kernels directly. Gaussian kernels also use the dot product code through the use of the following identity:

$$\|x-y\|^2 = x*x-2*x*y + y*y \qquad \text{Eqn 5.1}$$

For every input, the dot product of the input with itself is pre-computed and stored to speed up the Gaussians even further.
For a linear SVM, the weight vector is not stored as a sparse array. The dot product of the weight vector w with a sparse input vector

(id, val) can be expressed as ∑w[id[i]]*val[i] (i goes from 0 to number of sparse input vectors).

```
int p1=0, p2 =0,   a1=0, a2=0;
double dot = 0;
while (p1<num1 && p2<num2) {
        a1=x[xindex][p1].id;
        a2=y[yindex][p2].id;
        if (a1 == a2) {
          t1= (x[xindex][p1].value)*(y[yindex][p2].value);
          dot+=t1;
          p1++; p2++;
        } else if (a1 > a2) {
            p2++;
        } else {
          p1++;
        }
      }
```

Fig 5.1 Code for computing dot products of two sparse vectors

For binary inputs, storing the array *val* is not even necessary, since it is always 1. In the sparse dot product multiplication code, the floating point multiplication becomes an increment. For a linear SVM, the dot product of the weight vector with a sparse input vector becomes ∑w[id[i]] (i goes from 0 to number of sparse input vectors). Other code optimizations can also be used, such as using look-up tables for the non-linearities or placing the dot products in a cache. The latter can substantially speed up many SVM QP algorithms, at the expense of added complexity and memory usage.

The pseudo-code listing for Platt's implementation of the SMO algorithm is given below:

```
target = desired output vector
point = training point matrix

procedure takeStep (i1,i2)
if (i1==i2) return 0
alph1 = Lagrange multiplier for i1
y1=target[i1]
E1 = SVM output on point[i1]-y1 (check in error cache)
s=y1*y2
Compute L, H
if (L ==H) return 0
k11 = kernel (point[i1], point[i1])
k12= kernel(point[i1], point[i2])
k22=kernel(point[i2], point[i2])
eta=2*k12-k11-k22
if (eta < 0) {
   a2=alph2- y2*(E1-E2)/eta
   if (a2 < L) a2 = L
  else if (a2 > H) a2 = H
} else {
  Lobj = objective function at a2=L
  Hobj = objective function at a2=H
  if (Lobj > Hobj + eps) a2 =L
  else  if (Lobj < Hobj - eps) a2 =H
  else a2= alph2
}
if (a2 < 1e-8) a2 = 0
else if (a2 > C - 1e-8) a2 =C
if  (|a2-alph2| < eps*(a2+alph2+eps)) return 0
a1=alph1+s*(alph2-a2)
Update threshold to reflect change in Lagrange multipliers
Update weight vector to reflect change in a1 and a2, if linear SVM
Update error cache using new Lagrange multipliers
Store a1 in the alpha array
Store a2 in the alpha array
return 1
endprocedure

procedure examineExample(i2)
   y2 = target[i2]
  alph2 = Lagrange multiplier for i2
  E2 = SVM output on point[i2]-y2 (check in error cache)
  r2=E2*y2
  if (( r2 < -tol && alph2 < C) || (r2 > tol && alph2>0))
{
     if (number of non-zero & non-C alpha > 1){
        i1 = result of second choice heuristic
        if takeStep(i1, i2) return 1
     }
loop over all non-zero and non-C alpha, starting at random
{
    i1 = loop variable
    if takeStep (i1, i2) return 1
  }
}
return 0
endprocedure

main routine:
   initialize alpha array to all zero
   initialize threshold to zero
   numChanged=0;
   examineAll =1;
while (numChanged > 0 | examineAll) {
         numChanged = 0
         if (examineAll)
          loop I over all training examples
            numChanged( += examineExample(I)
        else
          loop I over all examples where alpha is not zero
```

Fig 5.2 Platt's pseudo-code for the SMO algorithm

The following is a summary of the SMO algorithm:

1. Iterate over the entire training example, searching for a λ_1 which violates the KKT condition.
   If λ_1 is found, go to step 2
   If we finish iterating over the entire training example, then we iterate over the non-bound set.
   If λ_1 is found, go to step 2.
   We will alternate iterating over the entire training set and non-bound set looking for a λ_1 which violates the KKT conditions until all λ's obey the KKT condition.
   Then we exit.
2. Search for λ_2 from the non-bound set.
   Take the λ which gives the largest value of $|E_1-E_2|$ as λ_2
   if the two examples are identical, then abandon this λ_2. Go to step 3.
   Otherwise, compute L and H value for λ_2.
   if L=H, then optimization progress cannot be made. Abandon this λ_2. Go to step 3.
   Otherwise, calculate the η value.

If it is negative, calculate the new λ_2 value.
if it is not negative, then calculate the objective function at the L and H points and use the λ_2 value which gives the higher value of the objective function
if $|\lambda_2^{new} - \lambda_2^{old}|$ is less than an ε value abandon it. Go to step 3. Otherwise go to step 4.
3. Iterate over the non-bound set starting at a random point in the set until a λ_2 that can make optimization progress in step 2 is found.
if it is not found, then iterate over the entire training example, starting at a random point, until a λ_2 that can make optimization progress in step 2 is found.
If no such λ_2 is found after these two iterations, then we skip the λ_1 value found and go back to step 1 to find a new λ_1 which violates the KKT condition.
4. Calculate the new λ_1 value.
Update the threshold b, error cache and store the new λ_1 and λ_2 values. Go back to step 1.

The next section goes on to describe the API used for classification and training using the above algorithm.

VI API DESCRIPTION

The software for this project has been adapted from Ginny Mak's Master's thesis on *'The Implementation of Support Vector Machines using Sequential Minimal Optimization'* conducted at the School of Computer Science at McGill University in 2001. Further information about the same can be found here [3]. The software is made up of two packages, the training or learning program *smoLearn* and the classification program *smoClassify* Fig 6.1 below shows the typical use cases:

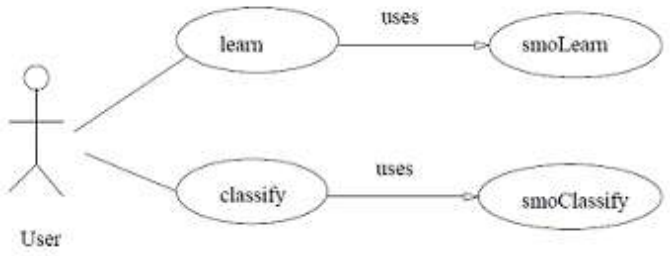


Fig 6.1 Typical use case of the software packages

smoLearn takes an input file whose format is compatible with that of *svm*^{light}, which is an SVM implementation developed by Thorsten Joachim in 2001. Files readable by the former are also readable by this software package. *smoLearn* also takes in the kernel type and a constant as an argument. Currently, three different kernels are supported:

- *Linear* - type 0
- *Polynomial* - type 1
- *Radial basis function* - type 2.

These kernels have already been described in Sec. III. In the case of the first two kernel types, the C parameter and the degree of the polynomial are also indicated by the user. In the case of the RBF kernel, the square of the variance is specified, from which the rbfConstant can be calculated using $1/(2*\text{variance}^2)$.
Five modules make up the smoLearn package. They are:

- Main module
- initializeTraining module
- learn module
- utility module
- result module

Main module:
The module **smoLearn.c** makes up the main module and orchestrates the whole learning process.
smoLearn.c will do its type checking for correct types of parameters based on the kernel chosen by the user. The module also checks for existence of the training file supplied by user and opens the model file with name supplied by the user for writing the training result to. After all the type checking, **smoLearn.c** records the type of kernel chosen by the user, the C parameter, the degree if polynomial kernel is selected, the square of the sigma (variance) if an RBF kernel is used, and the names of the training and model files supplied by the user. It opens the training file for reading and the model file for writing. If there are problems in opening those files, then the program will inform the user the errors and abort. Otherwise **smoLearn.c** will ask the **initializeTraining module** to do all the preparation tasks before the training process starts. These tasks include reading the training file and storing those training data and initializing all the data structures for the training process. If the initialization process fails, then **smoLearn.c** will abort the program. After initializing, **smoLearn.c** calls the **learn module** to start learning and starts a timer. At the end of learning, **smoLearn.c** calculates the time it took the learning algorithm to run and calls the **result module** to write the result of training in the model file. **smoLearn.c** then writes out the following statistics.

- The number of successful kernel iterations.
- The total number of iterations.
- The threshold value, *b*.
- The norm of the weight vector if a linear kernel is used.
- The number of non-bound support vectors and the number of bound support vectors.

initializeTraining module:
The tasks of the **initializeTraining module** consist of allocating memory for the following data structures and reading the training file:
Data structures initialized by **initializeTraining module**:

- A structure called feature consists of an integer element for storing the id of the feature and a double element for storing the value of that feature is defined. Only features with non-zero feature value are stored. Therefore each training example has a number of these structures allocated for it according to the number of non-zero valued features it has.
- A two-dimensional arrays of pointers to the structure feature, **example ****, is allocated for storing all the information of feature id/value pairs of the examples. Hence, example[*i*][*j*] will point to the *j*th feature structure which has non-zero feature value of the *i*th example.
- An array of integers, **target**, is used to store the class labels of the training example. Class labels are '+1' or '-1'. Therefore **target[*i*]** is either +1 or -1 for the *i*th example. These labels are read from the training file supplied by the user.
- An array of integers, **nonZeroFeature**, is used to store the number of non-zero valued features an example has.
- An array of doubles, **error**, is used to store the error of each example. This is the difference between the output of the support vector machine on a particular example and its class label. For the *i*th example, error[*i*] stores the output of support vector machine - target[i].
- An array of doubles, **weight**, is used to store the weight vector of the support vector machine. This is necessary only for a linear kernel.

- An array of doubles, **lambda**, is used to store the Lagrange multipliers of the training examples during training.
- An array of integers, **nonBound**, is used to record whether an example has a non-bound Lagrange multiplier during training, i.e. the value of the multiplier is in the interval (0, C) where C is the parameter which determines the trade-off between training error and margin.
- An array of integers, **unBoundIndex**, is used to index the examples which have non-bound Lagrange multipliers.
- An array of integers, **errorCache**, is used to index examples with non-bound Lagrange multipliers in the order of their increasing errors during training.
- An array of integers, **nonZeroLambda**, is used to index the examples whose Lagrange multipliers are non-zero during training.

Another task performed by **initializeTraining module** is to read the training file and store the information of the training examples.
There are 3 functions in this module:

- **initializeData()**,
- **readFile()**
- **initializeTraining()**

They are described below:
int initializeData(int size):
This function initializes the data structure **example ****, **target**, **nonZeroFeature** and **error** to the size passed in the function argument. It will return 1 if initialization is successful, otherwise it will return 0.

int readFile(FILE *in):
This function assumes that the file pointed to by file pointer '*in*' has been opened successfully. The function makes two passes of the training file. In the first pass, it just counts the number of lines in the file to have an over estimate of the number of training examples. Using this estimate, the arrays **example**, **target**, **nonZeroFeature** and **error** are allocated. In the second pass, it reads each non-comment line and stores the class label either +1 or -1 for each example in the **target** array and the feature id/value pairs in the **example** array. Only non-zero valued features are stored. While reading the lines, the function also keeps track of the maximum feature id it has encountered so far. At the end of the reading of the file, this maximum feature id is taken to be the number of features the training examples have. The user does not have to supply the number of features of the examples. The function returns 1 if reading is successful, otherwise it returns 0.

int initializeTraining(void):
This function initializes the data structures **weight**, **lambda**, **nonBound**, **unBoundIndex**, **error-Cache** and **nonZeroLambda**. It returns 1 if the operation is successful, otherwise it returns 0.

Learn module:
This module runs the training Sequential Minimal Optimization algorithm. It follows the pseudo codes described in Fig 5.2 (except for the modification in the second choice heuristic in the **examineExample** function) and the equations derived in Sec IV. There are five functions in the module.

void startLearn(void):
This function first iterates over all the training examples and calls function **examineExample()** to check if the example violates KKT condition at each iteration. Then the examples with non-bound Lagrange multipliers are iterated over. The function alternates between iterating over all examples and all non-bound examples until all the Lagrange multipliers of the examples do not violate the KKT conditions. At this point, the function exits.

examineExample(int e1):
This function checks for violation of KKT conditions of example e1. If it does not violate the KKT condition then it returns 0 to the function **startLearn()**, otherwise, it uses heuristics as outlined in Sec IV to find another example for joint optimization. In the second hierarchy of the heuristic, it iterates over the non-bound examples starting from a randomly chosen position. If this does not produce a candidate example for joint optimization, then it iterates over all examples looking for another candidate example. Since structurally, the separating plane orients itself over the optimization process, it is more likely that a bound example will become non-bound than for an example with zero Lagrange multiplier becoming a support vector. Another hierarchy was added right after the second hierarchy of the second heuristic and the last hierarchy was changed. If iteration over non-bound examples does not produce an optimization then the next iteration is carried out over the bound examples in selecting the second example for joint optimization. The iteration starts at a random position over the bound examples. If this still does not work out, then the iteration will be done over the examples with zero Lagrange multipliers. This function returns 1 if optimization is successful, otherwise it returns 0.

int takeStep(int e1, int e2):
Two arguments e1 and e2 are passed to the function. They are the examples chosen for joint optimization. Having chosen a second example for joint optimization, **examineExample()** calls **takeStep()** to carry out the optimization. The optimization step follows closely with what is described in Sec IV. If the second derivative eta, (Fig 5.2) is negative, then we calculate the new lambda for the second example and clip it at the ends of the line $L2$ and $H2$. Otherwise, we choose either $L2$ or $H2$ as the new lambda value depending on which one gives a higher objective function value. If joint optimization is successful, **takeStep()** returns 1 to the function **examineExample()**, otherwise it returns 0. At the end the function updates the condition of the support vector machine and all the data structures, which are necessary not only for the optimization process, but also for the **examineExample()** function. Each optimization step occurring in the function **takeStep()** will result in a change in errors for the non-bound examples and change in the threshold value. The first hierarchy of the second choice heuristic used by **examineExample()** requires choosing the second example with the largest positive error if the error of the first example is negative and one with the largest negative error if the error of the first example is positive. The second hierarchy of the second choice heuristic requires iterating over the examples with non-bound lambdas. The third hierarchy requires iteration over the bound examples. An array named **errorCache** holds the index to the non-bound examples arranged in order of increasing error. An array named **unBoundIndex** holds the index to the non-bound examples. An array named **nonZeroLambda** holds the index to the examples whose lambdas are not zero. These arrays are updated at the end of each successful optimization. If after an optimization, the lambda of the *i*th example is non-bound whereas before it is not non-bound, then the index *i* is added to the end of the **errorCache**, and the **unBoundIndex** arrays. If that example has zero lambda before and has non-zero lambda after

optimization, then it is added to the **nonZeroLambda** array. Then **takeStep()** will call the **quicksort()** function to sort the **unBoundIndex** array and the **nonZeroLambda** array in ascending order. In this case, we always have a sorted index array to examples with non-zero lambdas and non-bound lambdas. Sorting those indexes lets us quickly locate an index by using a binary search. This is necessary when we have to remove them from the array because the example has its lambda changed from non-bound to bound or in the **nonZeroLambda** array case, when its lambda changes from non-zero to zero. To remove the index from those arrays, we use a pointer named unBoundPtr for the **unBoundIndex** array and lambdaPtr for the **nonZeroLambda** array. These pointers will always point to the last element of the respective arrays. After we locate the position of the example index and we want to remove the index from either the **unBoundIndex** array or the **nonZeroLambda** array, we mark that position with the number equal to the total number of example + 1. That index will be at the end of the array when we quicksort the array. We just decrement the pointer to remove the index from the list of examples of non-bound lambdas or the list of examples with non-zero lambda after quicksorting. The same method applies to the **errorCache** array. The array holds indexes of examples with non-bound lambdas sorted in increasing order of errors. We use the **qsort2()** in the *utility* module to sort the indexes in ascending order of error after all the errors of non-bound examples have been updated at the end of optimization. A pointer, errorPtr, always points to the last element of the array. When an example whose lambda was not non-bound, becomes non-bound after optimization, we add that index to the end of the array and call *qsort2()* to update the array. If example *i* has its lambda changed from non-bound to bound, we will put into error[*i*] an error which is one greater than the largest error in the non-bound examples. Calling **qsort2()** will then put *i* at the end of the **errorCache** array. We can remove the index *i* just by decrementing the errorPtr.

double dotProduct(FeaturePtr *x, int sizeX, FeaturePtr *y, int sizeY):
The **dotProduct()** function calculates the dot product of two examples and returns the result to the calling function. For a sparse binary matrix, the calculation is sped up by the method suggested by Platt. Instead of doing multiplication, we just increment by 1 whenever the two vectors have the same non-zero valued feature.

double calculateError(int n):
This function is called by **takeStep()** at the end of a successful optimization to update the errors of all the examples with non-bound lambdas. It calculates the error of a non-bound example. The updating just follows the equations in Sec IV.

Utility module:
This module contains 6 functions which provide support for the **learn** module to carry out its task. They are functions which are not specific utility functions and can be used for many applications.
double power(double x, int n):
This function just calculates and returns the value of a double x, raised to the power of an integer, n. It returns 1 when n is zero. It assumes that n is zero or a positive integer.

int binSearch(int x, int *v, int n):
This function searches for x in an integer array v of size n. It returns -1 if x is not in the array, otherwise it returns the array position where x is found.

void swap(int *v, int i, int j):
This function swaps the ith element with the jth element of an integer array.

void quicksort(int *v, int left, int right):
This function uses the quicksort algorithm to sort the elements bounded by the left and right indexes of the integer array v in ascending order.

qsort2(int *v, int left, int right, double *d):
This function sorts the elements bounded by the left and right indexes of the integer array v in an order such that d[v[*i*]] < d[v[*i*+1]] in array d for the *i*th and (i+1)th elements of array v.

int myrandom(int n):
This function returns a random integer in the interval [0, n-1]. The seed is initially set at 0 and is automatically increased by one at each call.

Result module:
There is only one function in this module.
void writeModel(FILE *out):
This function takes the passed argument, an open file pointer, for writing and writes the results of the training in a fixed format. It assumes that the file has been open successfully for writing. See [3] for details on the model file format.

Three modules make up the smoClassify package. They are:

- Main module
- initialize module
- classify module

They are described below:

Main module:
This module consists of **smoClassify.c** which directs the classification process. **smoClassify.c** opens the given model file, test data file for reading and the prediction file for writing. Then it calls the **initialize module** to prepare data structures for the classification task. If there is a problem in preparing the data structures and reading the model and data files, then the program aborts, otherwise, **smoClassify.c** calls the **classify module** to classify the test data and writes the result to the prediction file. If there is an error in the classification process, the program aborts. It also calls *synth_top* to perform the classification, *write_result* to write the results of classification to a file, and *diff_files* to compare the results from hardware and software.

initialize module:
The tasks of **initialize module** are to initialize all the data structures for storing information which will be read from the model file. The two public interfaces are **int readModel(FILE *in)** and **int readData(FILE *in)**. The followings are the data structures initialized by this module.

- **example**, a two-dimensional array of pointers to the structure feature. The structure feature is defined as:
  struct feature {
  int id;
  double value;
  }
  This array will store the feature id/value pairs of each test data read from the test data file.
- **sv**, an array of pointers to the structure features (same as **example**) for storing the id/value pairs of the support vectors read from the model file.
- **target**, an array of integers for storing the class label of the test data read from the test data file. In this software,

this information is not used at all. The user can supply the class label if s/he knows the class label of the data ahead of time and wants to see how a particular trained model performs. If the user is only interested in the classification of some unclassified data, then s/he can put down any class labels in the data file.

- **nonZeroFeature**, an array of integers for storing the number of non-zero valued features each test data has.
- **lambda**, an array of doubles for storing the Lagrange multipliers of each test data.
- **svNonZeroFeature**, an array of integers for storing the number of non-zero valued features a support vector has.
- **weight**, an array of doubles for storing the weight vector of a trained support vector machine. This is only necessary for linear kernel.
- **output**, an array of doubles for storing the output of a trained support vector machine on each test data.

This module is made up of 5 functions:
int synth_top(Feature example[NUM_EXAMPLES]
[MAX_NUM_OF_FEATURES_PER_EXAMPLE],
Feature sv [NUM_SUPPORT_VECTORS]
[MAX_FEATURES_REQUIRED],
double lambda[NUM_SUPPORT_VECTORS],
int svNonZeroFeature[NUM_SUPPORT_VECTORS],
int nonZeroFeature[NUM_EXAMPLES],
double weight[MAX_FEATURES_READ+1],
double output[NUM_EXAMPLES],
int kernelType) :
This is the top-level function for synthesis that calls writeResult for computing and storing the results in the output prediction file.

int readString(char *store, char delimiter, FILE *in):
This function assumes the file pointed to by file pointer '*in*' has been opened successfully for reading. It reads all the characters in a file up to the delimiter and stores the characters read in the given array **store**. It returns 1 if reading is successful, otherwise it returns 0.

int readModel(FILE *in):
This function assumes that the file pointed to by file pointer '*in*' has been opened successfully for reading. This function reads the model file to find out the following:

- the number of features of the examples used in the training
- the kernel type
- the weight vector if the training was done using a linear kernel
- the threshold *b*
- the *C* parameter
- the number of support vectors found in the training
- the feature id/value pairs
- the product (class label*Lagrange multiplier) of the support
- vectors listed in the model file.

The function returns 1 if reading was successful, otherwise it returns 0.

int readData(FILE *in):
This function assumes that the file pointed to by file pointer '*in*' has been opened successfully for reading. It behaves like the function **readFile()** in the **initializeTraining** module of the program **smoLearn**. It just reads the test data file to store the feature id/value pairs of each test data. The function returns 1 if reading is successful, otherwise it returns 0.

void skip(char end, FILE *in):
This function assumes that the file pointed to by file pointer *'in'* has been successfully opened for reading. It skips all the characters read in the file until the end character is encountered.

classify module:
This module is responsible for calculating the output of the trained support vector machine on each test data and for writing out the classification result to a given prediction file. The public interface of the module is the function **writeResult(FILE *out)**. The module is made up of the following functions:
static double dotProduct(int xdim1, int xdim2, Feature x[xdim1][xdim2], int sizeX, int sizeY, int ydim1, int ydim2, Feature y[ydim1][ydim2], int xindex, int yindex);
Two vectors of the data are passed as argument together with the number of non-zero valued features they have to this function. The function calculates the dot product of the two vectors and returns the dot product.

static double wtDotProduct(int wdim, double w[wdim], int sizeX, int sizeY, int ydim1, int ydim2, Feature y[ydim1][ydim2], int yindex);
If the kernel used in training is the linear kernel, the weight vector was calculated and written to the model file at the end of training. The output of the support vector machine can be more quickly calculated by finding the dot product of the weight vector with the test data vector. This function is passed the weight vector, the vector of data together with the number of nonzero valued features of the weight vector and the data vector. It calculates and returns their dot product.

int writeResult(Feature example[NUM_EXAMPLES]
[MAX_NUM_OF_FEATURES_PER_EXAMPLE],
Feature sv [NUM_SUPPORT_VECTORS]
[MAX_FEATURES_REQUIRED],
double lambda[NUM_SUPPORT_VECTORS],
int svNonZeroFeature[NUM_SUPPORT_VECTORS],
int nonZeroFeature[NUM_EXAMPLES],
double weight[MAX_FEATURES_READ+1],
double output[NUM_EXAMPLES],
int kernelType):
The function calls the function **classifyLinear()**, or **classifyPoly()**, or **classifyRbf()** depending on whether the kernel used in training was linear, polynomial or radial basis function respectively. It returns 0 if classification and writing is successful, otherwise it returns 1.

int classifyLinear(Feature example[NUM_EXAMPLES]
[MAX_NUM_OF_FEATURES_PER_EXAMPLE],
Feature sv [NUM_SUPPORT_VECTORS]
[MAX_FEATURES_REQUIRED],
double lambda[NUM_SUPPORT_VECTORS],
int svNonZeroFeature[NUM_SUPPORT_VECTORS],
int nonZeroFeature[NUM_EXAMPLES],
double weight[MAX_FEATURES_READ+1],
double output[NUM_EXAMPLES],
int kernelType) :
This function times the classification and writes the time to stdout at the end of the classification. The function also calculates the output of the support vector machine on each test data. If the output is positive, the class label of the data is +1, else it is -1. The output of the support vector machine for a linear kernel is calculated by means of the following equation.

SVM *output* = w*x + b Eqn 6.1

where **w** is the weight vector of the SVM and *b* is the threshold.

int classifyPoly(Feature example [NUM_EXAMPLES] [MAX_NUM_OF_FEATURES_PER_EXAMPLE], Feature sv [NUM_SUPPORT_VECTORS] [MAX_FEATURES_REQUIRED], double lambda[NUM_SUPPORT_VECTORS], int svNonZeroFeature[NUM_SUPPORT_VECTORS], int nonZeroFeature[NUM_EXAMPLES], double weight[MAX_FEATURES_READ+1], double output[NUM_EXAMPLES], int kernelType):
This function times the classification and writes the time to stdout at the end of the classification. The function also calculates the output of the support vector machine on each test data. If the output is positive, the class label of the data is +1, else it is -1. The output of the support vector machine on a test data **x** for a polynomial kernel is calculated by means of the following equation.

SVM *output* = $\sum\lambda_i*y_i(1+x_i*x)^d$ - b Eqn 6.2

where λ_i is the Lagrange multiplier, *yi* is the class label of a support vector x_i, *n* is the number of support vectors and *b* is the threshold.

int classifyRbf(Feature example [NUM_EXAMPLES] [MAX_NUM_OF_FEATURES_PER_EXAMPLE], Feature sv [NUM_SUPPORT_VECTORS] [MAX_FEATURES_REQUIRED], double lambda[NUM_SUPPORT_VECTORS], int svNonZeroFeature[NUM_SUPPORT_VECTORS], int nonZeroFeature[NUM_EXAMPLES], double weight[MAX_FEATURES_READ+1], double output[NUM_EXAMPLES], int kernelType):
This function times the classification and writes the time to stdout at the end of the classification. The function calculates the output of the support vector machine on each test data. If the output is positive, the class label of the data is +1, else it is -1. The output of the support vector machine on a test data **x** for a radial basis function kernel is calculated by means of the following equation.

SVM *output* = $\sum\lambda_i*y_i*\exp(-|x_i-x|^2/(2*\sigma^2))$-b Eqn 6.3

where λ_i is the Lagrange multiplier, y_i is the class label of a support vector x_i, n is the number of support vectors and b is the threshold.

The next section goes on to describe research (both past and present) conducted in the field of efficient hardware implementations for the SMO algorithm.

VII. HARDWARE IMPLEMENTATIONS OF THE SMO ALGORITHM

Anguita et al [4] proposed a VLSI friendly training algorithm, based on which a digital architecture was presented and implemented on an FPGA. Wee and Lee [5] proposed a concurrent architecture for training SVM's based on the kernel Adatron algorithm. Unfortunately, both the serial and fully parallel architectures are difficult to scale, and thus a trade-off between performance and hardware costs is not possible, making these architectures less suitable to be used in embedded systems, where scalability is an important concern. To address this problem, Cao et.al [6] propose a parallel and scalable architecture, the main features of which include mapping the error cache updating task to multiple units in parallel, and adjusting the number of processing units based on the memory requirements. Their architecture is based on the modifications proposed by Keerthi [7] to Platt's SMO implementation. They have conducted experiments which show that if the input vector dimensions are not large enough, the computation of the Lagrange multipliers (Eqns. 4.7, 4.8 and 4.9) can become a bottleneck. They have therefore chosen to parallelize them in order to train the SVM efficiently. Analyzing Eqns. 4.10-4.12 reveal that each error element is updated independently with one training sample at a time. This allows for the partition of all training examples into smaller subsets and updating the smaller subsets of the error cache accordingly. This will reduce the processing time in parallel hardware almost linearly. Partitioning the array also allows for updating the local values of b simultaneously.
Hence, based on the above analysis, the task of calculating the Lagrange multipliers was mapped to hardware. It is characterized by one Lagrange multiplier optimizer (LMO) and multiple error cache updating units (ECU), all being connected to the cache unit controller (CUC). To support parallel processing, each ECU has its own memory units, holding a portion of training samples. Operation phases of this architecture include device configuration, training data loading, SVM training, and outputting of the result. The global FSM corresponding to the above is shown in Fig 7.1.
The LMO controls the entire process of training. It updates the Lagrange multiplier and checks the KKT conditions at each iteration step. The associated training samples are input by CUC. Because the Gaussian kernel evaluator is incorporated in this architecture, a lookup table (LUT) is contained in this module for exponential operations.

ECU is the basic parallel processing unit of this architecture. The most important design consideration of this module is memory organization. To improve the processing performance, the memory unit of each ECU is divided into two blocks to support parallel memory accessing. Before SVM training, the input patterns of training samples should be stored into the ECUs, each holding a different portion of the input patterns x_j, j=1,2,...,L, where L is the number of training samples in the current ECU. At each iteration, the correct indices are loaded from the corresponding ECUs by CUC and then stored to all ECUs concurrently before updating e_j, j=1,2,...,L. The indices should be stored in a different memory block other than the one that holds x_j to support concurrent memory accessing. With this design, the ECU is able to perform a multiply-accumulate (MAC) operation in a single clock cycle, and the loading and storing of the indices can be pipelined.

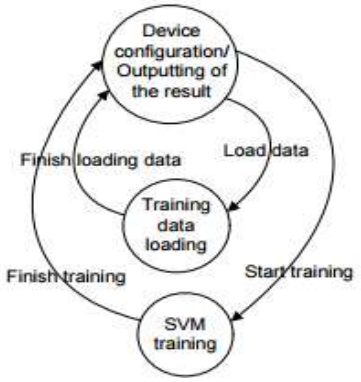


Fig 7.1 Global finite state machine of Cao et. al's architecture

CUC manages the data transfer among different modules. After all ECUs finish their updating tasks in the current iteration step, the CUC determines the final indices by merging the partial results achieved in each ECU. These values are then passed to the LMO module to start the next iteration step. The interconnection between CUC and ECUs is based on a bus-like structure with broadcasting ability. The CUC is the only master device on the bus, while all ECUs act as slave devices, and thus no arbiter is needed for the bus, making it simple, efficient, and easy to extend.
See Fig A.1 in Appendix A for a diagram showing the scalable training architecture described above.

Another system for training has been proposed by Patel et al. [8]. They recognize that a larger sub-problem size (> 2) not only reduces the number of iterations on an average, but also requires computing more kernel columns at each iteration, which is beneficial for the system, having a co-processor to compute kernel entries. Extending the sub-problem involves major changes in two steps of SMO:
• *Extending working set selection*: They use an extended variant of first order derivative based heuristics, which they refer to as Maximal Violating Quad (MVQ) to select a larger working set at each iteration.
• *Solving the extended sub-problem*: For a 4 multiplier sub-problem, using a Gaussian (RBF) kernel matrix of size 4 × 4 requires evaluation of 6 kernel entries. The host solves the 4-multipliers sub-problem using SMO and the co-processor computes the kernel columns.

In spite of parallel computation of multiple kernel entries in the co-processor, fetching kernel entries from a local cache on the host is much faster. They propose changing the size of the working set dynamically, depending on availability of kernel columns in the cache. This utilizes the reduction in iterations offered by using the larger working set as well as the higher probability of availability of only two columns in cache. Their co-processor architecture (see Fig A.2 in Appendix A) consists of the following units:
1. Sparse vector streaming:
They use the DDR3 controller provided by Xilinx Memory Interface Generator. Vector addresses in DDR3 are read from the Vector Translation Lookaside Buffer (TLB), filled by the host. Vector Stream Controller (VSC) continuously issues read commands to DDR3 controller and the corresponding mapping to the control buffer. Vectors i, j, k and l (input indices), which are read at the beginning of the computation, are mapped into column Feature Stream Engines (FSE) memory. The training set is read batch-wise by VSC, from which each vector is mapped one-to-one to a particular row FSE memory along the column. FSEs serve the function of streaming these vectors to the particular KPU in the grid.
2. Kernel Processing Unit:
The Kernel Processing Unit (KPU) evaluates the Gaussian kernel function (see Sec. III). Inputs x_i and x are sparse vectors streamed by the corresponding row and column FSEs into the KPU input buffers. The KPU's finish computation after an arbitrary number of cycles, which is signaled individually using *done*. The KPU result write controller writes the computed kernel entries into the kernel results buffer on receiving done from the KPU grid. The datapath has a 29-stage pipeline latency and uses arithmetic cores from FloPoCo[9] operating at a frequency of 200 MHz.

In the case of SVM classification, numerous articles can be found on how it can be sped up via parallelization using GPU's. However, very few, if any, architectures exist that serve as custom accelerators for the same. Mahmoodi et. al [10] propose a system of performing classification using linear and non-linear SVM's on FPGA's. The training is conducted offline via software, and classification is done in hardware. In this research maximum frequency of 202.840 MHz in linear classification, and classification accuracy of 98.67% in nonlinear one has been achieved. The parameters which are extracted from offline training are used for testing on hardware. In order to realize designed architecture and desired results, three different classes of data related to Persian handwritten digits are used for training and testing samples of the SVM. 800 data from each class (total 2400 samples) are provided. From each class of data, 600 samples have been used for training and 200 samples for testing. The data has feature vector dimension of 7 and 24, which have been extracted by using the geometric moments approach. LibSVM [11] is used for offline training and testing.
Experiments with the linear kernel show that by changing the parameter *C*, the classification error rate will change too (see Fig 7.2). For the Gaussian RBF kernel (see Sec. III), the error does not depend on C, but on γ instead. Choosing appropriate γ value is very important in order to find the optimal classification error rate. Exact formulation doesn't exist to calculate an appropriate value of γ, and often trial and error methods are used to get that (see Fig 7.3 for results of changing γ on the error rate). They have attempted to merge dot products of the Lagrange multiplier (α) and the output (y) into a single vector before multiplying with the support vector matrix and the input matrix x.

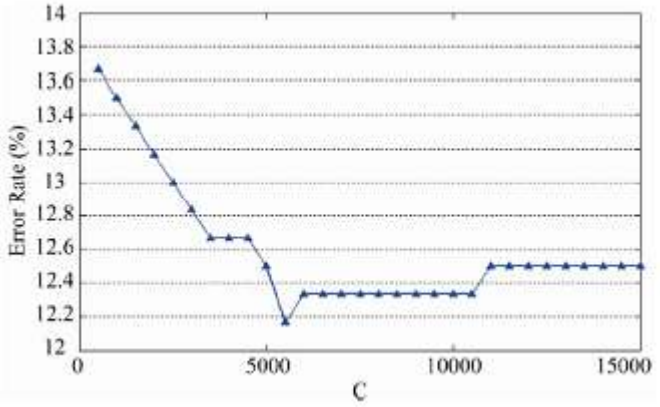


Fig 7.2 Diagram curve of linear SVM classification error rate versus changing parameter C.

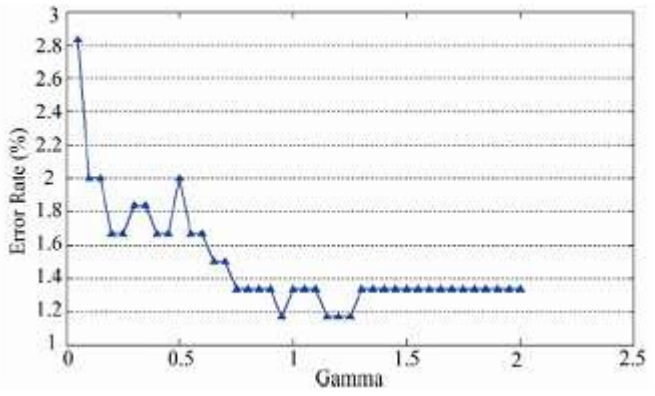


Fig 7.3 Diagram curve of non-linear SVM classification error rate versus changing parameter γ.

To design hardware architecture of above function, a combination of series and parallel methods have been used. Fig A.3 in

Appendix A shows the block architecture for 3-class SVM classification. Fig A.4 in Appendix A shows part of designed architecture for linear SVM, and Fig A.5 shows the same for non-linear SVM. Each array of rows of matrix *SV* is located in *SV ROM* 1 to *SV ROM* 7 outputs, which is indexed by counter1. Test data are in blocks *X_Test ROM*1 to *X_Test ROM* 7 addressed by *Counter* 2 with a period of 70 (as the number of support vectors of *Class*1 & 2). Vector of yα is located in *YAlpha ROM* whose counter is the same as *SV ROM*s. Blocks of *Mult*1 to *Mult*7 and *Addsub*1 to *Addsub*6 perform inner product between test data and support vectors; this result is multiplied by corresponding array of yα vector in *Mult*9. The remaining operations are sum of these values, that is done serially by *Accumulator* and *+b* blocks.

The next section goes on to describe the architecture and methodology chosen for synthesis.

VIII. ARCHITECTURE AND METHODOLOGY CHOSEN

The tool chosen for synthesis was Vivado (2015.1) and Vivado HLS (2015.1) from Xilinx. The Vivado suite of tools is useful for converting high-level (C/C++/SystemC) code into RTL and synthesizing the latter using a particular technology library for FPGA's. The methodology used for conversion is as follows:

- Write and test high-level code outside the Vivado environment.
- Synthesize the high-level code into a gate-level netlist by specifying the device and the board.
- Do C/RTL co-simulation and ensure that hardware performance matches that of software .
- Export RTL with the necessary interfaces for integration with the appropriate processor in the Vivado environment.
- Download and test on the board if available.

While all of these would be possible in the ideal case, there are certain issues with the Vivado tool suite that do not allow a user to synthesize highly complex and compute-intensive algorithms in a scalable and dynamic fashion. They are outlined below, and are a result of experiments conducted with algorithms of varying complexity:

- Need to specify the sizes of the input and output at function interfaces for those functions that lie in the synthesis path.
- Inability to synthesize library functions such as malloc(), pow(), and file I/O functions such as printf(), sscanf() etc.
- Inability to update global variables properly (All variables that are updated within a function need to be local to that function).
- Loop limits must not exceed 2000 iterations.
- Lack of sufficient resources on boards available under Vivado HLS.
- Input and output sizes must be limited. Board resource limits and clock timings are violated if they exceed a certain size, and it varies from application to application.
- Need to specify loop bounds for loops in code that does not lie in the critical path for synthesis (for latency calculations).
- Recursive functions are not supported.
- Classes and templates are not completely supported.
- Need to use specialized hardware for floating point operations to obtain reasonable accuracy (within 24% for the SMO classification code above).
- Inability of the tool to inform the user about the type of logic which a particular piece of code maps to. This often leads to cases wherein the co-simulation feature runs for hours and never finishes.
- Segmentation faults if the improper interface type is specified during RTL export.
- Generation of test vectors (in the *cosim_tv.exe* file) for every input combination in the design. While this may exercise the design thoroughly, any stuck-at faults or X propagation issues that are encountered in this process cannot be traced back to the high-level code, since these result from the logical behaviour of the RTL synthesized by the tool. The resultant effect is that co-simulation can hang sometimes and never finish.
- Vector-less activity propagation is run during bitstream generation when the IP is integrated with the Zynq 7000 SOC. This results in a *low confidence* level when the design is tested on the board. The tool requires the user to input a SAIF file or a VCD file for dynamic power estimation using Primetime. Only then can the user achieve a *high confidence* level on their design.

In light of the issues above, it was decided that the classification portion of SVM would be synthesized using Vivado, and that the training would be conducted offline. The Zedboard Zynq Evaluation and Development Kit was chosen in order to prototype and test the design. The device chosen was xc7z020clg484-1. Further information about the same can be found here [12]. The feature callout for the Zedboard is shown in Appendix B. The default clock period was 10 ns, with a default uncertainty of 1.25 ns. Using the methodology for conversion as described previously, three different kernels (linear, polynomial and radial basis function) were implemented and tested on the board. The training data consisted of words which referred to articles related to corporate acquisitions in *Reuters* articles. A model for each kernel was developed using *smotrain*, and this was then used along with test data by *smosynth* (which was the synthesized version of *smoClassify*) in order to predict as to which words were likely to belong to the same category as described above. A multi-class classification system similar to Fig A.3 in Appendix A was used, with +1 indicating a positive example and -1 a negative example. The architecture chosen was one which was similar to Fig A.1 and Fig A.5 in Appendix A, with separate compute units being assigned to all inputs when sharing of resources was not possible, and the dot products implemented via sparse vector encoding as in Fig 5.1. The synthesis and test results are elaborated upon in the next section.

IX. SYNTHESIS RESULTS

Three different solutions were developed, one for each type of kernel function. These solutions were a result of different synthesis techniques (via incremental application of directives within the tool) being applied to each kernel, and have been optimized to the full extent while keeping in mind the tool limitations described in the previous section. They are outlined below:

1. **Solution 1**:

In solution 1, the critical path of the design is from *synth_top* to *classifyLinear*. Hence, we use the *'loop_tripcount'* directive to specify the minimum and maximum iterations of the loops in the design. The *'pipeline'* directive is also used on the same loops in order to reduce the overall latency. The result of applying the above is that the latency is now *loop_body_latency* + *loop_iteration_count*, instead of being *loop_body_latency* * *loop_iteration_count*. The *'loop_tripcount'* directive has also been applied to the while loop in *wtDotProduct*, which computes the

dot product for sparse input vectors. The same function has been inlined as well using the HLS '*inline*' directive, and automatic inlining has been disabled. Hence, resource utilization increases, as Vivado attempts to reduce the initiation interval (i.e the interval required to process successive inputs) by utilizing more resources. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis phase are shown in Appendix C.

2. **Solution 2**:

In solution 2, the *'loop_tripcount'* directive was used along with the '*pipeline*' directive. These directives were applied to the inner-most loops on the critical path (*synth_top -> classifyPoly*) Although the '*pipeline*' directive could have been applied to the outermost for loop in '*classifyPoly*', doing so resulted in resource utilization far exceeding the limits for the Zedboard. It also resulted in a timing violation since the clock period now went up to 10.71 ns from 10 ns, making the design slower. Hence, the initialization portion of the output was moved into a separate for loop (*POLY_INIT_FOR_LOOP*), and this along with the inner for loop (*POLY_INNER_FOR_LOOP*) was pipelined. Hence, this meant that the critical path of the design was now changed, as a pipelined version of *POLY_INNER_FOR_LOOP* was now available for every instance of the outer for loop (*POLY_OUTER_FOR_LOOP*), meaning that the inner loop iterations could execute concurrently. This also resulted in register retiming being performed on the critical path, and clock period went down from 10.71 ns to 8.23 ns while the resource utilization did not change at this stage. The *'loop_tripcount'* directive had also been applied to the while loop in *dotProduct*, which computed the dot product for sparse input vectors for the non-linear case. The same function has been inlined as well using the HLS '*inline*' directive, and automatic inlining has been disabled. Hence, resource utilization increased, as Vivado attempted to reduce the initiation interval (i.e the interval required to process successive inputs) by utilizing more resources. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis phase are shown in Appendix D.

3. **Solution 3**:

In solution 3, a similar strategy was employed as in solution 2, and was applied to a different critical path (*synth_top->classifyRbf*) . Here too, the clock period increased from 10 ns to 10.71 ns, when the outer loop was pipelined and initializations were performed with the same. Employing a similar directive application strategy as in solution 2 resulted in the final clock period being 9.40 ns. The *dotProduct* function was inlined in this case as well, resulting in increased resource utilization. In addition, it was found that the co-simulation of the design ran for hours without finishing, and on examining the waveforms, it was found that the results of the pipeline were not being flushed before the next input came in. Hence, in this case, the *'pipeline*' directive was employed with the *'-enable_flush*' option to force the pipeline to flush at every stage. A screenshot of the directives used, along with the performance and utilization estimates from the synthesis phase are shown in Appendix E.

Also note that in all three cases, the number of support vectors required were different (37 in the linear case, 50 in the polynomial case, and 17 in the radial basis function case). The number of calls to the product functions were 3x in the radial basis function case, in comparison to the polynomial and linear case. Without proper application of directives, the resource usage would have shot up by a factor of 3x for the same. However, the effects of proper synthesis techniques are clearly visible, as the maximum usage has shot up by only 1.65x in the worst case (see Appendix F), indicating that the designs are highly optimized versions. The best case clock frequency is 121.5 MHz, while the worst-case clock frequency is 106.38 MHz, which are still well above the minimum clock frequency supported by the board (100 MHz). Also, in all the solutions listed above, the '*dataflow*' directive is applied to the top-level *'synth_top'* function, along with the directives listed in that solution. Performing the above optimization results in the insertion of FIFO's between functions to ensure the next function can begin operation before the *'synth_top*' has finished. Since all of the memory accesses in the design are fully sequential in nature, the use of FIFO's is justified. In addition, the use of the '*dataflow*' directive results in reduced utilization of FF's and LUT's, with the exception of additional DSP48E's. The latter result from the tool mapping all Multiply-Add-Accumulate (MAC) operations for floating-point numbers to the DSP48E units. The LUT's and FF's are used as memory elements for looking up indices or values for the arguments passed to the dot product function invocations

It was found that the design was bandwidth-limited due to the use of dual-port RAM's. Hence, the *'array_partition'* directive was used to split the example and support vector memories into multiple smaller ones. This is useful to increase the potential for parallel access. If type is *block* then the source array will split into chunks. If it is *cyclic*, then the elements will be interleaved into the destination arrays. In both cases *factor* is the number of smaller arrays to create. If type is *complete* then factor is ignored and the array is totally split into component registers, thereby not using any BRAM's. Since the BRAM resources on the board were limited, and since the minimum clock period constraint needed to be met, the '*cyclic*' option was chosen with a factor of 4 in all cases. A screenshot of the directives used at the top-level is shown in Appendix F.

Since we added the *-DHW_COSIM* flag to our testbench, Vivado understands that we would like to run C/RTL co-simulation. Hence, we run the same from Vivado with the simulator language set to Verilog. The C/RTL simulation passes in all cases, albeit it runs for very long and produces huge log files. They have been removed in the final package due to size constraints and can be reproduced easily. Appendix G compares the clock period and resource utilization for all three kernels.

The next section goes on to describe the system setup for testing on the board, and comparison of the results of the same with software.

X. INTEGRATION WITH THE ZYNQ 7000 SOC

In order to integrate with the Zynq 7000 SOC system, the IP needs to be exported as an IP-XACT adapter. This involves the addition of '*RESOURCE*' directives to create AXI4LiteS adapters in order to create the IP-XACT adapter. The setup for testing consists of 2 GPIO's (*start_comp* and *finish_comp*). The former consists of a single switch (*SW[0]*) which signals the start of computation. When this is done, the data from the model file is loaded into the local memory banks of the SOC (see Appendix H for the block diagram). The same is done for the test data. Using the DMA controller, the data is then transferred from the memory to the CPU (one of the ARM Cortex A9 cores in the APU). This is then split into the appropriate segments (as indicated in the

'array_partition' directive above) and offloaded onto the SVM classification IP (*synth_top*). The latter functions as a custom accelerator for classification. Once done, the results are sent back to the CPU and written to a file on disk. One of the LED's on board is also lit up to indicate completion. This is then compared to the reference results on disk and the accuracy is measured. The test program is written using the Xilinx SDK, and the software interface to the IP is provided via the drivers created for the IP in the Export RTL stage. In the entire process, the CPU acts as the master, while the IP acts as a slave. Hence, the *s_axilite* interface is used to represent the AXI4Lite slave interface and is applied to all input and output ports of the IP at the top-level, as well as the return value. The return value maps to the signal *'ap_return'*. The *'ap_start*' signal indicates the start of computation, the *'ap_stop*' signal indicates when the computation is complete, and *'ap_idle*' indicates when the IP is in idle mode. The CPU, by default, has the *'m_axi'* interface enabled , which allows it to control our IP. The comparison of software with hardware is shown below for all three kernels (assuming 100% software accuracy):

Table 10.1 Comparison of hardware/software performance of all three kernels

Kernel	Hardware accuracy (%)	% difference from software
Linear	84.616	15.384
Polynomial	75.066	24.934
Radial basis function	99.583	0.417

As can be seen, the RBF kernel has the best accuracy (99.583%) followed by the linear kernel (84.616 %), and then the polynomial kernel (75.066%). The choice of a particular kernel depends on the accuracy required, and the latter can often vary from application to application. For example, a linear kernel will work well for text-based data, while the polynomial or RBF kernel will work better for image-based data.
Appendix I lists the constraints used during placement and bitstream generation. Appendix J contains a snapshot of the project report and the bitstream generation report for all three kernels, while Appendix K contains a block diagram of the entire system, and the synthesized and the implemented designs for all three kernels.

XI. RECOMMENDATIONS FOR IMPROVING DESIGN AND TOOL EFFICIENCY

The implementation so far has been done keeping in mind the constraints present in the tool, as well as the board resource limitations as well. However, this does not mean that this is the best we can achieve in terms of design efficiency. The following are some of the recommendations that can be used to improve tool and design efficiency:

- Use of a Laplacian kernel instead of a Gaussian kernel for a more hardware friendly implementation (exponentiation and multiplication can be implemented via shifters).
- Use of fixed point calculations so that less memory and processor time are required and computation time is reduced.
- Since increasing the number of examples and features can improve accuracy, use of a device or board with more logic elements would be good.
- Use of Support Kernel Machines [13] as an alternative to SVM's.
- Use of algorithms listed in [7] to check for optimal indices concurrently.
- Allowing the user to choose the set of test vectors that the Vivado tool must use.
- Increasing loop limits from the original value of 2000 iterations.
- Support of dynamic input/output sizes.
- Support for synthesis of library functions.
- Ability to allow the user to know which portion of the device executes what portion of the code. This can be very useful if one would like to instantiate multiple copies of the accelerator IP, each working on a different portion of the input (similar to hardware threads in CUDA).
- Ability to check various categories of constraints such as Satisfiability, Observability and Controllability constraints.
- The tool should have the ability to list all stuck-at faults in the design based on the above.
- It would be nice if the tool could provide a scripting API instead of the user having to invoke the IP via the driver functions through the SDK.
- Weka [14] is a Java implementation that combines the algorithm recommendations listed above. It is very efficient, and if a C/C++ implementation were available, the resultant hardware would be very efficient computationally.

XII. CONCLUSION

In this report, I have tried to implement the SMO algorithm for SVM on an FPGA using HLS. The algorithm and the architectures explored have been presented, along the decisions behind choosing the best. The IP has been exported and integrated with the Zedboard Zynq 7000 SOC successfully. It is hoped that the learning outcomes resulting from this assignment will help those who wish to conduct research in the field of custom accelerators for data mining algorithms using FPGA's.

XIII. ACKNOWLEDGEMENTS

I would also like to thank Professor Zhu for allowing me to take on this project. I would also like to thank Ginny Mak for allowing me to use her source code as a starting point. This code and synthesis results for this project can be found under https://github.com/venkatesh20/SVM, which is a public repository on github.

XIII. REFERENCES

1. http://en.wikipedia.org/wiki/Support_vector_machine
2. http://research.microsoft.com/pubs/69644/tr-98-14.pdf
3. http://www.cs.mcgill.ca/~hv/publications/99.04.McGill.thesis.gmak.pdf
4. Anguita, D., Boni, A., Ridella, S., 2003. A digital architecture for support vector machines: theory, algorithm, and FPGA implementation . IEEE Trans. Neur. Networks, 14(5):993-1009. [doi:10.1109/TNN.2003.816033]
5. Wee, J.W., Lee, C.H., 2004. Concurrent support vector machine processor for disease diagnosis. LNCS, 3316:1129- 1134. [doi:10.1007/b103766]
6. www.zju.edu.cn/jzus/opentxt.php?doi=10.1631/jzus.C091050
7. Keerthi, S.S., Shevade, S.K., Bhattacharyya, C., Murthy, K.R.K., 2001. Improvements to Platt's SMO algorithm for SVM classifier design. Neur. Comput., 13(3):637-649. [doi:10.1162/089976601300014493]
8. http://labs.dese.iisc.ernet.in/APC13/pdf/07_alap_sriram.pdf
9. F. de Dinechin and B. Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. Design Test of Computers, IEEE, 28(4):18–27, July-Aug 2011.
10. www.scirp.org/journal/PaperDownload.aspx?paperID=5072
11. http://www.csie.ntu.edu.tw/~cjlin/libsvm/

12. http://www.xilinx.com/support/documentation/
boards_and_kits/UG926_Z7_ZC702_Eval_Kit.pdf
13. http://www.di.ens.fr/~fbach/CSD-04-1307.pdf
14. http://www.cs.waikato.ac.nz/ml/weka/

APPENDICES

APPENDIX A

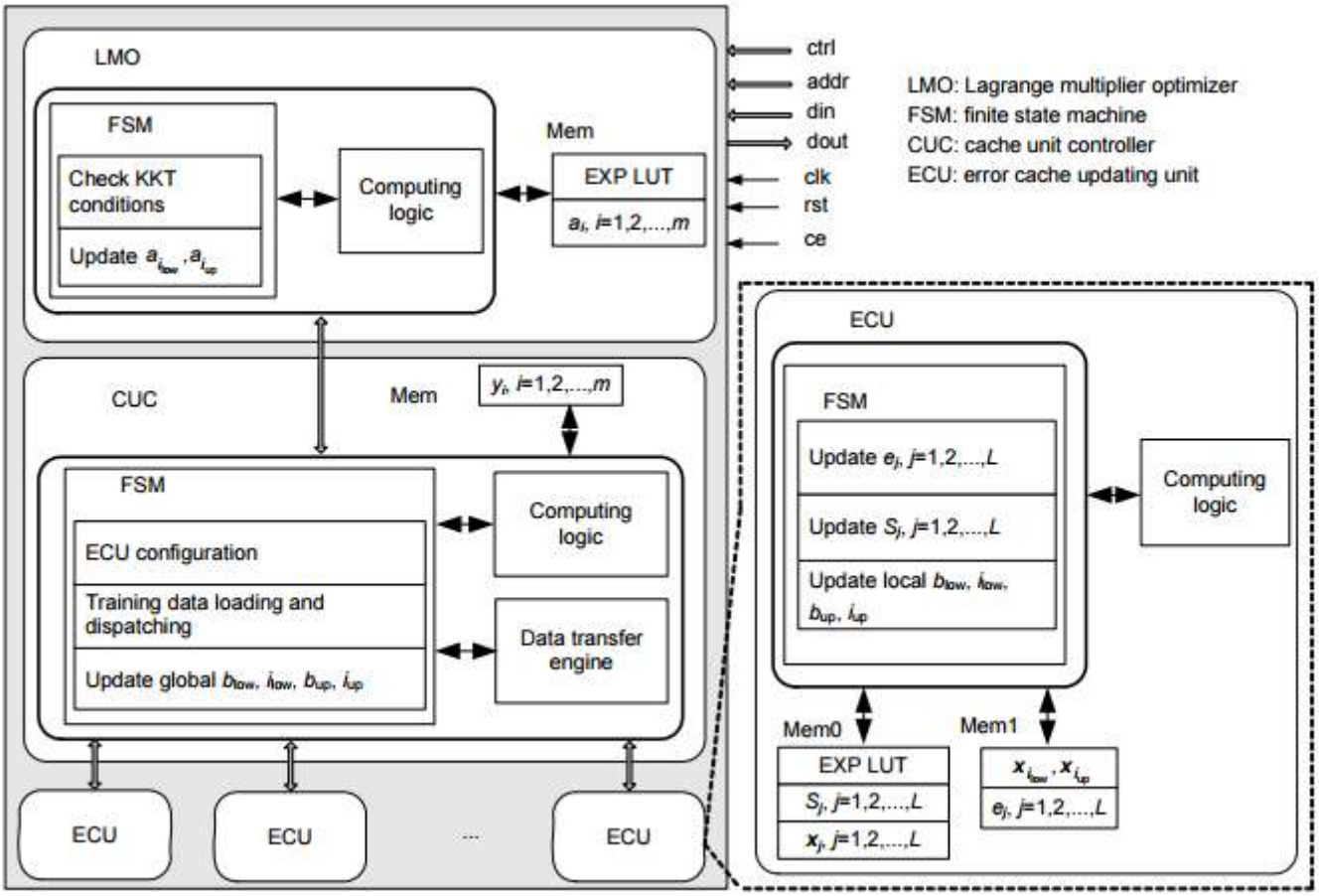


Fig A.1. Scalable digital architecture used by Cao et. al for training SVM's

APPENDIX A (CONT.)

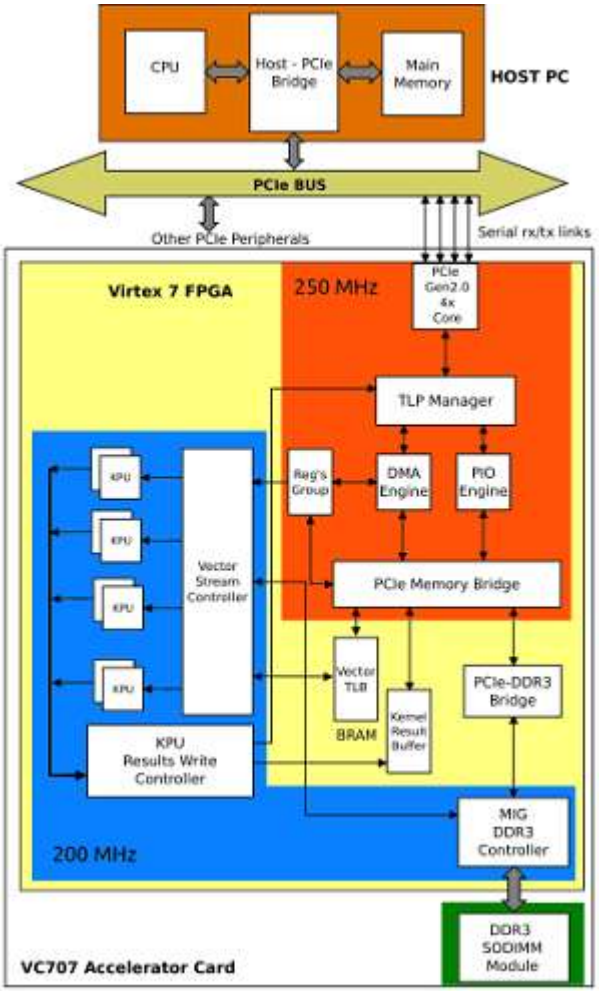


Fig A.2 System architecture (Patel et. al)

APPENDIX A (CONT.)

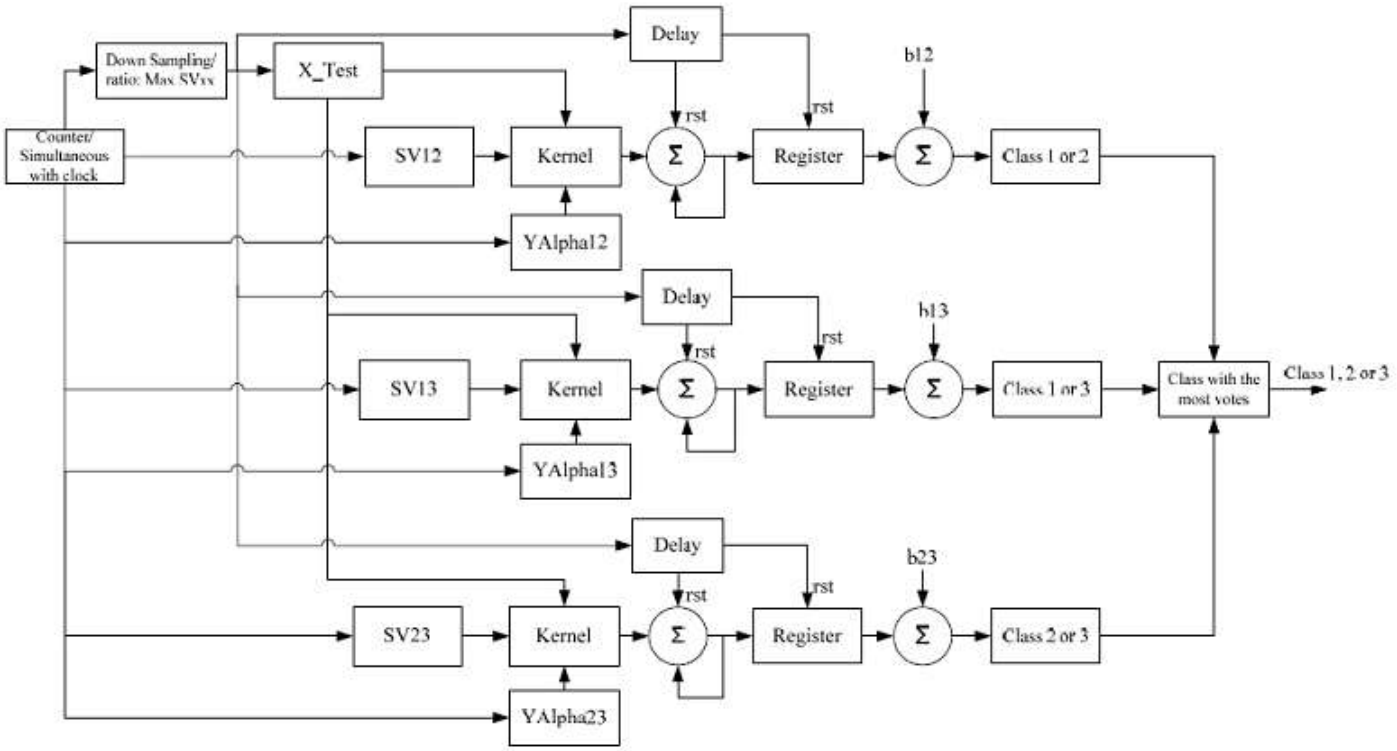


Fig A.3 Block diagram of architecture for 3-class SVM classification used by Mahmoodi et. al

APPENDIX A (Cont.)

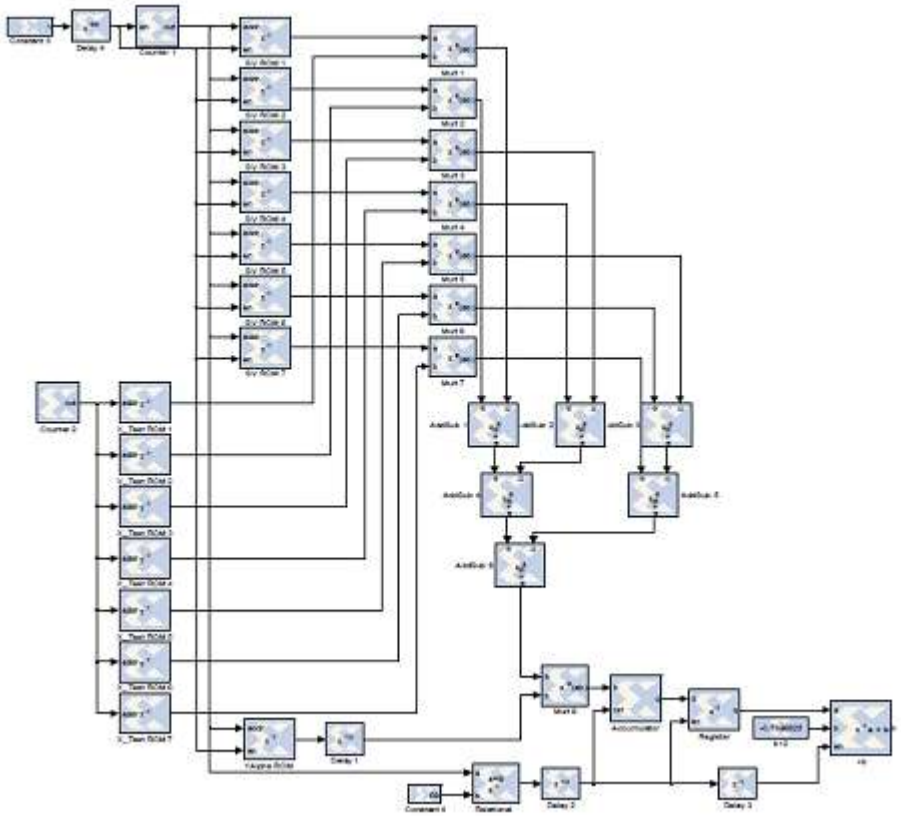


Fig A.4 Block diagram of architecture for linear SVM classification used by Mahmoodi et. al

APPENDIX A (Cont.)

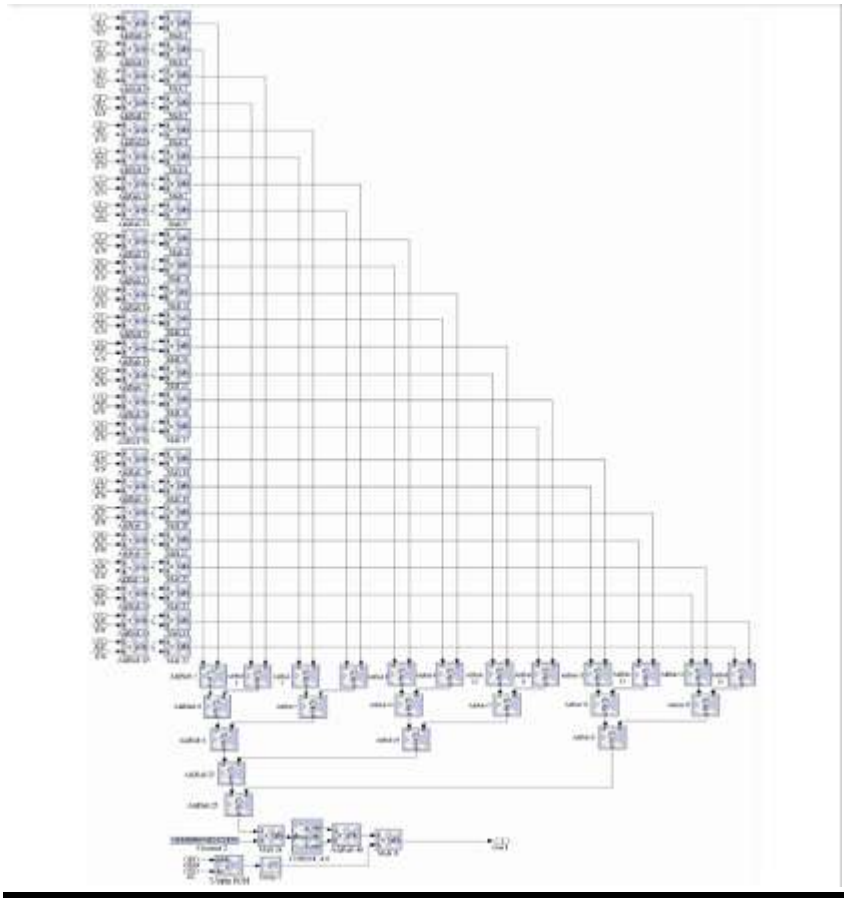


Fig A.5 Block diagram of architecture for non- linear SVM classification used by Mahmoodi et. al

APPENDIX B

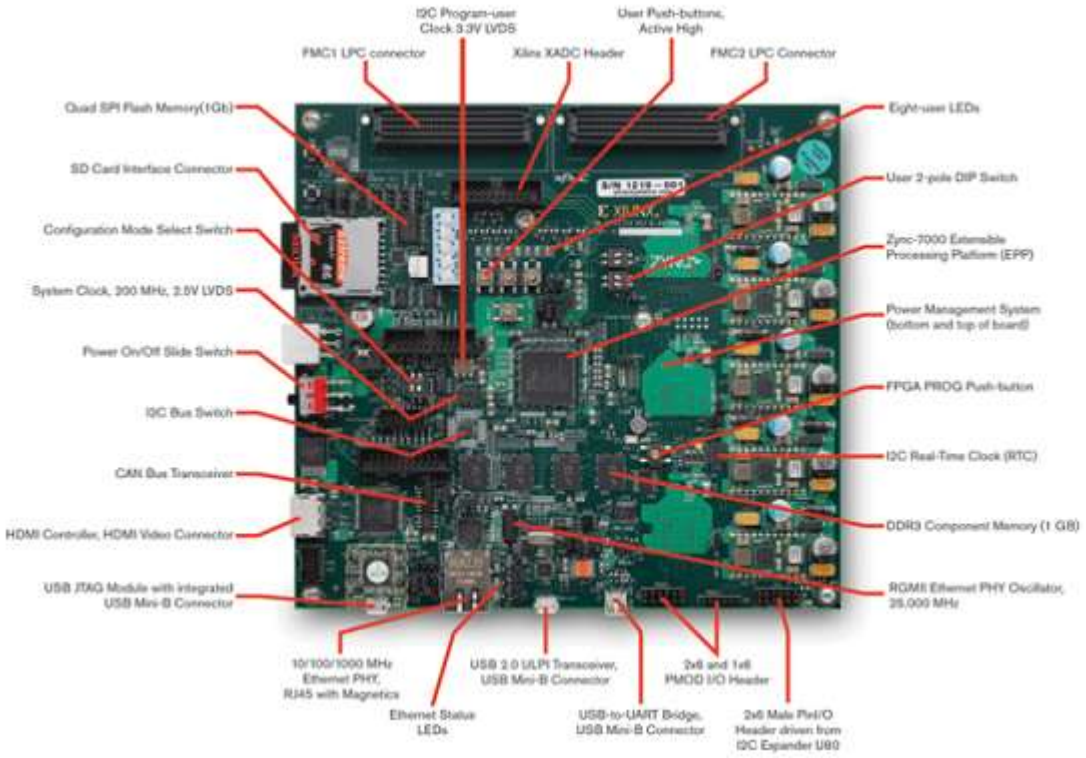
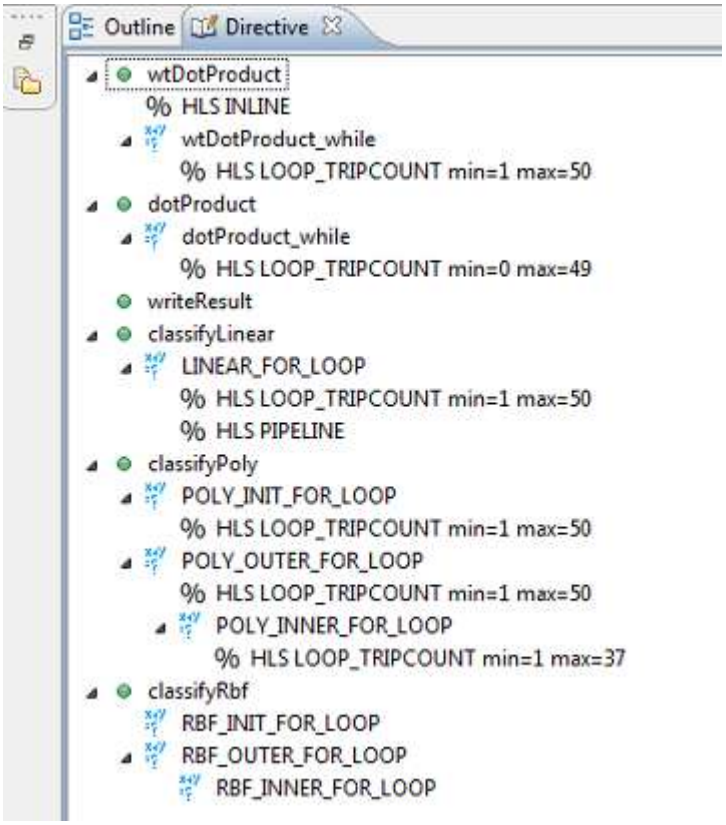


Fig. Feature callout for the ZC702 board

APPENDIX C

1. Directives used for solution 1:



2. Synthesis report:

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.23	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1691053	2	1691054	dataflow

Detail

Instance

Loop

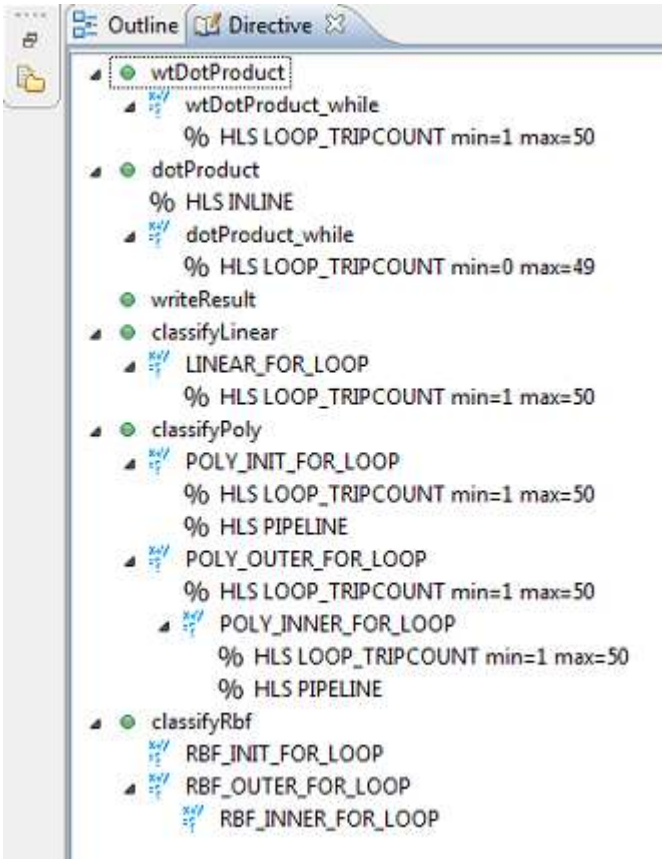
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	64	137	10748	15618
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
Total	64	137	10750	15618
Available	280	220	106400	53200
Utilization (%)	22	62	10	29

APPENDIX D

1. Directives used for solution 2:



2. Synthesis report:

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.23	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	2280053	2	2280054	dataflow

Detail

Instance

Loop

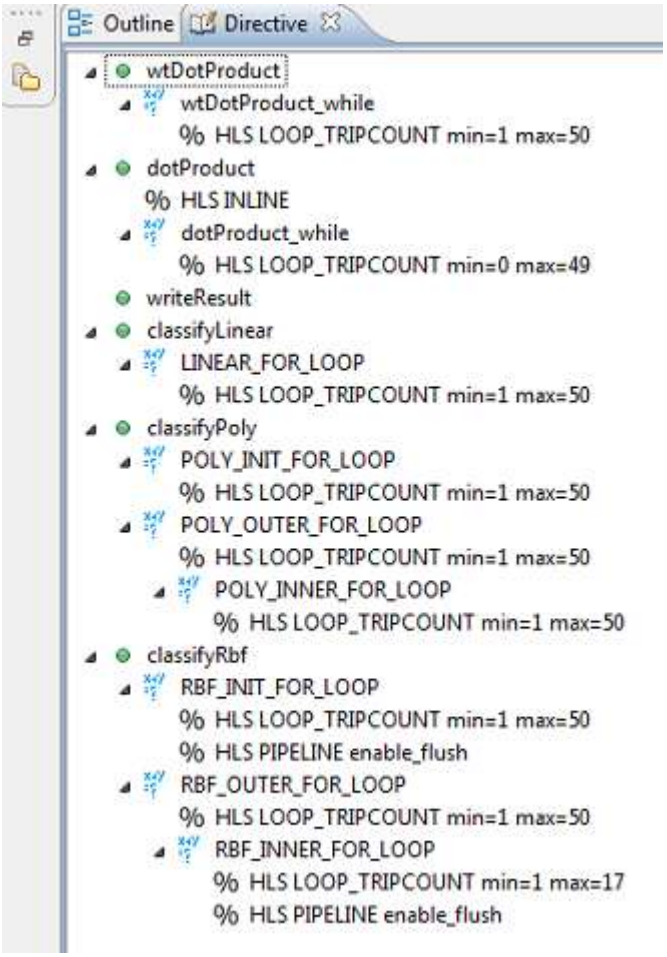
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	64	123	9633	13705
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
Total	64	123	9635	13705
Available	280	220	106400	53200
Utilization (%)	22	55	9	25

APPENDIX E:

1. Directives used for solution 3:



2. Synthesis report:

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	9.40	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	2256803	2	2256804	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	64	166	14264	22741
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
Total	64	166	14266	22741
Available	280	220	106400	53200
Utilization (%)	22	75	13	42

APPENDIX F

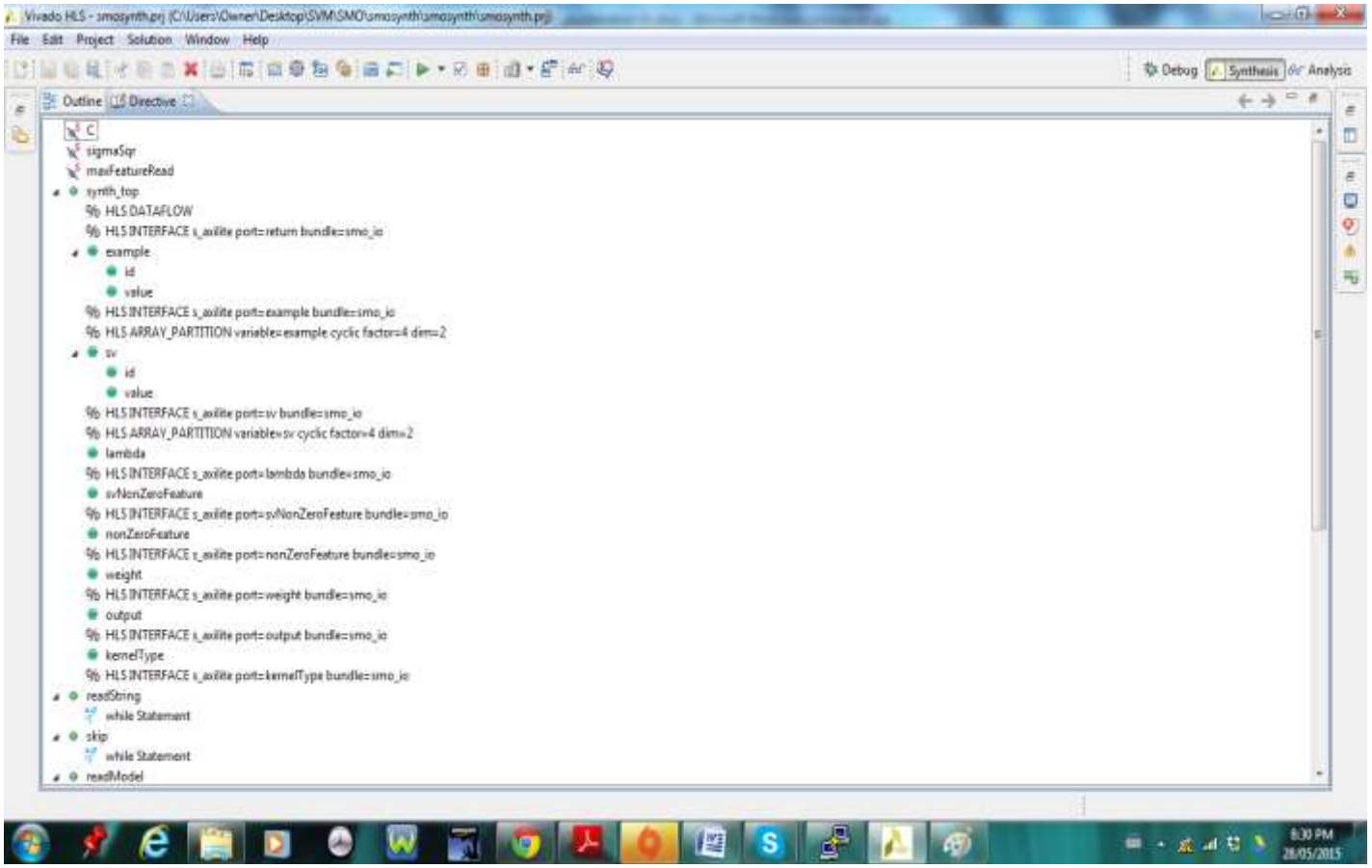


Fig. Snapshot of directives used at the top level

APPENDIX G

Vivado HLS Report Comparison

All Compared Solutions

solution1: xc7z020clg484-1
solution2: xc7z020clg484-1
solution3: xc7z020clg484-1

Performance Estimates

Timing (ns)

Clock		solution1	solution2	solution3
default	Target	10.00	10.00	10.00
	Estimated	8.23	8.23	9.40

Latency (clock cycles)

		solution1	solution2	solution3
Latency	min	1	1	1
	max	1691053	2280053	2256803
Interval	min	2	2	2
	max	1691054	2280054	2256804

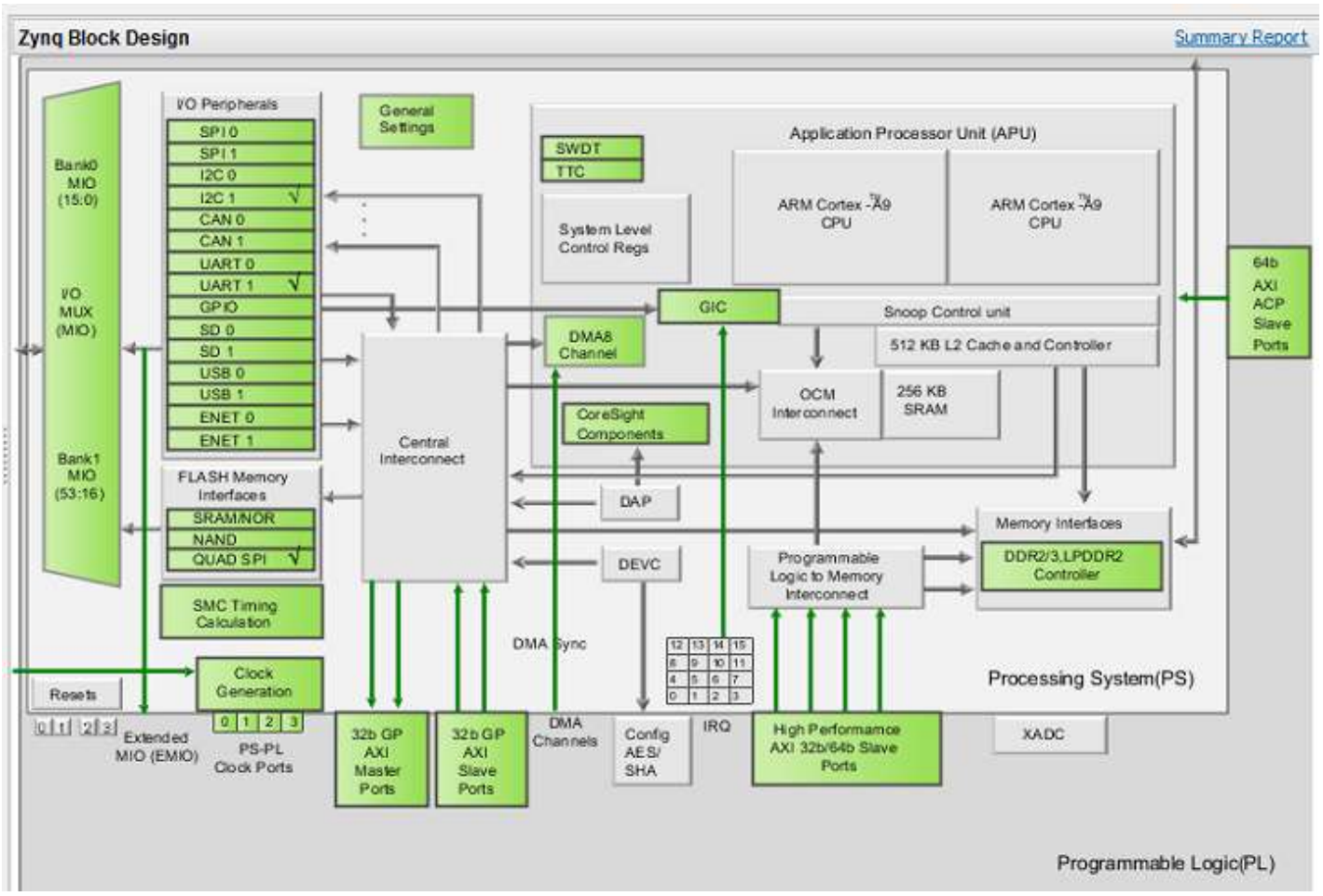
Utilization Estimates

	solution1	solution2	solution3
BRAM_18K	64	64	64
DSP48E	137	123	166
FF	10750	9635	14266
LUT	15618	13705	22741

Export the report(.html) using the Export Wizard

Fig. Comparison of timing and resource utilization across all three kernels

APPENDIX H


Zynq Block Design
Summary Report
Bank0 MIO (15:0)
I/O MUX (MIO)
Bank1 MIO (53:16)
I/O Peripherals
SPI 0
SPI 1
I2C 0
I2C 1
CAN 0
CAN 1
UART 0
UART 1
GPIO
SD 0
SD 1
USB 0
USB 1
ENET 0
ENET 1
FLASH Memory Interfaces
SRAM/NOR
NAND
QUAD SPI
SMC Timing Calculation
General Settings
SWDT
TTC
System Level Control Regs
Application Processor Unit (APU)
ARM Cortex -A9 CPU
ARM Cortex -A9 CPU
GIC
Snoop Control unit
512 KB L2 Cache and Controller
DMA8 Channel
OCM Interconnect
256 KB SRAM
CoreSight Components
DAP
DEVC
Central Interconnect
Memory Interfaces
DDR2/3,LPDDR2 Controller
Programmable Logic to Memory Interconnect
64b AXI ACP Slave Ports
DMA Sync
Resets
Clock Generation
Extended MIO (EMIO)
PS-PL Clock Ports
32b GP AXI Master Ports
32b GP AXI Slave Ports
DMA Channels
Config AES/ SHA
IRQ
High Performance AXI 32b/64b Slave Ports
XADC
Processing System(PS)
Programmable Logic(PL)

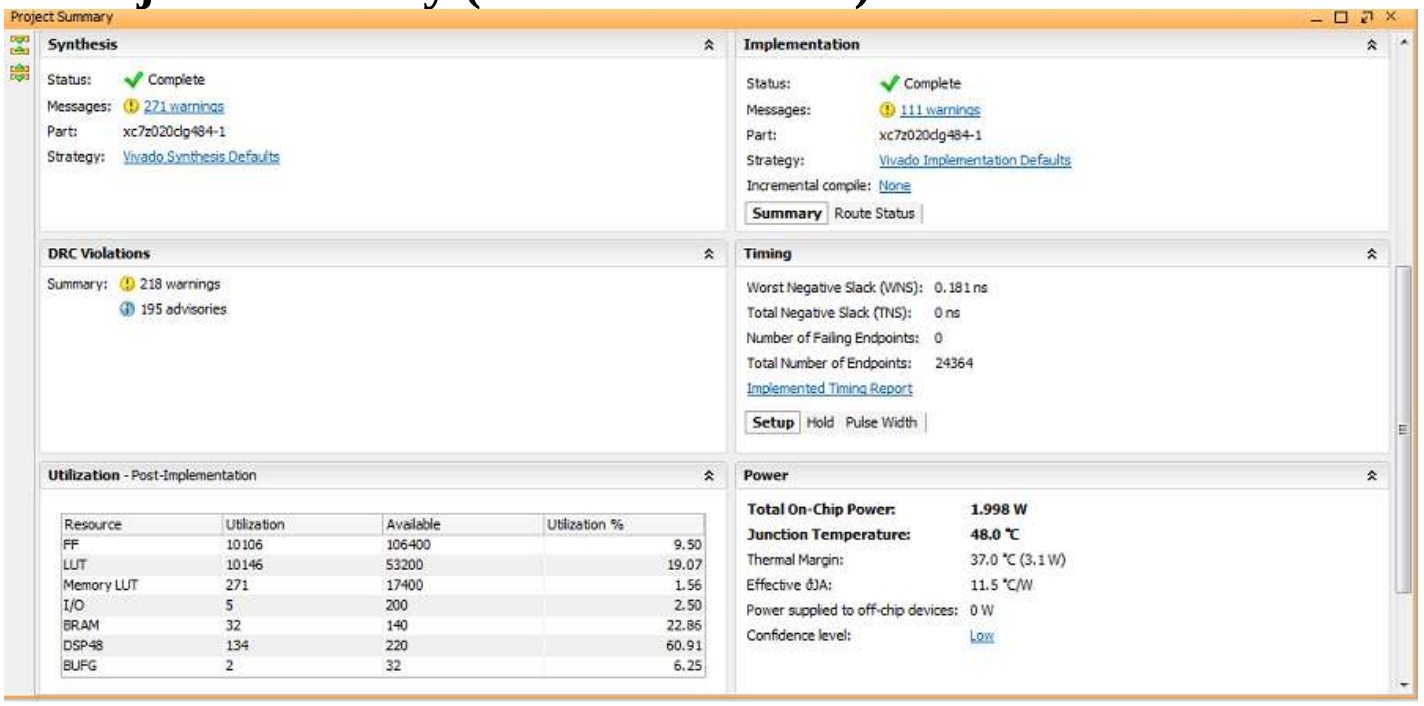
APPENDIX I

```
#MCLK
set_property PACKAGE_PIN AB2 [get_ports FCLK_CLK1]
set_property IOSTANDARD LVCMOS33 [get_ports FCLK_CLK1]

set_property PACKAGE_PIN AB4 [get_ports iic_1_scl_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_1_scl_io]

set_property PACKAGE_PIN AB5 [get_ports iic_1_sda_io]
set_property IOSTANDARD LVCMOS33 [get_ports iic_1_sda_io]

#SW0
set_property PACKAGE_PIN F22 [get_ports {gpio_io_i[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_io_i[0]}]

#LED0
set_property PACKAGE_PIN T22 [get_ports {gpio_io_o[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {gpio_io_o[0]}]

# To get rid of  OPMODEREG=0 issues
set_property SEVERITY {Warning} [get_drc_checks DSPS-11]
```

Fig. Constraints used in the XDC file

APPENDIX J

1. Project summary (for linear kernel):



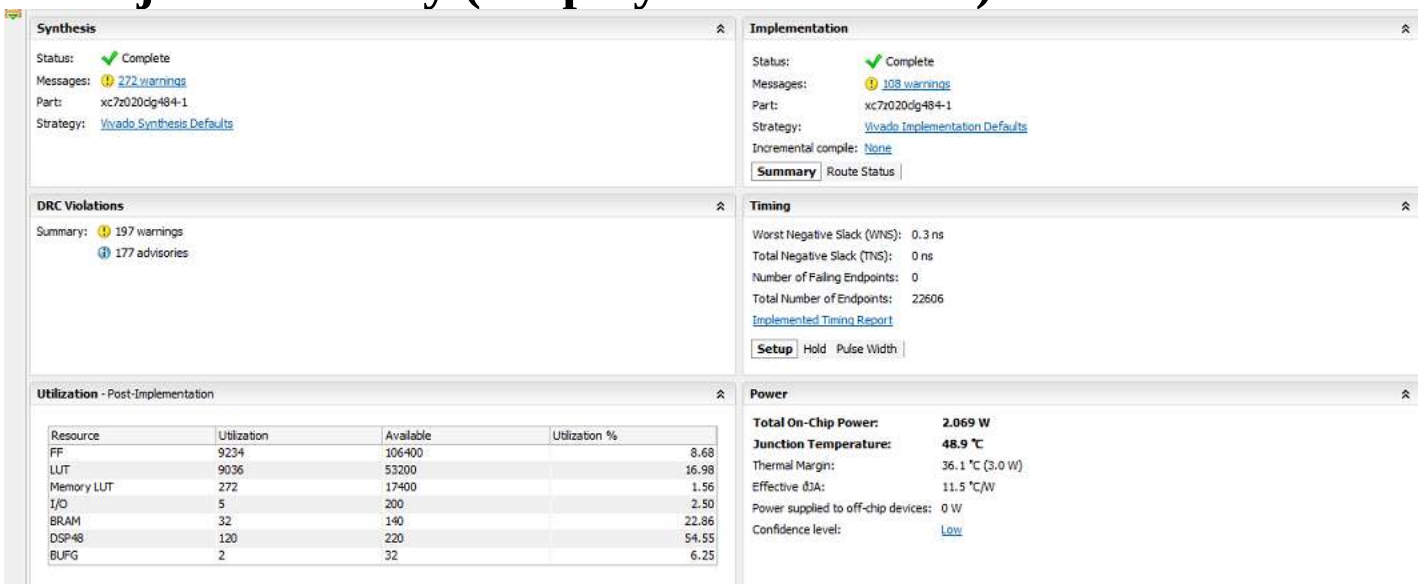
Resource	Utilization	Available	Utilization %
FF	10106	106400	9.50
LUT	10146	53200	19.07
Memory LUT	271	17400	1.56
I/O	5	200	2.50
BRAM	32	140	22.86
DSP48	134	220	60.91
BUFG	2	32	6.25

2. Bitstream generation (for linear kernel):

```
Running DRC as a precondition to command write_bitstream
INFO: [Drc 23-27] Running DRC with 2 threads
INFO: [Vivado 12-3199] DRC finished with 0 Errors, 217 Warnings, 195 Advisories
INFO: [Vivado 12-3200] Please refer to the DRC report (report_drc) for more information.
Loading data files...
Loading site data...
Loading route data...
Processing options...
Creating bitmap...
Creating bitstream...
Writing bitstream ./smo_linear_system_wrapper.bit...
INFO: [Vivado 12-1842] Bitgen Completed Successfully.
INFO: [Project 1-118] WebTalk data collection is enabled (User setting is ON. Install Setting is ON.).
INFO: [Common 17-186]
'C:/Users/Owner/Desktop/SVM/SMO/smosynth/smosynth/smo_linear/smo_linear.runs/impl_1/usage_statistics_we
btalk.xml' has been successfully sent to Xilinx on Sat May 23 12:44:36 2015. For additional details about this file,
please refer to the WebTalk help file at C:/Xilinx/Vivado/2015.1/doc/webtalk_introduction.html.
INFO: [Common 17-83] Releasing license: Implementation
write_bitstream: Time (s): cpu = 00:01:18 ; elapsed = 00:01:45 . Memory (MB): peak = 1753.883 ; gain = 365.531
INFO: [Common 17-206] Exiting Vivado at Sat May 23 12:44:36 2015...
```

APPENDIX J (Cont.)

3. Project summary (for polynomial kernel):



Synthesis

Status: Complete
Messages: 272 warnings
Part: xc7z020clg484-1
Strategy: Vivado Synthesis Defaults

Implementation

Status: Complete
Messages: 108 warnings
Part: xc7z020clg484-1
Strategy: Vivado Implementation Defaults
Incremental compile: None
Summary | Route Status

DRC Violations

Summary: 197 warnings
177 advisories

Timing

Worst Negative Slack (WNS): 0.3 ns
Total Negative Slack (TNS): 0 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 22606
Implemented Timing Report
Setup | Hold | Pulse Width

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	9234	106400	8.68
LUT	9036	53200	16.98
Memory LUT	272	17400	1.56
I/O	5	200	2.50
BRAM	32	140	22.86
DSP48	120	220	54.55
BUFG	2	32	6.25

Power

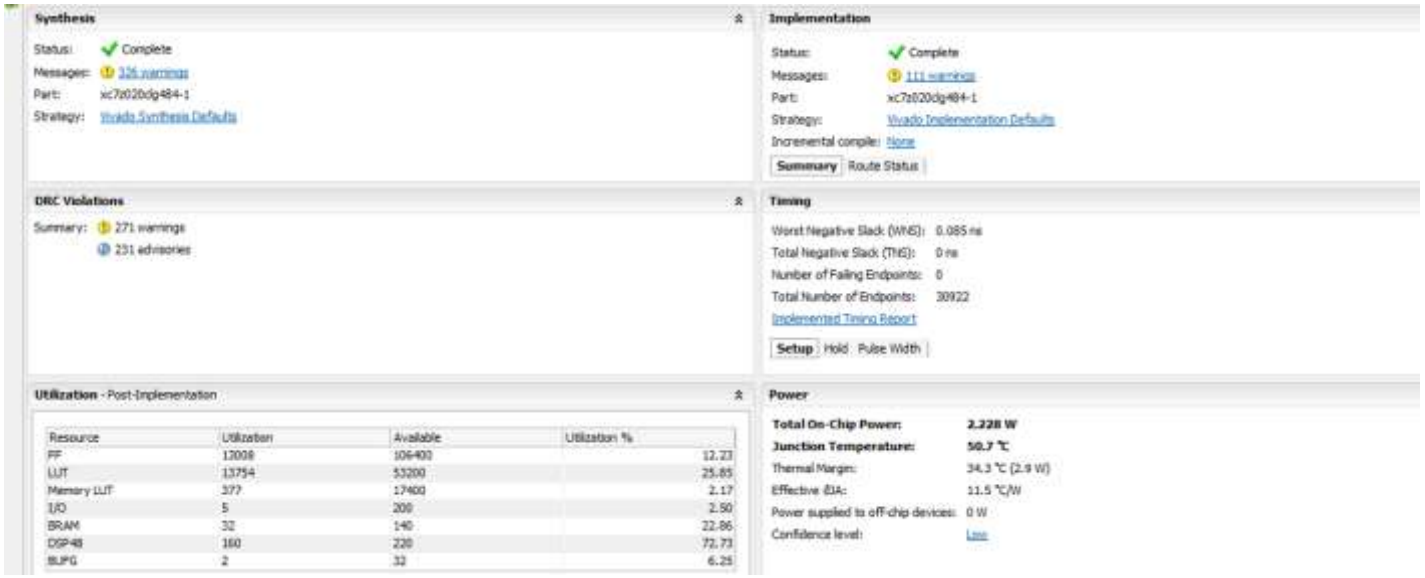
Total On-Chip Power: **2.069 W**
Junction Temperature: **48.9 °C**
Thermal Margin: 36.1 °C (3.0 W)
Effective θJA: 11.5 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low

4. Bitstream generation (for polynomial kernel):

```
Running DRC as a precondition to command write_bitstream
INFO: [Drc 23-27] Running DRC with 2 threads
INFO: [Vivado 12-3199] DRC finished with 0 Errors, 196 Warnings, 177 Advisories
INFO: [Vivado 12-3200] Please refer to the DRC report (report_drc) for more information.
Loading data files...
Loading site data...
Loading route data...
Processing options...
Creating bitmap...
Creating bitstream...
Writing bitstream ./smo_poly_system_wrapper.bit...
INFO: [Vivado 12-1842] Bitgen Completed Successfully.
INFO: [Project 1-118] WebTalk data collection is enabled (User setting is ON. Install Setting is ON.).
INFO: [Common 17-83] Releasing license: Implementation
write_bitstream: Time (s): cpu = 00:01:13 ; elapsed = 00:01:42 . Memory (MB): peak = 1721.598 ; gain = 340.672
INFO: [Common 17-206] Exiting Vivado at Sat May 23 19:07:00 2015...
```

APPENDIX J (Cont.)

5. Project summary (for radial basis function kernel):

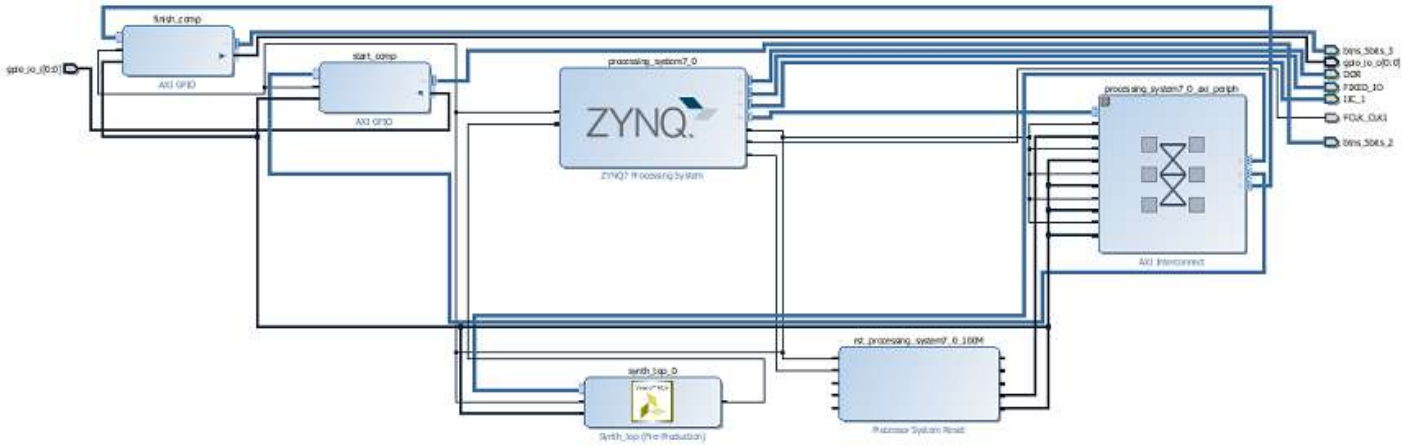


6. Bitstream generation (for radial basis function kernel):

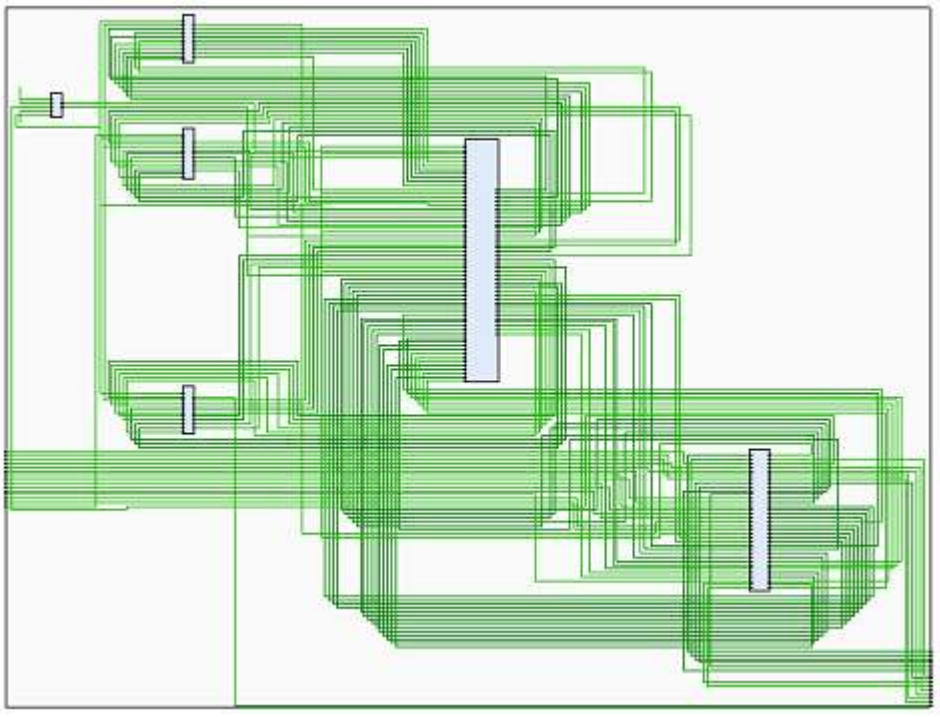
```
Running DRC as a precondition to command write_bitstream
INFO: [Drc 23-27] Running DRC with 2 threads
INFO: [Vivado 12-3199] DRC finished with 0 Errors, 270 Warnings, 231 Advisories
INFO: [Vivado 12-3200] Please refer to the DRC report (report_drc) for more information.
Loading data files...
Loading site data...
Loading route data...
Processing options...
Creating bitmap...
Creating bitstream...
Writing bitstream ./smo_rbf_system_wrapper.bit...
INFO: [Vivado 12-1842] Bitgen Completed Successfully.
INFO: [Project 1-118] WebTalk data collection is enabled (User setting is ON. Install Setting is ON.).
INFO: [Common 17-186]
'C:/Users/Owner/Desktop/SVM/SMO/smosynth/smosynth/smo_rbf/smo_rbf.runs/impl_1/usage_statistics_webtalk.
xml' has been successfully sent to Xilinx on Wed May 27 00:21:54 2015. For additional details about this file,
please refer to the WebTalk help file at C:/Xilinx/Vivado/2015.1/doc/webtalk_introduction.html.
INFO: [Common 17-83] Releasing license: Implementation
write_bitstream: Time (s): cpu = 00:01:24 ; elapsed = 00:01:46 . Memory (MB): peak = 1838.168 ; gain = 376.352
INFO: [Common 17-206] Exiting Vivado at Wed May 27 00:21:55 2015...
```

APPENDIX K

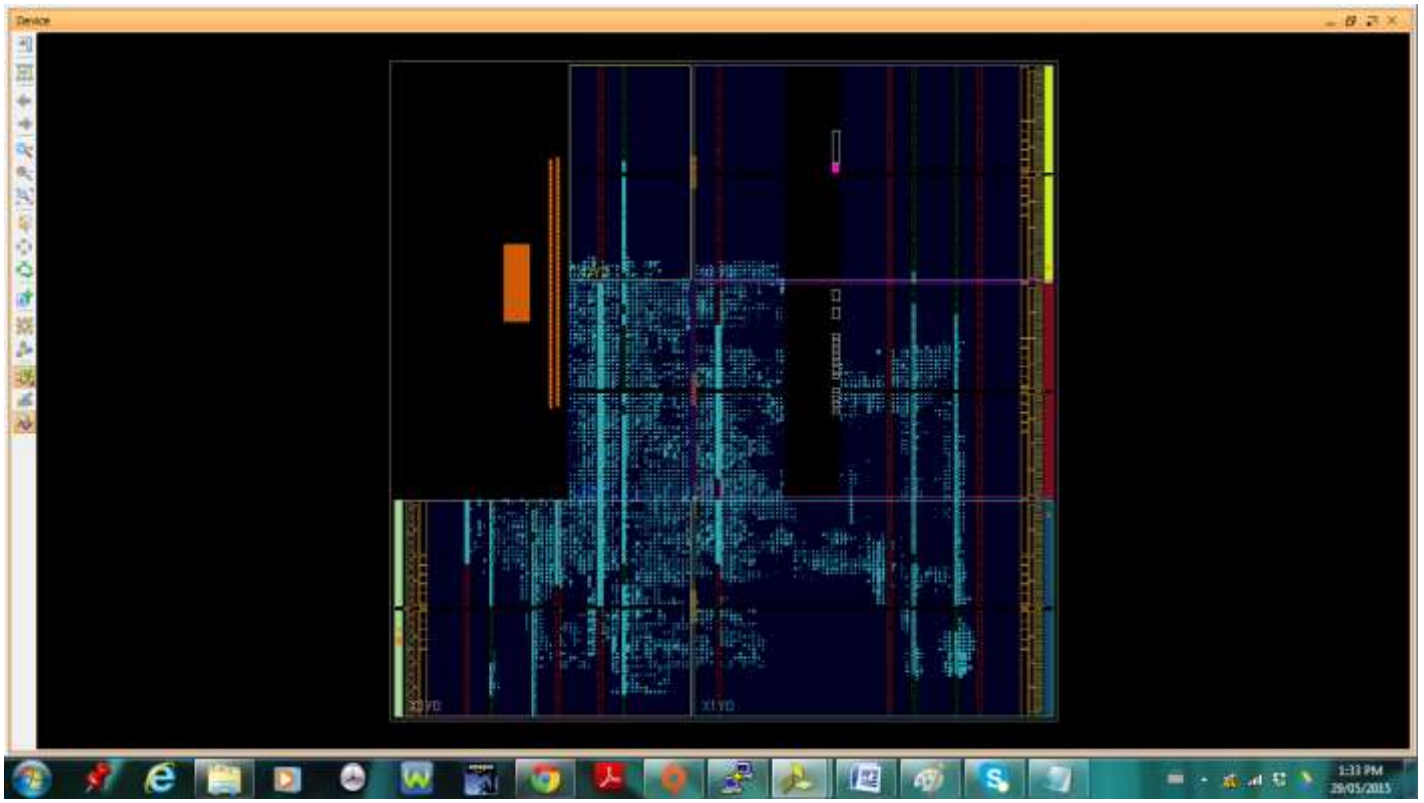
1. Block diagram (for linear kernel):



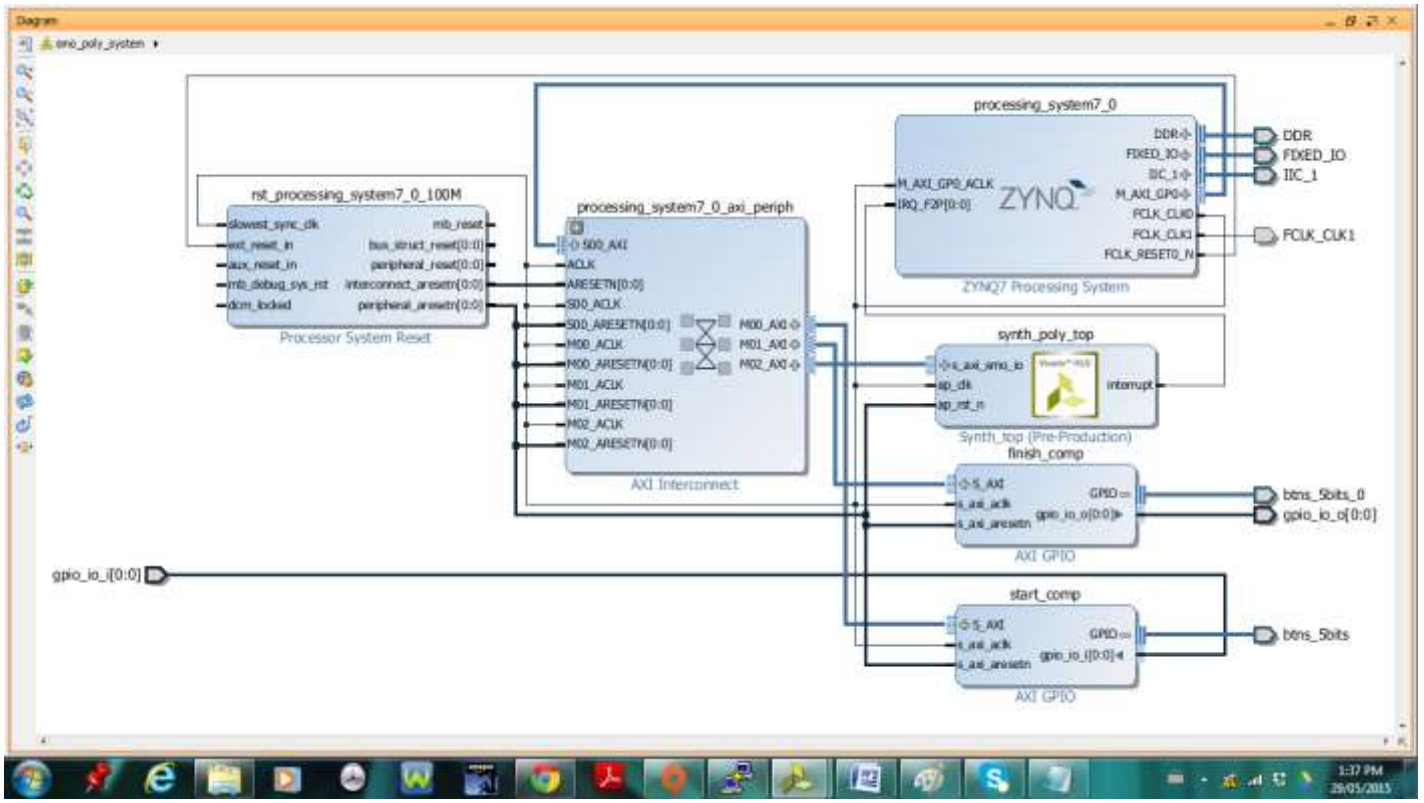
2. Synthesized design (for linear kernel):



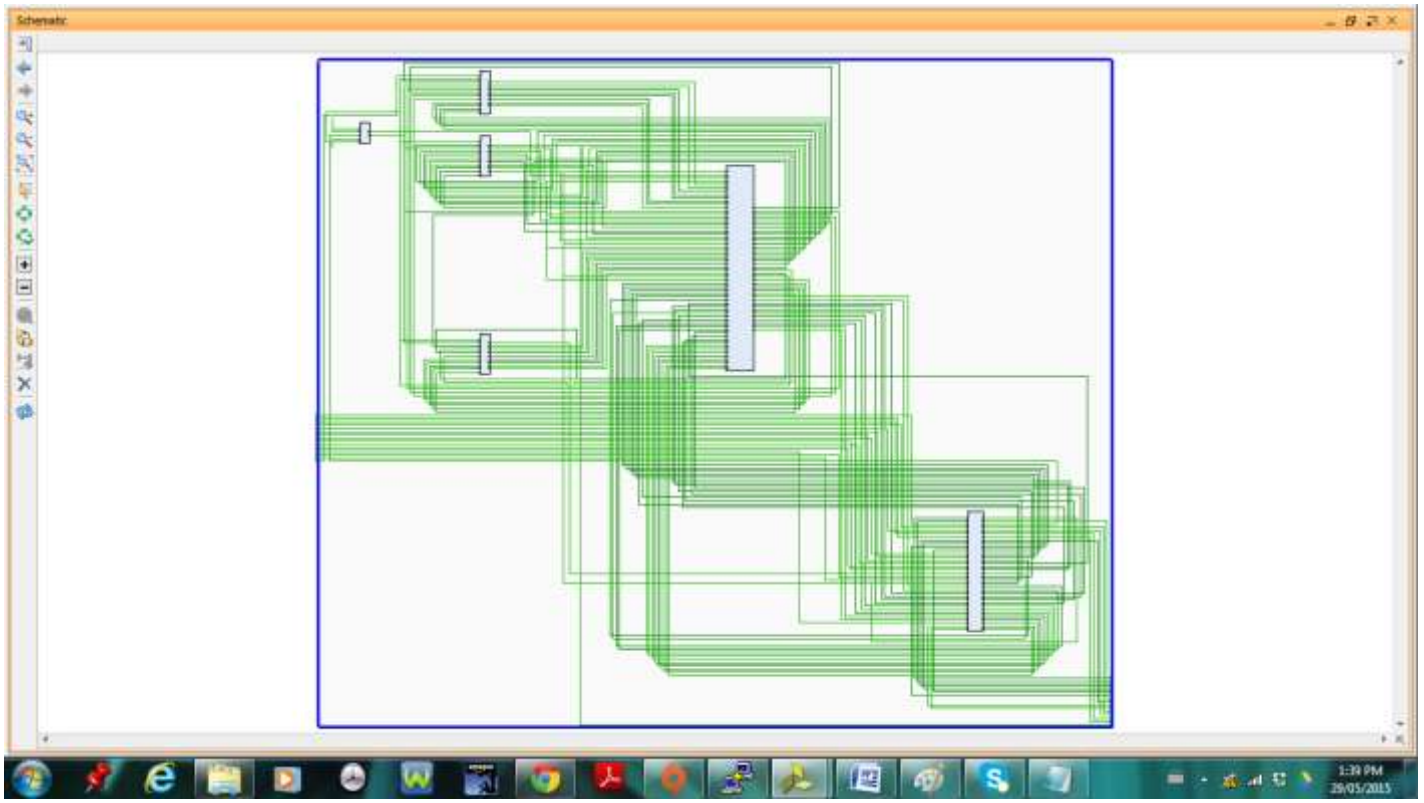
3. Implemented design (for linear kernel):



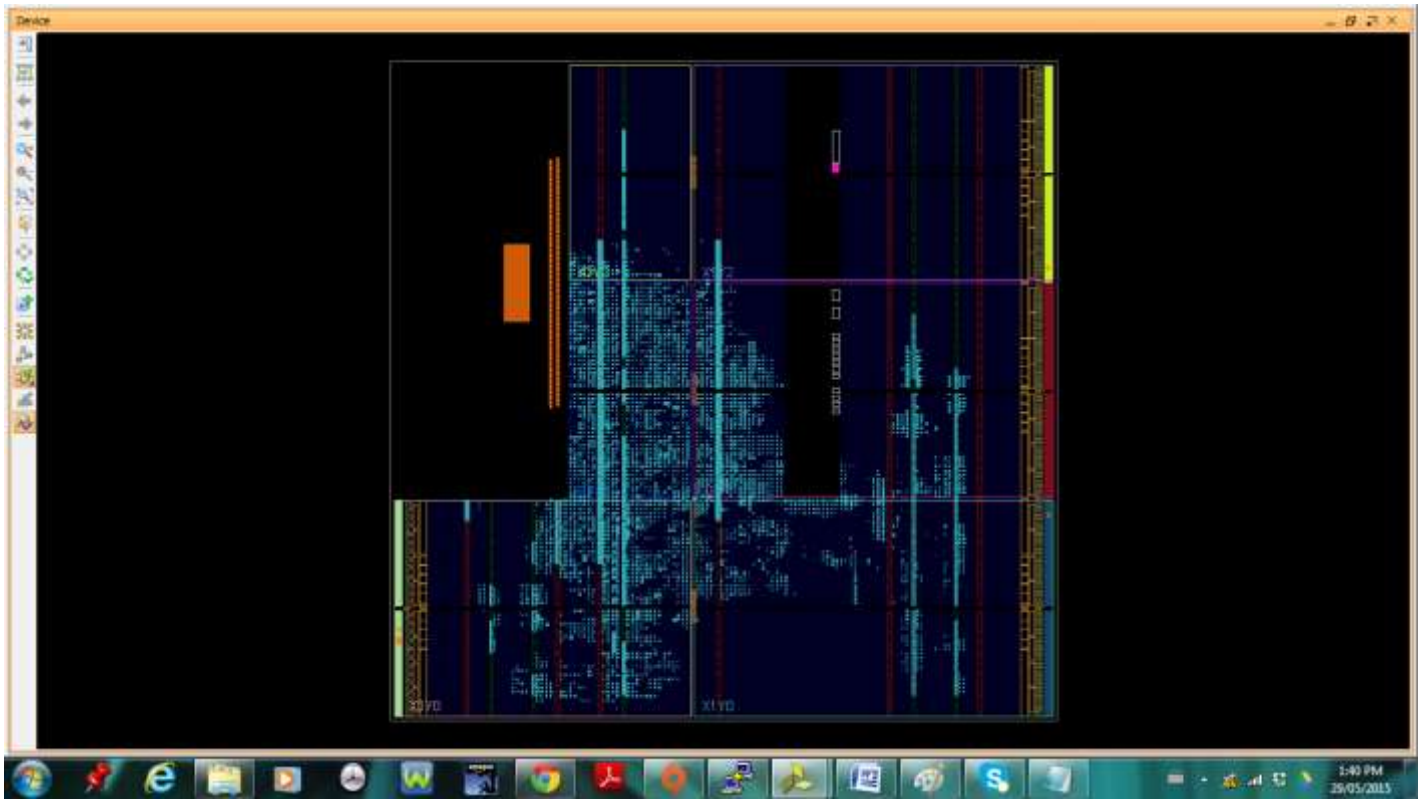
4. Block diagram (for polynomial kernel):



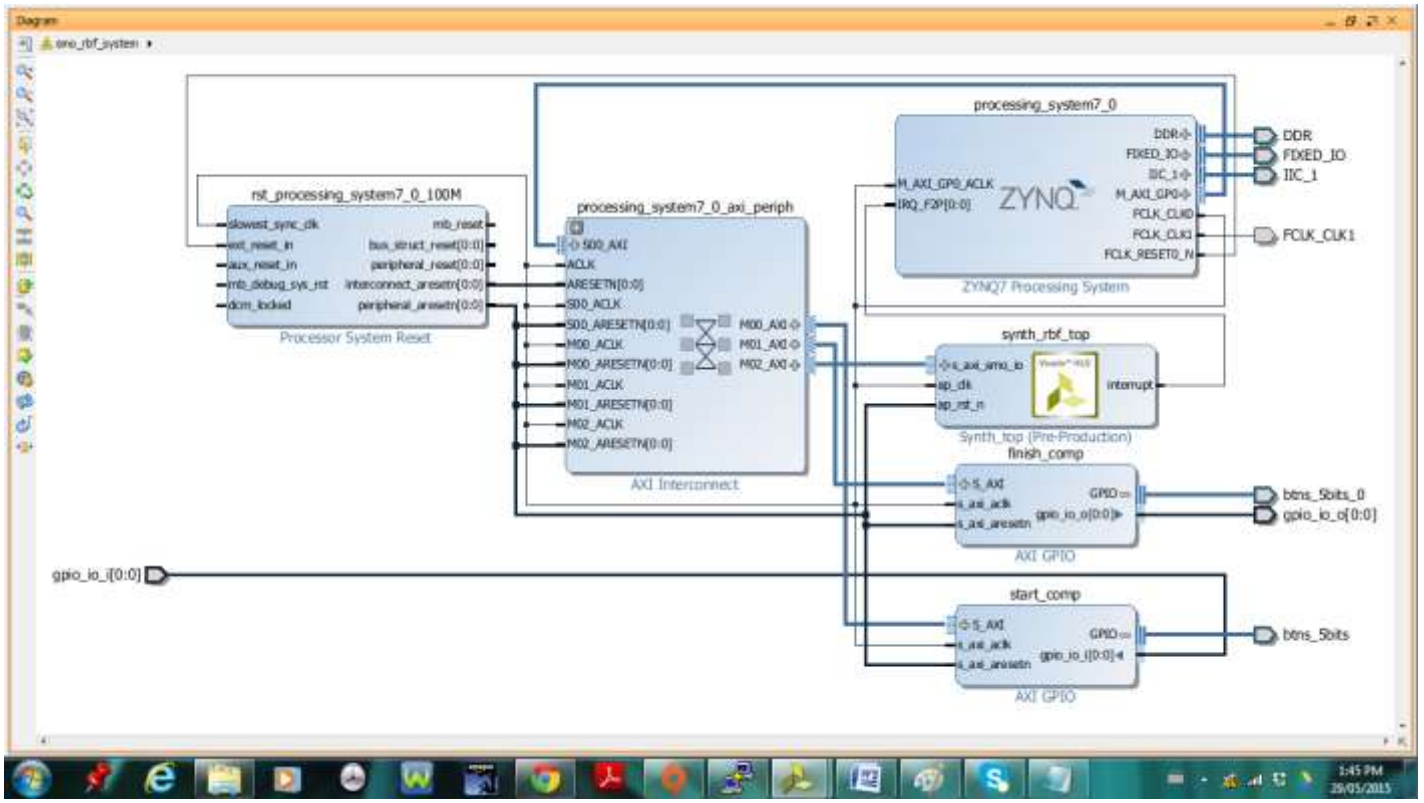
5. Synthesized design (for polynomial kernel):



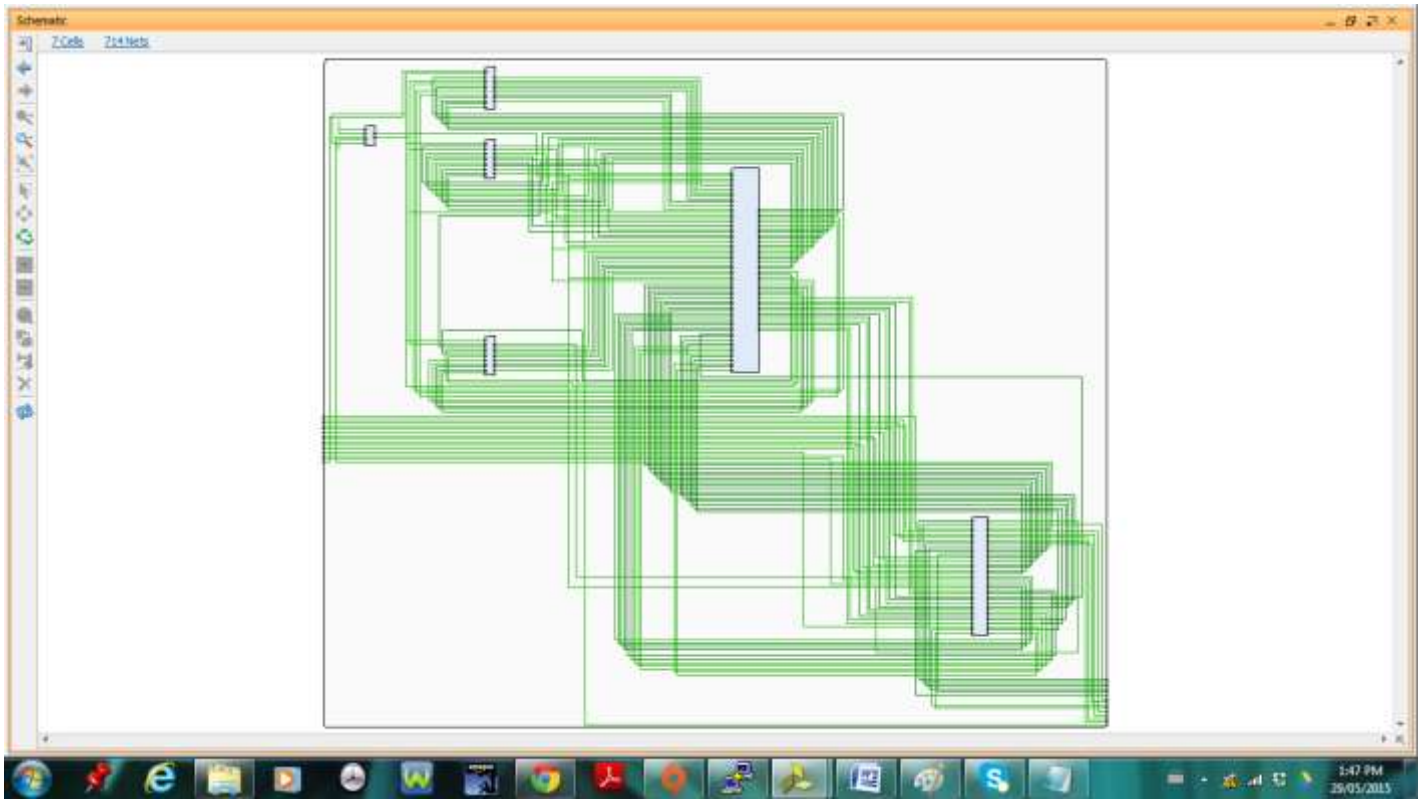
6. Implemented design (for polynomial kernel):



7. Block diagram (for radial basis function kernel):



8. Synthesized design (for radial basis function kernel):



9. Implemented design (for radial basis function kernel):