Project Report for Lab 3

Phil Tomson
ECE590
6/5/2005

Project:

VHDL implementation of a Support Vector Machine with a Gaussian kernel.

Backround:

Support Vector Machines (SVMs) are a machine learning method somewhat related to neural networks and perceptrons. SVMs are used to classify a set of inputs into categories. They are used for applications such as image and pattern recognition. Different types of kernels are used by SVMs to map higher dimensional spaces to two-dimensional spaces for binary categorization (or N-dimensional spaces for one-of-N categorization).

A Gaussian kernel is of the form:

$$\text{bias} + \sum_{i=0}^{NSV} \text{alpha}[i] * e^{-k*\gamma} \quad \text{(Eq.1)}$$

Where NSV is the number of support vectors, bias is a constant, alpha is a vector of values and k is a vector norm determined by:

$\langle x_i, x \rangle$ the vector norm (or difference) between the input vector (x_i) and the support vector (x) determined when the SVM was 'trained'.

NOTE: My implementation will be of an already-trained SVM where X and the alpha vector are already determined and provided. I have two trained SVMs and datasets to choose from (both from the University of Genova).

Problem Statement:

I plan to use an example SVM produced by a program called cSVM from the University of Genova. The example SVM is known as 'banana' (the name refers to the shape of the area of the graph). There is a dataset with 4900 examples to present to the SVM along with expected results for each example.
Several problems need to be solved:
0) A parser needs to be developed for the .aaa file description of the SVM which comes from the cSVM program (this part is already done).
1) The exponential part of the equation needs to be implemented in a lookup table. The smallest-sized lookup table needs to be determined for the given problem. Since it is a negative exponential we only need to worry about a range of input values from 0 to -N where all values less than -N will be considered to produce an output of 0. The value of N needs to be determined depending on the given problem. To determine how

small the lookup table needs to be we need to answer two questions:

1) What is the smallest range of 0 to -N that will give satisfactory answers (meaning that the SVM still produces the same answer from both the meta-model and the HDL model.

2) What level of granularity is needed in the lookup table; ie. How many entries from 0 to -N are needed in the lookup table in order for the HDL model to give satisfactory results (meaning results with will match our meta model).

(Some parts of #1 have already been done.)

Method of solution

- **Numerical representation/precision:** fixed point (integer,fractional); sizes to be determined experimentally.
- **Programmatic modeling language**: Ruby (http://www.ruby-lang.org) with some added modules for fixed point math.
- **HDL**: VHDL
- **VHDL Simulator:** GHDL
- **Synthesis tool**: Xilinx ISE 7.1 (no backup plan)

Goals of Solution:

- Functional equivalence of the Ruby and VHDL models will be verified using provided datasets. (see Verification section below)
- A testbench will be generated from the Ruby model.
- The VHDL model will be synthesized with Xilinx ISE. Timing and area of synthesized solution will be supplied.

Model Levels of solution:

1) System Meta model: to be written in Ruby. Model will implement the SVM with Gaussian kernel function. Exponential function initially uses built-in *exp()* function from the Ruby Math module. Model will be able to run example datasets through the SVM to determine accuracy of results.

2) Hybrid Meta model/HDL model: As minimal-sized lookup table is determined, the lookup table will initially be implemented in Ruby and will replace the built-in *exp()* function with lookup table. After this, the hybrid-meta model/HDL model can be developed where the lookup table is implemented in VHDL and one of the following options is used:

1) call the VHDL function from the Ruby meta-model: in this scenario, the Ruby model drives the interaction. Values are passed from the Ruby meta-model to the VHDL lookuptable and lookup results are passed back to the Ruby side for evaluation in the kernel formula.

2) use the Ruby meta-model as a foreign model callable from the VHDL side: In this scenario, the testbench drives the interaction. Values from the VHDL lookup table are passed to the Ruby meta-model and the results are passed back to the VHDL side.

3) Synthesizable HDL model: to be written in VHDL. Synthesizable with Xilinx ISE. (this will be implemented in Project 3)

Verification Plan:

Use the datasets for the SVM obtained from the University of Genova. Several hundred testcases exist for each of the two SVMs. One of the two SVMs will be selected (depending on which one is most amendable for hardware implementation) and the corresponding dataset will be run against it.

Since the SVMs in question classify a set of inputs into one of two categories (binary classification) we need to determine an acceptable level of mis-classifications for any given lookup table size and fixed point precision. For example: given a data set with 1000 input vectors it may be acceptable if the hybrid and HDL models mis-categorize 5% of classifications (as compared with the pure meta-model) if it means that the resulting synthesizable HDL design can fit into a smaller FPGA (or *any* FPGA targeted by our synthesis tool) and/or runs faster.

Project2 Results:

Meta Model: Since the .aaa file parser was already implemented, the first step was to implement the meta-model in Ruby. There are at least 2 levels of meta-model:

1) The purely mathematical implementation: meaning that the SVM formula was implemented in terms of mathematical operators available in Ruby's Math library using only numbers of type Float (Ruby's native floating point type) [this model had already been implemented.] (this will be referred to as the *level 1 meta-model*)
2) The Fixed Point implementation: In this model the SVM formula was implemented in terms of Fixed Point numbers for all values (based on a Ruby class I developed called *FixedPt*) and used a lookup table of fixed point values instead of Ruby's built-in *exp()* function. [this model had been implemented, but there were some issues with the *FixedPt* class which needed to be resolved]. This meta-model moves closer to the hardware implementation. (this will be referred to as the *level 2 meta-model*)

The level 1 meta model (the purely mathematical implementation) was able to correctly categorize all but 15 of the 4900 testcases presented to it, this represents the baseline which we would like to achieve as we move our meta-models closer to a hardware implementation. [NOTE: an SVM will sometimes mis-categorize an input. In this case 15 mis-categorizations out of 4900 was considered acceptable for this particular SVM (an accuracy of 99.69%)].

After moving to the level 2 (fixed point) meta-model there were several parameters which needed to be tuned in order to try to avoid increasing the number of miscategorizations:

1) The number of entries in the lookup table for the *exp(-k*gamma)* function (here after referred to simply as the *exp() lookup table*) needed to be tuned.
2) The accuracy (number of fractional bits) of each of the entries in this lookup table needed to be tuned.
3) The optimal range of input values (0 to -N) for the function *exp(-k*gamma)* needed to be determined. Since the input to the *exp()* function in our SVM formula is always

negative we only need to worry about a range of inputs from 0 to -N where numbers less than N are considered to return 0 from our *exp()* lookup table. See Figure below.

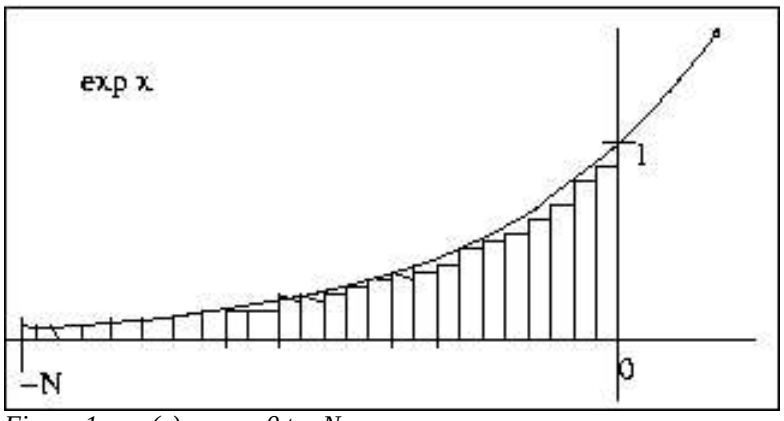

Figure 1: exp(x) range: 0 to -N

Several experiments were run on the level 2 (fixed-point, with exp lookup table) meta-model to determine the optimal values of these three parameters (where *optimal* implies the smallest hardware resources required) with the following results:

- Number of entries in the lookup-table for the exp() function: 128 (7 bit address)
- minimum N = -12.25
- Fixed Point representation: 16 bits with with binary point also at bit 16.

The level 2 meta-model of the SVM with these parameter settings actually resulted in 14 mis-categorizations out of 4900 test cases which is actually one less than the results from the level 1 meta-model. This improvement is really only a fluke result: some test cases result in a very close answer (close to 0.0), it's most likely that the use of fixed point arithmetic pushed one of those very close results over to the opposite side of 0.0 (changed sign from positive to negative or vice-versa depending on the result) and this happened to match the sign of the expected result.

The Hybrid Meta-Model:

After the level-2 meta-model was complete and tuned, I wrote some Ruby code to generate a VHDL implementation of the *exp()* lookup table. The VHDL implementation is follows Xilinx's template for inferring ROMs (actually a BRAM is loaded with the data). The idea of the hybrid meta-model was that the *exp()* lookup table would actually be running in the VHDL simulator while the level 2 Ruby model would make calls to the VHDL lookup table to obtain the value for the *exp* part of the formula. This was accomplished by using pipes to communicate between the Ruby model and the VHDL model. Each time the Ruby model needs a result from the exp lookup table, it sends an address (via the pipe) to the running VHDL model of the lookup table and the VHDL model sends back the result. On the VHDL side this was accomplished by writing a foreign-language extension (in C) for GHDL to handle the communication (see the files *communicate.vhd*, *external_comm_pkg.vhd* and *external_func.c* in the aaa2vhdl/TESTS subdirectory).

The results for the hybrid meta-model match those from the level 2 meta-model (14 miscategorized out of 4900 testcases), however it does take much longer to run (not a surprise).

Comment on the hybrid meta-model: Getting the communication synchronized between the Ruby model and the VHDL model took a significant amount of the effort involved in this segment of the project. The value of the hybrid meta-model vs the level 2 meta-model is debatable as they should be expected to give exactly the same answer (as they did) due to the use of FixedPt values in the level 2 meta model. However, the communication mechanism developed for the hybrid meta-model should be valuable in testing the next step of this project (Project 3) as it can be used for testing the resulting synthesizable VHDL code. It can also be used to migrate parts of the algorithm over to VHDL from Ruby, while maintaining the ability to simulate the entire algorithm before having a complete VHDL implementation. For example, it is clear that some sort of state machine will be needed in the VHDL implementation in order to synchronize fetching of data from the various lookup tables and generation of a final output. The model could be simulated without this state machine existing in VHDL using only the lookup tables in VHDL with the rest of the control maintained on the Ruby-side of the model. The state machine could then be implemented in VHDL with the knowledge that the model worked prior to it's introduction.

Also, it should be possible to automate the process of generating the boiler-plate C and VHDL code needed for the communication which should speed future development which uses this methodology.

Project #2 Deliverables:

- **Ruby meta-model(s)**
  The Ruby meta-models exists in the file aaa2vhdl/csvm.rb. The level 1 (golden) meta-model exists in the method *gaussian_kernel_gold.* The level 2 meta-model is implemented in the method *gaussian_kernel_fixedint*.
- **Hybrid Ruby/VHDL meta-model**
  The hybrid meta-model is implemented in the method *gaussian_kernel_hybrid* also in the csvm.rb file.
- **Synthesis results for the VHDL lookup table portion** (area and speed)
  In order to estimate how much area will be used by the final design VHDL lookup tables were generated for each of the lookup tables which will be in the final design:
  - exp() table (128 entries X 16bits/entry) in the file aaa2vhdl/TESTS/explut.vhd
  - alpha lookup table (241 entries X 16bits/entry) in the file aaa2vhdl/TESTS/alpha_lut.vhd
  - support vector lookup table (483 entries X 16bits/entry) in the file aaa2vhdl/TESTS/sv_lut.vhd

- **Verification results using provided datasets:** 14 mis-categorizations out of 4900 testcases

Project #3 Deliverables:

- **Synthesizable VHDL model for SVM:** The design consists of the following files (in the aaa2vhdl/TESTS subdirectory):
  - **csvm.vhd** : toplevel VHDL file.
  - **csvm_fsm.vhd** : state machine for controlling lookup table access and calculations.
  - **sv_lut.vhd** : support vector lookup table. The entries in this table represent the support vectors which result from training the machine.
  - **explut.vhd** : lookup table for exp() function.
  - **alphalut.vhd**: lookup table for alpha array.
- **Synthesis results (area and speed/throughput):**

Area:

```
Device utilization summary:
---------------------------

Selected Device : Spartan 3 (3s200ft256-5)

 Number of Slices:                      142  out of   1920     7%
 Number of Slice Flip Flops:            112  out of   3840     2%
 Number of 4 input LUTs:                191  out of   3840     4%
 Number of bonded IOBs:                  24  out of    173    13%
 Number of BRAMs:                         3  out of     12    25%
 Number of MULT18X18s:                    2  out of     12    16%
 Number of GCLKs:                         1  out of      8    12%
```

Comment on device utilization:

Device utilization is actually somewhat lower than expected. The Spartan 3 family has lots of nice features for implementing mathematically intensive designs like this one: multipliers, BRAMs. Much of this design is in the lookup tables. It's probable that this design would fit in the next smaller device. It would be interesting to see if the same design could fit in this device using case statements for the lookup tables instead of using the internal BRAMs. It's possible that the design would have higher throughput in that case since it would eliminate extra clock cycles currently needed to wait for data from the BRAMs (which are synchronous). I chose to use the Spartan3 200 device because I have a Spartan3 StarterKit from Xilinx which includes a prototyping board with this device.

Timing Report (pre-place & route):

```
Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
clk                                | BUFGP                  | 112   |
-----------------------------------+------------------------+-------+
Timing Summary:
---------------
Speed Grade: -5
   Minimum period: 8.272ns (Maximum Frequency: 120.893MHz)
   Minimum input arrival time before clock: 11.787ns (Tsetup)
```

```
   Maximum output required time after clock: 6.441ns (Tco)
```

Post Place & Route Timing:

Setup/Hold to clock clk (only worst-case signals shown)

Source	Setup to clk (edge)	Hold to clk (edge)	Internal Clock(s)	Clock Phase
xin<8>	14.571(R)	-6.670(R)	clk_BUFGP	0.000

Clock clk to Pad

Destination	clk (edge) to PAD	Internal Clock(s)	Clock Phase	
ready	8.634(R)	clk_BUFGP	0.000	(worst Tco)
result_sign	6.403(R)	clk_BUFGP	0.000	
valid_answer	6.403(R)	clk_BUFGP	0.000	
xin_addr<0>	8.429(R)	clk_BUFGP	0.000	

Clock to Setup on destination clock clk

Source Clock	Src:Rise Dest:Rise	Src:Fall Dest:Rise	Src:Rise Dest:Fall	Src:Fall Dest:Fall	
clk	16.231				(worst Tsu)

Analysis:

Max clock frequency: 1/(Tsu + Tco): 1/(16.231ns + 8.634ns) = 40.22 Mhz

Throughput:

For this particular SVM, each input to be classified consists of two numbers therefore to compute the vector norm between the input vector and each support vector would require at least two clock cycles given our architecture . However, we also need to access the support vector LUT to obtain each component of the support vector (2 numbers in this case) and since our LUT is implemented in a BRAM which is synchronous we need calculate the address of the LUT in one clock cycle and get the result in the next one. This means that for the inner loop of the algorithm (see equation 1 – the inner loop in this case refers to the vector norm operation) we need 4 clock cycles for this SVM. Two more clock cycles are then needed to take the output of vector norm calculation and use it to address the exp() LUT and get an output. This comes to a total of 6 clocks for the inner loop. Our SVM has 242 support vectors, so 242 X 6 = 1452 clocks to get an answer. Given that our clock is about 40MHz that comes to 36.3 microseconds per classification or 27,548 classifications/second.

One way to speed things up would be to use case statements to represent the lookup tables. Currently I use an array and a Xilinx recommended template that causes a BRAM to be inferred. BRAMS are synchronous and therefore require that the address be supplied with enough setup time prior to the clock. The data becomes available some

time after the clock arrives. Since I generate addresses synchronously this means that an extra clock is required to access data and this has been added to the controlling state machine (in csvm_fsm.vhd). Not using the internal BRAMs will take much more space in the device, but it would reduce the number of clocks for each inner loop calculation from 6 to 3 which would theoretically double our throughput to about 55000 classifications/second (however, post place & route timing would probably reduce the clock frequency given that more of the device will be utilized, so realistically throughput would not double).

- **Testbench and Verification results**

The testbench is contained in the file tb_csvm.vhd. I do not have results for an exhaustive run of all testcases. I have gone through 4 testcases and compared the intermediate results (by looking at wavforms in the simulator) with the results the meta-model reported and the SVM seemed to be doing what it should be for those particular testcases. However, when I began to run more testcases with the testbench I saw more failures than I would have expected at this point (given that the first 4 testcases seemed to be working correctly on close examination). At this point, I am not sure if this is a problem with the VHDL implementation of the SVM or the testbench.

General comments on the project:

- The level 2 meta-model was useful in determining sizes for representation of data. It was fairly easy to modify it so that it would report the largest integer part of a fixed point number seen during the run and this was used to determine how many bits would actually be needed in the VHDL implementation. Since the goal is to programatically generate all of the VHDL files from a given .aaa file, it would be nice if this determination were more automatic. For example: as testcases are run through the meta-model each of the fixed point variables in the meta-model could be monitored to determine the smallest number of integer and fractional bits that would be needed when the VHDL code is generated.
- One of the main problems encountered in the VHDL version was keeping track of the binary point after various operations. The proposed fixed_pkg from IEEE would certainly help with this if it worked with ISE.
- I would like to come up with a more automated way of comparing various stages of the computation between the Meta-model and the VHDL implementation. This might be a better use of the hybrid meta-model which was used in the second phase of the project: Signals from the state machine could be fed from the VHDL design back to the Hybrid meta-model and at given states the values of certain VHDL signals could be compared with the values of their counterparts in the hybrid meta-model. Differences could be reported. This would be a great debugging aid. As it was I did a lot of these comparisons by hand which was quite tedious. It's one thing to know that your meta model results do not match your HDL implementation, but it would be much more informative to know exactly where in the design the results begin to diverge. If the communication between the two could be setup automatically, say by providing a list of signals that you want to watch in both models (it should be possible) this would be another use for meta-models: debugging aids. Maybe there's a product idea in there somewhere ;-)

Directory Structure:

/ECE590/phil_tomson/proj3/

report/ - project report
aaa2vhdl/ - meta-model files
aaa2vhdl/TESTS – tests, vhdl files
aaa2vhdl/TESTS/proj3_6_4 – ISE report files

Running the models:

Requirements:

- GHDL 0.18 (available from: http://ghdl.free.fr)
- Linux (for running GHDL)
- Ruby 1.8.2 (available from http://www.ruby-lang.org)
- GCC (for compiling C extensions for Ruby and GHDL)

Procedure:

- CD to the aaa2vhdl/TESTS subdirectory
- run the build script to build the GHDL foreign models.
- Run 'make' to build Ruby extension
- To run the level-1 meta model: *ruby run_banana.rb*
  (this will take 5 to 10 minutes depending on how fast your PC is)
- To run the hybrid meta model: *ruby run_banana_hybrid.rb*
  (this will take a while 30 to 60 minutes depending on how fast your PC is)