Imperial College London

ADVANCED DSD COURSEWORK

Project – Acceleration of an SVM classifier

Individual Report

by

Lei Kuang

(lk16)

12th of March 2019

CONTENTS

1. INTRODUCTION

This report focuses on design, optimization, and implementation of an embedded system running on Zynq, for the purpose of accelerating a computationally intensive application. The specific application that will be demonstrated for acceleration in this report is a support vector machine (SMV) classifier which has already been trained on Matlab. All necessary parameters of the trained SVM model are provided.

The SVM classifier is required to be synthesized into a customized hardware block (IP core) for acceleration by using the high-level synthesis (HLS) tool provided by Xilinx. Then an embedded system based on Zynq will be implemented to transfer the test data (a vector) from the arm core (processing unit) to the customized IP core that implemented by the traditional FPGA programmable logic (PL). Finally, the classification result will be read back to the arm core for further processing.

With keeping a hardware concerned programming style, the synthesis result always gives a minimum and maximum latency (or interval) that are equal. Which means all loops are convergent no routine of the program is returned earlier. As a result of this, the minimum and maximum value of the latency (or interval) are considered to be equal if no additional notation is applied.

In this report, the SVM classifier has been designed and optimized into an IP core successfully with a tuned latency of 84 clock cycles. Then a Zynq system which interfaces the IP core through AXI-Lite has been implemented, achieving a computation time of 11.98 ms, which is 222 times faster than the software-based version where the provided C code is directly executed on a bare metal Zynq system.

To further eliminate the communication overhead, the IP core is then modified to be interfaced through AXI-Stream where the test vectors are transferred to the DDR3 memory without waiting for the SVM classifier to finish its current work and then acts like a FIFO which is processed by a direct memory access (DMA) controller. In this case, the accelerator of SVM classifier acts as a consumer, once it finishes its current computation, it will request a new test vector from the FIFO. In addition, as the arm core is a 32-bit RSIC CPU, its internal data bus supports a data width of 32 bits, in order to fully utilize its bandwidth, the elements of the test vector are parallelized two by two into 32 bits before being transferred. By means of employing these two approaches, the total computation time is further reduced to only 1.77 ms, which is approximate 1500 times faster and the associated communication overhead is dramatically reduced from 10.3ms (AXI-Lite) to 0.07ms (AXI-Stream) for processing 2000 test vectors.

1.1. Support Vector Machine Classifier

The following SVM classifier with a kernel of hyperbolic tangent is required to be accelerated:

$$f(x) = sgn\left(\sum_{i=1}^{N_{SVs}} y_i a_i K(\vec{x}, \overrightarrow{SV_i}) + b\right) = sgn\left(\sum_{i=1}^{N_{SVs}} \alpha_i K(\vec{x}, \overrightarrow{SV_i}) + b\right), N_{SVs} = 1050$$

Where:

$\vec{x}$:	Input vector for classification
$\overrightarrow{SV_i}$:	The support vectors returned from the training stage
N_{SVs}:	Number of support vectors
y_i:	The labels of the support vectors
a_i:	The weights of the support vectors
α_i:	The product of the label and the weight of the support vectors
b:	Bias of the classifier
$K(\vec{x}, \vec{y})$:	Kernel function, which is defined as $\tanh(2 * \vec{x} \cdot \vec{y})$
$sgn(u)$:	The sign of the input number, more specifically, returns 0 if $u >= 0$

Both input vector and support vector have a dimension of 16, where each element is a type of 16-bit signed fix-point variable with 12 fractional bits, having an exact value range of [-7.999755859375, 7.999755859375].

The specific SVM model processed in this report has 1050 support vectors. The weights and labels are not provided, instead, their products alphas are available.

Figure 1 presents the computation flow of the SVM classifier. The classifier firstly computes the dot product of the input vector and a support vector, the result is then multiplied by 2 and input to the hyperbolic tangent function. Next, the product of the hyperbolic tangent result and alpha will be accumulated until the same operation has been processed with all remaining support vectors and their corresponding alphas. Finally, the SVM bias is added to the accumulator and the classification result depends on the sign of the final sum.

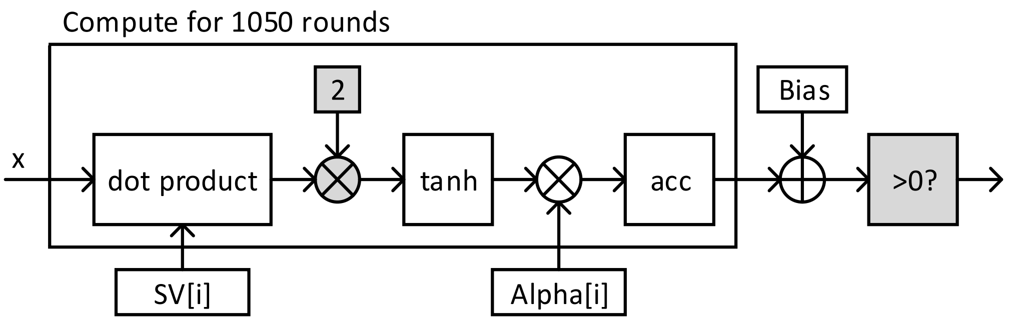

Figure 1 Block diagram of SVM classifier computation

From the view of the top level, there are two computation blocks (in grey) can be optimized through wiring:

1) For a digital system, "multiply by 2" is equivalent to "shift to left by 1 bit" which is almost free on hardware (FPGA), only costing wiring. Thus, a '0' can be directly appended to be the least significant bit of the dot product result.

2) As the entire system will be optimized through defining all variables as type of fixed-point (arbitrary precision) whose most significant indicates the sign by defination, thus the classification result can be directly determined by the most significant bit of the final sum.

1.2. Overview of Optimization Strategy

The original implementation of the SVM classifier is based on software approach, where variables are defined as C-type int or double, which are 32-bit and 64-bit respectively. However, as the specification defined, both the input vector and support vector have only 16-bit elements, it is meaningless to define them as C-type double which brings redundant bits and costs more resources when doing the synthesis on HLS. For example, if elements of $\vec{x}$ and $\overrightarrow{SV}$ are defined as double, a 64-bit multiplier will be synthesized on hardware to compute the multiplication which is not necessary and costs more resources compared to a 16-bit multiplier that it is supposed to be.

In order to accelerate the SVM classifier, arbitrary-precision variable will be used to save hardware resources and analysis will be given in the synthesis part, such that for the dot product block, it utilizes a 16-bit multiplier to compute the product, resulting in fewer than a tenth the resources required for the C-type double implementation. At the same time, the latency is also reduced which gives the indication that using arbitrary-precision variable can not only optimize the resource utilization, both also boost the performance.

When doing the division, the divider synthesized by HLS takes a quite long time to compute and the precision of the quotient cannot be tuned. In order to maximize the throughput of the SVM classifier, a customized binary divider is proposed to be designed and used for computing the hyperbolic tangent where the number of the iteration loops controls the precision of the quotient.

At the Zynq system level, both AXI-Lite and AXI-Stream interfaces have been evaluated. In order to reduce the communication overhead between the arm core and the accelerator as much as possible, AXI-Stream is proposed to be the best choice where one-time transmission can be performed on all test vectors that are required for computation on the accelerator. In addition, with employing data flow and direct memory access techniques, the performance of the accelerator is further optimized.

2. TESTBENCH FOR TUNING

In order to tune the designed accelerator with tolerable computation error compared to the provided C-based SVM classifier, the difference of the computation results between the software and designed hardware for specific variables are evaluated. As presented in Figure 2, variables that come from different computation stages of SVM classification are probed. HLS only synthesizes the variables that are associated with the output which means that inserting some debug variables will have no impact on the synthesis result.

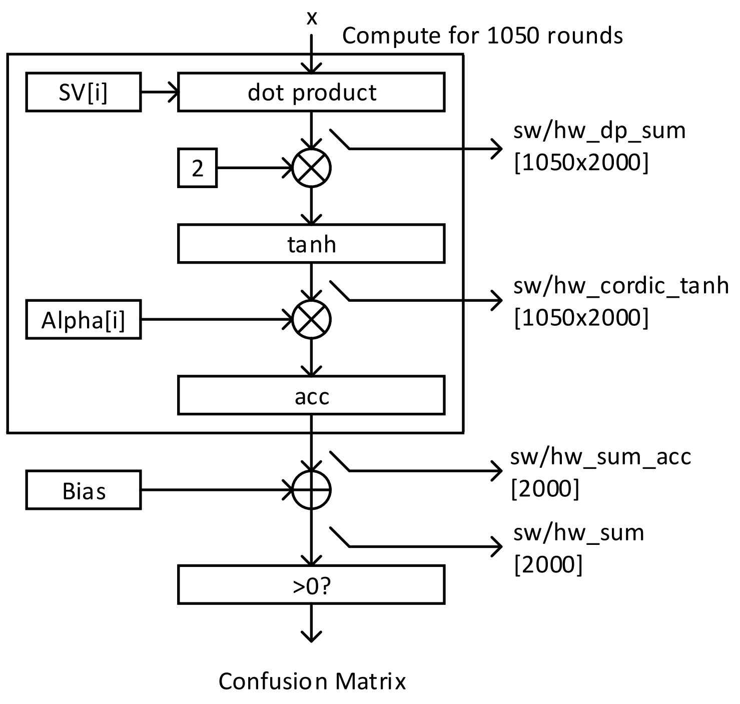

Figure 2 Testbench for system tuning

The testbench always outputs a log message as shown in the following block, which presents the difference of the computation results between the software and tuned hardware for all test vectors. The summary presents the maximum positive error (Max) and minimum negative error (Min) that have been processed which is convenient for the designer to tune the system.

```
…
1998 SW:sum :   286.8360864817646529   HW:sum :  284.4348144531250000   E:  -2.401272
1999 SW:sum :   150.1784354111215691   HW:sum :  148.8398437500000000   E:  -1.338592
Summary:
Max:  3.827697
Min: -4.291227
Error Cnt:170

Classification Error Rate: 0.085000

Confusion Matrix:
Golden  Res
1826    98
72      4
```

All tuned HLS core presented in the following part will always have a classification error rate of 0.085 and the confusion matrix is always the same as the one presented in the log block.

3. DESIGN AND OPTIMIZATION OF SUBFUNCTIONS

This part demonstrates the design of dot product function and hypobaric tangent function, associated with the trade-offs that are made to optimize the resource utilization and performance which aims to achieve the minimum latency without losing much precision.

3.1. Dot Product

Dot product computes the multiplication of two elements that come from two equal-length sequences respectively and outputs the accumulated summation of all resulted products.

The dot product function can be realized by the following snippet of code in HLS:

```
dp_sum = 0.0;
for (unsigned int i=0; i<VEC_D; i++) {      // #define VEC_D 16
    dp_pro = ap_vec[i] * ap_SVs[i];         // Multiplication
    dp_sum = dp_sum + dp_pro;               // Accumulation of the sum
}
```

3.1.1. OPTIMIZATION THROUGH USING ARBITRARY-PRECISION VARIABLE

Originally, both the input vector and support vector are defined as C-type double which consists of 1-bit sign, 11-bit exponent and 52-bit fraction. It is impossible for FPGA hardware to compute the multiplication of two double-type variables in one clock cycle as the critical path of the synthesized hardware is quite long compared to the clock period of 10ns. In addition, it is a waste of FPGA resources to use a 64-bit double multiplier for computing the multiplication of two variables that are actually 16-bit.

By means of using arbitrary-precision variables where variables are defined as fixed-point two's complement number with necessary length of bits, the drawback of C-type double can be eliminated. Figure 1 shows the analysis of the precision of internal variables when processing the computation of the dot product function. Symbol "fn.m" is used to represent a variable that is defined as fixed-point ('f') number consisting of n-bit integer and m-bit fraction. As the elements of the input vector and the support vector are both 16-bit with 4 bits representing the sign and the integer, the product will become 31-bit because both the width of integer without sign (3-bit) and the width of fraction (12-bit) are doubled during multiplication. Then all products will be summed which leads to a number that 16 times (critical case) larger than the original product, thus the final result of the dot product is 35-bit with 11 bits representing the sign and integer whereas the length of fractional part remains the same after the addition.

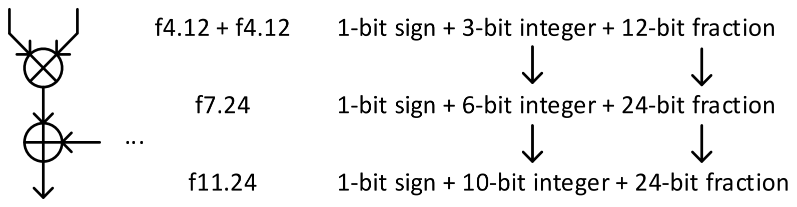
...
f4.12 + f4.12
1-bit sign + 3-bit integer + 12-bit fraction
f7.24
1-bit sign + 6-bit integer + 24-bit fraction
f11.24
1-bit sign + 10-bit integer + 24-bit fraction

Figure 3 Computation Precision of Dot Product Subfunction

The synthesis result (Table 1) shows that the resources required for dot product to be implemented using arbitrary-precision variable (without losing precision during computation) are less than tenth resources utilized by the original double-type implementation. The precision of the internal variables can be further tuned in order to save resources and speed up the computation when evaluating the entire performance.

3.1.2. OPTIMIZATION OF PERFORMANCE

In order to achieve the minimum latency and maximize the throughput, the loop of executing dot product function is fully unrolled where 16 DSP multipliers are utilized to perform the involved multiplication of the input vector and the support vector. The block diagram of the unrolled loop is shown in Figure 4, where the multiplicatiosn of two vector elements are computed in parallel, then an adder network is used to compute the summation which outputs the final dot product result.

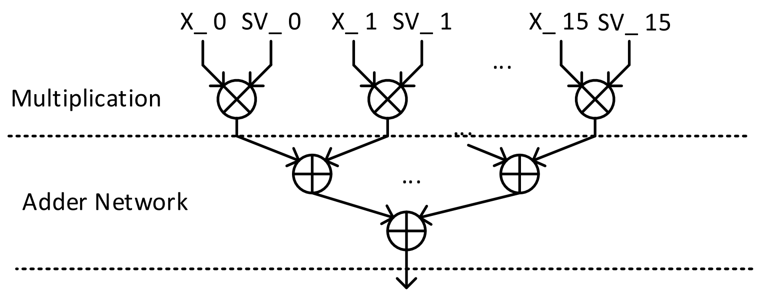

Figure 4 Block diagram of the fully unrolled dot product

In addition, as the input vector and support vector are both defined as an arrays, the HLS will automatically synthesize them as ROM or RAM where the element can only be accessed one by one (or two if dual port). This significantly limits the performance even the loop is unrolled as consumers (16 DSP multipliers) do not have enough input data to process. As a result of this, directive ARRAY_PARTITION must be applied to make all elements from the array become individual so all of them can be accessed at the same time so that the computation can be processed in parallel.

Note that complete ARRAY_PARTITION is only applied on support vector when synthesizing the dot product function because, at the current level, the input support vector only has a dimension of 16. When it comes to the top level, this directive becomes valid as it is impossible to completely partition a very large array due to the limitation of FPGA resources, whereas cyclic ARRAY_PARTITION will be used instead.

3.1.3. COMPARISION OF DIFFERENT IMPLEMENTATIONS

Table 1 concludes the HLS synthesis results based on different design strategies.

Implementation	Directive	Latency	Interval	BRAM_18K	DSP48E	FF	LUT
Double	-	209	209	0	14	1042	1824
Arbitrary Precision	-	49	49	0	1	80	13
Arbitrary Precision	Pipeline	19	19	0	1	90	113
Arbitrary Precision	Partition + Unroll	2	2	0	16	567	617

Table 1 HLS synthesis results of the dot product

It is noticeable that by means of using arbitrary precision, both the resource utilization and performance can be improved significantly which leads to resources that even fewer than tenth resources that required by the C-type double implementation.

Pipeline technique is an approach which can boost the performance without increasing the resource utilization much. However, in order to ultimately minimize the latency of the dot product function and maximize its throughput, loop unrolling with partitioning the input vector array becomes the best choice whereas the drawback is that the resource utilization increases quite a lot.

3.2. Hyperbolic Tangent

In this part, the design and optimization of hyperbolic tangent function that based on CORDIC algorithm will be demonstrated.

3.2.1. CORDIC ALGORITHM

CORDIC algorithm is a hardware-efficient approach that is capable of computing complicated functions such as trigonometric function on hardware through pre-computing some loop-up tables and iterating operations that only involves shift and add/sub, which significantly boosts the hardware performance.

The generic CORIC equation for hyperbolic rotation can be found in [1], as shown below, the derivation of these function will not be discussed as the focus is on its application for computing the hyperbolic tangent.

For the CORDIC algorithm that runs in rotation mode:

$$x_{i+1} = x_i + y_i \cdot d_i \cdot 2^{-i}$$
$$y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}$$
$$z_{i+1} = z_i - d_i \cdot \tanh^{-1}(2^{-i})$$

Where $d_i = -1\ if\ z_i < 0, otherwise, d_i = +1$

Through assigning the input argument value as the initial value of z, the hyperbolic sine and cosine can be computed through iteration where $[\cdot 2^{-i}]$ is equivalent to "shifting to the right" on hardware and $[\tanh^{-1}(2^{-i})]$ are pre-computed as look-up table. The principle of the iteration is to perform a binary search over the input argument and the precision of the result depends on the number of iterations and the precision of the pre-computed values. After rotating the input vector (x_0, y_0) for n times, the vector after rotation becomes:

$$x_n = A_n[x_0 coshz_0 + y_0 sinhz_0]$$
$$y_n = A_n[y_0 coshz_0 + x_0 sinhz_0]$$
$$z_n = 0$$
$$A_n = \prod_n \sqrt{1 - 2^{-2i}} \approx 0.8$$

Substitute the initial vector as $(x_0, y_0) = (1, 0)$ into the equation above, hyperbolic sine and cosine can be computed:

$$x_n = A_n[x_0 coshz_0 + y_0 sinhz_0] = A_n coshz_0$$
$$y_n = A_n[y_0 coshz_0 + x_0 sinhz_0] = A_n sinhz_0$$

However, the final result is not exactly $coshz_0$ and $sinhz_0$, a constant factor that has a value of $1/A_n$ is required for compensation where the initial vector becomes $(x_0, y_0) = (1/A_n, 0) = (1.25, 0)$

3.2.2. OPTIMIZATION OF HYPERBOLIC TANGENT

As the CORDIC algorithm can only compute the hyperbolic sine and cosine directly, the hyperbolic tangent can only be obtained through the division of sine and cosine, as defined by Equation 1:

$$tanh(\theta) = \frac{sinh(\theta)}{cosh(\theta)}$$

Equation 1 Hyperbolic Tangent

Just like the trigonometric function which can only accept an input angle between 0 and 90 degrees [1]. For hyperbolic function, it has the same limitation as for the trigonometric function where the value of the input argument is limited to an approximate range of (0, 1). This limitation can lead to a large computation error when the input argument is out of the range.

However, as the sum of arguments equation defined (Equation 2), the computation of hyperbolic tangent for an argument with large value can be realized through separating the input argument as two individual parts – integer and fraction. For the integer part, pre-computed loop-up tables can be used to store the hyperbolic tangent results so that the CORDIC algorithm only needs computing the fractional part. This makes the CORDIC algorithm capable of computing an input argument with arbitrary value whereas the drawback is that an additional operation of division is required which can take dozens of clock cycles to compute, leading to a significant increase of latency for hyperbolic tangent function.

$$tanh(\theta) = tanh(\theta_{int} + \theta_{fra}) = \frac{tanh(\theta_{int}) + tanh(\theta_{fra})}{1 + tanh(\theta_{int})tanh(\theta_{fra})}$$

Equation 2 Sum of arguments for hyperbolic tangent

In order to eliminate the additional division required when optimizing the input argument range, Equation 3 shows how to compute the hyperbolic tangent through hyperbolic sine and cosine with the optimization that still works. It is noticeable that the computation of the hyperbolic tangent through this proposed approach only involves one division operation, however, nothing comes for free, this approach leads to four additional multiplication operations. This is a typical scenario where the resources are sacrificed in order to achieve better performance.

$$tanh(\theta) = tanh(\theta_{int} + \theta_{fra}) = \frac{sinh(\theta_{int} + \theta_{fra})}{cosh(\theta_{int} + \theta_{fra})} = \frac{sinh(\theta_{int})cosh(\theta_{fra}) + cosh(\theta_{int})sinh(\theta_{fra})}{cosh(\theta_{int})cosh(\theta_{fra}) + sinh(\theta_{int})sinh(\theta_{fra})}$$

Equation 3 Sum of arguments for hyperbolic sine and cosine

In addition, as the tangent value can go up to 0.99933 when the input number approaches a value of 4, the implantation of hyperbolic tangent can be further optimized through direct multiplexing out value 1 when the input argument is larger than 4.

3.2.3. OPTIMIZATION OF DIVISION

HLS will automatically synthesize a hardware divider in order to compute the division ('/') involved in the code. However, in order to tune the entire system, especially, tuning the precision of the division result, the proposed approach is to implement a customized binary divider which is considered capable of reducing the latency of the hyperbolic tangent function by means of limiting the precision of the division result.

A basic binary divider can be implemented through subtraction and shift, just like the add-and-shift multiplier but with a opposite process. Figure 5 shows how the binary division is processed through subtract and shift.

The computation of the first bit of the quotient starts from conditioning whether the dividend is larger than the divider. If the divider is larger, the dividend will be shifted to the left which is equivalently being multiplied by two. Then, the shifted dividend will be conditioned again, this time, it is larger than the divider, thus subtraction is performed between the dividend and the divider, and the difference will be shifted to the left again. Following the same process as described, the quotient bit can be computed one by one through iteration, where the computation result approaches the real result as the number of iterations (steps) increases. Thus, the number of iterations can be used to control the precision of the division. Fewer the iteration loops, lower latency it can achieve, however, larger the error becomes.

	0 1 1 1 0	**Step**	**Condition**	**Quotient Bit**	
1001	**1 0 0 0**	[1]	1000 < 1001	0	0
	0 0 0 0				
	1 0 0 0 0	[2]	10000 > 1001	1	0.5
	1 0 0 1				
	0 **1 1 1** 0	[3]	1110 > 1001	1	0.75
	1 0 0 1				
	0 1 0 1 0	[4]	1010 > 1001	1	0.875
	1 0 0 1				
	0 0 0 1 0	[5]	10 < 1001	0	0.875
	… … … … …		…	…	…

Figure 5 Binary Davidson Example: 8/9=0.888…

When computing the hyperbolic tangent, step 1 of the division process can be optimized. As the output value of the hyperbolic tangent function is always less than 1, which means, with the same input argument as tangent, the corresponding result of hyperbolic sine is always smaller than hyperbolic cosine. Base on this discovery, the computation of hyperbolic tangent can just start from computing first bit of the fraction part instead of the integer part, with the dividend hyperbolic sine shifting to the left at initial. However, as both variables that hold the result from hyperbolic sine and cosine, are defined as a fixed-point number with necessary bits representing their maximum value, further shifting to the left will lead to the overflow of the variable. The solution that overcomes the overflow issue is to shift the divider to right instead of shifting the dividend to left before starting to compute the quotient.

The following snippet of HLS code realizes the customized binary divider, where variable sinh and cosh are the dividend and divider respectively, resulting in a quotient tanh_abs:

```
cosh = cosh >> 1;                      // Shift the divider to right
CORDIC_DIVISION_LOOP:
for(unsigned int j=0; j<7; j++){
    #pragma HLS UNROLL
    if(sinh>cosh){
        tanh_abs[ORDIC_N*2-1-j] = 1;   // ap_ufixed<CORDIC_N*2, 0> tanh_abs;
        sinh = (sinh - cosh) << 1;
    }
    else{
        tanh_abs[ORDIC_N*2-1-j] = 0;
        sinh = sinh << 1;
    }
}
```

In HLS, the iteration loop of the division is fully unrolled in order to achieve the best performance.

For a particular precision of 7-bit (after tuned), the computation of the quotient only takes 3 clock cycles which is a seventh the clock cycles (21) that the HLS synthesized divider requires.

Table 2 shows the HLS synthesis results where the customized 7-bit divider has excellent latency and resource utilization compared to the HLS synthesized divider through sacrificing the division precision which is tolerable after tuning.

Implementation	Directive	Latency	Interval	BRAM_18K	DSP48E	FF	LUT
HLS Divider	-	20	20	0	0	236	237
Customized Divide	-	8	8			51	113
Customized Divide	Unroll	2	2	0	0	51	269

Table 2 HLS synthesis results of the divider

3.2.4. REALIZATION OF HYPERBOLIC TANGENT

Figure 6 presents the block diagram of the proposed architecture that includes all optimizations.

At the very beginning, the most significant bit of the input argument theta is stored which represents its sign. Then, according to the sign, the remaining bits of the input argument will be directly multiplexed out if the sign if positive, otherwise, they will be flipped and increased by one which is the process of inverting a two's complement number, resulting in absolute value. Once the absolute value of the input argument is ready, its integer and fractional parts can be directly obtained through wiring as it is defined as a type of fix-point variable where bits on the left side of the point represent the integer and the right side of the point are fraction part.

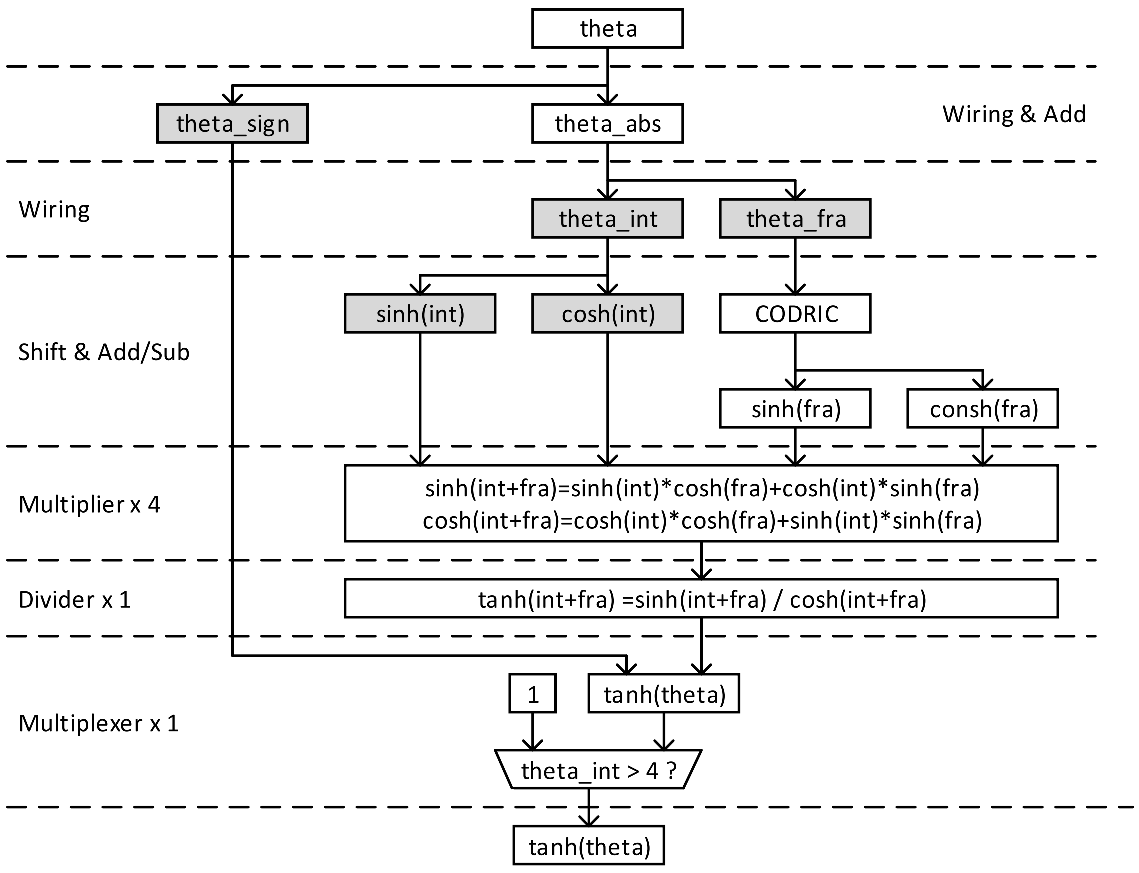

Figure 6 Block Diagram of Implementation of Hyperbolic Tangent

After obtaining the integer part and the fractional part of the absolute value of the input argument, the next step is to compute the hyperbolic sine and cosine for the integer and fraction respectively. For the integer part, the result can be obtained directly through accessing the pre-computed loop-up table whereas the fractional part will be processed by the CORDIC block where the iteration loop is fully unrolled in order to achieve the best performance.

When the CORDIC loop finishes the iterations, sinh(theta_fra) and cosh(theta_fra) are ready for the division which results in tanh(theta_fra). However, as the trade-off that has been made, in order to eliminate the additional division that brought by optimizing the acceptable range of the input argument, the final hyperbolic tangent value will be computed after computing the sum of arguments for hyperbolic sine and cosine functions, which instead, requiring four additional multiplication operations.

Finally, a multiplexer determines if the output is 1 or the computation result of hyperbolic tangent from the CORDIC block, by conditioning whether the integer part of the input argument is larger than 4. The reason why the C-like branching is not executed at the very beginning of the process entrance which can decrease the latency is that, as the entire system will finally be pipelined in order to maximize the throughput where different branch conditions can bring data and control hazards to the system which is quite hard to debug. Even through HLS tool is considered that will optimize the hazards automatically, however, the designer is supposed to fully understand this challenge when writing the code, resulting in a better performance estimation during hardware synthesis. In other words, the maximum and minimum latency (or interval) are supposed to be equal if the HLS code is written following a hardware-concerned style.

3.2.5. COMPARISON OF DIFFERENT IMPLEMTATIONS

The implementation based on C-type double utilizes the math library to compute the hyperbolic tangent function which results in a latency range from 2 to 75. Different minimum and maximum latency indicate that there is a branch condition at the entrance of the function which is expected to directly return 1 if the input argument is larger than a specific constant which has been mentioned in the previous part.

Through unrolling both iteration loops of the CORDIC block and the customized divider, a maximum (=minimum) latency of 5 clock cycles can be achieved which is remarkable compared to the reference implementation using the math library and the C-type double variable.

Implementation	Directive	Latency	Interval	BRAM_18K	DSP48E	FF	LUT
Double	-	2~75	2~75	11	56	7981	10239
Arbitrary Precision	-	29	29	0	0	333	961
Arbitrary Precision	Unroll (CORDIC)	23	23	0	0	338	1038
Arbitrary Precision	Unroll + 7-bit Divider	5	5	0	0	154	1099

Table 3 HLS synthesis results of the hyperbolic tangent

Note that all implementations that have been demonstrated have been carefully tuned with classification results that are exactly the same as the reference implementation.

4. IMPLEMENTATION AND OPTIMIZATION OF THE TOP LEVEL

The top level of the SVM classifier executes the iteration loop where the same computation process is performed with one input vector and different support vectors. For original iteration loop that is not unrolled, there are 1050 rounds of computation which is equal to the number of support vectors.

The performance can be boosted through unrolling the iteration loops where each loop will have its own blocks for computation. In other words, the same hardware for iterating the loop sequentially will be duplicated in order to perform the computation in parallel, which will boost the resources dramatically. In addition, due to the limitation of the resources of the Zynq board provided, it is impossible to fully unroll the top-level iteration loop, instead, it can be unrolled partially.

Figure 7 shows the block diagram of the partially unrolled top level of the SVM classifier, where the block of multiplying the dot product output by 2 is ignored as it is considered to be free.

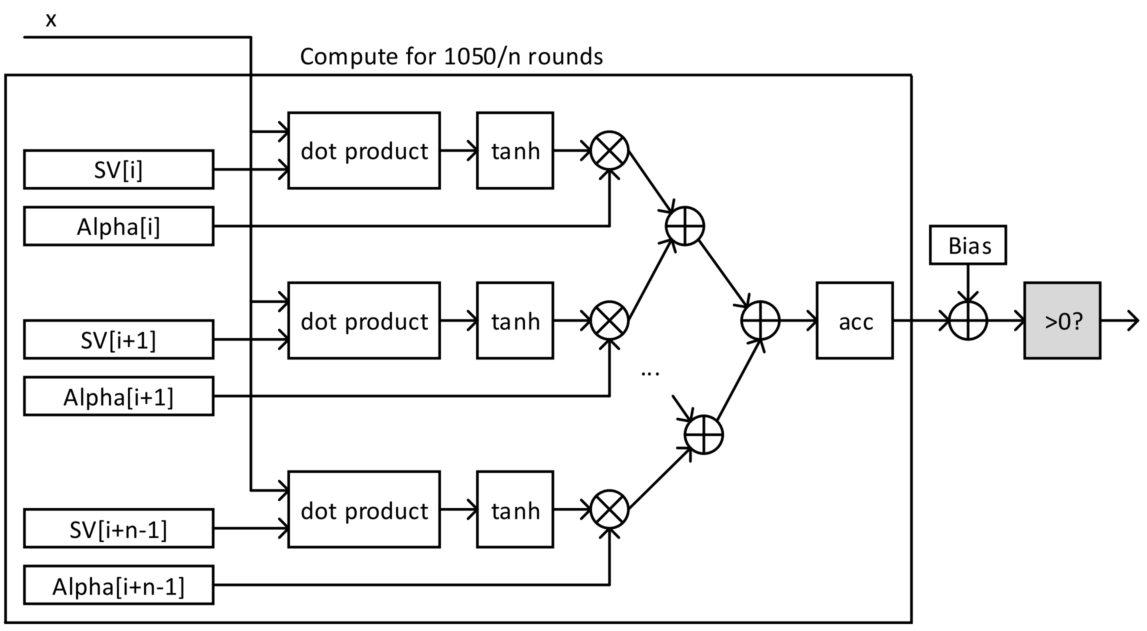

Figure 7 Block Diagram of the Partially Unrolled Top Level

Original Array	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	->	0

Block Partition	0	1	2	3	->	0
	4	5	6	7	->	4
	8	9	10	11	->	8
	12	13	14	15	->	12

Cyclic Partition	0	4	8	12	->	0
	1	5	9	13	->	1
	2	6	10	14	->	2
	3	7	11	15	->	3

Figure 8 Difference between Block Partition and Cyclic Partition

It is noticeable that in order to execute the top-level loop in parallel, multiple sets of support vectors and their corresponding alphas are required to be accessed at the same time. The first ideal that comes is to perform a complete partition on support vectors and alphas to make all of them accessible at any time. However, due to the limitation of FPGA resources, it is impossible to achieve the complete partition. Instead, the cyclic partition which can automatically map an array into multiple smaller arrays through element interleaving can be used. Figure 8 shows how the cyclic partition works and the difference compared to block partition where block partition maps the original array into several smaller arrays through separating the original array as sequential blocks without element interleaving.

In addition, the partial unrolling factor is supposed to be the multiplication of any divisors of 1050 which consists of 5 elements "{2, 3, 5, 5, 7}". In other words, 1050 should be divided by the factor exactly. If this requirement is not satisfied, additional costs will be punished for the data path and loop control or even crashes the HLS.

Table 4 summarizes the HLS synthesis results with different directives.

Implementation	Directive for Top Level	Latency	Interval	BRAM_18K	DSP48E	FF	LUT
[1] Arbitrary Precision	Partition	13651	13651	34	16	1222	1720
[2] Arbitrary Precision	Partition + Pipeline	1063	1063	34	16	1496	1829
[3] Arbitrary Precision	Partition + Pipeline + Unroll(2)	538	538	34	32	2444	3517
[4] Arbitrary Precision	Partition + Pipeline + Unroll(10)	119	119	170	160	10073	17912
[5] Arbitrary Precision	Partition + Pipeline + Unroll(15)	84	84	272	210	14784	33423

Table 4 HLS synthesis results of the SVM classifier

A full set of support vector must be partitioned for subfunction to compute the dot product in parallel, and this is the reason that directive partition is applied for implementation [1].

For implementation [2], directive pipeline is used for boosting the throughput of the SVM classifier whereas the resource utilization only slightly increases.

When it comes to unroll the iteration loop, it is noticeable that as the factor of the partial unroll increases, the resources are consumed dramatically. It is also noticeable that the number of DSP48E multipliers required is proportional to the unroll factor as each unrolled loop required its own individual computation blocks.

When the unroll factor goes up to 15, DSP48E multipliers will no longer be enough for dot product functions, the solution is to force HLS to synthesize more multipliers through consuming the resources of LUT, supporting further unrolling. However, the unroll factor cannot increase any more due to the limitation of the resources.

The best implementation with unroll factor 15 achieves the minimum latency and maximum throughput, consuming 97% of BRAM_18K, 95% of DSP48E, 13% FF and 62% LUT. Resources must not be fully utilized as the Zynq system will consume additional resources in order to implement other instances.

5. IMPLEMENTATION THE ZYNQ SYSTEM AND TIME MEASUREMENT

This part focuses on the implement of the Zynq-based embedded system which runs the SVM classifier through two approaches. The first approach directly runs the SVM classifier on Zynq without HLS accelerator (IP core) to obtain a reference execution time of classifying 2000 test vectors. The second approach implements the accelerator that synthesized from HLS and moves the computation of classification on the accelerator for the purpose of accelerating the computation. Two types of AXI interfaces will be used for establishing the communication between the Zynq system and the accelerator and the performance will be evaluated and compared.

5.1. Zynq-only System

The Zynq-only System directly executes the classification of the input vector through a software approach. The performance will be used as a reference for evaluating how much performance can be improved by the HLS accelerator.

Through directly running the C code provided on the arm core whose clock frequency is set to be 650MHz for stability concern, it takes approximately 2.67 seconds to classify 2000 test vectors.

5.2. Zynq System with AXI-Lite Interface

This part introduces the Zynq system that employs the SVM IP core derived from HLS for computation acceleration. The specific interface for Zynq system to communicate with the accelerator is AXI-Lite.

AXI-Lite interface is a lightweight bus protocol that only involves memory mapping, where a single set of data (typically 4 bytes) can be accessed through providing the address where the first byte of the data locates. The difference between the AXI-Lite and standard AXI bus protocol is that AXI-Lite does not support data burst where through providing the address of the first byte of the data, a number of data bytes can be accessed with automatically memory address increment, which means, only the address for the first data byte is required in order to access a number of the following data bytes.

Figure 9 demonstrates how AXI-Lite and standard AXI interface write data to the memory. All AXI interfaces employ a hand-shake mechanism to ensure the data transfer successfully. The process starts from the master sending the request of writing to the slave (memory) first, then wait for the ready response which indicates the slave is ready for data accessing. Only then the master can send out the address and data, and keep the current state until the slave acknowledges the operation. Not like the standard AXI interface, in order to send a number of data bytes (larger than 4 bytes) to the slave, the same hand-shake process needs processing for a number of times which increases the communication overhead significantly. This kind of overhead can also be eliminated through using the AXI-Stream interface which will be discussed in the following part.

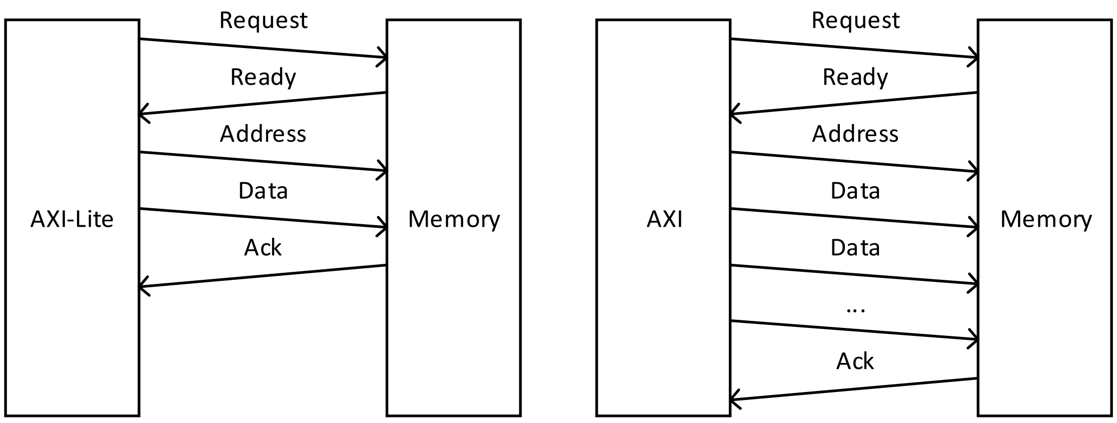

Figure 9 Difference between AXI-Lite and standard AXI Interface (Write Operation)

In addition, for this implementation that based on AXI-Lite interface, interrupt is used to trigger the arm core to read back the classification result which frees the arm core from polling the result so that it can process other tasks during the computation of classifications on the accelerator.

By means of employing the accelerator, the computation of classifying 2000 test vectors is successfully accelerated, achieving an overall execution time of 11.98ms which is approximately 222 times faster than the software approach running on Zynq.

5.3. Zynq System with AXI-Stream Interface

As AXI-Lite interface does not support data burst, the overhead of transferring a large amount of data bytes is quite large compared to the standard AXI interface. However, as the interface of the SVM classifier only involves the input vector which consists of 32 bytes (each element is 2-byte and the vector has a dimension of 16) and the output result is only 1-bit, even standard AXI interface is employed which supports data burst, the communication overhead cannot be reduced much as the burst length required for data transfer is quite short. As a result of this, AXI-Stream interface is proposed to be the best choice where all test vectors are transferred directly to the DDR3 memory and then acts like a FIFO. The accelerator is the corresponding FIFO consumer who consistently reads the available test vector and performs the computation until all test vectors are processed.

By means of employing AXI-Stream interface, two additional techniques Dataflow and Direct Memory Access (DMA) can be utilized.

5.3.1. DATAFLOW

The accelerator is required to accept the input vector as a data stream when AXI-Stream interface is employed. As a result of this, an additional task is implemented who reads the input vector elements from the data stream one by one. The accelerator can only start to process the SVM classification when all elements of the input vector are available.

In addition, in order to fully utilize the data width of the arm core which is a 32-bit RISC core, every two elements of the input vector are concatenated to be 32-bit. At least 8 clock cycles are required for the accelerator to accept one input vector with 8 pair of elements after concatenation.

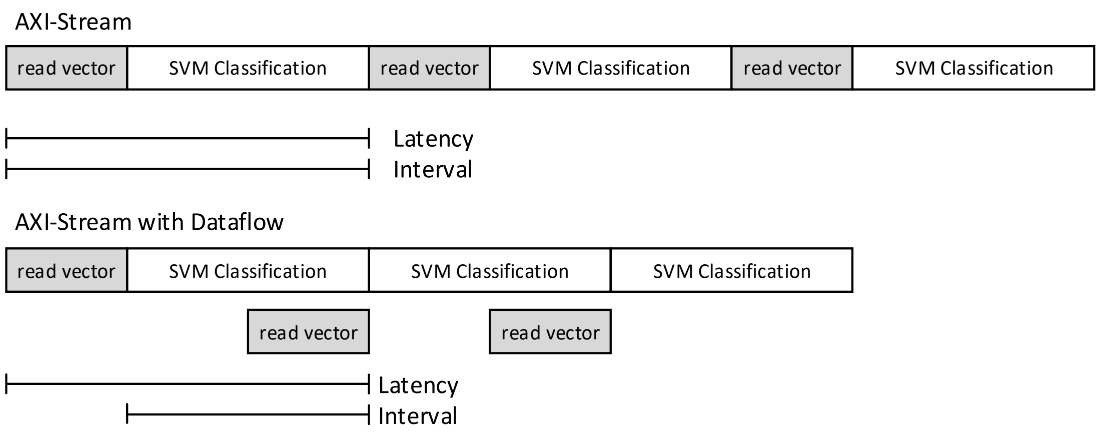

Figure 10 AXI-Stream Interface with Dataflow technique

Dataflow directive is applied in HLS on the top level to enable task-level pipelining. As shown in Figure 10, the top level without dataflow directive executes tasks of reading vector from the data stream and SVM classification in sequential. Whereas, if dataflow directive is used, the read vector task can be pipelined so that it will be processed during the computation of classification for the previous input vector.

As states in Table 5, the synthesis result with dataflow does not reduce the latency as expected, because the first operation of reading the input vector from the data stream cannot be pipelined. However, when considering the latency from the system level where 2000 test vectors are processed as a stream, the throughput is increased significantly.

Implementation	Directive for Top Level	Latency	Interval	BRAM_18K	DSP48E	FF	LUT
AXI-Lite Interface	Partition + Pipeline + Unroll(15)	84	84	272	210	15111	33913
AXI-Stream Interface	Partition + Pipeline + Unroll(15)	95	95	272	210	15090	33680
AXI-Stream Interface	Partition + Pipeline + Unroll(15) **Dataflow**	95	85	272	210	15201	34431

Table 5 HLS Synthesis Results for Accelerators with Different Interfaces

5.3.2. DIRECT MEMORY ACCESS

Direct memory access (DMA) is a technique that assists the processing unit in dealing with the data transfer between the memory and the peripherals. By means of using the DMA technique, the arm core no longer needs to manipulate the data transfer but ask the DMA controller to process the job so that it can process other tasks during data transfer.

Figure 11 demonstrates how the DMA technique frees the processing unit. In the beginning, the processing unit flashes necessary data into the memory for the DMA controller to access. Then it sends a request to the DMA controller and switches to other tasks. After accepting the request, the DMA controller streams the data to the accelerator for computation and streams the results to memory for the processing unit to access. After finishing the job, the DMA controller sends an interrupt signal to the processing unit to indicate that the job has been done. Finally, the processing unit switches back to the original task and processes the results.

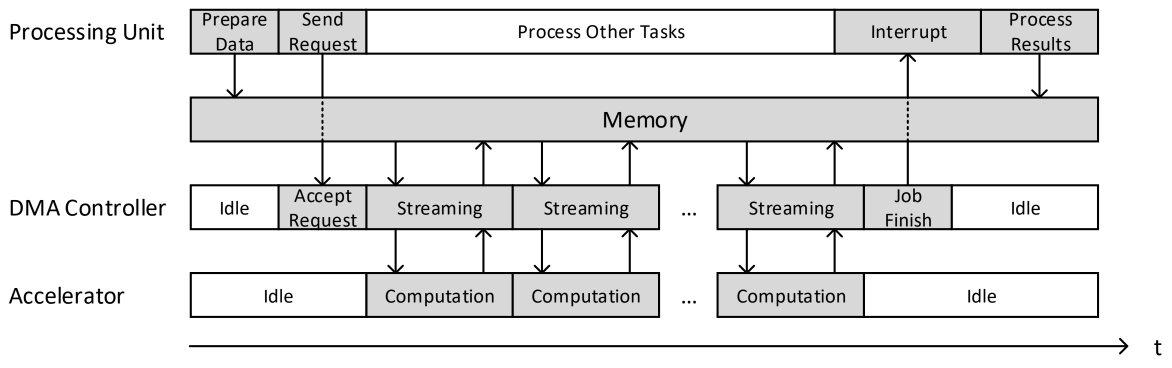

Figure 11 Process of Direct Memory Access

5.4. Time Measurement for Different Implementations

Table 6 summarizes the execution time that measured by the Zynq library (#include "xtime_l.h") for different implementations:

Implementation	Zynq Execution Time	Core Computation Time	Communication Overhead	Acceleration Ratio
Zynq Only (reference)	2.67 [s]	-	-	x1
Zynq + AXI-Lite HLS Accelerator	11.98 [ms]	1.68 [ms]	10.3 [ms]	x222
Zynq + AXI-Stream HLS Accelerator	1.77 [ms]	1.70 [ms]	0.07 [ms]	x1500

Table 6 Acceleration Summary of the SVM classifier

Zynq execution time measures the time required for the Zynq system to process 2000 test vectors and obtain all classification results. If the HLS accelerator is employed, the time spent on transferring all test vectors to the accelerator and reading back all classification results is also included.

Core computation time only estimates the time required for the accelerator to finish the computation of 2000 test vectors based on the interval from the synthesis result, assuming that test vectors (or stream) are always available:

For the accelerator with AXI-Lite interface, the core computation time is:

$$84 \times 2000 \times 10 = 1.68 \times 10^6\ [ns] = 1.68[ms]$$

For the accelerator with AXI-Stream interface, the core computation time is:

$$85 \times 2000 \times 10 = 1.700 \times 10^6[ns] = 1.70[ms]$$

It is noticeable that the accelerator with AXI-Stream interface has slightly longer computation time, the reason behind is that the accelerator with AXI-Stream involves the process of reading the input vector as a stream whereas the accelerator with AXI-Lite interface does not have such process and just assumes that the input vector is always available at the interface, thus one more clock cycle is considered to be used for buffering the input vector which must be kept the same during the computation of the classification.

6. CONCLUSION

This report demonstrates how to accelerate a software SVM classifier using Vivado HLS through moving the computation intensive blocks to the FPGA hardware. The result shows that for an accelerator with lightweight AXI-Lite bus interface, the computation of 2000 test vectors can be speed-up 222 times faster than the original computation on Zynq through software approach.

In order to further reduce the communication overhead between the arm core and the accelerator, interface AXI-Stream is proposed to be the best choice which achieves an excellent acceleration performance, which is almost 1500 times faster than the software approach.

7. REFERENCE

[1] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," presented at the Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Monterey, California, USA, 1998.