Report 2

Advanced Digital Systems Design

Antoniadis Dimitris, dimitris.antoniadis20@imperial.ac.uk

Ramirez Pelayo pelayo.ramirez20@imperial.ac.uk

Group 2

1. INTRODUCTION

This document is the report of the final project of the *Advanced Digital Systems Design* Class. The aim of this project is to improve the Classification algorithm of a Support Vector Machine (SVM). The primary objective is to achieve as great throughput as possible.

This paragraph briefly introduced the goals of the project. The next section discusses the advantages of the CORDIC algorithm on FPGAs and a way to extend its input range. The 3rd section presents and justifies the choices that were made during the algorithm design regarding the bit lengths of the variables and the accuracy of the Classifier. The 4th section discusses various improvements on throughput via different parallelisation and acceleration methods. In the 5th section, the problems and the results from the final implementation are presented. Finally, the 6th section concludes the results and presents the trade-offs, the improvements, and the problems the authors encountered during the design.

2. CORDIC ALGORITHM

Before delving deeply into the CORDIC algorithm, the kernel will be briefly discussed. The gaussian kernel used is a separable kernel, in other words, it can be described as N different mapped subspaces separated by a hyperplane in the N-dimensional space. Therefore, it will allow the SVM to classify the dataset into different subspaces easily by raising the dimensions of the problem. To implement the exponential function, the CORDIC algorithm will be used.

CORDIC stands for Coordinate Rotation Digital computer and it is an algorithm used to calculate hyperbolic and trigonometric functions (among many other functions) in a way that is very efficient to implement on an FPGA. This hardware efficiency is due to the fact that CORDIC only performs additions, subtractions, and bit shifts. The operations performed by the algorithm represent rotations, $F(\theta)$, in a 2D hyperbolic space, the Lorentz-Minkowski space, whose metric tensor is given by $g_{\mu\nu}$.

$$g_{\mu\nu} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad (1)$$

$$F(\theta) = \begin{pmatrix} \cosh(\theta) & \sinh(\theta) \\ \sinh(\theta) & \cosh(\theta) \end{pmatrix} \qquad (2)$$

With the inner product defined by $g_{\mu\nu}$, the rotations can be seen as a mapping along the hyperbolas $xy = C$, where x and y are the coordinates in the 2D space, and C are constants representing the family of hyperbolas. On the other side, $F(\theta)$ is a non-orthogonal matrix with determinant equal to 1 that belongs to the Lorentz group. As a consequence, it performs the Lorentz transformations and will be the basis of the CORDIC algorithm.

From a high-level perspective, the algorithm rotates a vector an angle θ by doing a series of rotations θ_i using minimal computation resources. However, not all the θ_i can be efficiently implemented on the FPGA. Therefore, we must add a constraint to the angle values [1] in order to implement it as a bit shift.

$$tanh(\theta_i) = 2^{-i} \quad (3)$$

These angle values will be obtained previously with MATLAB and stored in a LUT, so that the algorithm can access them easily. To apply straightforwardly this condition on our rotation matrix, we need to rewrite it in the following way:

$$F(\theta) = \frac{1}{\sqrt{1-\tanh^2\theta}} \cdot \begin{pmatrix} 1 & tanh\theta \\ tanh\theta & 1 \end{pmatrix} \qquad (4)$$

Now, combining (3) and (4), we can write the algorithm for one iteration. However, a new variable, δ^i, will be first introduced. It is called the "decision variable" and will take the values ± 1. As a consequence, it will allow the algorithm to choose whether to perform a clockwise rotation or an anticlockwise rotation depending on its sign.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \frac{1}{\sqrt{1-2^{-2i}}} \begin{pmatrix} 1 & \delta^i 2^{-i} \\ \delta^i 2^{-i} & 1 \end{pmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \qquad (5)$$

As it can be seen from the above equation, not only is the vector rotated in each iteration, but also scaled a factor $K_i = 1/\sqrt{1-2^{-2i}}$ each time. Therefore, we must discuss what the initial values of the coordinates x_0, y_0 are. The final aim of the algorithm is to implement the $\exp(\theta) = \cosh(\theta) + \sinh(\theta)$, therefore, we need the final value to be consider these factors to give the appropriate solution. Instead of obtaining the factor K_i for each iteration, the accumulative value for the fixed number of iterations used will be obtained with MATLAB and set as x_0 and y_0.

$$K_n = \lim_{N\_iter \to \infty} \prod_{i=1}^{N\_iter} \frac{1}{\sqrt{1-2^{-2i}}} \to 1.2051363 \qquad (6)$$

This algorithm can be implemented in two ways: rotation or vectoring mode. As it can be deduced from the previous explanation, the rotation mode will be used. In this method, the angle z will be driven from the argument of the function, Z_0, to zero after the iterations [2]. As a consequence, both the x and y will yield the exponential value because of the initial value selection for x_0 and y_0.

$$z_{i+1} = z_i - \delta^i \cdot \operatorname{atanh}(2^{-i}) \qquad (7)$$

$$x_n, y_n = \frac{1}{K_n}(x_0 \cosh(z_0) + y_0 \sinh(z_0)) = \exp(z_0) \quad (8)$$

However, this CORDIC algorithm has to deal with two mathematical issues: convergence and its range. While the former is easily solved, the latter needs a greater modification of the algorithm. Regarding the first one, elemental hyperbolic rotations do not converge, however, we can achieve convergence by repeating the following iterations i=4,13,40,…,k,3k+1,… [2]. To solve the second issue, we must first study the range of convergence of our algorithm. As it has been explained, the rotation mode will drive the z_0 angle to zero. Therefore, a condition on the maximum value of it must appear.

$$|z_0| \le \theta_{N_{iter}} + \sum_{i=1}^{N_{iter}} \theta_i \qquad (9)$$

$$|z_0|_{max} \approx 1.182 \text{ for } N_{iter} \to \infty \qquad (10)$$

This limits the maximum angle that can be introduced in the algorithm to obtain a correct result [2]. In other words, this is the maximum angle that the algorithm can drive to zero (subtracting θ_i in every iteration). As a consequence, a modification has to be included to expand the range of convergence from [-1.182,1.182] to higher values. This will add a section of code prior to the calculation of x_n and y_n [3]. If the argument of the exponential, z, is greater than the allowed $|z_0|_{max}$[1], then it will be split in two terms, q and r. We have to consider that the angle will always be negative as it is obtained from the product of $-\gamma$ (with $\gamma > 0$) and the L2 norm $||\vec{s}||$ of a vector which is always positive.

[1] Note that this value will be adjusted in the code depending on the number of iterations used in order to achieve the maximum accuracy.

$$z = q \cdot Ln(2) - r \qquad (11)$$

We can exploit the fact that we are using fixed point representation to implement q as simply the division of $-z/Ln(2)$ and r as $z + q \cdot Ln(2)$. Where q is always positive and r negative. Thus, we can ensure the following is satisfied:

$$|r| < Ln(2) < 1.182 \qquad (12)$$

To obtain the final value of the exponential, we need to right shift q times the x_n returned by the algorithm after the iterations. In other words:

$$\exp(z) = \exp(r) \cdot 2^{-q} \text{ with q>0} \qquad (13)$$

With this implementation, the range of convergence of the algorithm has been expanded and is now limited by the number of bits specified to q and the exponential. Our choices will be discussed in the following section.

3. DATA REPRESENTATION AND THEIR OPTIMISATION

In this section the data types, word sizes and their effects in terms of accuracy will be discussed. Moreover, some limits calculated with MATLAB prior to the simulation of the algorithm will be commented to optimise accuracy even further. The accuracy aim will be 99.66%, which is the accuracy obtained for the implementation given. The accuracy of our implementation will be increased along this section and, at the end of it, some will be traded for time optimisation.

First of all, some data definitions and word sizes were given to us in the coursework instructions and, therefore, they will not be the aim of these optimisations. However, there are some key variables in our implementation that implicitly contain the trade-off between resources and accuracy. As a first optimisation, if it can be considered as such, the *classify* function's code was translated to MATLAB and ran there. This first approach yielded the maximum and minimum value a variable will carry, as well as the number of decimals required. Fostered by the results obtained with this software, we concluded that at least one decimal was required in all our model variables. We decided to adopt a fixed-point representation due to this latter argument and to the fact that this variable definition has great advantages in hardware implementation; for instance, it helps control the resource costs of the FPGA.

Variable	**Max. value**	**Min. value**	**Type definition**	**Description**
L2Squared	24.177,50	46,75	ap_fixed<24,16>	Stores the value of the L2 norm.
Angle_rad	-0,0468	-24,1775	ap_fixed<16,6>	The argument of the exponential for the CORDIC algorithm.
Exp_value	0,9543	$3{,}1612 \cdot 10^{-11}$	ap_fixed<16,2,>	Returns the exponential value after the CORDIC algorithm.
Sum	2,9723	-3,2058	ap_fixed<16,4>	Carries the value of the sum of all the alphas times the exponential for all the SV in every image and then adds the offset *bias* to it.
q	34	1	ap_fixed<8,7>	Stores the value described in (11).

Table 1. Values obtained with MATLAB for the different variables that will be used. The maximum and minimum values shown are rounded. The notation of type definition ap_fixed<w,i> stands for arbitrary precision fixed-point representation <word length, bits for integer part>. Note this definition is for signed numbers and, therefore, we must allocate an extra bit for the sign. The w chosen will be discussed in the following paragraphs.

As it can be seen from the above table, the integer bit length has been appropriately selected to be able to represent the whole range of numbers the variable can store, plus one extra bit for the sign. Thus, the first optimisation of resources has been made. Nevertheless, these definitions have a degree of freedom which is the number of bits left for the decimal representation (w-i). This is the case of study for the next optimisation, however now a trade-off should be considered between resources and accuracy of the model.

Firstly, we will consider the first four variables: L2Squared, Angle_rad, Exp_value and Sum. As it can be seen from Table 1, they carry a non-negligible decimal part, therefore, it would be expected to achieve an accuracy function that increases with the word length. To study the accuracy, a set of events will be presented for easier comprehension of the plot.

	L2Squared	Angle_Rad	Exp_value	Sum
Case 1	ap_fixed<20,16>	ap_fixed<12,6>	ap_fixed<12,2,>	ap_fixed<12,4>
Case 2	ap_fixed<22,16>	ap_fixed<14,6>	ap_fixed<14,2,>	ap_fixed<14,4>
Case 3	ap_fixed<24,16>	ap_fixed<16,6>	ap_fixed<16,2,>	ap_fixed<16,4>
Case 4	ap_fixed<26,16>	ap_fixed<18,6>	ap_fixed<18,2,>	ap_fixed<18,4>

Table 2. Four different cases considered. The number of fractional bits per variable increases as the case number increases. Case 3 corresponds with the values presented in Table 1.

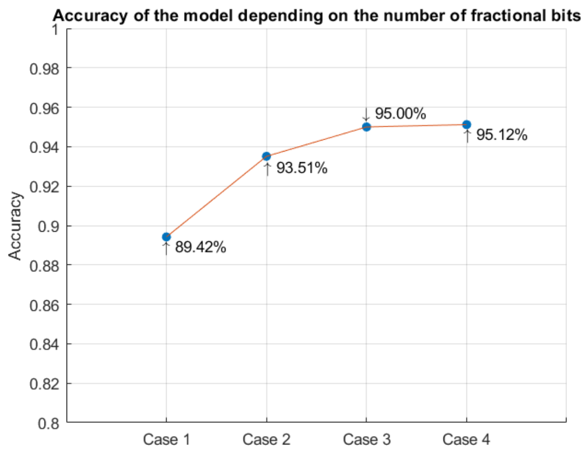

Figure 1. The accuracy of the model is represented for each of the cases shown in Table 2.

It can be seen that the accuracy improvement of these word-size optimisations is asymptotically limited around 95%. However, the resource costs are not limited by this asymptote, they keep increasing as the number of fractional bits increase. Therefore, it would be useless to increase the number of fractional bits further than Case 3 or 4, as the accuracy gain will be very small but it will leave less space for future unrollments or partitions. If we compare Case 3 with Case 4, we see a slightly better accuracy of the latter, but a resource-cost study yielded a 2% more utilisation of the total resources available of BRAM_18K (25%) and a 1% more of DSP48E (4%). As a consequence, we considered that Case 3 was the optimal scenario. Even though we lose 0.12% accuracy, we are saving resources which will be shown to be an issue in the following sections. Moreover, it will be shown that future optimisations will compensate this loss leading to even higher values of accuracy.

Secondly, we will consider the variable q on its own because it has been proven to be the variable that most affects the accuracy of the model. The MATLAB simulation showed that 59.13% of the arguments of the exponential exceed this range of convergence and, therefore, pass through q. This variable has some implicit effects over the angle that inputs the CORDIC if the range of convergence [-1.182,1.182] is exceeded, because it will determine the number of decimals that Vivado considers for the $Ln2$ in (11). While the integer part of q is fixed to 7 bits, the fractional number of bits can be modified.

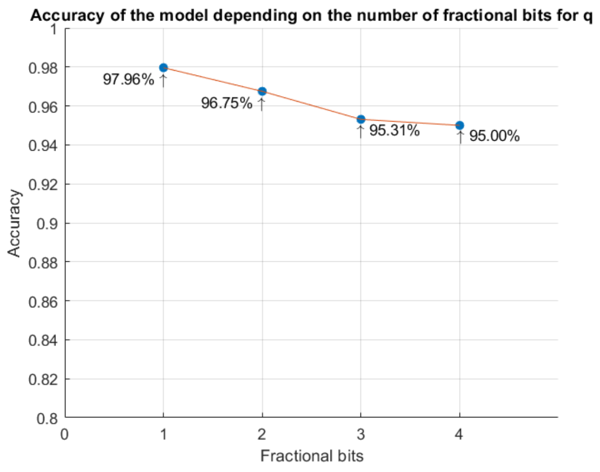

Figure 2. The accuracy of the model is represented against the number of fractional bits allocated for the variable q.

On first inspection, it will be expected to observe an accuracy improvement as the number of fractional bits increases, however, this is not the case. The reason why this happens is based on the way q is implemented. As it was explained, q is the integer division between -z and Ln2, so its value will always be different than the one obtained and that should be used to calculate r (see equation 11). Luckily, the truncation done to Ln2 with only one fractional bit compensates better the overall loss of decimals in q for all the arguments of the exponential. Therefore, we obtain the best accuracy with one, while we minimise the resources used.

So far, with the right choice of variable's sizes, the accuracy of the model has been improved up to a value of 97.96%. The next case of study is the implementation of both the CORDIC (equation 5) and the product $-\gamma \cdot \|\boldsymbol{x}_i - \boldsymbol{x}_j\|$ in the argument of the exponential. If we look carefully, both operations can be either implemented as a product, "*", or as a bit shift. Before proceeding with the results, the implementation as a bit shift operation will be presented. Equation 5 shows there is a 2^{-i} operation in every iteration that can be performed with a bit shift. For the second product, we can implement $\gamma = 0.001$ as a normal product or as a right shift of 10 bits. Both implementations have been analysed and the conclusions extracted are obvious. The bit shift gives a more efficient implementation (uses less resources) and improves even further the accuracy.

Implementation	**Accuracy**	**DSP48E resources used**
Bit shift	98.11%	3%
Product operation	97.96%	5%

Table 3. Accuracy of the model and percentage of usage of the DSP48E available resources. The BRAM_18K (23%), FF (1%) and LUT (5%) rounded percentage of use is the same for both implementations.

As it was described in the first paragraph of the section, we have increased the accuracy with all the optimisations done and, now, we are in the position to trade some of it for less computational time. In the CORDIC section, the number of iterations was introduced and it has been omitted until now. It is straightforwardly seen that the higher this number, the higher the computational time will be, but also the higher the accuracy. Before discussing these results, equation 10 has to be brought to our attention. The series in equation 9 converges to $z_0 = 1.182$ if and only if $N_{iter} \to \infty$, however, this cannot be considered the case and its value should be obtained externally and introduced in the code (summing the series up to the selected N_{iter}). This value will determine the number of angles that go through q and it has to be very accurately calculated if we want to optimise our accuracy. Having said that, three different numbers were studied: 5,9,15 with accuracy values of 97.56%, 97.88%, 98.11%, respectively. In order to do a time study, C simulation times were measured for the accuracy obtention. It is important to remark that these values refer to the performance of our local computers and not of the FPGA, however, they provided strong evidence of the time reduction. Both simulation times of $N_{iter} = 9{,}15$ will be compared to the result obtained for $N_{iter} = 5$. For 9 iterations, the C simulation took 4% longer and, for 15 iterations, it took 7% longer. Although these values will not be the same on the FPGA implementation, the relations will be similar and, as a consequence, we decided to stick to $N_{iter} = 5$, leaving a final accuracy of our CORDIC implementation of 97.56%.

4. Synthesis Optimizations and Implementation

This section presents an investigation around the optimizations. Trade-offs between synthesis results and implementation are also presented. Finally, a different approach is proposed as the authors believe important speed up can be achieved by changing the classification algorithm.

Synthesis Optimizations

Initially a communication interface was introduced as a directive for the Classifier Function which was responsible for getting an image and providing the result of the SVM Classification algorithm back. The interface that was chosen is AXI LITE and it was specified both for input image and output sum.

Loop Unrolling

The Classification Algorithm that was used was the same as the one in the testbench. It contains two loops that can be unrolled to optimize throughput. These loops are not independent of each other, therefore unrolling does not provide optimal results. However, it improves by far the throughput. A different approach will be discussed later on this document.

Initially the outer loop was totally unrolled. However, the timing constraints were not met, therefore a partial unroll should be considered. Various unroll loop factors were considered and, initially, a clear pattern could not be obtained. This fact can be verified by the following table, where throughput is defined as $t = 1/(T * Interval)\ and\ T$ is the clock period equal to 10ns. The classifier can classify one image, or $t = images/second$.

Unroll Factor	Interval	Throughput
100	401656	248.9693
50	182020	549.3902
25	323975	308.6658
10	240077	416.533
1	392536	254.7537
2	260480	383.9066
5	78508	1273.756
15	26170	3821.169
33	11896	8406.187
55	7138	14009.53

Table 4. Various Unroll Factors of the Outer Loop. With Green Factors of 165 producing a pattern of throughput improvement.

However, by inspecting the following graph, it becomes clear that a factor of the total number of SVMs, which is equal to 165, provides a throughput maximization pattern. The factors of the Number of the Support Vectors provide greater throughput. After profiling the various unroll factors the following reasoning was extracted: if an unroll factor of 2 is chosen, then the loop will be split in two parts. One part will execute 82 iterations, while the other one will execute 83. Therefore, one part of fabric will be idle while waiting for the other to finish its final iteration. On the other side, if factors of 165 are chosen, all unrolled loops will finish their execution approximately simultaneously.

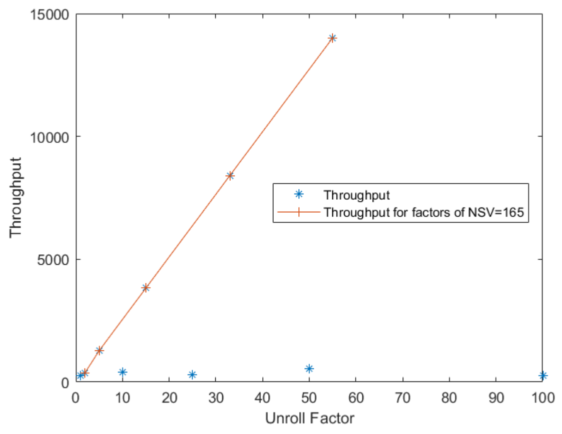

Figure 3. Throughput of various unroll factors and the improvement pattern of factors of 165 (Number of Support Vectors).

Based on this assumption and based on timing constraints results provided by synthesis an unroll loop factor of 55 was chosen. The constraints result obviously, do not allow a big unroll factor. Therefore, the inner loop of image table was also unrolled. The inner loop contains 784 iterations. Therefore, factors of 784 were chosen. The results obtained are presented on the following table. The same pattern as above is followed. Finally, Loop Merge pragma was used to slightly improve the throughput, allowing greater flexibility on synthesis tool.

Unroll Factor	Interval	Throughput
8	3022	33090.6
16	2782	35945.3
56	2518	39714.0

Table 5. Unroll Loop Factor for Inner Loop.

Memory Partitioning

Since Unrolled Loops have dependencies, the authors of this document, assumed that by partitioning the memories storing SVM values, alpha values and x values, better throughput would be achieved. So far, memories are BRAMs so they have only two ports. By partitioning the memories using the same factor as the unroll loop factors or a multiple of this factor, deadlocks on accessing the same resources would be reduced. Initially, SVMs and alphas were chosen to be split in 55 blocks and Image was chosen to be split in 56 blocks. Two time consuming attempts were made but they failed to complete. Values of 5 and 4 were chosen respectively, however the throughput did not improve. The calculated throughput was equal to 617.3. After profiling the operations it was discovered that due to the loop's dependency, the partitioning makes even more complex the synthesis procedure, as such greater FSM has to be synthesized and finally the design becomes slower. A complete partitioning significantly improved the throughput, however the utilization exceeded 600% and it was not feasible.

CORDIC

CORDIC algorithm is already optimized on its own, since it uses only shifts and add/subtract operations. Therefore, inline pragma was used to facilitate greater flexibility, leading to minor improvement to throughput. The CORDIC algorithm was implemented inside function *my_exponential(…)*. Additionally, a static table with only 5 elements of rotating angles was initialised inside function, so they maintain their value on every iteration. The reason only 5 elements were chosen and therefore only five iterations, is because the sixth iteration has almost most of the times the same value as the fifth. Therefore, for an accuracy of approximately 97.5 percent, only 5 iterations were necessary.

A Potential Different Approach

The authors of this document firmly believe that this acceleration method is not the optimal. As a starting point, loop dependencies should be removed. Additionally, an $lSquared[IMG\_SIZE]$ table should be introduced to allow the total independent operation of the loops. In this way, the outer loop can be partially unrolled and the inner loop can be pipelined, since $lSquared$ value is accumulated. This operation could be facilitated by partitioning the SVMs table in a cyclic way so that the inner loop has faster access on sv[i][j] value. In this way, while the loop is going through the j variable, the same memory block would be used. A complete partitioning or partial partitioning equal to the unroll factor of the outer loop is suggested in order to allow parallel write operations on $lSquared[i]$ elements. A cyclic partitioning is also suggested on $x[]$ table. In this way, when a bit of image is chosen, it will be from a different memory and not all the loops will try to read the same read block. At the end of these loops an $lSquared[i]$ will be ready to be used for exponential calculation. By partitioning $K[]$ and unrolling the loop, $K[i]$ values can be calculated parallelly. Finally, $K[]$ values have to be multiplied by corresponding alphas and be accumulated. Therefore, a pipeline sum would be fit for this operation.

An attempt on this approach was made, but time-consuming synthesis failed to finish. A snippet of the proposed classifier function is provided below.

```
void classifier(fx_svm x[IMG_SIZE], fx_sum *sum){
#pragma HLS ARRAY_PARTITION variable=fx_svs cyclic factor=33 dim=1
    static fx_sum bias = fx_bias[0];
    *sum = 0.0;      // the biggest sum value is less than 32768.multiple of 0.25
    fx_exp_value K[NSV];
    #pragma HLS ARRAY_PARTITION variable=K block factor=33 dim=1
    static fx_l2Squared l2Squared[NSV];
#pragma HLS ARRAY_PARTITION variable=l2Squared complete dim=1 // static always initialised to 0

IM
#pragma HLS UNROLL factor=33
G:
for (int j=0; j<IMG_SIZE; j++){
    fx_x _x = x[j];

#pragma HLS PIPELINE
    SVM:
    for (int i=0; i<NSV; i++){
        fx_svm _sv = fx_svs[i*IMG_SIZE + j];
        l2Squared[i] = l2Squared[i] + ((fx_l2Squared) (_sv - _x) * (_sv - _x));
    }
}

#pragma HLS UNROLL factor=33
    EXP:
    for(int i=0; i<NSV; i++){
        K[i]= my_exponential(-(l2Squared[i]>>10));
    }

    SUM:
    for(int i=0; i<NSV; i++){
#pragma HLS PIPELINE
        fx_alpha alpha = fx_alphas[i];
        *sum +=  alpha*K[i];
    }

    *sum = *sum + bias;
}
```

Code 1. Extract of the code proposed for the classifier function.

5. Implementation

Based on the analysis made on Synthesis procedure, an implementation where the Unroll loop factors are 55 and 56 respectively use around 23% of BRAM and 46% of DSP Resources. The rest of the FPGA blocks are used in less than 10%. Therefore, having in mind that it may lead to a design with negative slack, these settings were chosen in order to implement the design. However, as it was proved the implementation of the design was not feasible because the total negative slack was huge. The following graph shows multiples attempts that were made in order to achieve zero total negative slack. This target was achieved for an unroll inner loop factor of 8.

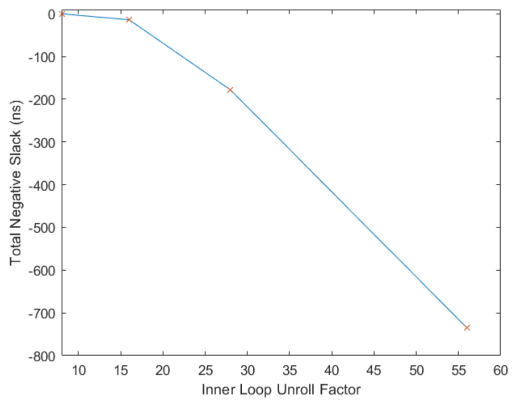

Figure 4. Total Negative Slack compared to the unroll factor of the inner loop.

However, this came at a cost. First of all the throughput decreased and the resources utilisation also was reduced. More specifically, the interval was equal to 2998, as such the throughput was equal to 33355.5 images per second. The BRAM utilisation remained 23% however the DSP blocks declined drastically to 5% utilisation.

Based on the above statistics the design is expected to achieve a poor performance. Though, it is expected not to suffer from negative slack.

A testbench was created in Vivado SDK including two implementations. One implementation was about the FPGA and one implementation was about the ARM CPU. Both of them were supposed to implement the same function, since for the ARM implementation is a copy of the Vivado_HLS function. However, weird outputs were obtained on the output of FPGA, while ARM implementation maintained the same accuracy as Vivado_HLS simulations.

More specifically, in order to write to the FPGA an image the following command was used:

```
XClassifier_Write_x_V_Words(&HlsClassifier, 0, ((int*) (&x[0])), IMG_SIZE/4);
```

The above command casts the address of the first element of the image (ap_fixed<8,7>) to a type int pointer. The type int is 32 bits long, therefore, the length of write is equal to the image size divided by 4, since int is 4 times greater than an element of the image.

In order to read the results of the Classification function for an image the following function was used:

```
XClassifier_Get_sum_V_o(&HlsClassifier);
```

The output of this function is a 32-bit unsigned integer. The authors expected that inside these 32 bits, a 16-bit ap_fixed could be extracted. However, the results were different. For example, $score[0] = -2.545410$, but by casting to ap_fixed<16,4> the result of the output the value obtained was 288. Then the bits of the output were printed and the bits were 0000 0000 0000 0000 0000 0001 0010 0000. No resemblance could be extracted from the output and the simulations made previously.

The authors believe that it may be a result of source code error, since the implemented design passed all checks and validations through the whole procedure. The ap_fixed library was difficult to be included properly in a C++ file. Additionally, in the Vivado SDK C++ file wizard there is not a template file that could be used like in the case of the Hello World in a C file project.

Finally, at this point an assumption can be made that the FPGA operates correctly and the FPGA version can be compared to the ARM version. For the comparison, the TTC Timer has been used. COUNTS_PER_SECOND variable was used. This variable is equal to half the frequency. The clock used is the ARM PLL which has a frequency of 666 MHz. By using this counter and obtaining its value, execution time can be measured.

The execution time of FPGA was 27307507.98 us, therefore 2061 images in 27307507.98 us leads to a throughput of 75 images/second. The execution time of the ARM function was 3945083738.44 us, therefore the throughput is equal to 0.522 images per second. FPGA implementation is much faster than the ARM implementation. A reason why ARM implementation is extremely low is that the same algorithm is used and CORDIC algorithm is not oriented for generic CPUs. A better exponential implementation could be used for the ARM algorithm, but in that case the points that were calculated by ARM implementation would be biased and the final result would not be consistent.

The accuracy obtained on the FPGA implementation is about 43% and for the ARM function is 97% as it was in the synthesis stage simulations.

Based on the execution time results, an interrupt vector can be exploited to split the workload between the ARM function and the FPGA implementation. The FPGA is about 150 times faster, therefore a ratio of 1/150 images could slightly improve the execution time exploiting both CPU and FPGA parallelly. Another potential option is to use 2 threads, one for controlling the CPU implementation and one for controlling the FPGA.

6. Conclusions and Discussion

The aim of this project was to maximise the throughput of the SVM Classifier. The authors of this document would like to note that a well optimized SVM Classifier needs multiple iterations in order to achieve great performance. This can be verified by the fact that multiple analyses were made around the word lengths in order to decide the optimal point of trade off between area and accuracy. Then, the authors would like to bring to the reader's attention the efforts that are needed in order to improve the throughput result of synthesis. Multiple techniques can be used, however, a synthesis procedure is time-consuming. Therefore, choices have to be made carefully and they have to be examined earlier before they are performed. Even if there is a faithful synthesis result with good throughput, area utilisation and timing, this circuit may fail to be implemented and may suffer from Negative Slack. This part is the most time consuming because the generation of the circuit takes many hours, in our case, every time, this part on its own needed 5-6 hours. The testbench implementation on the SDK was a demanding procedure since a template is necessary to provide a decent result. The ap_fixed library was not compatible with the C style Hello world template and therefore a C++ empty application without template was chosen. A lot of part of a C style template was copied to the C++ empty application but many compile errors were encountered. Finally, the authors agree that Vivado is a quite difficult tool and needs a lot of experience in order for the user to feel confident using it.

Regarding the results of this report, the more the bits of the words the greater the accuracy and the resources utilisation. Additionally, the greater the parallelisation the greater the throughput. Furthermore, the more independent the operations are the greater the throughput is. An important optimisation can be performed by inspecting the iterations on a repetitive procedure, since there is a saturation point.

Taking the above observations into consideration, a faithful implementation can exploit a parallel implementation on FPGA and ARM CPU to further improve the throughput.

7. REFERENCES

[1] Kastner, R., Matai, J., and Neuendorffer, S., *Parallel Programming for FPGAs*, arXiv e-prints, 2018.

[2] Daniel R. Llamocca-Obregón, Carla P. Agurto-Ríos, *A fixed point implementation of the expanded hyperbolic cordic algorithm*, Latin American Applied Research. 37. 83-91, January 2007.

[3] L. Moroz, V. Samotyy, *The CORDIC Method of Calculation the Exponential Function*, e-Journals, Vol. 4, March 2018.