A Parallel Implementation of Sequential Minimal Optimization on FPGA

Daniel H. Noronha[a], Matheus F. Torquato[b], Marcelo A. C. Fernandes[c,*]

[a] *University of British Columbia, Vancouver, BC V6T 1Z4, Canada*
[b] *College of Engineering, Swansea University, Swansea, Wales, SA2 8PP, UK*
[c] *Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte (UFRN), Natal, RN, 59078 970, Brazil*

Abstract

This paper proposes a parallel FPGA implementation of the training phase of a Support Vector Machine (SVM). The training phase of the SVM is implemented using Sequential Minimal Optimization (SMO), which enables the resolution of a complex convex optimization problem using simple steps. The SMO implementation is also highly parallel and uses some acceleration techniques, such as the error cache. Moreover, the Hardware Friendly Kernel (HFK) is used in order to reduce the kernel's area, enabling an increase in the number of kernels per area. After the parallel implementation in hardware, the SVM is validated by bit-accurate simulation. Finally, analysis associated with the temporal performance of the proposed structure, as well as analysis associated with FPGAs area usage is performed.

Keywords: SVM, SMO, FPGA, Support vector machine, Sequential minimal optimization, Hardware.

1. Introduction

In the last few years, the requirements for electronic systems have drastically changed, mainly due to new applications that are emerging. One of the fields

*Corresponding author
Email addresses: danielhn@ece.ubc.ca (Daniel H. Noronha), m.f.torquato@swansea.ac.uk (Matheus F. Torquato), mfernandes@dca.ufrn.br (Marcelo A. C. Fernandes)

that recently gained much attention is the field of Artificial Intelligence (AI), more specifically the Machine Learning (ML) field. This area of study increased the need for higher processing speed and lower power consumption. It is well-known that high throughput is usually required for many ML applications. This is especially true in the fields of Computer Vision Zepeda & Perez (2015) and Big Data Huang & Liu (2014). This increasing need for speed and low power makes the implementation of those algorithms in reconfigurable hardware even more desirable.

Moreover, with the increase of the number of embedded systems that use ML techniques and with the advent of the Internet of Things (IoT), the low power consumption of those systems became even more critical. In order to allow this decrease in power consumption, it is necessary to decrease the clock of those devices. Nevertheless, in order to avoid the reduction of the throughput of those devices, it is necessary to allow parallelizations, especially using reconfigurable hardware.

The importance of Field-Programmable Gate Arrays (FPGAs) as compute accelerators has dramatically increased during the last couple of years. Many companies such as Amazon, IBM and Microsoft included FPGAs in their data centers aiming to accelerate their search engines. In the center of those applications are many machine learning algorithms, such as Support Vector Machines (SVMs). For FPGAs to thrive in this new role, the efficient usage of FPGA resources is required.

One of the most used machine learning algorithms is the Support Vector Machine (SVM). This algorithm has many different application, specially involving classifications and regressions in the fields of natural language processing (NLP). The SVM algorithm shows many parallelization opportunities and is therefore very interesting for a hardware implementation.

In order to implement the hardware of the already consolidated SVM algorithm, a study was performed to identify the main elements that must be present in its implementation. This study included the new heuristic proposals for the Sequential Minimal Optimization (SMO) and the different kernels proposed to

be implemented in hardware. Next, a proposal was made for parallel implementation of the algorithm in order to allow the use of any number of support vectors and configurable kernel parameters.Thus, the target of this work is the development on FPGA a parallel algorithm of the training phase of a support vector machine using SMO.

1.1. Related work

Several works in the literature show the implementation of AI algorithms in reprogrammable hardware. The majority of works in this area are justified by the enormous gain in speed and energy savings, which are possible in this type of hardware Torquato & Fernandes (2019); de Souza & Fernandes (2014); Da Silva et al. (2019); Da Costa et al. (2019); Coutinho et al. (2019). There is a growing interest in the use of SVMs in applications related to image processing and embedded classification systems. In this context, the high computational cost of SVM, especially in large data sets, requires the acceleration of this algorithm. The biggest problem with software implementations is that, while these implementations have good numerical accuracy, they are not able to meet the requirements of low power consumption and real-time processing. This occurs in some cases, such as in image processing for example. This fact motivated the implementations of the SVM in reconfigurable hardware.

Several works have implemented only the SVM inference. Patil et al. (2012); Hussain et al. (2013, 2014) implemented the inference phase using systolic arrays to accelerate calculations and reduce the FPGA usage area. Initially, Patil et al. (2012) proposed an implementation for the classification of facial images in order to differentiate six basic expressions: smile, surprise, sadness, anger, disgust and fear. The structure with a systolic array enabled the efficient use of memory and the low complexity of the proposed architecture. This work, however, did not focus on creating a structure with high throughput, but on using the same structure for several different binary classifications, resulting in a multi-class classification. To do this, the author used the partial Xilinx reconfiguration tool to dynamically modify logic blocks while the rest of the logic continued

running without interruptions. Moreover, Hussain et al. (2013) proposed an implementation with systolic arrays for the classification of DNA microarrays. This work, later expanded by the same authors in Hussain et al. (2014), also used partial reconfiguration as in Patil et al. (2012). The proposed architecture, however, focused on the acceleration of the classification of a small number of elements of enormous size, as is the case of DNA microarrays. In addition, the kernel used in this paper was the linear kernel. This kernel, although easily implemented in hardware, has a low capacity to classify complex problems.

Ruiz-Llata et al. (2010); Jallad & Mohammed (2014); Pan et al. (2013) implemented the same phase using the hardware-friendly kernel (HFK) obtaining significant reductions in the space used by the FPGA. This occurred since this kernel does not need the multiplication blocks present in conventional kernels. The HFK kernel, which was also used in this work, uses a structure very similar to the Gaussian kernel and performs its more complex operations on a CORDIC (COordinate Rotation DIgital Computer) containing only shifts and sums as shown in Section 4. The main contribution of Ruiz-Llata et al. (2010) was the introduction of an architecture for the inference phase that allowed both classification and regression reusing the same hardware components. The proposed hardware was used to classify between 4 different classes of 32-bit 32-pixel images in 8-bit gray scale. The regression was tested with the *sinc* function. The maximum clock reached by the implementation was 30 MHz, however several cycles were required for classification due to the HFK. In Jallad & Mohammed (2014), the developed hardware intended to be used in satellite image-based imaging applications. The main contribution of the work was the low area used due to the use of the HFK in combination with a control block. However, much of the computation was performed serially through the control block, making the classification take 90 clock cycles per pixel of the image to be analyzed. The hardware proposed in Pan et al. (2013) had its architecture similar to that proposed in Ruiz-Llata et al. (2010). However, the L1 norm of the HFK kernel was calculated in parallel rather than serially. Thus, the number of clocks required to compute the kernel was no longer dependent on the dimensionality of the

input.

A few other works have done the implementation of the training phase of the SVM. Ta-Wen et al. (2012) implemented SVM training using the SMO. The proposed architecture consisted of three main blocks that represented the main functions of the SMO and were controlled by a finite state machine (FSM). However, the large area gain that could be obtained with the HFK kernel was not taken into account. In addition, the kernel used was the linear kernel, limiting the classification capacity of the proposed architecture. The work presented in Ta-Wen et al. (2012) uses a semi-parallel implementation when it regards the implementation proposed in this paper.

Cao et al. (2010) also proposed an architecture based on the SMO. The main contribution of this work was the flexibility given the proposed structure, which was customizable between fully parallel and serial. The purpose of this architecture was to be used in embedded applications where fully parallel implementations exceed FPGA resources and serial implementations did not meet the temporal requirements of the problem.

In Bustio-Martínez et al. (2010) a hybrid software and hardware architecture was proposed to accelerate the SMO training. In the proposed implementation the SMO algorithm was partitioned so that the serial processing portion of the algorithm and the control signals were executed in a General Purpose Processor (GPP) while an FPGA performed tasks that could be executed in parallel (in this case, the vector product of the linear kernel). The proposed architecture with the co-processor obtained a speedup of 178.7× when compared to a software-only implementation in a GPP.

In Filho et al. (2010) a dynamically reconfigurable architecture was proposed for the training of the SVM using the SMO. The major contribution of this work was the proposal of a modular architecture capable of being reprogrammed during its execution. The kernel used was the HFK, providing reduction in area usage when compared to similar implementations using other kernels. The proposed architecture was able to achieve speedups of up to 12.5× when compared to implementations designed purely in software. In addition, a study of the nu-

merical accuracy required for the training using the proposed architecture was made, showing that 24 bits were sufficient to represent the input data (6 bits representing the integer part and the others 18 for the fractional part).

Venkateshan et al. (2015) proposed a different SVM training algorithm, known as the Hybrid Working Set (HWS). In the proposed architecture the FPGA was only used as co-processor of the kernel functions. In this implementation, the kernel used was the Gaussian kernel. The implementation of the FPGA co-processor achieved a speedup of 25× when compared to a purely software implementation.

An SMO IP core was developed in Madadum & Becerikli (2017) using high-level synthesizes (HLS) strategy. In this work, the SMO code was made in C++ and transform to VHDL using the Xilinx Vivado HLS. The SMO implementation had as target the xc7z020clg484-1 FPGA working at 100 MHz and it has a time per iteration of about 2.69 ms.

Finally, the work presented in Feng et al. (2018) proposed a hardware implementation of the modified SMO (MSMO) algorithm. The proposed scheme was synthesized to Altera Cyclone II FPGA with 50 MHz and used the fixed-point strategy with 1 sign bit, 3 integer bits, and 12 decimal bits. This work achieved a speedup of 69× when compared to a software-only implementation in a 3.7 GHz Dual-Core CPU (i3-4170).

1.2. Organization

In Section 2 a background about the SVM algorithm will be shown. Then, in Section 3, the Sequential Minimal Optimization algorithm will be presented. In this section, are going to be discussed the heuristics for the selection of the elements to be optimized, the calculation of the optimization of these elements, the calculation of the threshold of the SVM b, as well as methods for accelerating the SMO. In Section 4, the details of the proposed blocks will be presented in a generic way, allowing the implementation of the algorithm proposed regardless of the FPGA model. The results of specific tests showing the proper functioning of the proposed architecture, as well as the results of the synthesis showing

the features required for the proposed architecture in the target FPGA will be presented in Section 5. Finally, in Section 6, the general conclusions about the design implementation will be presented.

2. Support vector machine (SVM)

Support vector machines were initially developed as a binary classification Haykin (2008). In this type of problem, there is a training set $(\vec{x_i}, y_i)_N^{i=1}$, where x_i is the i-th element and y_i its respective class. Classes should always assume that $y_i = \pm 1$. The training for binary classification consists of solving the quadratic problem with linear constraints as presented in

$$\begin{aligned} Maximize \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\vec{x_i}, \vec{x_j}) \\ Subject\ to \quad & 0 \leq \alpha_i \leq C \quad for\ i = 1, 2, ..., N \\ & \sum_{i=1}^{N} \alpha_i y_i = 0, \end{aligned} \tag{1}$$

where $K(\vec{x_i}, \vec{x_j})$ represents the kernel function and C is a regularization parameter that acts as a limiting value on the Lagrange multipliers α_i.

The feed-forward classification phase, which consists of the classification of a new $\vec{x}$ vector, is given by

$$y(\vec{x}) = sgn\left(\sum_{i=1}^{N_{SV}} y_i \alpha_i K(\vec{x_i}, \vec{x}) + b \right), \tag{2}$$

where the α_i and b parameters are given in the training phase. All Lagrange multipliers, α_i, that have values other than zero are called Support Vectors (SV). It is also important to note that the sum of Equation (2) extends only to the support vectors number N_{SV}, rather than extending to all N points in the training set. The SVM classification inference architecture can be seen at the top of Figure 1. In it, the SVM inference is illustrated in conjunction with the training phase as seen in Equation 1.

In the regression, the problem is to estimate the value of a function for any point from a training set with a number of finite points. In a similar way to the

classification, there is as input of the training phase a set $(\vec{x_i}, y_i)_N^{i=1}$, where $\vec{x_i}$ is the i-th reference point and y_i its respective output. It is important to note that, unlike the classification, the values of y_i in the regression can assume any real value.

Therefore, including the parameter ε in the equation proposed by Vapnik Cortes & Vapnik (1995), the training phase of the regression is given by

$$\begin{aligned} \textbf{\textit{Maximize}} \quad & \sum_{i=1}^{N}(\alpha_i^* - \alpha_i)y_i - \varepsilon\sum_{i=1}^{N}(\alpha_i^* + \alpha_i) \\ & - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)K(\vec{x_i}, \vec{x_j}) \qquad (3) \\ \textit{Subject to} \quad & 0 \leq \alpha_i^*, \alpha_i \leq C \quad For\ i = 1, 2, ..., N \\ & \sum_{i=1}^{N}(\alpha_i^* - \alpha_i) = 0, \end{aligned}$$

where, as in the classification, $K(\vec{x_i}, \vec{x_j})$ represents the kernel function and C is a regularization parameter that acts as a limiting value on the Lagrange multipliers α_i. ε is a positive parameter that defines the insensible zone within which errors are ignored. Note that both C and ε are constants defined before the training, according to the characteristics of the input set.

The feed-forward regression phase, which consists in estimating the value of a new point x, and is given by

$$y(x) = \sum_{i=1}^{N_{SV}}(\alpha_i^* - \alpha_i)K(\vec{x_i}, x) + b, \qquad (4)$$

where the α_i^*, α_i and b parameters are the result of the training phase. In addition, as in the classification, only the support vectors are used for the summation.

3. Sequential minimal optimization (SMO)

3.1. Karush-Kuhn-Tucker conditions (KKT)

The training of a SVM requires the solution of a quadratic programming optimization (QP) problem as shown in Equation (1). SMO breaks this big

QP problem into a series of QP problems of the smallest possible size Platt (1999). These small QP problems can be solved analytically, which avoids costly numerical computation of the major QP problems.

A point is an optimal point from Equation (2) if and only if the KKT conditions are met. The KKT conditions are given by

$$\begin{aligned} \alpha = 0 \quad &\Rightarrow \quad y_i f(\vec{x_i}) \geq 1 \\ \alpha = C \quad &\Rightarrow \quad y_i f(\vec{x_i}) \leq 1 \\ 0 < \alpha < C \quad &\Rightarrow \quad y_i f(\vec{x_i}) = 1, \end{aligned} \tag{5}$$

where f is the SVM inference phase. Thus, the SMO iterates until these conditions are met, guaranteeing the convergence of the algorithm.

In order to find this optimal point, the SMO selects through a heuristic a pair of αs (α_i and α_j) to optimize, then optimizes them, recalculates the value of b according to the new αs and repeats the process until the KKT conditions are met (within a certain tolerance).

3.2. Heuristics for αs selection

Part of the SMO algorithm is dedicated to choosing the pair of αs to be optimized. In the literature, there are several different ways of choosing these elements to be optimized. However, there is no "wrong" way to make that choice, however the order of these choices can change the speed of SMO convergence. There are two heuristics used in choosing the Lagrange multipliers, one for the first Lagrange multiplier in an external loop (α_i) and other one for the second in an inner loop (α_j).

In the heuristic of the first choice (external loop) an iteration is performed throughout the entire training set. When an element that does not obey the KKT conditions is found, this element is selected as α_i to be optimized. In order to accelerate the SMO, for each loop performed in the entire data set, it is executed a loop only with the elements whose Lagrange multipliers have no value 0 or C (unlimited multipliers).

The heuristic of the second choice (internal loop), in turn, consists in choosing as α_j the element that makes possible the biggest step, being the error given

by

$$E_i = f(\vec{x_i}) - y_i \tag{6}$$

and the step given by

$$|E_i - E_j|. \tag{7}$$

If the α_j chosen with this heuristic does not result in any convergence improvement, a loop is performed on all unlimited αs until the first one results in some convergence improvement. Finally, if none of them results in any improvement, a loop is executed throughout the entire training set.

3.3. α_i and α_j Optimization

Once the values of α_i and α_j to be optimized are chosen, it is necessary to find the margins Lo and Hi such that $Lo \leq \alpha_j \leq Hi$ so as to make α_j meet the constraint $0 \leq \alpha_j \leq C$ after the optimization . If y_i and y_j have different values

$$Lo = max(0, \alpha_j - \alpha_i), \tag{8}$$

$$Hi = min(C, C + \alpha_j - \alpha_i) \tag{9}$$

and if y_i and y_j have the same value

$$Lo = max(0, \alpha_i + \alpha_j - C), \tag{10}$$

$$Hi = min(C, \alpha_i - \alpha_j). \tag{11}$$

The next step is to find the value of α_j that maximizes the Equation (2). The optimal point α_j is given by

$$\alpha_j^{new} = \alpha_j - \frac{y_j(E_i - E_j)}{\eta} \tag{12}$$

where

$$\eta = 2K(\vec{x_i}, \vec{x_j}) - K(\vec{x_i}, \vec{x_i}) - K(\vec{x_j}, \vec{x_j}). \tag{13}$$

If the value of α_j is not between Lo and Hi, the value of α_j is saturated

$$\alpha_j^{new} = \begin{cases} Hi & if\ \alpha_j^{new} > Hi \\ \alpha_j^{new} & if\ Lo \leq \alpha_j^{new} \leq Hi \\ Lo & if\ \alpha_j^{new} < Lo. \end{cases} \tag{14}$$

Finally, once the value of α_j has been found, one can find α_i using

$$\alpha_i^{new} = \alpha_i + y_i y_j (\alpha_j - \alpha_j^{new}). \tag{15}$$

It is important to note that α_i is only calculated when the value of α_j undergoes a minimally considerable change. Thus, α_i is calculated only when

$$|\alpha_j - \alpha_j^{new}| \geq \varepsilon(\alpha_j + \alpha_j^{new} + \varepsilon), \tag{16}$$

where the positive parameter ε defines the insensible zone within which errors of the KKT conditions are ignored. This parameter usually has a value between 10^{-2} and 10^{-3}.

3.4. Calculating the b threshold

Only after optimizing α_i and α_j is it possible to select the threshold b to satisfy the KKT conditions. The threshold b_1 is given by

$$\begin{aligned} b_1 = &-E_i - y_i(\alpha_i^{new} - \alpha_i)K(\vec{x_i}, \vec{x_i}) \\ &- y_j(\alpha_j^{new} - \alpha_j)K(\vec{x_i}, \vec{x_j}) + b \end{aligned} \tag{17}$$

and it is valid when α_i is not within its limit (that is, $0 \leq \alpha_i \leq C$), because it forces the correct SVM output when the input is $\vec{x_i}$. The threshold b_2 given by

$$\begin{aligned} b_2 = &-E_j - y_i(\alpha_i^{new} - \alpha_i)K(\vec{x_i}, \vec{x_j}) \\ &- y_j(\alpha_j^{new} - \alpha_j)K(\vec{x_j}, \vec{x_j}) + b \end{aligned} \tag{18}$$

and it is valid when α_i is not within its limit because it forces the correct SVM output when the input is $\vec{x_j}$. If both b_1 and b_2 are valid, they will have the same value. If the new αs are both at the edges (i.e., $\alpha_i = 0$ or $\alpha_i = C$ and $\alpha_j = 0$ or $\alpha_j = C$) then all values between b_1 and b_2 satisfy the KKT conditions. In these cases b is chosen as the intermediate value between b_1 and b_2. Thus, b is given by

$$b = \begin{cases} b_1 & if\ 0 < \alpha_i < C \\ b_2 & if\ 0 < \alpha_j < C \\ (b_1 + b_2)/2 & otherwise. \end{cases} \tag{19}$$

3.5. Error cache

The error cache refers to a SMO acceleration technique that consists in the continuous error storage of all the unlimited elements of the training set. This technique is advantageous, since, due to the linearity of the feed-forward phase of the classification presented in Equation (2), it is possible to keep the error always updated by doing the calculation

$$E_k^{new} = E_k + y_i(\alpha_i^{new} - \alpha_i)K(\vec{x_i}, \vec{x_k}) + y_j(\alpha_j^{new} - \alpha_j)K(\vec{x_j}, \vec{x_k}) - b + b^{new} \quad (20)$$

only when some α is optimized.

The continuous calculation of the error avoids the extremely costly feed-forward calculation phase of the classification, which would occur every time that the error of any point in the input set needed to be calculated.

4. SMO Implementation

4.1. General structure implementation

The general structure of the proposed architecture for the SMO is present in Figure 2. This structure was developed according to the SMO algorithm proposed in Section 3 and it is formed by four main layers that will be explained in detail in the following sections. The structure of Figure 2 corresponds to the bottom of Figure 1.

In Figure 2, the notation $[n.b]$ indicates that n bits are used in that bus, of which b bits are used for the integer part representation and $n - b$ bits are used for the integer part representation. The signals that are not labeled are binary signals, that is, [1.0]. The logic operation of the other layers that make up the structure in discussion will be explained in the subsections 4.2, 4.3, 4.4 and 4.5.

4.2. Hardware friendly kernel (HFK)

A kernel is a similarity function between two or more elements. Although the polynomial and Gaussian kernels are the most common, they were not developed focusing on taking advantage of the features present in reconfigurable architectures. A list of common kernels can be found in Table 1.

In order to solve this problem, the hardware friendly kernel (HFK) was proposed in Anguita et al. (2006). This kernel generates a resemblance function similar to the Gaussian kernel. The HFK has the advantage of being implemented in hardware only with shifts and additions rather than multiplications. This makes the area occupied by the HFK much smaller than the area occupied by the other kernels shown in Table 1.

It is important to note that since in the sequential minimum optimization the elements to be optimized are called $\vec{x_i}$ and $\vec{x_j}$, the kernel friendly hardware equation can be rewritten as

$$K(\vec{x_i}, \vec{x_j}) = 2^{-\gamma||\vec{x_i} - \vec{x_j}||_1}, \tag{21}$$

where $||\vec{x_i} - \vec{x_j}||_1$ is the norm L1 of the difference between the vectors $\vec{x_i}$ and $\vec{x_j}$. γ is a constant that depends on the dimension dim of these vectors and is given by

$$\gamma = 2^{-log_2(dim)}. \tag{22}$$

For example, if $\vec{x_i} \in \Re^2$, so $\gamma = 2^{-1}$.

Figure 3 shows an overview of the hardware friendly kernel implementation. Initially, the initial CORDIC error from block Ec_1 is calculated, which is given by

$$Ec_1 = -\gamma||\vec{x_i} - \vec{x_j}||_1. \tag{23}$$

The CORDIC (COordinate Rotation DIgital Computer) algorithms are part of a class of algorithms that replace complex operations, such as multiplications by shifts and sums.

Then E_1 is divided between its integer part I and its fractional part F in the FI block. The CORDIC block then receives the fractional part F and performs

the operation $B_1 2^F$, where B_1 is a value to be multiplied by the kernel which can be useful as shown in 4.5. Finally, the value of $B_1 2^F$ is shifted to the right I times, resulting in the final value of the kernel, since $B_1 K(\vec{x_i}, \vec{x_j}) = B_1 2^{Ec_1} = B_1 2^{F+I} = B_1 2^F 2^I$.

The calculation of E_1, in turn, is executed as shown in Figure 4. First, the subtraction of each dimension of $\vec{x_i}$ by its dimension in $\vec{x_j}$ is performed. Then, the magnitude value of each of these elements is calculated and a tree sum is executed. Finally, the value is shifted to the right q times, where $q = log_2(\gamma)$ and then its signal is inverted. Thus, the calculation of Ec_1 is performed in only one cycle in a parallel configuration and without using multiplications. However, it is important to note that if q is not an integer value, it is necessary to replace that block with a multiplication by γ.

The CORDIC block responsible for calculating $B_1 \times 2^F$ is illustrated in Figure 5.

The CORDIC block is responsible for the iterative calculation of equations

$$B_{i+1} = B_i(1 + d_i 2^{-i}) \tag{24}$$

and

$$Ec_{i+1} = Ec_i - log_2(1 + d_i 2^{-i}), \tag{25}$$

where d_i can assume the values of 0, $+1$ and -1 and it is chosen such that $|Ec_{i+1}| \leq |Ec_i|$, making the value of E_i decreases over the iterations. When $Ec_i \to 0$ then $B_i \to B_1 2^{Ec_1}$, which happens after b iterations as proved in Anguita et al. (2006).

Equation 24 is calculated using the sum of B_i with the result of a Barrel-Shifter, which is responsible for shifting the value of B_i to the right i times. The Ec_i value is calculated by subtracting Ec_i by the values of $log_2(1 + 2^{-i})$ saved in a ROM within each CORDIC block.

It is important to note that the value of Ec_1 is always negative, as shown in the Equation 23. Thus, in order to simplify the proposed hardware, the integer I will always be positive and the fractional part F always negative. This allows d_i to always be 0 or $+1$, avoiding the storage of $log_2(1-2^{-i})$ values in the ROM.

4.3. α_i and α_j Optimization

The calculation of α_i and α_j is performed as discussed in Section 3.3 An overview of the α values optimization implementation is illustrated in Figure 6.

It is important to note that in Figure 6 only the calculation of a kernel is required. This is because when $\vec{x_i}$ equals $\vec{x_j}$, the result of the HFK kernel is 1. Thus, Equation 13 can be rewritten as

$$\eta = 2K(\vec{x_i}, \vec{x_j}) - K(\vec{x_i}, \vec{x_i}) - K(\vec{x_j}, \vec{x_j}) = 2K(\vec{x_i}, \vec{x_j}) - 2. \tag{26}$$

The block a_j is composed of two simple subtractions and one division. This block receives as input the old value of α_j and y_j and the E_i and E_j errors. Then, the new value of a_j is calculated using the equation 12 with η being calculated according to Equation 26.

The calculation of the lower limit Lo and the upper limit Hi are performed in the Lo\Hi block as shown in Figure 7. The calculation of these constraints is executed as described in Equations 8, 9, 10 and 11.

After calculating Lo and Hi the value of α_j is limited using these values in the Limit block as shown in Equation 14. Simultaneously, in the check status block, it is checked whether it is advantageous to continue the optimization. For this, three conditions are verified: if i is equals to j, if Lo is equals to Hi and if α_j has not changed significantly. If any of these conditions are true, a pulse will be emitted to the skip output instead of the done output, causing the iteration to be discarded and new values of i and j chosen for the optimization.

In addition, as seen in Figure 6, the value of α_i is also calculated after calculating α_j. This value is calculated as shown by the Equation 15.

4.4. Bias optimization

After optimizing the values of α_i and α_j it is necessary to update the bias value. An overview of this calculation is illustrated in Figure 8. This Figure shows the selection between the values of $b1$ and $b2$ as described in Equation 19.

The calculation of the values of $b1$ and $b2$ is illustrated in Figure 9. In this block, the calculation of $b1$ is performed as described in Equation 17 and the calculation of $b2$ is performed as described in Equation 18. It is important to note that most of the operations required to calculate $b1$ and $b2$ are identical in both equations, which makes it possible to reuse these signals in hardware.

4.5. Data storage and error calculation

After optimizing the values of α_i and α_j and adjusting the value of the *bias* it is necessary to store this data for the next iteration. However, before that, a new error value is calculated for each of the elements of the training set. In the proposed implementation, the calculation of all the errors is performed in parallel, since the used HFK kernel does not need multiplication blocks and occupies a very small area when compared to the polynomial kernel and Gaussian kernel. As discussed in Section 3.5, the cache error was created in order to avoid the costly feed-forward phase calculation during the optimization of α_i and α_j. The general architecture of the storage block is illustrated in Figure 10.

As shown in Figure 10 both the values of the input set elements and the class to which these elements belong are stored in ROMs, since those values are not overwritten during training. In a scenario where the same hardware will be used to train several data, it will be necessary to replace these ROMs with RAMs. The values of the Lagrange multipliers and the *bias* are stored in RAMs as well. It is also important to note that the value of these blocks are only updated after the errors calculation are completed.

Initially, in the block that calculates the errors, the values that are common to the calculation of all errors of the equation 20 as shown in Figure 11 are calculated. These values are:

- The subtraction of the new value of *bias* by the old value of *bias*
- A multiplication of the i class by a subtraction of the new value of α_i by the old value of α_i

- A multiplication of the j class by a subtraction of the new value of α_j by the old value of α_j.

It is important to note that since classes can only assume the value of $+1$ or -1, the multiplications in question can be replaced by simple conditional inversions.

The error calculation is performed by N blocks as shown in Figure 12, where N is the number of elements of the training set. Then, the outputs of the i-th and j-th errors (E_i and E_j) are selected using a N-input multiplexer.

The individual error calculation illustrated in Figure 12 consists mainly of the computation of two kernel functions: $k(i,k)$ and $k(j,k)$, where k is the error index to be calculated ranging from 1 to N. Since each block is always responsible for calculating the same k-th error (E_k), the value of the k element is saved to a register within the error calculation block itself. Finally, after calculating the error according to Equation 20, the value of E_k is stored in a register.

5. Results

5.1. Methodology

This section aims to present the tests and results of the architecture described in Section 4. In order to test the implemented hardware, the exclusive OR (XOR) gate and the Fisher's Iris data set were trained. In this data set, presented in Fisher (1950), the 'setosa' and 'versicolor' types of plants of the Iris genus are distinguished according to the height and width of both petal and the sepal.

The XOR logic gate was chosen as a test case for classification because it is a simple example, common in the literature Gu & Han (2013) and not linearly separable. The Fisher's Iris data set was chosen to test the classification of a three-dimensional data set, thus serving as a test to verify that the implemented structure is able to perform satisfactorily in more challenging cases.

The final values of the *bias* as well as the value of the Lagrange multipliers for each of the elements of the input set are stored and processed, allowing

the generation of graphs and numerical data that prove the effectiveness of the proposed architecture in the classification training.

As in Section 4, a usage analysis of the target FPGA Virtex 6 xc6vlx240t-1ff1156 is also performed based on the report generated during the proposed architecture synthesis.

5.2. Training results

After training the XOR function using the proposed hardware structure, the bias and support vectors SV, the Lagrange multipliers α were found. Using these values and calculating the classification feed-forward phase presented in Equation 2 it was possible to find the curve that delimits the two classes.

Looking at Figure 13, it is possible to see that the four points training sets became support vectors. In addition, it is also noted that the classification curve generated by the implemented algorithm separates the classes maximizing the borders.

Figure 14, shows the result of the second proposed test, where the class 'setosa' is represented by class +1 and the 'versicolor' and 'virginica' classes are represented by class -1 . For the proposed test, the first 128 elements of this set were used, as well as the first three dimensions of each of these elements.

The results of the second proposed test, as seen in Figure 14, show that the endpoints and those closest to the opposite class became supporting vectors. Visually it is noticed that the support vectors generated a classification curve capable of dividing the two classes in order to maximize the edges between them, thus fulfilling the objective proposed by the SVM.

In both proposed tests the KKT conditions were satisfied, allowing the algorithm convergence. Thus, it was possible to perceive that the hardware presented in this work is functional and therefore a valid alternative for the training of the SVM using SMO.

5.3. Synthesis results

The Table 2 shows the synthesis result for a Virtex FPGA 6 xc6vlx240t-1ff1156 regarding the architecture proposed in Section 4.

As can be seen in Table 2, the proposed hardware synthesis was accomplished using both 32 bits (with 24 bits for the fractional part) and 16 bits (with 10 bits for the fractional part). In addition, tests were conducted with the proposed hardware supporting up to 8, 16, 32 or 64 elements in the training set.

It is important to note that for each training cycle (or training iteration), a maximum of

$$2b + w \tag{27}$$

clock cycles are required, where w is the number of clock cycles required for the optimization splitting block of α_i and α_j added to the overhead of the control signals between the blocks. The value of w varies with the numerical precision of the implementation and usually has a value between 6 and 10. $2b$ is the number of clocks needed to calculate the kernel during the optimization of α_i and α_j and during the error calculation. However, many times less clocks than $2b + w$ are required, since the proposed hardware is able to skip iterations that do not contribute significantly to the training convergence, as discussed in Section 4.Based in Equation 27, the throughput of the system, th, in iteration per second (IPS) can be expressed as

$$th = (2b + w) \times f_c \tag{28}$$

where the f_c is the clock frequency in cycles number per second.

Using the polynomial kernel, the number of multipliers did not grow with the increase of the number of elements. This behavior is expected, since the used HFK kernel was implemented without using any multiplication. This feature shows one of the great advantages of using HFK rather than the polynomial kernel. It is observed that the proposed hardware is highly parallelizable since there is no performance loss with the increase of the maximum number of elements of the training set. This behavior was expected, since the implementation bottleneck is in the calculation of the error, which was executed entirely in a parallel configuration.

Figure 15 shows the regression analysis ($R^2 = 1$ for both, 16.10 and 32.24 bits) of the occupation area to the number of logic cells, N_{LC}, and the number

of elements, N, of the SMO algorithm. Using the regression analysis can be observed that the N_{LC} has a linear growth for both, 16.10 and 32.24 bits and the regression equations can be expressed as

$$\tilde{N}_{LC}^{16} = 776.4N + 1102 \tag{29}$$

and

$$\tilde{N}_{LC}^{32} = 1620N + 2243 \tag{30}$$

where $\tilde{N}_{LC}^{16}$ and $\tilde{N}_{LC}^{32}$ are the estimates of the logic cells number for 16.10 and 32.24 bits, respectively. R-squared, R^2, is a statistical measure (also called coefficient of determination) of how close the data are to the fitted regression curve. The R^2 varies between 0 and 1, indicating, in percentage, how much the model can explain the observed values. The higher the R^2, the more explanatory the model is, the better it fits the sample.

Based in the Equations 29 and 30 observe that there is a linear relation with the number of bits, n, enabling the creation of a two-dimensional regression analysis (see Figure 16) expressed as

$$\tilde{N}_{LC} = 1198N + 1653n - 38002 \tag{31}$$

where $\tilde{N}_{LC}$ is the estimates of the logic cells number on FPGA.

Figure 17 shows the scalability of the proposed hardware regard of the number of elements, N. The curves were built based in Equations 29, 30 and 32 show the logic cells required for high element number (until 10000). Figure 17 also shows the estimate of the elements number for some commercial FPGAs XILINX Virtex-6 (2018); XILINX Virtex-7 (2018); XILINX Virtex Ultra (2018). The resume of the values about the maximum number of logic cells and the elements number are presented in Table 3.

Finally, Figure 17 shows the two-dimensional regression analysis regard of the throughput in KIPS ($\times$1000 iteration per second). Two-dimensional regression equation found with $R^2 = 0.9993$ can be expressed as

$$\tilde{th} = -0.3071N - 44.58n + 1859 \tag{32}$$

where $\tilde{th}$ is the estimates of the throughput in KIPS. It is essential to observe that throughput is more influenced by the number of bits than the number of elements and this is a crucial result given that the number of bits can be customized for each application. Works in the literature have found that a wide variety of applications can work with 16 bits of fixed-point resolution. Works in the literature have found that a wide variety of applications can be designed with a fixed-point resolution between 16 and 20 bits.

5.4. Real-world experiments

In order to illustrate the applicability of the hardware architecture of the SVM training presented in this work, it was estimated the convergence time of the algorithm for three examples from the literature. The first example called 'Iris' is the full version of the Matlab 'fisheriris' data set (used as an example in Subsection 5.2). As seen in Section 5, this data set was presented in Fisher (1950) and it distinguishes between the 'setosa' and 'versicolor' types of plants of the genus Iris according to height and width both the petal and the sepal. The second data set, presented in Street et al. (1993) and titled Breast Cancer Wisconsin, aims to classify binary tumors as malignant or benign using 10 parameters obtained from 569 different tumors. These parameters include radius length, texture, perimeter, area, among others. Finally, the third data set, called Banknote Authentication, is used to classify 1372 banknotes between real or false as seen in Lohweg et al. (2013). For this, 4 attributes are used after the wavelet transform of the original image: the transformed image variance, the obliquity of the transformed image, the kinetosis of the transformed image and the entropy of the original image.

The estimated convergence time for the cited data sets can be seen in the Table 4. The first column contains the reference of the data set used. In the second and third columns are presented the dimensions of the problem (number of elements and dimensionality of each entry). In the next column the clock obtained by extrapolating the Table 2, in the case the SVM training was implemented in the proposed architecture on a sufficiently large hardware,

is presented. In the fifth column are presented the approximate numbers of iterations necessary for the problems, according to their dimensions, to converge. In the last column, from the number of iterations and the maximum sampling frequency, an estimation of the convergence time of the mentioned problems is displayed. The data presented in Table 4 were generated according to a 16-bit architecture, 10 bits for the fractional part. Thus, 30 clocks per iteration of the training phase are required. The examples set forth herein are also available from the UCI Machine Learning Repository.

In a problem such as the 'Iris' data set, approximately 500 iterations were required for the convergence of the algorithm. Thus, given the maximum frequency of 33 MHz, the final result was available after 0.45ms. In the case of the Breast Cancer data set, with a clock of approximately 27 MHz, approximately 5200 iterations were required, which caused the convergence to occur in approximately 5.2ms. Finally, in the 'Banknote Authentication' data set training, a maximum frequency of 30 MHz was achieved, to run approximately 17000 iterations. In this way, the training result was available after approximately 17ms.

5.5. Comparisons with state of the art works

By extrapolating the data from Table 2 it is also possible to compare the proposed architecture with other implementation proposals in the literature. Table 5 resumes the comparative results, and it shows the throughput in KIPS ($\times$1000 iteration per second) for each work and the speedup achieved regard the implementation proposed in this work.

In the hardware presented in Filho et al. (2010), for example, approximately 100000 clock cycles were required for each iteration of the classification of the 'Breast Cancer' data set using 20-bit components. This is because the implementation in question serializes parallelizable parts of the code to save area. Thus, even if this implementation was executed at 100 MHz, as described in Filho et al. (2010), it achieved 1 ms per iteration (th = 1 KIPS). Considering a 32.24 bits numerical precision and a maximum clock rate of 24.385 MHz as

seen in Table 2, the implementation proposed in the current work would achieve 58 clocks per iterations ($2b + w$, $b = 24$ and $w = 10$) or 2.38 μs per iteration ($th = 420$ KIPS), and with that a speedup of $\approx 420\times$.

When compared to the architecture presented in Cao et al. (2010), the proposed architecture presented a speedup of $\approx 67\times$ in the 'Breast Cancer' data set, assuming a maximum clock of 24.385 MHz in a numerically accurate implementation of 32.24 bits (2.38 μs per iteration or $th = 420$ KIPS). According to the equations presented in the paper, the approximate number of clocks per cycle for this data set would be approximately 12000 iterations and the maximum frequency of the FPGA would be 75 MHz (160 μs per iteration or $th = 6.25$ KIPS). The acceleration obtained in comparison to this work is due to the fact that the number of clocks do no depend on the size of the input in the implementation proposed here, since the kernel calculation involving the dimensions (Block E_{c1} of HFK) is executed in a completely parallel configuration.

When compared to Bustio-Martínez et al. (2010), which used the FPGA only as kernel functions accelerators, the speedup of the present work was $\approx 48\times$. In this work, the maximum clock reached was 35 MHz when classifying a data set with dimensionality 14 and computing the linear kernel in only 3 clocks. However, after the hardware implementation of the kernel, the implementation bottleneck became the other part of the SMO algorithm executed in software. In this algorithm, each iteration (software + hardware) took an average of 115 μs ($th = 8.69$ KIPS), while in the implementation of the present work it would take approximately 2.38 μs ($th = 420$ KIPS) considering a clock of 24.385 MHz in an implementation with 32.24 bits.

The work presented in Ta-Wen et al. (2012) achieved 2.2 ms per iteration ($th = 0.45$ KIPS) considering a clock of 50 MHz and 8.15 bits format precision. Comparing Ta-Wen et al. (2012) with implementation proposed here (considering a clock of 24.385 MHz and 32.24 bits) there is a speedup about $\approx 933\times$. In Venkateshan et al. (2015) a co-processor of the kernel functions on FPGA is proposed and in this case, the co-processor achieved 3.14 ms per iteration ($th = 0.32$ KIPS) for the SensIT dataset, and 32 bits float-point format preci-

sion. Using the similar configuration, the strategy proposed here has 2.38 μs per iteration ($th = 420$ KIPS), and this is equivalent to a speedup of $\approx 1312\times$. An equivalent speedup, $\approx 1135\times$, it was found comparing the work presented in Madadum & Becerikli (2017). This work achieved a time per iteration of about 2.69 ms ($th = 0.37$ KIPS) considering a clock of 100 MHz and 32 bits float-point format precision.

Finally, the propose presented in Feng et al. (2018) achieved a time per iteration of about 87 μs ($th = 11.49$ KIPS) considering a clock of 50 MHz and 16.12 fixed-point bits format precision. When compared to Feng et al. (2018), the speedup of the present work was $\approx 98\times$. In this case, it was considered a clock of 33.383 MHz and 16.10 bits, in other words, 30 clocks per iteration or 0.89 μs per iteration ($th = 1123$ KIPS).

6. Conclusion

This work proposed a hardware parallel architecture of the Sequential Minimal Optimization (SMO) algorithm on FPGA. The SMO technique is one of the ways to the training phase of the Support Vector Machine (SVM) networks. A detailed analysis of the implementation was conducted, in addition to the analysis of simulation and synthesis data.

From the simulation data, the architecture was validated and the analysis of the synthesis data allowed to verify the behavior of the system regarding essential parameters, such as occupation area and throughput in iteration per second. By observing FPGA synthesis performed it was verified that with the development of this algorithm, directly in hardware, it is possible to reach high performance, especially regarding throughput when compared with other solutions in the literature.

Funding

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001.

Acknowledgments

The authors wish to acknowledge the financial support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for their financial support.

References

Anguita, D. et al. (2006). Feed- forward support vector machine without multipliers. *IEEE Transactions on Neural Networks*, *17*, 1328–1331.

Bustio-Martínez, L., Cumplido, R., Hernández-Palancar, J., & Feregrino-Uribe, C. (2010). Advances in pattern recognition. chapter On the Design of a Hardware- Software Architecture for Acceleration of SVM's Training Phase. (pp. 281–290). Springer.

Cao, K., Shen, H., & Chen, H. (2010). A parallel and scalable digital architecture for training support vector machines. *Journal of Zhejiang University SCIENCE C*, *11*, 620–628.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Kluwer Academic Publishers*, .

Coutinho, M. G. F., Torquato, M. F., & Fernandes, M. A. C. (2019). Deep neural network hardware implementation based on stacked sparse autoencoder. *IEEE Access*, *7*, 40674–40694. doi:10.1109/ACCESS.2019.2907261.

Da Costa, A. L. X., Silva, C. A. D., Torquato, M. F., & Fernandes, M. A. C. (2019). Parallel implementation of particle swarm optimization on fpga. *IEEE Transactions on Circuits and Systems II: Express Briefs*, (pp. 1–1). doi:10.1109/TCSII.2019.2895343.

Da Silva, L. M. D., Torquato, M. F., & Fernandes, M. A. C. (2019). Parallel implementation of reinforcement learning q-learning technique for fpga. *IEEE Access*, *7*, 2782–2798. doi:10.1109/ACCESS.2018.2885950.

Feng, L., Li, Z., & Wang, Y. (2018). Vlsi design of svm-based seizure detection system with on-chip learning capability. *IEEE Transactions on Biomedical Circuits and Systems*, *12*, 171–181. doi:10.1109/TBCAS.2017.2762721.

Filho, J. G., Raffo, M., Strum, M., & Chau, W. J. (2010). A general-purpose dynamically reconfigurable svm. *2010 VI Southern Programmable Logic Conference (SPL)*, (pp. 107–112).

Fisher, R. (1950). The use of multiple measurements in taxonomic problems. *Contributions to Mathematical Statistics*, .

Gu, Q., & Han, J. (2013). Clustered support vector machines. *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics (AISTATS)*, .

Haykin, S. (2008). *Neural Networks and Learning Machines*. Pearson Prentice Hall.

Huang, H., & Liu, H. (2014). Big data machine learning and graph analytics: Current state and future challenges. *IEEE International Conference on Big Data*, .

Hussain, H., Benkrid, K., & Seker, H. (2014). Novel dynamic partial reconfiguration implementations of the support vector machine classifier on fpga. *Turkish Journal of Electrical Engineering and Computer Sciences*, .

Hussain, H. M., Benkrid, K., & Seker, H. (2013). Reconfiguration- based implementation of svm classifier on fpga for classifying microarray data. *35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, (pp. 3058–3061).

Jallad, A. H. M., & Mohammed, L. B. (2014). Hardware support vector machine (svm) for satellite on-board applications. *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, (pp. 256–261).

Lohweg, V. et al. (2013). Banknote authentication with mobile devices. *Media Watermarking, Security, and Forensics*, .

Madadum, H., & Becerikli, Y. (2017). The implementation of support vector machine (svm) using fpga for human detection. In *2017 10th International Conference on Electrical and Electronics Engineering (ELECO)* (pp. 1286–1290).

Pan, X., Yang, H., Li, L., Liu, Z., & Hou, L. (2013). Fpga implementation of svm decision function based on hardware-friendly kernel. *International Conference on Computational and Information Sciences - ICCIS 2013*, (pp. 133–136).

Patil, R., Gupta, G., Sahula, V., & Mandal, A. (2012). Power aware hardware prototyping of multiclass svm classifier through reconfiguration. *25th International Conference on VLSI Design (VLSID)*, (pp. 62–67).

Platt, J. (1999). Fast training of support vector machines using sequential minimal optimization. *Advances in Kernel Methods—Support Vector Learning*, *3*.

Ruiz-Llata, M., Guarnizo, G., & Yébenes-Calvino, M. (2010). Fpga implementation of a support vector machine for classification and regression. *IEEE World Congress on Computational Intelligence*, .

de Souza, A. C., & Fernandes, M. A. (2014). Parallel fixed point implementation of a radial basis function network in an fpga. *Sensors*, *14*, 18223–18243.

Street, W. et al. (1993). Nuclear feature extraction for breast tumor diagnosis. *1993 International Symposium on Electronic Imaging: Science and Technology*, .

Ta-Wen, K., Jhing-Fa, W., Jia-Ching, W., Po-Chuan, L., & Gaung-Hui, G. (2012). Vlsi design of an svm learning core on sequential minimal optimization algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *20*, 673–683.

Torquato, M. F., & Fernandes, M. A. (2019). High-performance parallel implementation of genetic algorithm on fpga. *Circuits, Systems, and Signal Processing*, (pp. 1–26).

Venkateshan, S., Patel, A., & Varghese, K. (2015). Hybrid working set algorithm for svm learning with a kernel coprocessor on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *23*, 2221–2232. doi:10.1109/TVLSI.2014.2361254.

XILINX Virtex-6 (2018). Virtex-6 Family Overview. https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf. [Online; accessed 24-Dec-2018].

XILINX Virtex-7 (2018). 7 Series FPGAs Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. [Online; accessed 24-Dec-2018].

XILINX Virtex Ultra (2018). UltraScale Architecture and Product Data Sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. [Online; accessed 24-Dec-2018].

Zepeda, J., & Perez, P. (2015). Exemplar svms as visual feature encoders. *CVPR*, .

Table 1: Common kernels.

Kernel	Details*
Polynomial	$\left[(x^T x_i) + 1\right]^d$
Gaussian	$e^{\left(-\|x_i - x\|^2 / 2\theta^2\right)}$
Hardware-friendly	$2^{-\gamma \|x_i - x\|_1}$
Linear	$x^T x_i$
MLP	$tanh\left(k\, x^T x_i + \theta\right)$

* d, θ and k are user-defined parameters.

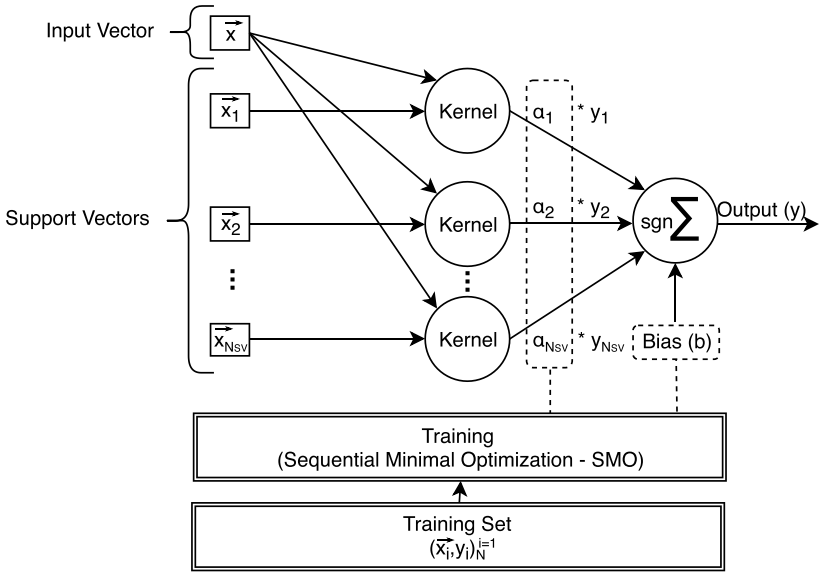

Figure 1: General architecture of SVM inference and training.

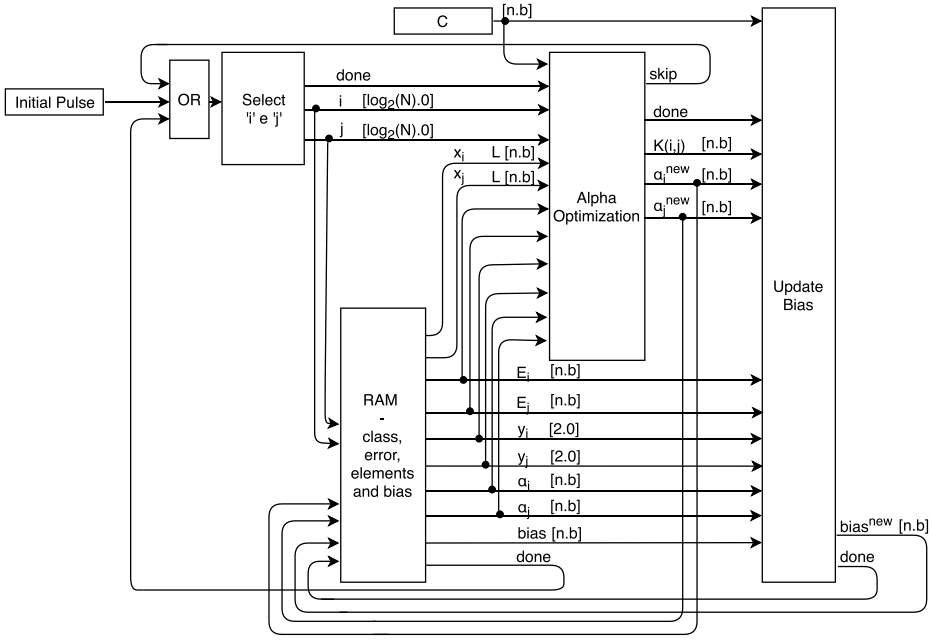

Figure 2: SMO general structure.

Figure 3: HFK Architecture.

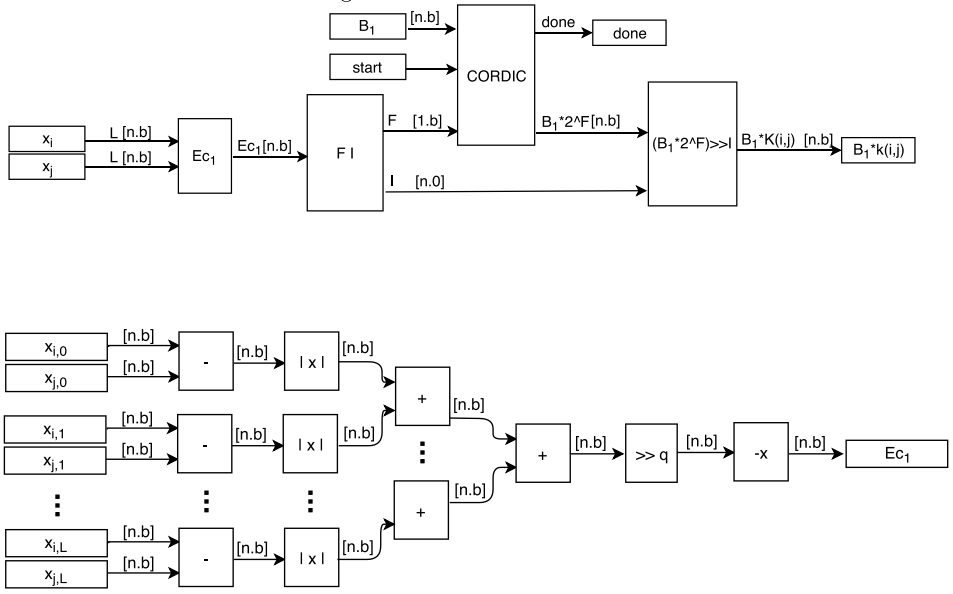

Figure 4: Ec_1 Block from HFK.

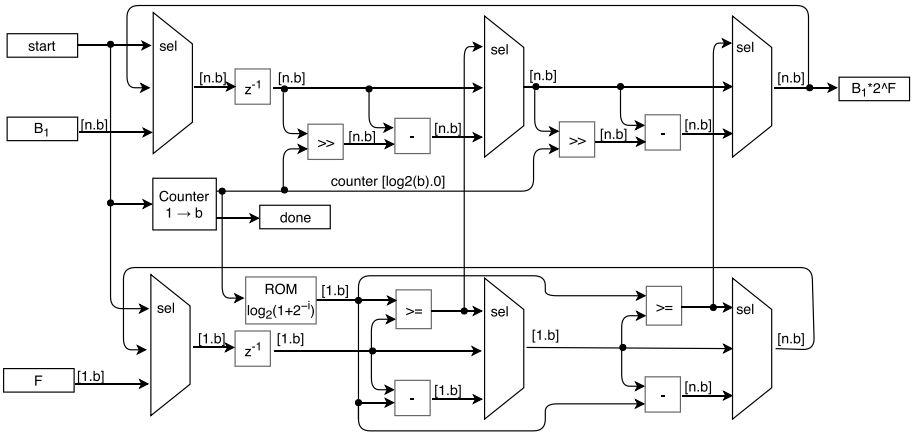

Figure 5: CORDIC Block from HFK.

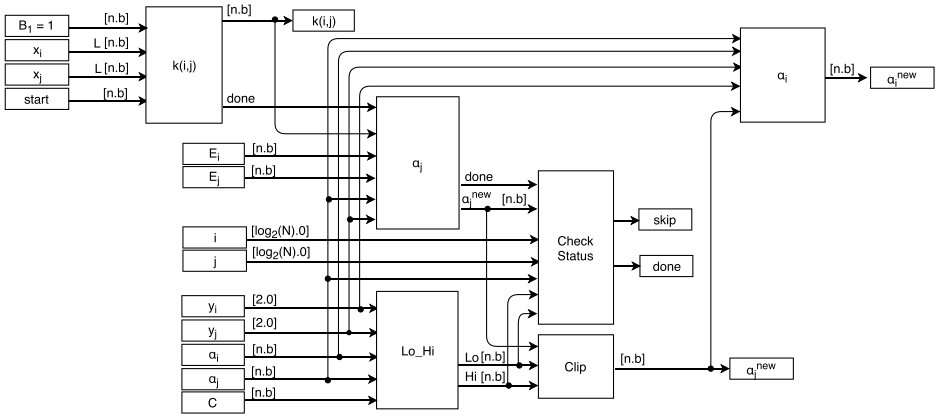

Figure 6: α_i and α_j Optimization.

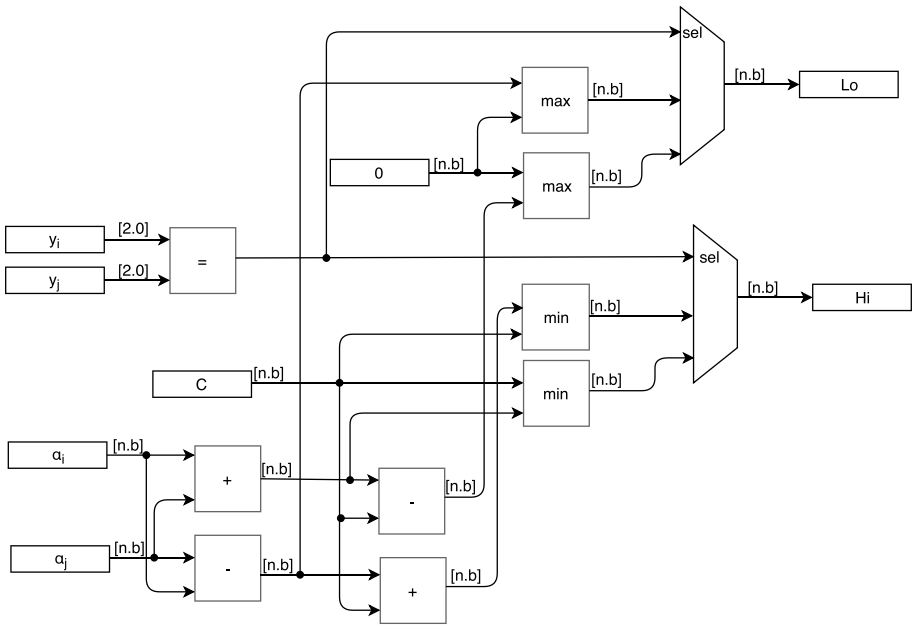

Figure 7: Lo and Hi Calculation.

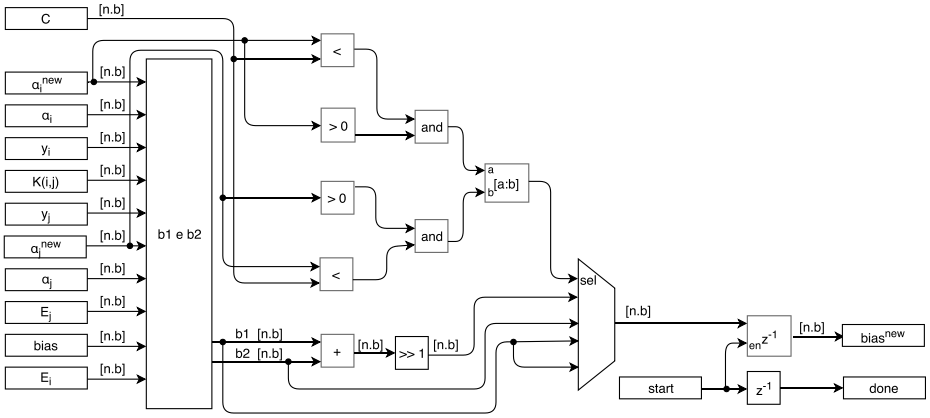

Figure 8: Bias calculation.

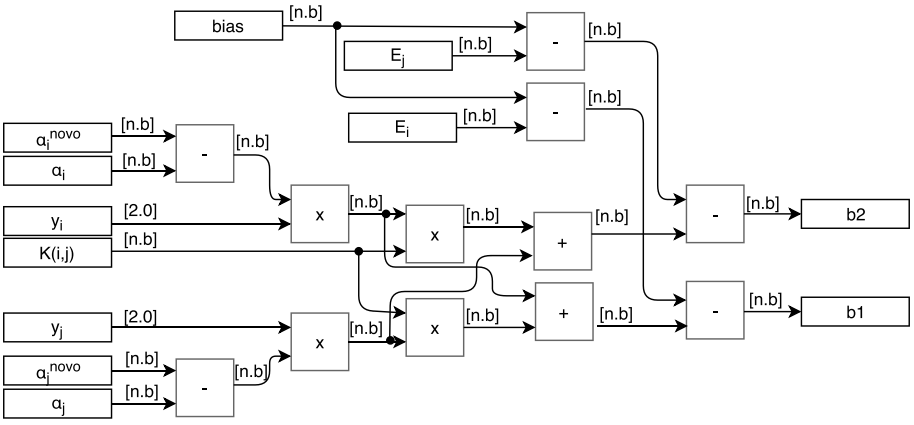

Figure 9: $b1$ and $b2$ Calculus.

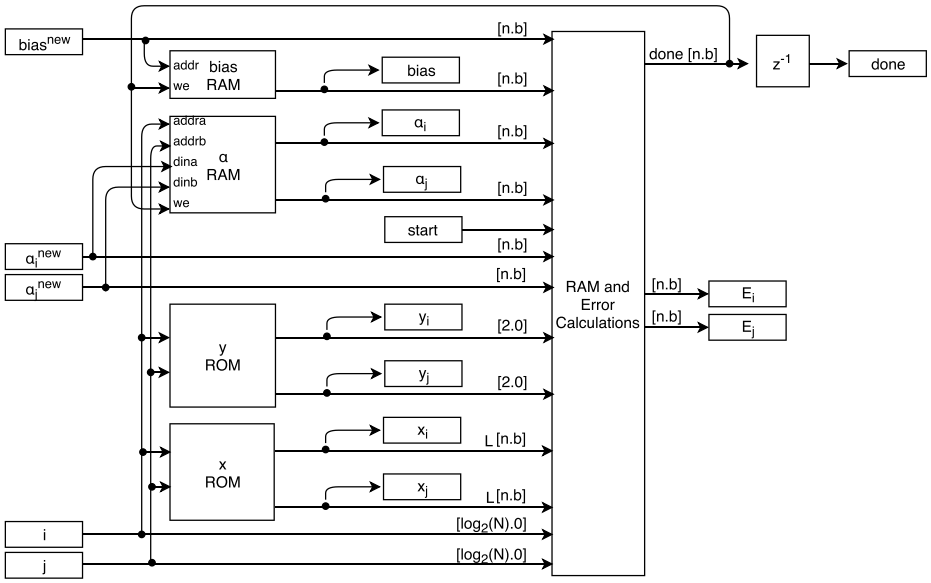

Figure 10: Overview of data storage and error calculation.

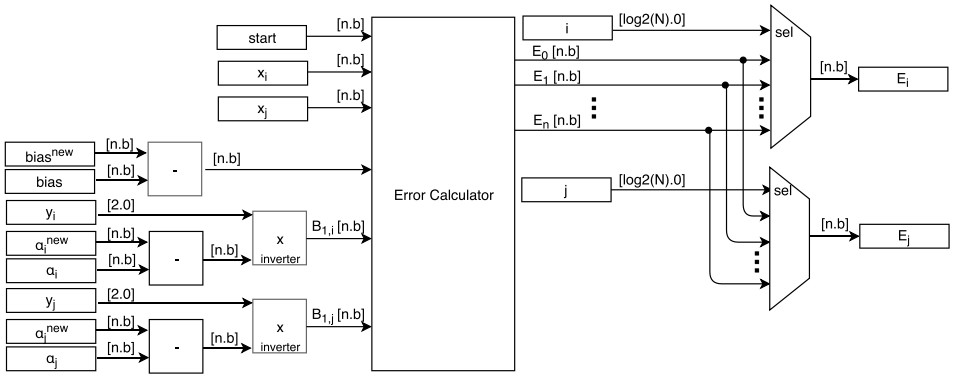

Figure 11: Error calculation.

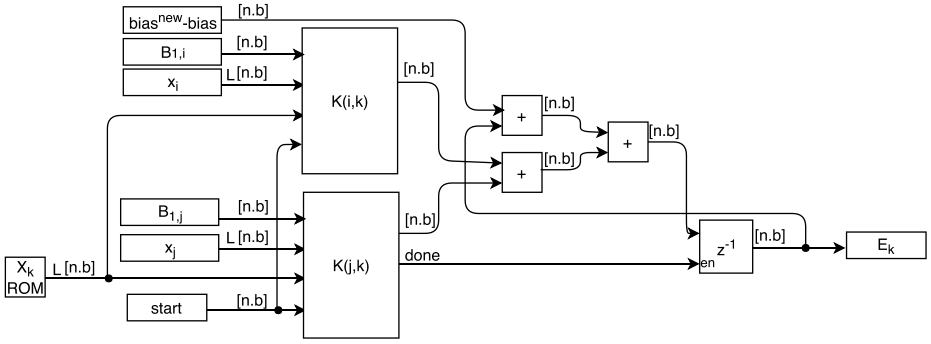

Figure 12: Individual error calculation.

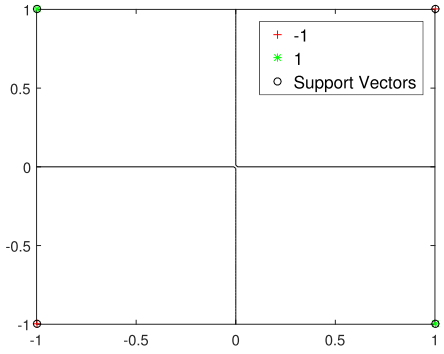

Figure 13: XOR Test result.

Table 2: SMO syntesis results.

Bits	Elements	Multipliers	Logic cells	Clock, f_c (MHz)
16.10	8	16 (2.0%)	7397 (4.9%)	34.620
32.24	8	31 (4.0%)	15320 (10.1%)	24.684
16.10	16	16 (2.0%)	13534 (8.9%)	33.980
32.16	16	31 (4.0%)	28158 (18.6%)	24.601
16.10	32	16 (2.0%)	25784 (17.1%)	34.426
32.24	32	31 (4.0%)	53887 (35.7%)	24.385
16.10	64	16 (2.0%)	50859 (33.7%)	33.383
32.24	64	31 (4.0%)	106008 (70.3%)	24.657

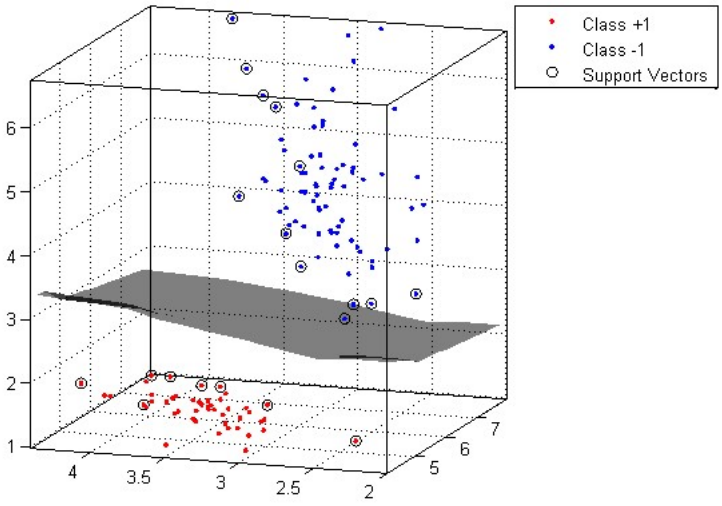

Figure 14: Fisher's Iris data set test result.

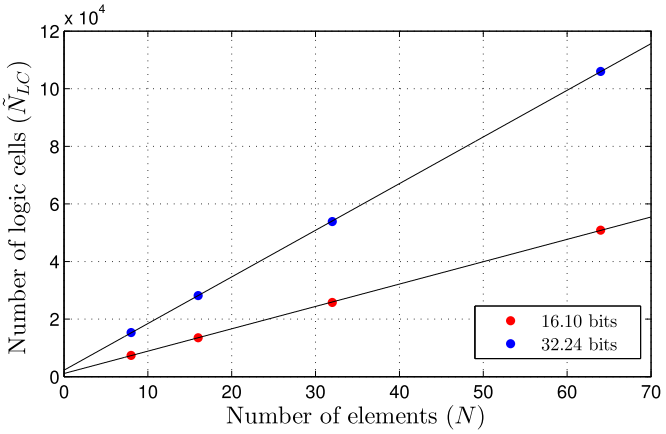

Figure 15: Regression analysis regard to the FPGA area occupation. $R^2 = 1$ for both, 16.10 and 32.24 bits.

Table 3: The estimate of the elements number for some commercial FPGAs.

FPGA	Logic cells	Elements		
		16 bits	24 bits	32 bits
Virtex 6 XC6VLX760	758784	975	631	467
Virtex 7 XC7V2000T	1954560	2516	1630	1205
Virtex UltraScale	5541000	7135	4623	3418

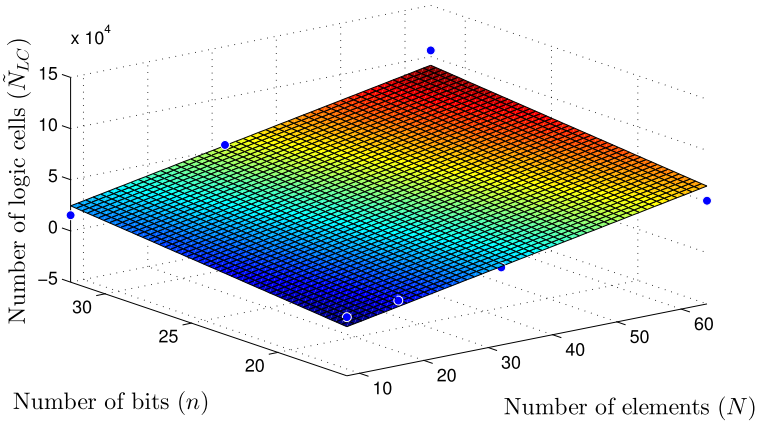

Figure 16: Two-dimensional regression analysis regard to the FPGA area occupation, $R^2 = 0.9108$.

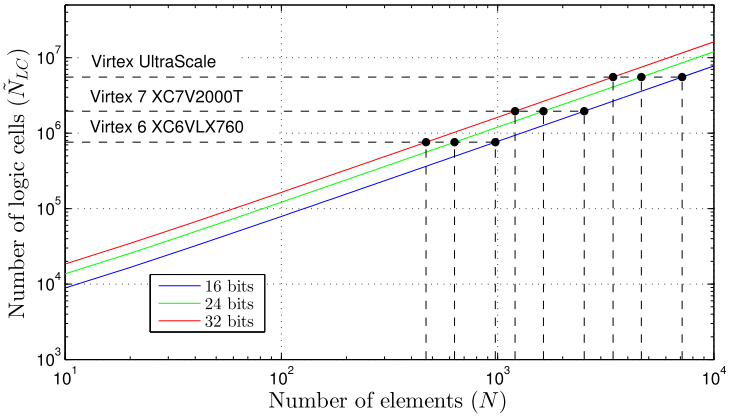

Figure 17: Scalability of the proposed hardware regard of the number of elements, N.

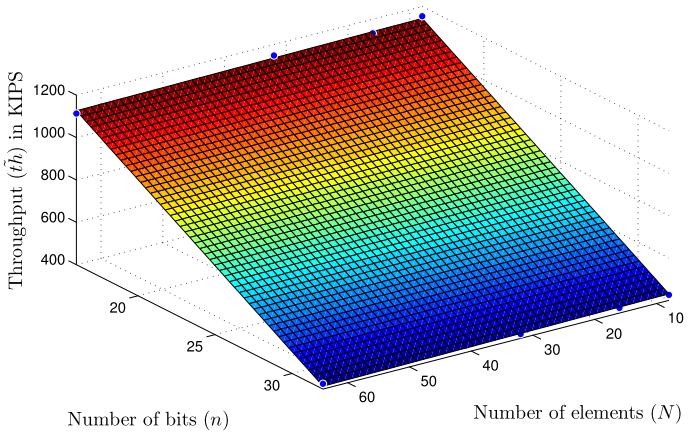

Figure 18: Two-dimensional regression analysis ($R^2 = 0.9993$) regard of the throughput in KIPS ($\times$1000 iteration per second).

Table 4: Convergence time and sampling rate of literature applications (SVM training with 16.10 bits).

Name	Elements	Dimension	Clock (f_c)	Iterations	Convergence
Iris	100	4	$\sim 33\,\text{MHz}$	$\sim 0.5\text{K}$	$\sim 0.45\,\text{ms}$
Breast Cancer	569	10	$\sim 27\,\text{MHz}$	$\sim 5.2\text{K}$	$\sim 5.2\,\text{ms}$
Banknote Auth.	1372	4	$\sim 30\,\text{MHz}$	$\sim 17\text{K}$	$\sim 17.0\,\text{ms}$

Table 5: Comparative table with state of the art works. Throughput results.

Reference	Throughput		Speedup
	Reference	This work	
Filho et al. (2010)	1 KIPS	420 KIPS	$\approx 420\times$
Cao et al. (2010)	6.25 KIPS	420 KIPS	$\approx 67\times$
Bustio-Martínez et al. (2010)	8.69 KIPS	420 KIPS	$\approx 48\times$
Ta-Wen et al. (2012)	0.45 KIPS	420 KIPS	$\approx 933\times$
Venkateshan et al. (2015)	0.32 KIPS	420 KIPS	$\approx 1312\times$
Madadum & Becerikli (2017)	0.37 KIPS	420 KIPS	$\approx 1135\times$
Feng et al. (2018)	11.49 KIPS	1123 KIPS	$\approx 98\times$