EE9-AO16 Advanced Digital System Design (2017-2018)

Supporting Vector Machine

Report

Group ID:	10
Group Member:	Jianyi Cheng
	01371255
	Yilin Weng
	01366683
Due Date:	16 March, 2018

Abstract

Supporting Vector Machine (SVM) has been widely used as machine learning models in various applications, such as character recognitions. In this work, a SVM classification IP core has been implemented and accelerated on a Zynq7000 SoC using High-Level Synthesis (HLS). Techniques such as parallelization, precision analysis, memory reconstruction have been carried out to achieve the maximum throughput. With performance bottleneck found in the data feeding rate, the dual IP core system controlled by two ARM cores have provided a maximum throughput of 1 result per 1.32 us.

Keyword: Supporting Vector Machine (SVM), FPGA, High-Level Synthesis (HLS).

Contents

1. Introduction

The objective of this project is the hardware implementation and acceleration of a Supporting Vector Machine (SVM) classification on a Zynq7000 System-on-Chip (SoC). Detailed optimizations have been made in several internal functional blocks including algorithm modifications, precision adjustments, operator analysis for hardware resource allocations and memory reconstruction. At top level, the weighted Kernel accumulation process in the SVM classification has been parallelized and pipelined to achieve highest computation speed and hardware reuse. This results in an optimal design with latency of 90 clock cycles with both DSP and LUT approaching to full usage. In addition, it has also been shown the whole system performance bottleneck has been found in the data transmission rate between ARM core and the implemented IP core when the IP core latency reached around 250 clock cycles. Hence a dual IP dual core system has been implemented and demonstrated providing doubled speed performance.

2. System Overview

Figure 2.1 demonstrates block diagram of the original SVM classification hardware flow, which can be generally divided into five stages. Firstly, a matrix of 16 16-bit input data was loaded into the IP core. At the initial stage, the dot product of the input data and a Supporting Vector (SV) set of the same size was calculated and multiplied by two. Subsequently the output was sent to the getTanh functional block resulting in single Kernel function output at the output. This was followed by a weighting process, where the output was scaled by the corresponding α. Then for each classification process, these weighted Kernel results were accumulated at Stage 4. It should be mentioned that the number of SV sets were 1050, which means that the first four stages were carried out by 1050 times to obtain all the weighted Kernel results. Finally, a bias value was added onto the accumulator resulting in a final classification decision in a single bit form. To be specific, the positive summation results leading to outputs of 0s while the negative values generated outputs of 1s. These single bit results of indicated that the character has been recognized as 'E'. In contrast, those 0s meant the chosen characters were not recognized as other characters. For maximum acceleration of the system shown in the figure, parallelism and pipelining have been implemented in both top level and internal functional blocks. Details of the design choices are included in the following sections.

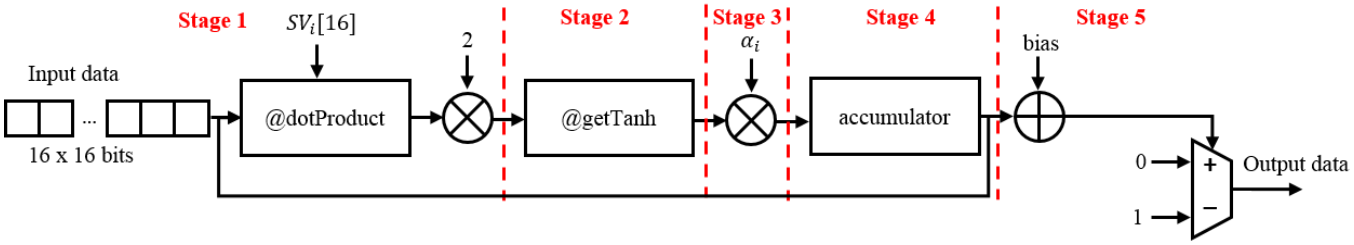


Figure 2.1: Block diagram of the SVM classification processing system.

3. Arithmetic Implementations and Design Choices

a. Datatype design choices

In this implementation, all the data have been decided to be implemented as fixed point due to three considerations.

- Fixed point data structures are simple, where the arithmetic processes can be considered as integer operations. With most suitable precisions, it is able to provide high performances at system level but may require large amount of design efforts on precision analysis to realize best results.
- Compared to the fixed point data type with the same bitwidth, the floating point data can have significantly larger dynamic range and higher computation speed. However, hardware costs are expensive when aligning are required between the registers, such as addition of two inputs. Hence this can be a performance bottleneck when working on a FPGA chip with limited hardware resources.
- Hybrid architecture that contains both fixed-point and floating point datatypes are also possible. However, it requires data type conversion between these two types, which can limit the system performances. In this work, typically it has been measured that the conversion from floating point to fixed point took 3 clock cycles, while the reversed the conversion required 9 clock cycles to finish. Considering the small number of latency of the total system (90 clock cycles), this approach was not considered.

Therefore, with all the data implemented in **fixed point types**, precision analysis has been carried out in each stage of the processing system to achieve lowest latency, which has been shown in the following sections.

b. Stage 1: dotProduct module and multiplication by 2

The hardware architecture of dotProduct functional block was shown in Figure 3.1. In the figure, it can be seen that two inputs, where each of them contained 16 elements, were sent to the block. Each element in the SV data set was multiplied with the element at the corresponding address in the input data set *x*. Subsequently, those multiplication results were summed to provide a single output indicating the dotProduct of two input data sets.

To optimize the computation speed of the block, several design choices have been made as follows:

- The multiplications of each element pair were **fully unrolled** and **pipelined** to allow parallelisation and maximum hardware reuse for highest possible throughput benefiting the latency of top level system.
- The multiplier types for the element pairs have been chosen to be **partially LookUp Table (LUT)** and **Digital Signal Processing (DSP) based**. This was to balance the DSP and LUT usage of the whole system both at high percentages making use of the hardware resources for possibly highest performance.
- The multiplication by 2 at the output of the dotProduct function block has been chosen to apply **bit shifting** techniques so that minimum resource and latency can be achieved typically in this part.

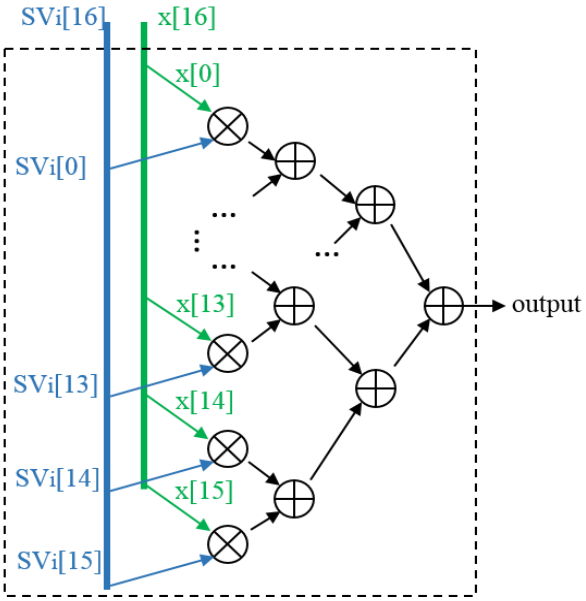


Figure 3.1: dotProduct function block implementation.

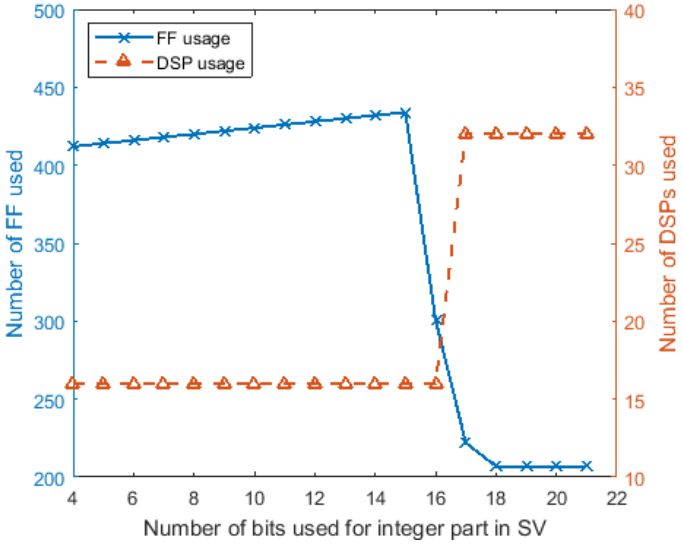


Figure 3.2: Precision analysis of DSP multiplier based dotProduct module with an input of 4-bit integer and 11-bit fractional parts and a SV set input of fixed 9 fractional part varying integer part.

Precision and hardware resources on DSP multiplier based dotProduct module

For precision analysis, the multiplier configurations need to be considered since the addition and shifting at output were implemented taking minority of the hardware resources in this block. Figure 3.2 shows the hardware usage of a DSP multiplier based dotProduct functional unit with variations in the integer bitwidth of SV set while keeping precisions of input data set and fractional part in SV set constant. In the figure, it can be seen that with the increasing precision in integer part of SV matrix, the number of DSP blocks used for the block was constant of 16 but jumped to constant 32 when the integer part reached 17 bits. It is known that a single DSP block on Zynq SoC can be applied as an 18 x 25 multiplier [1]. When the number of bits exceeded the available width of 25 bits, more DSP slices would be required. With 9 bits occupied as fractional part of the SV set, only 16 bits were available on the single DSP slice. Therefore, increase in number of DSPs was observed when the integer bits expanded to 17 bits. In addition, the Flip-Flop (FF) usage was slightly increasing with the bitwidth when a single DSP was used for each multiplication. However, significant decrease in FF usage was observed when two DSPs were used for each pair. This might be due to that the added DSPs contained available hardware resources, which replaced original FFs in the block. With increasing number of bits, available resources left in DSPs reduced requiring more FFs to construct the registers. Besides, the additional DSPs caused latency increasing by 1 clock cycle, which can be neglected since the block has been pipelined in the classification system resulting in an interval of 1 clock cycle.

	Input data <15, 4>		Supporting vector <13, 4>	
	Integer bitwidth	Fractional bitwidth	Integer bitwidth	Fractional bitwidth
DSP changing point	14	21	16	21

Table 3.3: Maximum bitwidth with other bitwidths constant when using single DSP for each pair multiplication.

Similar observations were found in precision analysis on other parts of the input. To be specific, the number of DSPs jumped from 16 to 32 at a certain bit number meanwhile FF usage dropped sharply after slightly increasing. Table 3.3 demonstrates the number of bit right before the number of DSPs got doubled. <m, n> means the total bitwidth of the data was m bits while the integer part has n bits. It can be seen that these measured maximum bitwidth follows the same rule shown in Equation 3.1. These results all agreed with the expectations.

Maximum number of bits for single DSP multiplication=25-bitwidth in other part (Equation 3.1)

Precision and hardware resources on LUT multiplier based dotProduct module

The other strategy is to use LUT multiplier instead of DSP based multiplier so that those DSPs would be replaced by LUTs. In this case, all the output would be generated from the LookUp Table applying distributed arithmetic architecture. Hence the latency of the multiplier would not change with larger bitwidths of inputs up to 32 bits as each multiplication was simply fetching results from LUT but required more LUT units to construct larger size tables. Figure 3.4 demonstrates the similar precision analysis to the DSP based dotProduct functional block. It can be seen that when varying each of bit number and keeping the rest bitwidth constant, the LUT usage increased linearly. This agreed with expectation since each additional bit resulted in one more addition and shifting process for the accumulator in each distributed arithmetic multiplication.

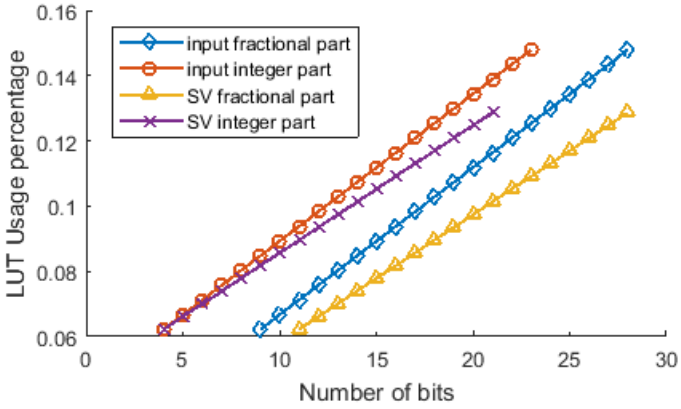


Figure 3.4: LUT usage of LUT based dotProduct block.

In this work, decision has been made to implement **3 LUT based multipliers** and **13 DSP based multiplier** to achieve most possible latency with both LUT and DSP usage under 100%. Besides, the precision of input data to the dotProduct block was **total 15 bits containing 4 integer bits** while the type of supporting vector was configured as **total 13 bits with integer part in 4 bits**.

c. Stage 2: getTanh module

It is known that function *tanh(x)* contains exponential terms which are challenging to implement. In this work, hardware approximation has been carried out using COordinate Rotation DIgital Computer (CORDIC) algorithm to achieve *tanh(x)* calculations with acceptable precision. In this work, the *tanh(x)* functional block has been designed based on the CORDIC algorithm in rotation mode, which basically iterates rotating the input angle approaching to the idea value in terms of the transformation as follows [2]:

$$\begin{pmatrix} X_{i+1} \\ Y_{i+1} \\ Z_{i+1} \end{pmatrix} = \begin{pmatrix} 1 & \delta_i 2^{-i} & 0 \\ \delta_i 2^{-i} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ \delta_i tanh^{-1}(2^{-i}) \end{pmatrix} \qquad (E3.2)$$

Figure 3.5 indicates the designed process to obtain *tanh(x)* through CORDIC iterations. Firstly, the input angle was checked if it was within the range, where the input with absolute value more than 5 would be directly output with value of 1 or -1 in terms of the sign bit in input data. In contrast, small value of angle would be sent to CORDIC iteration. Different from conventional CORDIC algorithm, design choices have been made for optimized hardware implementation with lower latency and possibly less hardware resources:

- The **integer and fractional parts** of input angle were **separated** for calculations, where different strategies have been applied on each part. To be specific, the integer parts had small number of bits, which can be implemented using a simple **LUT** with relatively small size and low latency. For the fractional parts, **CORDIC iterations** were carried out in rotation mode to obtain the corresponding *sinh* and *cosh* results. The benefits brought by this measure is data bit separation instead of the range checking within 1.13, where the range checking was data dependent to the iteration process and caused additional delays. In this approach, the process for integer part was simple and fast through LUT. For the fractional part, the input was always smaller than 1.13 so direct iterations were carried out.

- All the multipliers in this function block have been configured as **LUT multipliers** to balance the usage of DSP and LUT in the whole system.
- Within the fractional part calculation, the **number of iterations** have been decided to be **5**, where detailed correctness analysis has been shown in the following parts of the same section.
- In addition, it is known that additional iteration is required when the counter was in a form of 3k+1 ($k \in N^+$) to obtain more accurate results with i constant shown in Equation E3.2 in these two iteration cycles [2]. In this case, this occurred only when i was 4, so the **checking state** was carried out by a single comparison with a constant 4.
- Finally, since the outputs of *sinh* and *cosh* data sets were 4 values, simple arithmetic conversions have been carried out in terms the expressions as follows:

$$sinh(|\theta|) = sinh(|\theta_{int}| + |\theta_{frac}|) = sinh(|\theta_{int}|)\,cosh(|\theta_{frac}|) + cosh(|\theta_{int}|)\,sinh(|\theta_{frac}|) \quad (E3.3)$$

$$cosh(|\theta|) = cosh(|\theta_{int}| + |\theta_{frac}|) = cosh(|\theta_{int}|)\,cosh(|\theta_{frac}|) + sinh(|\theta_{int}|)\,sinh(|\theta_{frac}|) \quad (E3.4)$$

$$tanh(|\theta|) = sinh(|\theta|) \,/\, cosh(|\theta|) \quad (E3.5)$$

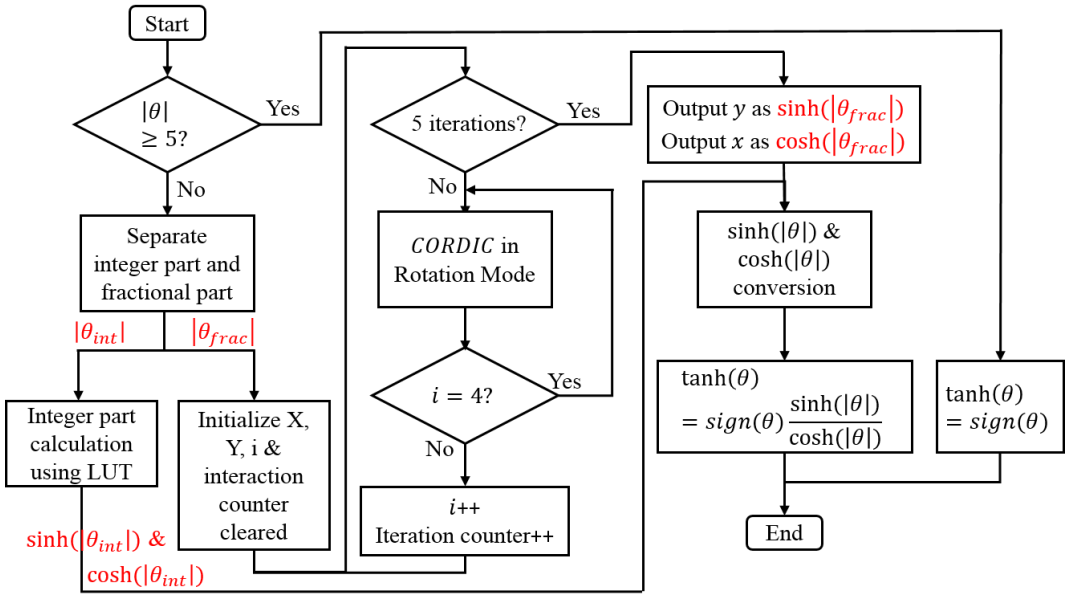


Figure 3.5: Flow chart of CORDIC algorithm to obtain *tanh(x)* result.

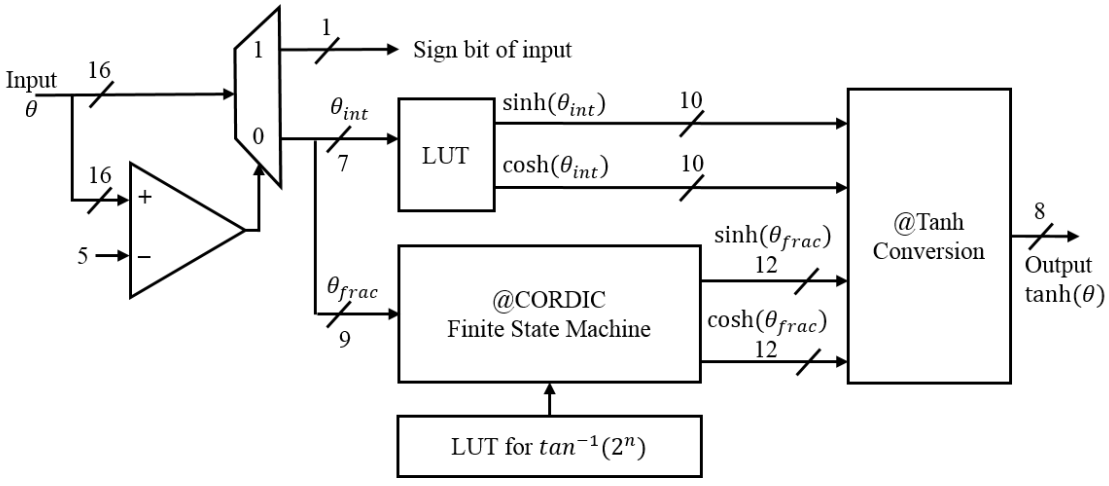


Figure 3.6: Hardware block diagram of *tanh(x)* implementation.

Figure 3.6 shows the hardware block diagram of the getTanh functional block to provide *tanh(x)* result based on CORDIC algorithm. As shown in the figure, it can be seen the range checking was achieved by a 16-bit comparator and multiplexer. In addition, with high precision calculation for the input angles within the range, parallel conversions were carried out for the integer and fractional parts of input. The integer part was designed as a LUT while the fractional part was processed through a finite state machine to perform CORDIC iterations. It should be mentioned that the CORDIC iteration loop has data dependency, where the current the output depended on the output in last

iteration cycle, so this was not unrolled but pipelined. Finally, a single conversion from four outputs were carried out with four multiplications, two additions and one division. The bit precision has been adjusted for minimum hardware usage and lowest latency when maintaining the output correctness.

Correctness analysis on number of iterations

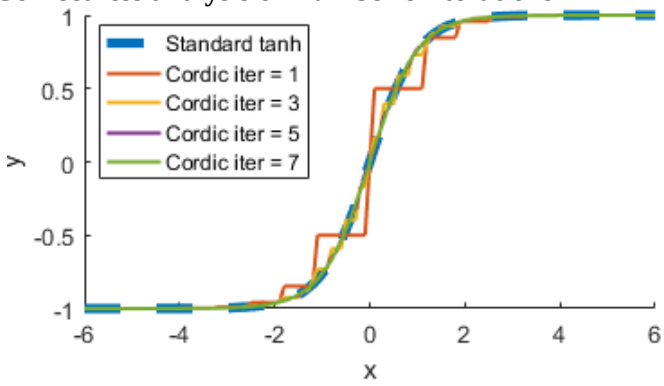


Figure 3.7: CORDIC approximated tanh function compared to the standard function curve.

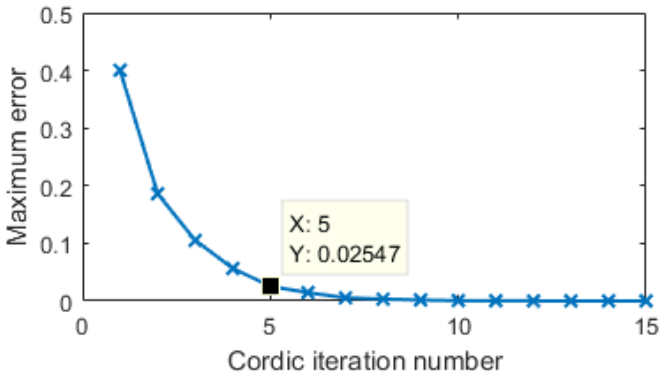


Figure 3.8: Maximum error in output using CODIC algorithm in terms of different number of iterations.

As mentioned that the iteration using CORDIC algorithm have data dependency and cannot be parallelized, the number of iterations, considered as most time-consuming operations in classification process, was correlated to the latency of whole system effectively. Figure 3.7 shows the approximated tanh function waveform using CORDIC algorithms with different number of iterations. It can be seen that with more iterations, the curve appeared to be more smooth and approaching to standard tanh function, which agreed with the expectations that the output value would be approaching to the correct value along with rotations. The maximum error between the outputs of CORDIC approximated algorithm and standard tanh function was shown in Figure 3.8. Generally, with more iterations, the error dropped, where the decreasing rate was also decreasing. Hence a trade-off can be found that small number of iterations may cause decision errors while accurate process with large number iterations would have longer latency. In this work, it has been found that with result correctness maintained, the **minimum iteration number was 5**, where the maximum tanh output error was 2.5%.

Optimized algorithm performances

	getTanh function block using conventional CORDIC algorithm	getTanh function block using optimized CORDIC algorithm in Figure 3.5
Latency	24	21
Interval	1	1
FF usage	873 (1%)	745 (1%)
LUT usage	2595 (4%)	1549 (3%)

Table 3.9: Performance and hardware utilization comparison between two CORDIC algorithms.

Table 3.9 shows the comparison between two genTanh function blocks. Generally, due to additional range checks around 1.13, the block with conventional algorithm has 3 more clock cycles of latency, which met the expectations. In addition, more hardware resource in FF and LUT are required. From the results demonstrated above, the optimized algorithm with separate calculations of integer and fraction parts has been chosen for the classification system implementation for better computation speed and reduced hardware usage.

Divider analysis

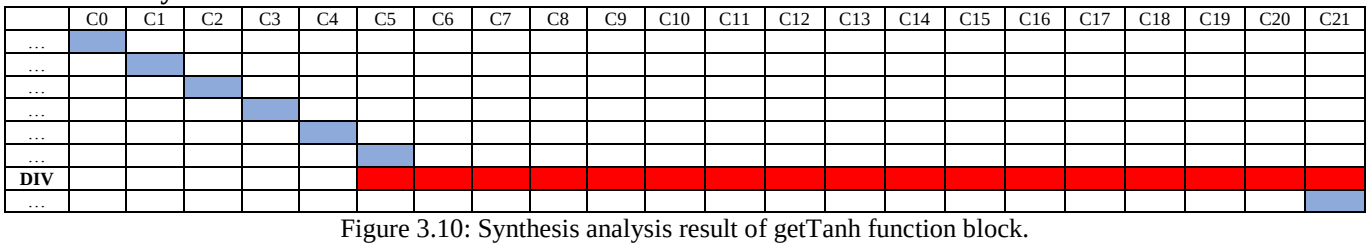


Figure 3.10: Synthesis analysis result of getTanh function block.

Figure 3.10 shows the synthesis analysis result of getTanh function block using lowest possible precision fixed point divider. In the figure, it can be seen that the latency of the divider was the majority of the total latency of this block, which took 17 clock cycles out of 22 clock cycles. Hence different approaches for use of divider need to be considered. Table 3.11 shows the comparison of four types of divider with lowest possible latency. Firstly, floating

point or half-precision divider appeared to have lower latency than the fixed point paying the price of more hardware resources, where especially the half-precision divider has approximately 1/3 of the fixed point latency. However, as mentioned that the whole system has been implemented in fixed point data types, the half-precision dividers were required to use with data conversions between fixed point and floating point. It is also shown in the figure that with conversions included, the use of LUT and FF significantly increased with a significantly longer latency. Therefore, it can be concluded that fixed point divider may be the most suitable divider for this block design.

		float point	half-precision	fixed point<8,3>	half-precision transfer to <8,3>
Synthesis Results	Latency	15	6	17	21
	Interval	16	7	18	22
	LUT	1009	177	160	2112
	FF	777	218	139	1474

Table 3.11: Comparison of divider choices for getTanh function block.

Precision analysis

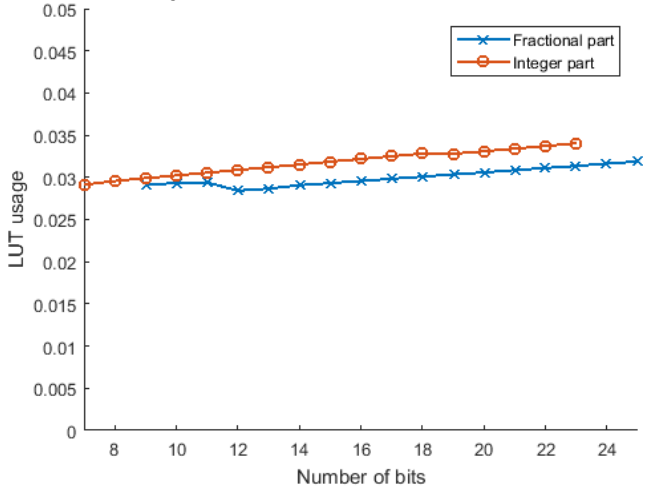


Figure 3.12: LUT usage of getTanh function with variations of input data precisions.

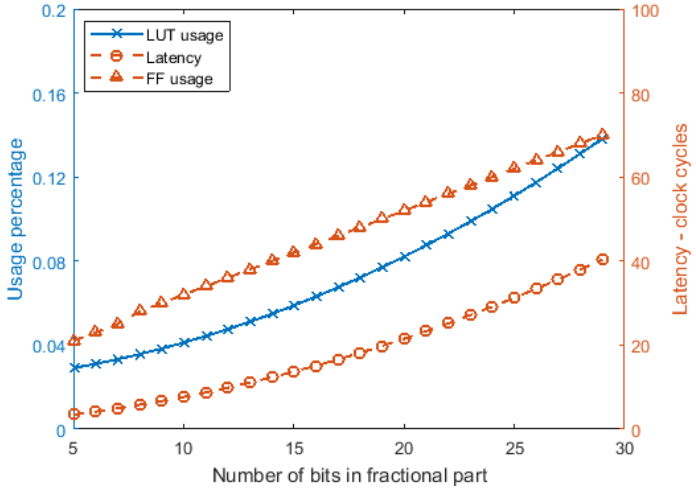


Figure 3.13: Performance and hardware usage of getTanh function with variations of fixed point divider precisions.

As for the precision analysis, there are two aspects need to be considered. Firstly, Figure 3.12 shows the effects of input precisions on LUT usages in getTanh function block. Generally, the LUT usages were slightly increasing with increasing number of bits. It should also be mentioned that the rest of the hardware resources were also maintained at same levels. In contrast, the variation in the precision of divider has noticeably affected the FF and LUT usages as well as the computation speed shown in Figure 3.13. With increasing number of fractional bits, the FF and LUT usage were rising linearly. In addition, the division was also delayed with higher precision. This agreed with expectations since the divider has contributed 77% of the total latency in this function block. With result correctness maintained, design choice has been made to configure input to getTanh function as **7 integer bits and 9 fractional bits** while the divider precision has been configured as **3 integer bits and 5 fractional bits**.

d. Stages 3 & 4 & 5: weighting, accumulation and decision

The implementations of the rest three stages were straightforward with less design complexity. Design choices have been made in Stage 3 to configure the multiplier as **LUT multiplier** to save DSP usage but took more LUTs. In addition, the precision of alpha data sets has also been adjusted as **5 integer bits and 8 fractional bits**.

e. Conclusion

In this section, the analysis on internal modules in the classification system has been carried out with design choices. Firstly, the data type for the whole system has been decided to use fixed point to perform highest computation speed with the same architecture. In addition, the types of multipliers have also been specified in each module to make use of all available resources, such as DSP and LUT. Besides, the precisions of the data have been analyzed on the effects on the system latency and hardware usages. For the CORDIC part, modifications of algorithm has been done to provide better performances with less hardware usage, These design choices that improved the performances of internal modules are expected to be beneficial to the top level system, which has been verified in the next section.

4. Systemic Analysis and Design Choices

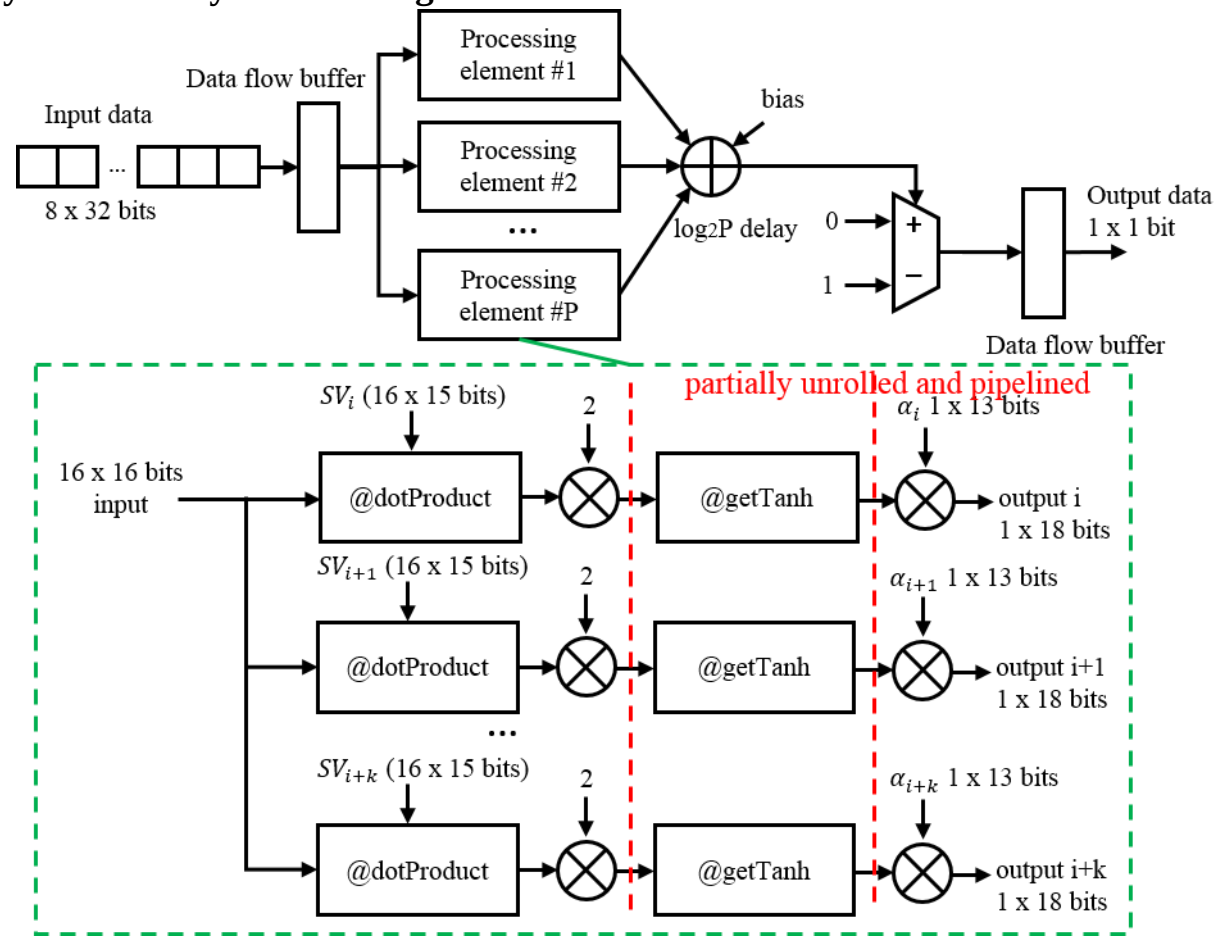


Figure 4.1: Block diagram of the SVM classification processing system (k*P = SV set number).

Figure 4.1 demonstrates hardware block diagram of the top-level system, which has been generally implemented and accelerated using parallel computations. To be specific, the input data was loaded into the data flow buffer, which allowed input preloading for the next classification while the operation in current cycle being processed. Subsequently, the whole classification operation has been divided into P parts and executed simultaneously. This was followed by all the weighted kernel results obtained from the processing elements. Then these values were summed with an additional bias for classification decision making, where the positive summation results leading to single bit outputs of 0s while the negative values generated outputs of 1s. The design choices have been taken in the following aspects:

- The top level design has been configured as **DATAFLOW**, where preloading buffers were assembled both at inputs and outputs enabling preloading during the classifications so that the data transmission cost can be hidden.
- Parallelisation of processing element has been achieved the maximum extent with all the available resources, where **P = 18**. Detailed analysis is included in the following parts of this section.
- The classification process in each processing element has also been **partially unrolled by factor of 4 and pipelined** to achieve maximum hardware reuse and reduced total latency.
- The memory architecture has been reconstructed in two aspects. Firstly, the **input data was reshaped** completely so that the bitwidth of the input can be maximized, where 2 sets of 16-bit data can be transmitted as a single 32-bit data leading to doubled data transmission rate. In addition, the SV set and α have been **partitioned in P sections** and **reshaped in a cyclic format**. This resulted in a working flow that each processing element was allocated with a partitioned memory bank of SV set and a bank of α set as shown in Figure 4.2. This improved memory bandwidth of the memory architecture instead of duplicating the whole memory block. Benefited from cyclic ordering, the paralleled system performed P Kernel function each time following the SV set order. An example of SV set mapping in a single processing element has been shown in the figure. This can be seen that after reshaping the array structure, the sequence of SV set has been modified and equally distributed in each processing element.

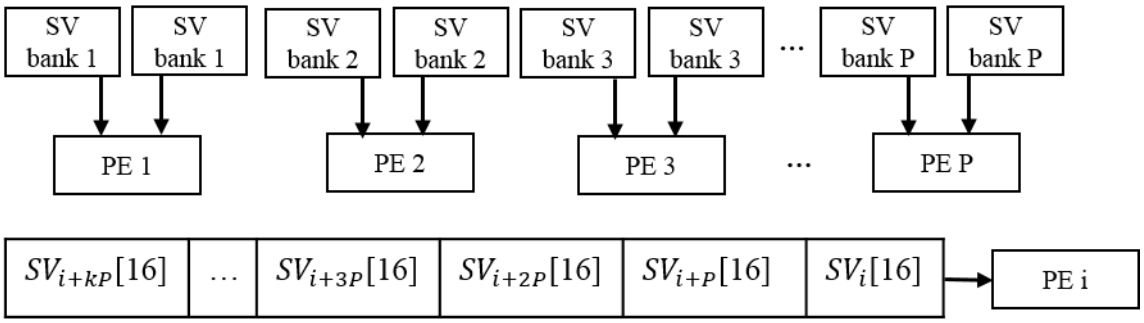


Figure 4.2: Memory architecture of SV and α sets in classification system (k*P = SV set number).

- All the internal functions have been **inlined** skipping the calling process. At hardware level, this indicated that no additional registers were placed between the internal functional blocks which might cause additional unnecessary delays.

a. Processing parallelization

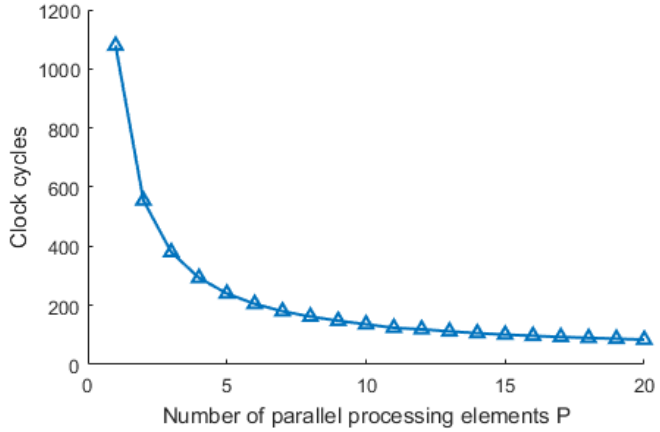


Figure 4.3: System computation speed in terms of number of paralleled processing elements.

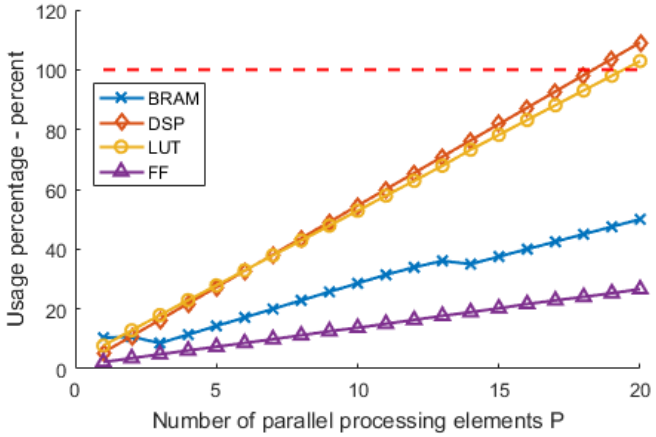


Figure 4.4: Hardware utilizations in terms of number of paralleled processing elements.

One of the most effective solution for computation acceleration is maximizing the number of parallel processes. Figure 4.3 shows the latency of the classification system with variations in the number of processing elements. As shown in the figure, it can be seen that more parallelism resulted in lower latency, which agreed with the expectations. The improvement was noticeable at low number of parallel processes, however, when the number of processing elements continued to increase in extensively paralleled system, the effects on speedup reduced. For hardware utilizations, Figure 4.4 shows the hardware resource utilization of the classification system with different extents of parallelism. In figure, it can be seen that with increasing number of processing elements, the utilization increased linearly. This agreed with expectations since the unrolling process was to simply duplicate the processing elements. In the figure, it can be seen that the DSP and LUT were increasing rapidly while BRAM and FF usages were increasing at low rates. Hence performance bottleneck can be found in the use of DSP and LUT when the number of parallel processes **P reached 18**, where both of these usage exceeded at P = 19. The final specification of this IP core is shown as follows:

Clock frequency	Timing results – clock cycles		Hardware usage			
	Latency	Interval	DSP	BRAM	LUT	FF
100 MHz	90	87	216 (98%)	126 (45%)	49511 (93%)	25568(24%)

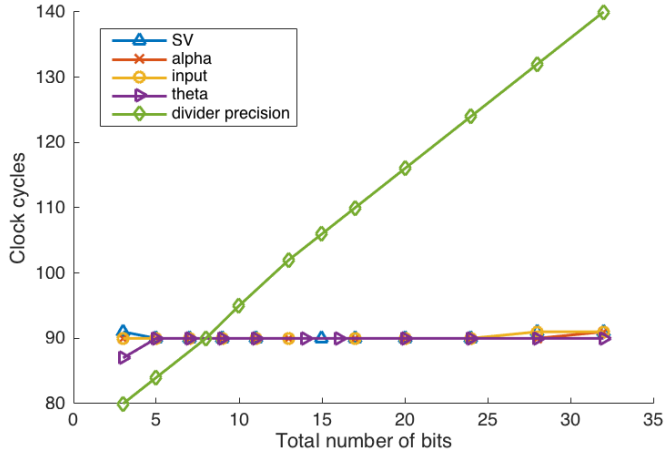
Table 4.5: Performance of optimal SVM classification IP core.

b. Systemic precision analysis

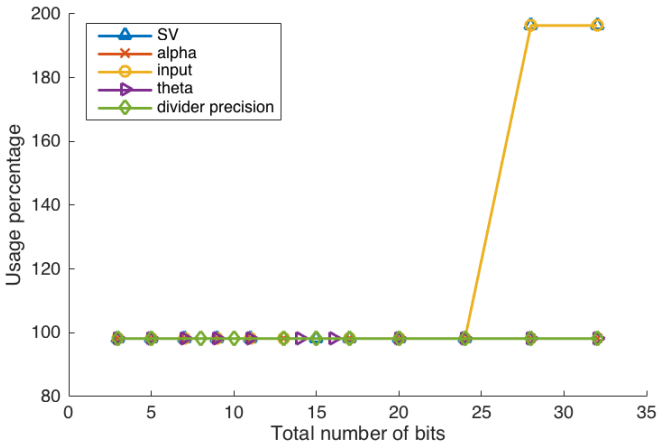
To evaluate the detailed precision effects at systemic level, single parametric analysis has been carried out on the performance and hardware utilizations of the system. Figures 4.6 show the precision analysis on the top level classification system with variations of a single parameter precision.

- Firstly, it can be seen in (a) that apart from parallelism, the latency of the system had a linear relation with the divider precision. Since the divider dominated the total latency of CORDIC algorithm, this agreed with the explanation in the previous section.

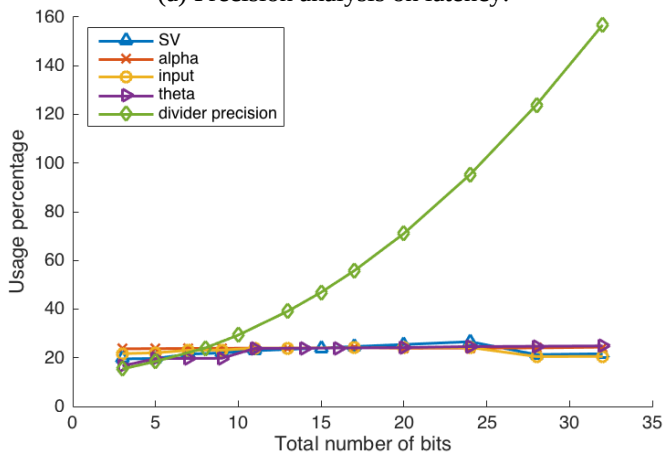
- Meanwhile, the implementation of divider at different precision also required relatively large amount of FF to store the temporary data. Hence in (c), the precision of divider also have caused increase in FF usage dramatically.
- In addition, the use of DSP was doubled when the input or SV set were more than 26 bits attributed to limited 25 bits in a DSP block. Since only the dotProduct module used DSP based multipliers while other parts assembled LUT based multipliers, only the use of SV sets and input data would require DSPs.
- The LUT utilization seemed related to most of the data variables since they were all involved with arithmetic operations such as additions. In (d) it can be seen that generally with increase of precisions, the LUT usage increased as well. More importantly, the divider precision appeared to have more effective impact on the LUT usage resulting in LUT increasing dramatically, while others were observed have only linear relations with LUT usage at low increasing rates.
- Finally, for the BRAM usage, since only the SV and alpha sets were stored in the memory, the other data had nearly no effects on the BRAM usage. Compared to alpha, the precision of SV set was observed more correlated to the total BRAM usage, which increased in sharper slope.



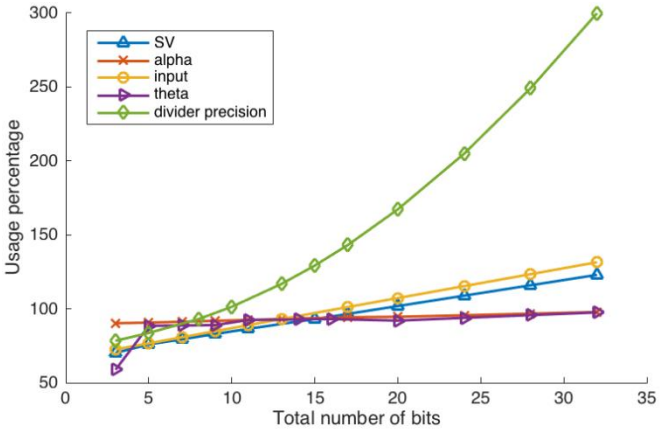
(a) Precision analysis on latency.



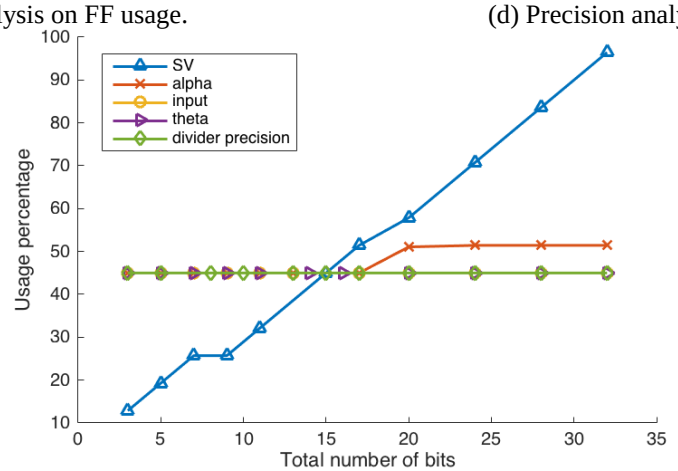
(b) Precision analysis on DSP usage.



(c) Precision analysis on FF usage.



(d) Precision analysis on LUT usage.



(e) Precision analysis on BRAM usage.

Figures 4.6: Precision analysis on the whole classification system.

c. Transmission cost hiding

From software point of view, the hardware computation process and input data can be carried out in parallel. Figure 4.7 shows the conventional stalling process and optimized preloading processes. In the figure, it can be seen that the processor was kept at idle state waiting for the hardware process to be finished in each classification operation. In addition, the hardware was also idle when receiving the input data. Therefore, a large number of time slots has been wasted in the total 2000 classifications known as transmission cost. To make use of the available time slots, the (n+1)th input data set can be preloaded onto the input buffer during which the hardware was calculating the result for the nth input set. In the figure, two situations when applying input preloading are demonstrated, which are longer transmission time and longer hardware execution time. When the transmission time in software was longer than the hardware execution time, the total classification time would be shown in Equation E4.1. In this case, the software was working in fullload resulting in the performance bottleneck of the classification system.

$$t_{TX>EXE} = 2000t_{TX} + t_{EXE} + 2000t_{RD} \qquad (E4.1)$$

where t_{TX} is the time for transmission of a single input data set, t_{EXE} is the time for a single classification process, t_{RD} is the time for a single result reading process.

In contrast, when the hardware execution time was longer, the software was idle, which can be working on other processes such as software calculations. The valid result could be fetched through interrupt triggering so that higher efficiency can be obtained when additional software computations were filled into the while slots in software time flow. In this case, the latency in hardware IP core is the bottleneck and the total operation time would be:

$$t_{TX<EXE} = 2000t_{EXE} + t_{TX} + 2000t_{RD} \qquad (E4.2)$$

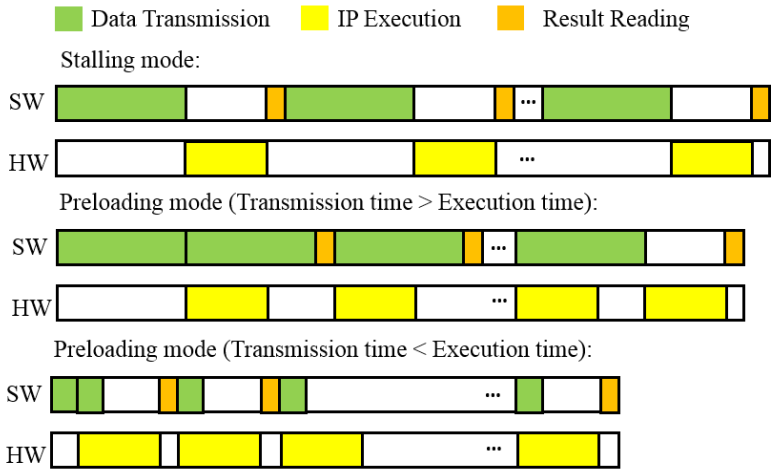


Figure 4.7: Timing diagram of transmission cost hiding techniques.

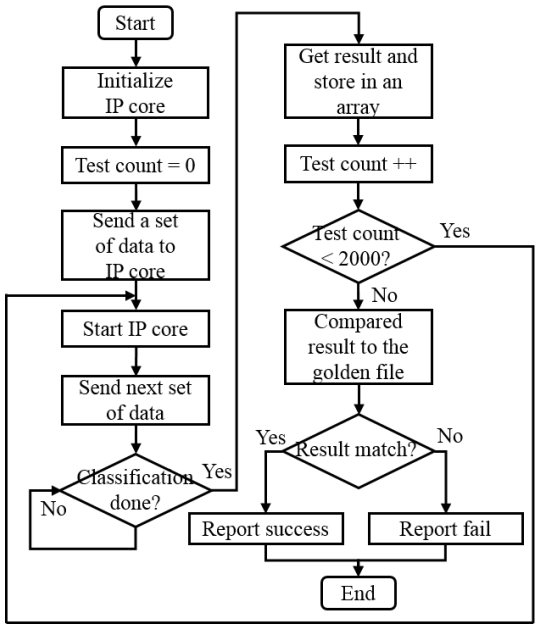


Figure 4.8: Preloading implementation in SDK.

The implementation of data feeding with preloading in software is shown in Figure 4.8. Firstly, the hardware IP core was initialized and checked availability. Subsequently, the first test data set was sent to the hardware followed by a start message. After that, instead of waiting for the result to be available, the processor sent the second data set before checking the state of first result. Then the first result was checked and fetched back to the processor. This was followed by starting the second execution and transmission of third data set. Hence the loop was repeated until all the test data has been classified. The reason for this implementation was that the transmission time was longer than the hardware computation time. Hence no interrupt was required since the processor has already been working in fully loaded.

Figure 4.9 shows the measured timing diagram of stalling mode system operation. In the figure, it can be seen that the throughput of system was 1/325 result per clock cycle. This contained 152 clock cycles for data transmission, where the output was valid after 122 clock cycles when the transmission was finished. This 122 clock cycles included total hardware execution time. For the preloading mode shown in Figure 4.10, the throughput of the system was

1/263 result per clock cycle. In the figure, it can be seen that the transmission time was the same as the time in Figure 4.9. However, it only took 17 clock cycles to validate result. This means that the hardware execution was being carried out during the transmission and possibly finished. Hence the preloading method has hidden the hardware execution time cost in this case leading to lower total classification time and 23.6% improved throughput.

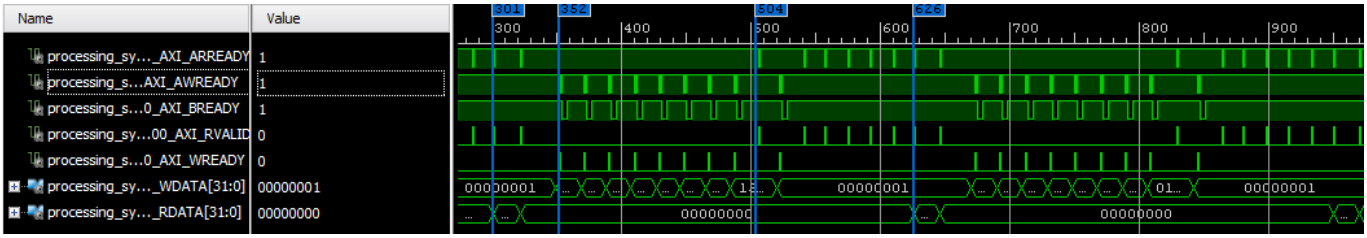


Figure 4.9: Timing diagram of classification process in stalling mode system.

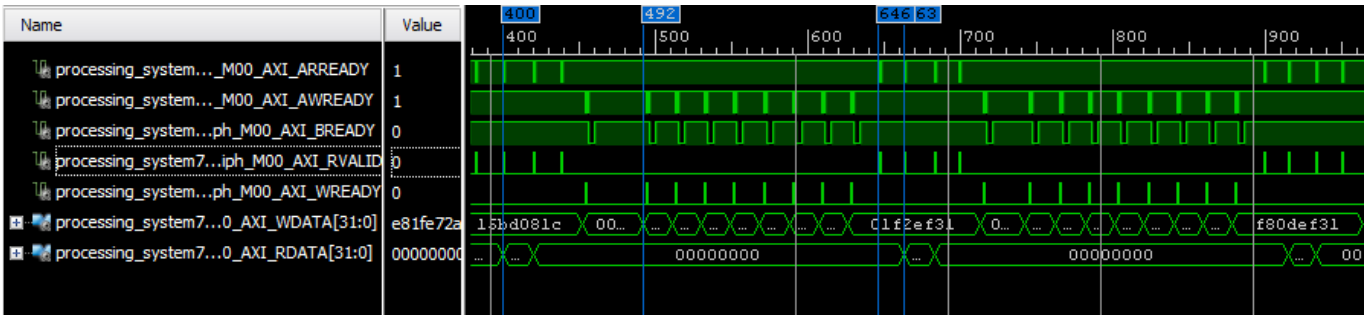


Figure 4.10: Timing diagram of classification process in preloading mode system.

d. Performance bottleneck analysis

Figure 4.11 indicates the measured total time of classification tests with different IP core latencies. This total time is defined as the amount of time from first input data to be sent to the moment when the last result has been obtained. In the figure, it can be seen that when the latency was higher than 250 clock cycles, the IP core computation speed was considered as the main performance bottleneck of the classification system. This is supported that when the latency increased above 250 clock cycles, the total time also increased linearly, which met the expectations in terms of Equation E4.2. In contrast, when the latency of hardware went below 250 clock cycles, the total operation time appeared to stay around 5 ms. This indicated that the transmission time has become the bottleneck as the hardware execution time has been hidden by input preloading, which might not affect the classification throughput effectively. Hence it can be concluded that further optimizations can be focused on increasing the transmission bandwidth of the processor.

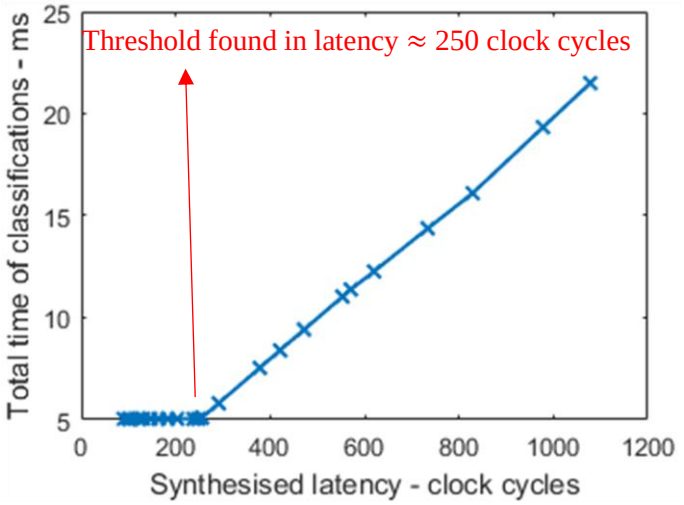


Figure 4.11: System performance analysis with variations in hardware computation speed.

e. Dual core operation

One solution to increase the transmission data rate is to implement dual IP core dual CPU system on Zynq SoC. As shown in Figure 4.12, both of ARM cores were connected to a private SVM IP Core. The test data sets were split into two parts so that two classification processes can be carried out in parallel. It should be mentioned that since this was a bare metal system, the memory was shared between two cores. Memory accessing rules have been defined in software for both of cores to avoid memory contention or write-before-read hazards. To be specific, both of cores had private memory region, which could only be overwritten by the corresponding core. However, these two cores could communicate through writing and reading the shared region to check status, where careful coding has been implemented to ensure proper timing.

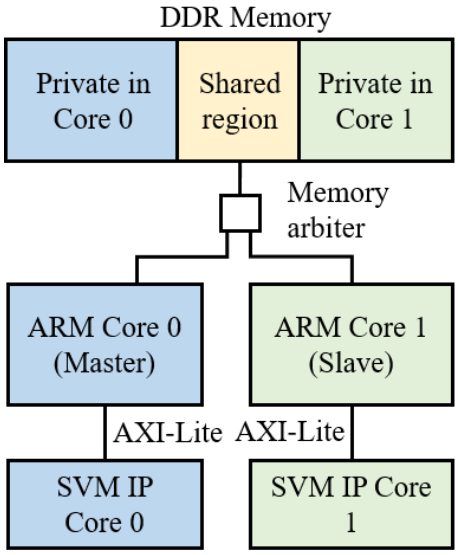


Figure 4.12: Dual core system overview.

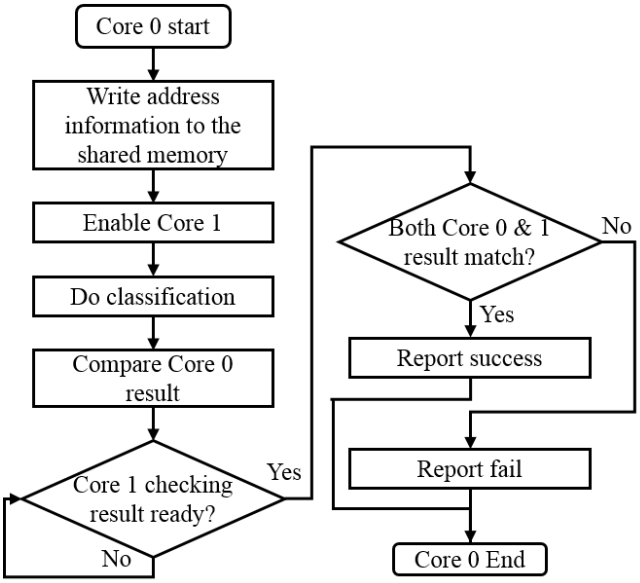


Figure 4.13: Software implementation of dual core classification system.

Figure 4.13 shows the implemented dual core working flow. Firstly, the address information such as test data, SV, alpha and gold result sets were written to the shared memory for slave core (Core 1) to access. Subsequently, classifications have been carried out in both master core and slave core in parallel with two transmission buses. After this process finished, the result checking parts were also executed separately. Then the matching information in the slave core would be stored in the shared region and checked by the master core. Hence the total test data set can be summarized in the master core providing the same results obtained in a single core system. In this case, the dual core operation simply doubled the workload capacity so that the transmission rate has increased by 100%. It was measured that the total time of classifications in a dual core system was 2.5 ms, which was half of the measured time in a single core system. Hence this implementation was successful and agreed with the expectations that the dual core system has doubled the throughput of the classification system when the transmission time was larger than the hardware execution time.

f. Hardware usage overview

Figures 4.14 and 4.15 show the synthesis reports of single core and dual core classification system. In the figures, it can be seen that both of the DSP and LUT have nearly reached the maximum limits. Since dual core system required two IP core implemented on the SoC, only maximum 50% of the hardware resources were allowed. Besides, it should also be mentioned that the single IP core has more extensive parallelism leading to lower latency than the implemented IP core for dual core system. However, as mentioned that the bottleneck of the system was transmission cost, when the latency was lower than 250 clock cycles, it was not expected to have different throughputs in the top level system with only hardware latency optimized.

Figures 4.16 and 4.17 show the total hardware utilizations on Zynq SoC, which appeared to be slightly different from those in synthesis reports. To be specific, the LUT and FF usage have been optimized by the software tool to reach a lower level than the expectations from synthesis reports. In addition, the use of BRAM for a single system increased and that in the dual core system was reduced. This might depend on the software backend optimization process. Finally, the Global Buffers (BUFG) were used for connections providing enhanced fan-out signals, where the usage was system connection dependent [4]. Hence this would not be concerned in this case.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.61	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
90	90	87	87	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	175	1208
Instance	126	216	25358	48233
Memory	-	-	-	-
Multiplexer	-	-	-	68
Register	-	-	35	-
Total	126	216	25568	49511
Available	280	220	106400	53200
Utilization (%)	45	98	24	93

Figure 4.14: Synthesis report of single core system implementation on Zynq SoC.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.66	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
148	148	145	145	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	130	938
Instance	72	108	13202	24471
Memory	-	-	-	-
Multiplexer	-	-	-	50
Register	-	-	26	-
Total	72	108	13358	25461
Available	280	220	106400	53200
Utilization (%)	25	49	12	47

Figure 4.15: Synthesis report of single core system implementation on Zynq SoC.

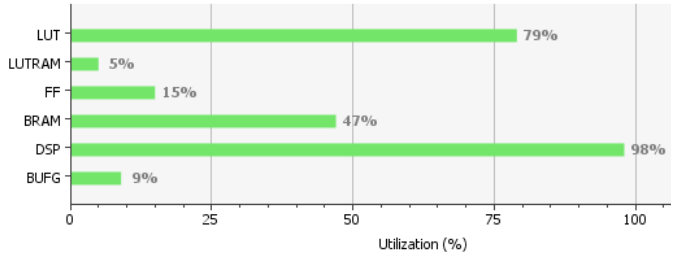


Figure 4.16: Hardware utilization of single core system implementation on Zynq SoC from bitstream report.

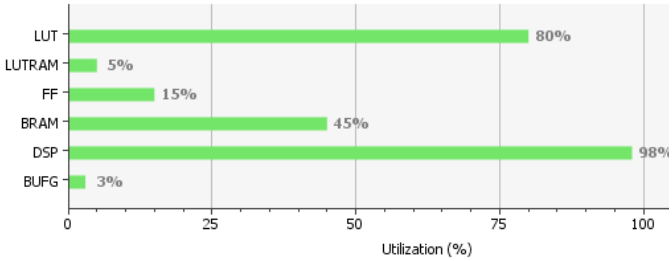


Figure 4.17: Hardware utilization of dual core system implementation on Zynq SoC from bitstream report.

g. Cascade SVM classification system simulation

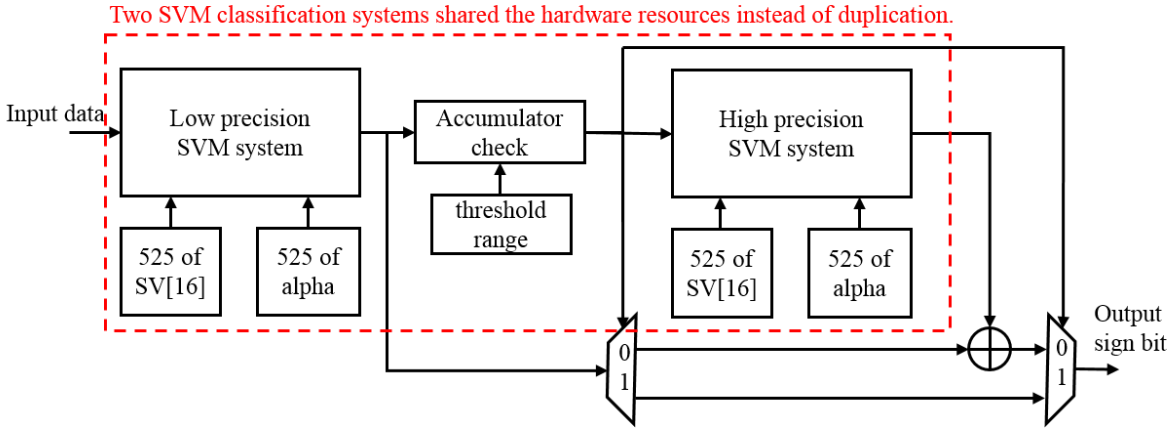


Figure 4.18: Cascade architecture of SVM classification system.

To further improve the hardware execution speed, cascade architecture of SVM classification system has been proposed in Figure 4.18. In the figure, it can be seen that the input data was sent to the first SVM block for low precision classification, where the

accumulator result would be checked if it was within the threshold range [5]. If the value was out of range, classification decision can be made directly from the low precision accumulator. However, when the accumulator result was in the threshold range, additional high precision process was enabled to provide more accurate classification result.

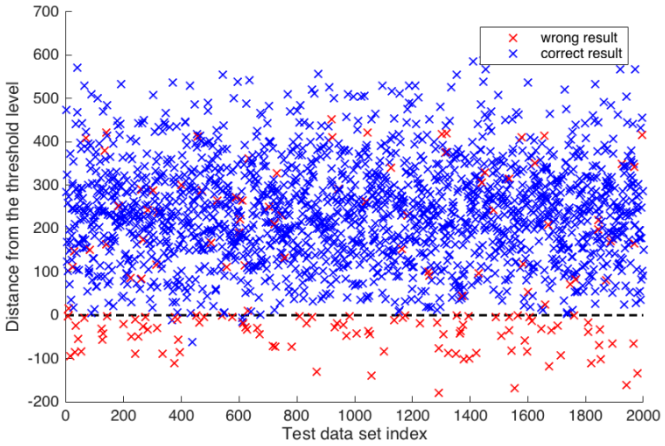


Figure 4.19: Accumulation results from pure high precision SVM classification system.

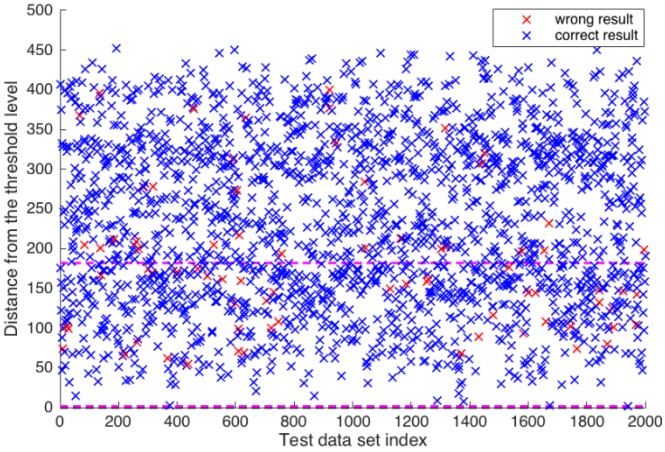


Figure 4.20: Accumulation results from pure low precision SVM classification system.

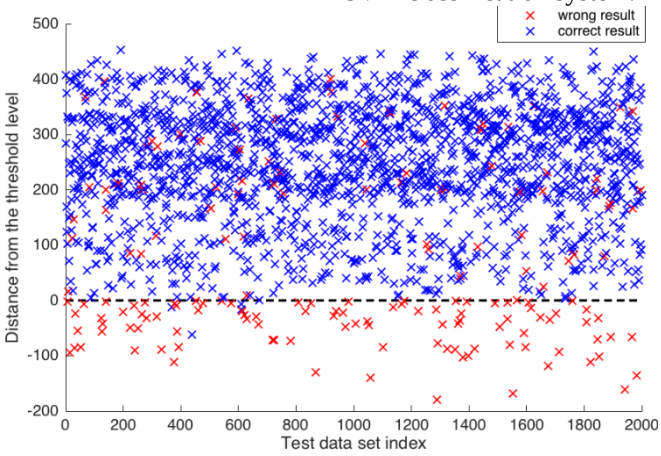


Figure 4.21: Accumulation results from cascade SVM classification system.

Figure 4.19 shows the accumulation result for each test data set using a single high precision SVM classification system. In the figure, it can be seen that the correct result was mainly distributed above the threshold line of 0. In the negative region, the system provided the wrong classification results. Figure 4.20 shows the results using a single low precision SVM classification system. In contrast, all the results were positive and the red crosses were appearing equally among the blue crosses. This indicated less effective classification performance and a threshold range was required to be defined to perform more accurate classifications. Hence as shown in the figure, the pink lines represented the upper and down threshold levels, where the data sets with accumulation results within the range were sent to high precision SVM system for further classifications. This cascade procedure ended up in the result distribution shown in Figure 4.21. Compared to original distribution, the number of crosses near 0 were significantly reduced. More importantly, the red crosses appeared to concentrate below 0. This was similar to the observations in Figure 4.19 indicating that the cascade architecture was able to perform the same accuracy as the high precision system and possibly had faster computation speed. The total time of classifications using cascade design is shown:

$$t_{cascade} = N_{LPE}t_{LP} + (N_{total} - N_{LPE})(t_{LP} + t_{HP}) \qquad (E4.3)$$

where N_{LPE} is the number of data sets only require low precision classification, t_{LP} is the execution time of low precision system and t_{HP} is the execution time of high precision system.

It should be mentioned that the performance of cascade architecture is test data dependent. When considering unified distributed input data, design choices need to be made in terms of t_{LP} and t_{HP}. Another trade-off can be found that with relatively higher accuracy in low precision block, latency would also be higher. In this cascade design, design choices have been made on the number of SV sets for low precision part as 350, which was typically 1/2 of the total number of full precision sets. Due to time limitation, detailed parametric analysis on the number of low precision part was unable to finish. Further researches can be carried out on SV set size for the effects on performances and hardware utilizations.

Figure 4.22 shows the performance of implemented cascade SVM classification system. In the figure, it can be seen that the minimum latency of the top level system has reduced to 62 with 30% improvements compared to the single core pure high precision implementation. However, the maximum delay was 30% longer. Hence the practical performance would depend on the input data distributions. To be specific, with large number of data away from the threshold, the overall performance of the top level system would be benefited from cascade architecture. When dealing with more data sets with small errors, the delay might be worse.

SVM level	Accumulator range at level 1		Data set execution records		
	Upper value	Down value	L1 precision	L1 & L2 precision	Total
2	180	-5	1560	440	2000

Table 4.23: Two level cascade results on test data.

Since the performance of this design is test data dependent, the simulation results for 2000 input test data sets have been shown in Table 4.23. With a threshold range from -5 to 180, 1560 data sets have been processed with only low precision part, while the rest of the data have gone through high precision calculations. Hence the average latency for this block would be:

$$latency_{av} = \frac{1560 \times 62 + 440 \times 121}{2000} = 75\ clock\ cycles \quad (E4.4)$$

Hence it can be concluded that the cascade implementation has provided better systemic computation speed than the single core high precision architecture.

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.61	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
61	120	62	121	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	73
FIFO	-	-	-	-
Instance	0	216	24938	47145
Memory	252	-	936	222
Multiplexer	-	-	-	4559
Register	-	-	256	-
Total	252	216	26130	51999
Available	280	220	106400	53200
Utilization (%)	90	98	24	97

Figure 4.22: Performance and hardware usage of two-level

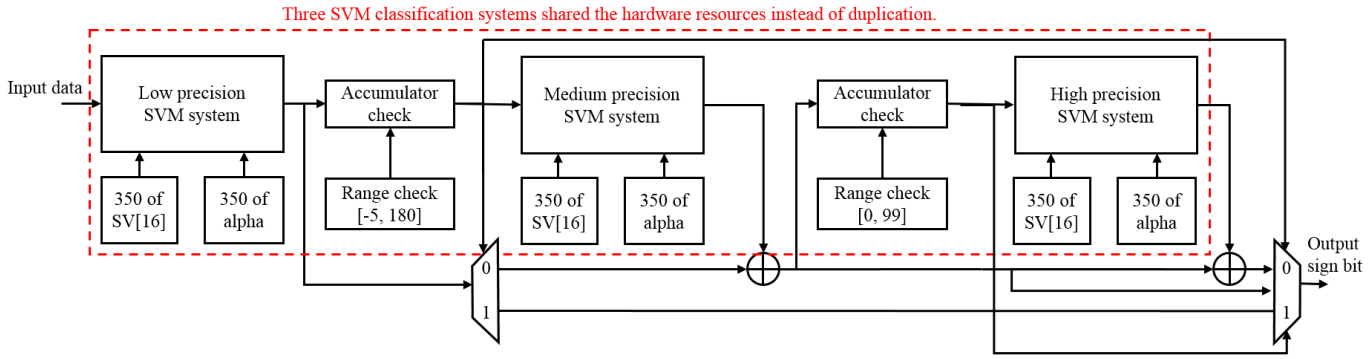


Figure 4.24: Block diagram of three level cascade SVM classification system.

SVM level	Accumulator range at level 1		Accumulator range at level 2		Data set execution records			
	Upper value	Down value	Upper value	Down value	L1 precision	L1 & L2 precision	L1 & L2 & L3 precision	Total
3	180	-5	99	0	1244	498	258	2000

Table 4.25: Three level cascade performance specifications.

For further analysis, multi-level cascade SVM classification system is possible. For this typical data set group, three stage cascade implementation has been carried out where each stage contains 350 SV and alpha sets. Table 4.25 shows the simulation results, where the number of the data set went through full precision calculations have further

dropped. However, the price of multi-level cascade design was the memory reconstruction cost, where advanced partitioning and reshaping techniques were required for efficient use. In this case, the BRAM in three-stage cascade has reached 135%, which failed the synthesis process.

h. Conclusion and system specifications

Figure 4.26 shows the performance specifications of three implemented designs. Firstly, the original architecture of SVM classifier appeared to have significantly high latency with relatively small hardware use percentages, where all the slices were used no more than 10% except DSPs. This resulted in a long total execution time. In contrast, the optimal design for single core system have occupied most of available DSPs and LUTs. This price caused dramatically improved IP core latency. For the optimal design in dual core system, the latency was higher than the previous design but only took half of the chip area. In this project, the hardware execution time has been covered by the data transmission time, where the latency of 90 would not improve the system performances. However, benefited from dual core operations leading to doubled data transmission rate so that the throughput of the system can be improved. Hence, the dual core system as best solution has performed throughput of 1 result per 1.32 us. The implementation of cascade architecture has also been carried out. However, due to the time limitation, the memory construction for dual core system was unable to be finished. However, this still has provided an expectation of total time improvements brought by the cascade implementation, where the maximum latency was higher than optimal design 1 but also had half minimum latency. This can result in a lower throughput in average improving the top level system performance.

	Original architecture	Optimal Design 1 (P = 18 single-IP single-ARM core)	Optimal Design 2 (P = 9 dual-IP dual-ARM core)	2-Level Cascade SVM IP core	2-Level Cascade SVM IP dual core
Synthesis latency	12648/79848	90	148	62/121	69/204
Synthesis interval	12649/79849	87	145	61/120	70/205
Co-smulation latency	19747/79223	169	229	-	-
Co-smulation interval	19747/79323	158	218	-	-
BRAM usage	7%	45%	26%	90%	67%
DSP usage	19%	98%	49%	98%	49%
FF usage	2%	24%	13%	24%	13%
LUT usage	4%	93%	48%	100%	50%
Total classification time	1687 ms	5 ms	2.5 ms	-	-
Throughput	1 result per 843.5 us	1 result per 2.63 us	1 result per 1.32 us	-	-

Figure 4.26: Performance specifications of implemented solutions
(SW took 2.7s in one ARM Core and 1.35s in dual Core system).

5. Future work

Due to the time limitation, the cascade design was unable to implemented onto the FPGA for software test. In addition, the transmission time has not been reduced to improve the system performances. For further researches, several considerations are suggested. Firstly, parametric analysis on the number of SV and alpha sets in the low precision classification part, where a sweet point may be found in the trade-off between the average latency of IP core and hardware resource usages. The low precision implementation may be more parallelized than the high precision part for improvements in total classification time. It would also be possible to maximize reusing of the two classification sections with similar architectures. Besides, analysis on the number of levels in multi-level cascade classification systems, total hardware usage, improved average latency can be carried out for most optimal solutions. For the data transmission part, it is possible to create direct memory access between the IP core and the DDR memory with optimal data interface such as streaming or Master burst mode. Finally, software coding can also be optimized using NEON instructions when the transmission time was covered by the hardware execution time so that part of the classification process can be carried out in the ARM cores resulting in speedup in total execution time.

6. Conclusion

In this work, the implementation of a SVM classification system has been implemented and optimized. The internal modules have been designed with consideration of precision control, operator type selection and algorithm improvements. At system level, the single core implementation provided the lowest latency with available hardware resources on Zynq SoC. Since the performance bottleneck has been found in the data transmission time, dual core system has been implemented for doubled data rate with simply two IP cores as the latency below 250 clock cycles was not affecting the system performances. This resulted in the throughput of 1 result per 1.32 us. However, the single core system can still have better performances than the dual core one when large numbers of SV and alpha sets are to be used for higher precision classifications. Further improvements can start with cascade design for improved average operation time.

References

[1] Xilinx, "Zynq-7000 All Programmable SoC Data Sheet: Overview", Product Specification, 2017.

[2] N. NEJI et al, "Architecture and FPGA Implementation of The CORDIC Algorithm for Fingerprints Recognition Systems", in *8th Intl. Multi-Conf. on Sys., Signals & Dev.*, 2011.

[3] D. R. Llamocca-Obregón and C. P. Agurto-Ríos, "A Fixed-Point Implementation of the Expanded Hyperbolic CORDIC Algorithm", in *IBERCHIP*, 2006.

[4] Xilinx, "Xilinx ISE 10.1 Design Suite Software Manuals and Help - PDF Collection", 2008.

[5] M. Papadonikolakis and C.S. Bouganis, "Novel Cascade FPGA Accelerator for Support Vector Machines Classification", *IEEE Trans. on Neural Net. and Learning Sys.*, vol. 23, no.7, pp. 1040-1052, 2012.

Appendix: Precision configuration list

	Variables	Description	Integer bits	Fractional bits
svm_classification.cpp	in	Entry of testData with 16 dimensions	4	12
	*out	Boolean output for the result of classification	1	0
	SVs	Supporting vectors with 16 dimensions	4	11
	alpha	Weight for each SVs	5	8
	s_in	Convert input data to lower precision datatype	4	9
	sum	The result of summation	10	8
	ch_sum[18]	Used for 18 channels	10	8
	bias	Parameter bias	10	8
	temp	Each element of summation	10	8
	parameter_k	The output from kernel	3	5
getKernel.cpp	x (input)	Entry of SV	4	11
	y (input)	Entry of transformed testData	4	9
	*out	The result of kernel (parameter_k)	3	5
	dotPro	Store the result of dot product	7	9
dotProduct.cpp	x (input)	Entry of SV	4	11
	y (input)	Entry of transformed testData	4	9
	pro	The result of dot product	7	9
	*out	The result of 2 times pro		
getTanh.cpp	theta_in (input)	Entry of angle to calculate tanh	7	9
	*out	The result of tanh(theta_in)	3	5
	temp	Temporary result of tanh(abs(theta_in))	3	5
	pro	The result of divider (sinh_out divided by cosh_out)	3	5
	sinh_out	sinh(theta)	3	5
	cosh_out	cosh(theta)	3	5
	neg	The flag that theta_in is negative	1	0
	theta_int	The integer part of theta_in	32	0
	sinh_lut	LUT of sinh(theta_int)	4	6
	cosh_lut	LUT of cosh(theta_int)	4	6
	pow_lut	LUT of arctanh() function	4	12
	sinh_frac	sinh(theta_frac)	4	12
	cosh_frac	cosh(theta_frac)	4	12
	sinsin	sinh(theta_int) times sinh(theta_frac)	4	12
	sincos	sinh(theta_int) times cosh(theta_frac)	4	12
	coscos	cosh(theta_int) times cosh(theta_frac)	4	12
	cossin	cosh(theta_int) times sinh(theta_frac)	4	12
cordic_sinh_cosh.cpp	theta_in	The fractional part of the input from getTanh.cpp	7	9
	*sinh	The result of sinh(theta_frac)	4	12
	*cosh	The result of cosh(theta_frac)	4	12
	x	Iteration for cosh	4	12
	y	Iteration for sinh	4	12
	x_next	Temporary storing the value of x	4	12