Acceleration of an SVM classifier

Zhipeng Cheng, *CID: 02249084, zc1022@ic.ac.uk* and Zhiyuan Zhu, *CID: 02261066, zz4722@ic.ac.uk*

***Abstract*—This report is aimed to illustrate the design of optimization of SVM classifier. During this project, Vivado HLS is used to provide the implementation and optimization for FPGA by creating a customized IP core, including optimizing loop structure, data type, exponential function, CORDIC iteration numbers, and interface. The simulation of speedup is also provided to demonstrate the effect of the accelerating technique.**

I. Introduction

THIS project is aimed to build and accelerate SVM classifier module. The design of the classifier should be packed as an IP core and communicate with ARM processor using AXI interface. This module consists of several parts. First of all, it needs to build a Radial Basis Function (RBF) kernel classifier. The key equation is shown below:

$$y = sgn(\sum_{i=1}^{N_{SV}} y_i a_i K(x_i, x) + b)$$

In the above equation, the input data x is the value of pixels of the image which have the size of 2601 $\times$ 784. Since each image has 28 $\times$ 28 pixels, each of them will calculate the kernel function with support vectors, $K(x_i, x)$. the kernel function contains the square of the difference between x_i, features of SVs, and x, test data, times a coefficient γ. Then calculate the exponent of the result and multiply it with another coefficient $y_i a_i$. Afterward, sum the results of 165 support vectors together and add a bias to it. This is the final value. To get the classification of an input between two digits "1" and "0", the classifier takes the sign of the output. If the output is less than 0, the prediction is 0. If the output is greater or equal to 0, the prediction is 1. By comparing the prediction result with the truth table, we can get the correctness of this classifier by dividing the correct number by the total number.

The above paragraphs are a brief description of SVM classifier. In this project, we use Vivado HLS to implement and optimize this module at a high level using the c++ language. The first step is creating a self-customized IP core and its corresponding test bench through programming in the HLS language. Then speed up the performance of the classifier module by utilizing various solutions and directives which include exploiting fixed-point data type, CORDIC function approximation, parallelism such as unroll and pipeline, and faster interface. Further, Vivado is used to integrate the self-customized IP core and generate the bit-streams. Finally, the firmware is downloaded to the development board and the program is created in SDK to calculate the time consumption of this module and the accuracy of the result.

II. Implementation of the Design

After understanding the basic operation of the classifier, the first attempt is building a basic system that can achieve the functionality of the classifier. The structure in test_classification.cpp is used and modified as the first version of our design. In HLS, the function of the classifier is picked out and configured as the source file. The rest of the code is set as the test bench. The final outputs of C Synthesis and C Simulation are shown in Figure 1. As shown in the figure, the latency is quite long, achieving 2334590 cycles, and the BRAM used by FPGA is exceeding the limitation, whereas the DSP and LUT only occupy a small portion. The simulation result shows 99.65% accuracy. This is the fundamental accuracy given to us. During the following processes of optimization, the remaining resources on the board will be utilized to speed up the calculation.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.232	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2334590	2334590	2334590	2334590	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	101
FIFO	-	-	-	-
Instance	-	40	2311	4326
Memory	514	-	0	0
Multiplexer	-	-	-	407
Register	-	-	641	-
Total	514	40	2952	4834
Available	280	220	106400	53200
Utilization (%)	183	18	2	9

(a) The C Synthesis result of the original version

```
INFO: [SIM 2] ************** CSIM start **************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../c_headers/SVM_tb.cpp in debug mode
   Compiling ../../../c_headers/SVM_1.cpp in debug mode
   Generating csim.exe
Classification Accuracy: 0.996540
Confusion Matrix (2601 test points):
0.455594, 0.001153
0.002307, 0.540946
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] ************** CSIM finish **************
```

(b) The C Simulation result of the original version

Fig. 1. The C Synthesis and Simulation results of the original design

III. CORDIC OPTIMIZATION

It's not enough to just have the bare code and run the program for a long time. The first point to optimize the classifier is the exponent function because it's a separate module where we can test and optimize easily. During this section, we utilize an iterative algorithm to substitute the exponential function named CORDIC.

The CORDIC (Coordinate Rotation Digital Computer) algorithm is a widely-used iterative algorithm that is used to calculate trigonometric, hyperbolic, and other transcendental functions. The algorithm was originally invented by Jack Volder in 1959 for use in digital computers that lacked hardware support for such functions. It operates by rotating a vector through a series of predetermined angles, in a way that ultimately results in the desired calculation. The algorithm is particularly useful in situations where hardware resources are limited, or where high precision is not required.

The CORDIC algorithm has been used in a variety of applications, including digital signal processing, image processing, and numerical analysis. It is also widely used in hardware implementations of digital signal processing algorithms, such as in digital signal processors (DSPs) and field-programmable gate arrays (FPGAs).

$$e^z = \sinh z + \cosh z$$

According to the equation above, in CORDIC algorithm, we can set x as $\sinh z$ and y as $\cosh z$, so the result of the exponential function is the sum of the final x and y. The code of our CORDIC function is shown in Figure 2

```
double cordic_exp(double angle)
{
        double cosh = 1.207;
        double sinh = 0;
        double factor = 0.5;

        int N=20;
        for (int a = 0; a < N; a++){
                int sigma = (angle < 0) ? -1 : 1;
                double temp_cosh =  cosh;
                double mul = sigma * factor;
                cosh = cosh + mul * sinh;

                sinh = sinh + mul * temp_cosh;

                double b = sigma * cordic[a];

                angle = angle - b;

                factor = factor / 2;
        }
        double ex = cosh + sinh;
        return ex;
}
```

Fig. 2. The code of the self-designed CORDIC function in Vivado HLS

In this CORDIC function, the original value of cosh is 1.207, which is the compensation of the length of the vector, since each time the vector rotates, its length of it will change a little bit. The value of this parameter is shown below:

$$\frac{1}{K'} = \prod_{i=1}^{\infty} \cosh 2^{-i} \approx 1.207$$

In this function:

$$x^{(i+1)} = x^{(i)} + d_i y^{(i)} 2^{-i}$$

$$y^{(i+1)} = y^{(i)} + d_i x^{(i)} 2^{-i}$$

$$z^{(i+1)} = z^{(i)} - d_i e^{(i)}$$

$$e^{(i)} = \tanh^{-1} 2^{-i}$$

In the above equations, x is $\sinh z$, and y is $\cosh z$. And z is the rotation angle, whose actual value is the input of this function. Each time the vector will rotate $e^{(i)}$ degrees. Furthermore, the direction of rotation depends on the previous angle, if the previous angle is less than 0, the vector will rotate in a positive direction. If the previous angle is greater than 0, the vector will rotate in a negative direction. After several iterations, the angle will approach 0, and the sinh and cosh will approach the actual result of sinh and cosh of the input value. Therefore, by adding these two values, we can obtain the final iteration result of CORDIC. The outputs of C Synthesis and C Simulation after CORDIC optimization are shown in Figure 3.

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.418	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2392010	2392010	2392010	2392010	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	237
FIFO	-	-	-	-
Instance	-	28	1654	3923
Memory	516	-	0	0
Multiplexer	-	-	-	573
Register	-	-	1233	-
Total	516	28	2887	4733
Available	280	220	106400	53200
Utilization (%)	184	12	2	8

(a) The C Synthesis result of the CORDIC optimization version

```
In file included from C:/Xilinx/Vivado/2018.2/include/hls_fpo.h:166:0,
                 from C:/Xilinx/Vivado/2018.2/include/hls_half.h:58,
                 from C:/Xilinx/Vivado/2018.2/include/ap_int_sim.h:73,
                 from C:/Xilinx/Vivado/2018.2/include/ap_int.h:65,
                 from C:/Xilinx/Vivado/2018.2/include/ap_fixed.h:57,
                 from ../../../c_headers/SVM_1.h:4,
                 from ../../../c_headers/SVM_1.cpp:7:
C:/Xilinx/Vivado/2018.2/include/floating_point_v7_0_bitacc_cmodel.h:135:0: note: th
 #define __GMP_LIBGMP_DLL 1

Classification Accuracy: 0.995771
Confusion Matrix (2601 test points):
0.456363, 0.000384
0.003845, 0.539408
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

(b) The C Simulation result of the CORDIC optimization version

Fig. 3. The C Synthesis and Simulation results of the CORDIC optimization design

According to Figure 3, we could find after CORDIC optimization, the latency even increases from 2334590 to 2392010, which is deviated from expectations. Besides, except the usage amount of resources changes a little bit, the accuracy decreases from 0.996540 to 0.995771. By printing and analyzing the range of the input, the data of the output, and the correct output, we find the range of the input is $(-16, 0)$, and if the input value is less than -1, it will exceed the suitable range of the CORDIC function, which will result in wrong output. Therefore, we come up with an idea to compensate for data exceeding the suitable range. An example of this solution is shown below:

$$e^{-13.56} = e^{-13} \times e^{-0.56}$$

According to this example, we can divide this function into 2 parts. The first one is exponential to the power of the integer part of the input, which can be reserved in lookup table, and the second one is exponential to the power of the decimal part of the input, which can be calculated through CORDIC function. Therefore, the code of this logic is shown in Figure 4.

```
order = ceil(-0.001 * l2Squared);     //round up of the input
double K = cordic_exp(-0.001 * l2Squared - order);    //exp(decimal part)
sum +=  alpha * K * Exponential[-order];
```

Fig. 4. The code of the method to optimize the CORDIC function in Vivado HLS

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2395640	2395640	2395640	2395640	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1564
FIFO	-	-	-	-
Instance	-	28	2066	4568
Memory	518	-	64	15
Multiplexer	-	-	-	669
Register	-	-	1627	-
Total	518	28	3757	6816
Available	280	220	106400	53200
Utilization (%)	185	12	3	12

(a) The C Synthesis result of the second CORDIC optimization version

```
                from ../../../c_headers/SVM_1.h:4,
                from ../../../c_headers/SVM_1.cpp:7:
C:/Xilinx/Vivado/2018.2/include/floating_point_v7_0_bitacc_cmodel.h:135:0: note: thi
 #define __GMP_LIBGMP_DLL 1

Classification Accuracy: 0.996924
Confusion Matrix (2601 test points):
0.454441, 0.002307
0.000769, 0.542484
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

(b) The C Simulation result of the second CORDIC optimization version

Fig. 5. The C Synthesis and Simulation results of the second CORDIC optimization design

The C Synthesis and C Simulation results of this method are shown in Figure 5. We could find that the accuracy increases to 0.996924, which means the error is resolved. As we have mentioned above, the latency of this solution is even larger than the original result, so we consider decreasing the times of iteration as well as remaining the same accuracy. The C simulation results of different iteration times are shown in Figure 6.

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../c_headers/SVM_1.cpp in debug mode
   Generating csim.exe
Classification Accuracy: 0.996924
Confusion Matrix (2601 test points):
0.454441, 0.002307
0.000769, 0.542484
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

(a) The C Simulation result of the second CORDIC optimization version when N=10

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../c_headers/SVM_1.cpp in debug mode
   Generating csim.exe
Classification Accuracy: 0.996924
Confusion Matrix (2601 test points):
0.454441, 0.002307
0.000769, 0.542484
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

(b) The C Simulation result of the second CORDIC optimization version when N=8

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../c_headers/SVM_1.cpp in debug mode
   Generating csim.exe
Classification Accuracy: 0.997309
Confusion Matrix (2601 test points):
0.454825, 0.001922
0.000769, 0.542484
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

(c) The C Simulation result of the second CORDIC optimization version when N=5

Fig. 6. The C Simulation results of the second CORDIC optimization design with different iteration times

After changing the value of N, we find the accuracy starts to change when it is less than 8. However, to our surprise, the accuracy increases when N is small. Although it is higher, we assume it is not accurate, so after consideration, we finally choose 8 as the value of N. And the C Simulation result when N = 8 is shown in Figure 7.

According to Figure 7, we find that when N = 8 the latency is 35640 less than the occasion when N = 20. Although it is still larger than the solution without CORDIC, we assume that after optimizing the loop structure, the advantage of the CORDIC function will be obvious.

IV. Data Type Optimization

Another method to offer higher performance is changing the data type from high precision type to fixed point. There is a trade-off between performance and correctness. Because when the design uses a lower precision data type, there is more fractional part being lost. The correctness may significantly decrease due to the precision loss. The floating point is used at the beginning to simulate the process of the loss of accuracy and improve performance. Where we replace all the double precision to float which is single precision. The result is shown in Figure 8 below:

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.975	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2360000	2360000	2360000	2360000	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1562
FIFO	-	-	-	-
Instance	-	28	2066	4568
Memory	518	-	64	15
Multiplexer	-	-	-	669
Register	-	-	1625	-
Total	518	28	3755	6814
Available	280	220	106400	53200
Utilization (%)	185	12	3	12

Fig. 7. The C Synthesis result of the second CORDIC optimization version when N=8

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.442	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2095505	2095505	2095505	2095505	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1517
FIFO	-	-	-	-
Instance	-	24	2164	4448
Memory	259	-	64	16
Multiplexer	-	-	-	620
Register	-	-	1267	-
Total	259	24	3495	6601
Available	280	220	106400	53200
Utilization (%)	92	10	3	12

(a) The C Synthesis result of the floating data type version

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../c_headers/SVM_1.cpp in debug mode
   Generating csim.exe
Classification Accuracy: 0.996540
Confusion Matrix (2601 test points):
0.454441, 0.002307
0.001153, 0.542099
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

(b) The C Simulation result of the floating data type version

Fig. 8. The C Synthesis and Simulation results of the floating data type design

Compared with the speedup obtained from the CORDIC optimization, It reduces 264495 cycles from 2360000 to 2095505. The latency saved is nearly 8 times more than the CORDIC algorithm which shows that the transformation to a shorter data type is an efficient option. Therefore, the floating point data type is replaced by a fixed point in the following step. The fixed point type is specifying the data with a fixed decimal point which is different from double or float. The fixed point doesn't require the calculation of a decimal point and only requires basic integer operations such as addition, subtraction, multiplication, and division. Therefore, it simplifies the process of calculation. On the other hand, it also occupies smaller storage space which can save BRAM resources on the board. The size of some parameters is given to us and we also try and test the suitable configuration for the CORDIC algorithm and classifier function including their intermediate variables. Finally, the simulation results are shown in the figure below:

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.516	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
798106	798106	798106	798106	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	16	40	5023
FIFO	-	-	-	-
Instance	-	0	440	692
Memory	259	-	64	16
Multiplexer	-	-	-	313
Register	-	-	2205	-
Total	259	16	2749	6044
Available	280	220	106400	53200
Utilization (%)	92	7	2	11

(a) The C Synthesis result of the fixed point data type version

```
Vivado HLS Console
                 from C:/Xilinx/Vivado/2018.2/include/ap_int.h:65,
                 from C:/Xilinx/Vivado/2018.2/include/ap_fixed.h:57,
                 from ../../../c_headers/SVM_1.h:4,
                 from ../../../c_headers/SVM_1.cpp:7:
C:/Xilinx/Vivado/2018.2/include/floating_point_v7_0_bitacc_cmodel.h:135:0: note: this
 #define __GMP_LIBGMP_DLL 1

Classification Accuracy: 0.996155
Confusion Matrix (2601 test points):
0.454056, 0.002691
0.001153, 0.542099
1.000000,
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

(b) The C Simulation result of the fixed point data version

Fig. 9. The C Synthesis and Simulation results of the floating data type design

As shown above, the latency has been significantly reduced from 2095505 to 798106, representing a reduction of one-third of the previous result. This improvement can be attributed to the use of fixed point data types. While the number of DSP, FF, and LUT decreases slightly, the reduction in latency is the primary benefit of this approach. The secondary advantage is the smaller storage space. When it comes to the accuracy, due to the suitable length of each variable, the accuracy is still relatively high which is 99.61%. The reason why the fixed-point data type can provide a significant improvement in latency and resource usage is that it uses a simple representation and calculation rules that only require basic integer operations. Fixed-point arithmetic also avoids complex operations such as exponentiation for the decimal point. Furthermore, it is widely supported and optimized in modern devices such as integer

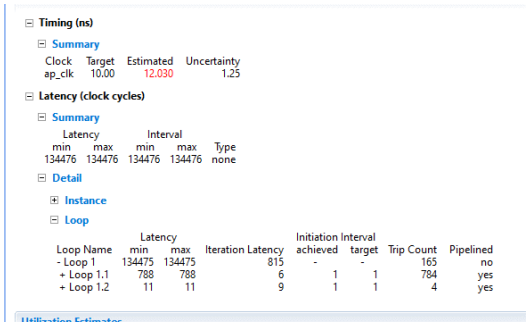
adders and multipliers. Therefore, the fixed-point data type can provide a tremendous improvement in latency. Additionally, it can also save storage space because fixed-point data typically use a fixed width, which limits their precision and range. Moreover, they do not require a calculation of exponent, which saves space compared to other data types.

V. Data process optimization

In this section, we implement some methods of data process optimization, which improves performance as well as sacrifices a little bit of accuracy. Firstly, in the kernel function, we are supposed to multiply a parameter called γ, which is 0.001. Since the binary value of γ approximate 0.0000000001, we change code to $m = -l2Squared >> 10$ instead of $m = -0.001 * l2Squared$. The latency decreases from 798106 to 797941, while the accuracy reduces to 99.5%, which is still acceptable.

Secondly, in CORDIC function, for 2^{-i} we modify $factor\,/\,2$ to $factor >> 1$. Although it does not change the latency, the usage volume of FF and LUT decreases a little bit to 2653 and 5960 respectively.

Thirdly, as we have mentioned before, we use the function ceil() to round up the data. Later we find that using (int), data type casting, could also achieve the same value. Compared to the accuracy decreasing to 99.46%, the latency falls sharply to 134476, which is shown in Figure 10. Considering a tremendous decrease in latency, this small loss of accuracy is acceptable.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	12.030	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
134476	134476	134476	134476	none

Detail

Instance

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	134475	134475	815	-	-	165	no
+ Loop 1.1	788	788	6	1	1	784	yes
+ Loop 1.2	11	11	9	1	1	4	yes

(a) The C Synthesis result of the remove ceil function version

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../c_headers/SVM_1.cpp in debug mode
   Generating csim.exe
Classification Accuracy: 0.994617
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

(b) The C Simulation result of the remove ceil function version

Fig. 10. The C Synthesis and Simulation results of the remove ceil function design

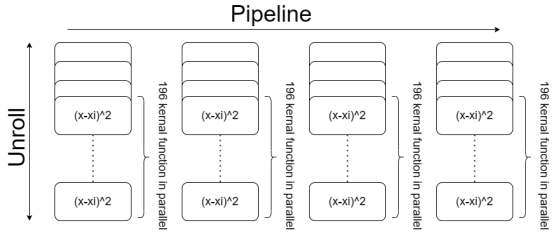
Lastly, when we split the kernel function, there is an unreasonable operation that appears which is UREM and it takes a lot of cycles to finish, which significantly affects our performance. After searching online and discussing with teammate, this operation seems to happen due to the possibility of overflow when indexing the support vector and input data using the loop variables. The HLS thinks that the index is possible to overflow, therefore, the index is performed an unsigned remainder operation whose abbreviation is exactly UREM. The solution is calculating the index in advance to prevent the compiler from misunderstanding it.

VI. Parallelism structure optimization

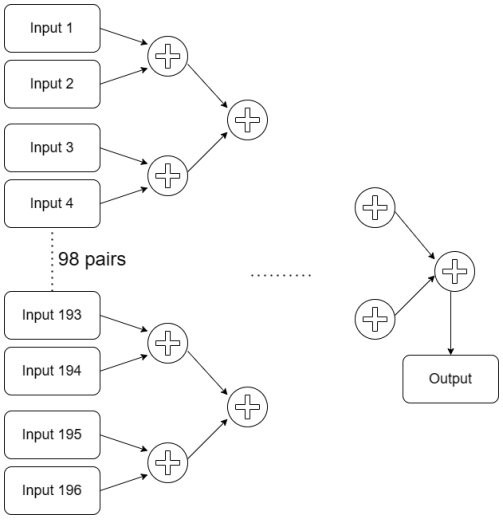
To take full advantage of the resource on the FPGA board to improve the overall performance, we need to utilize parallelism to trade the resource with speed. To specifically do it, the operation that doesn't have data dependency should be first captured. If we search the operation in the classifier, it is obvious that there are two loops that seem to have data dependency but actually, it's fake. The operation of kernel function and the summation of the weighted kernel output which is:

$$\sum_{i=1}^{N_{SV}} y_i a_i K(x_i, x)$$

The kernel function of each element can be calculated separately without interfering with each other. The summation of the kernel output can also be implemented by using an adder tree to improve the performance. The block diagram of these two methods is shown in the figure11:



(a) The block diagram of the data flow using unroll and pipeline



(b) The block diagram of the adder tree architecture

Fig. 11. The block diagram of the optimization of the architecture

As shown in the diagram, we choose to separate the 784 kernel function into 4 times 196 where 196 kernel operation is processed simultaneously and the out 4 loops is pipelined. The reason we optimize the architecture this way is that we find it is the best way of taking advantage of programmable

Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
37786	37786	37786	37786	none

Detail

Instance

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	37785	37785	229	-	-	165	no
+ Loop 1.1	200	200	25	-	-	8	no
+ Loop 1.2	13	13	9	1	1	6	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	98	-	-
Expression	-	111	0	13646
FIFO	-	-	-	-
Instance	-	0	315	767
Memory	99	-	64	14
Multiplexer	-	-	-	743
Register	0	-	7637	32
Total	99	209	8016	15202
Available	280	220	106400	53200
Utilization (%)	35	95	7	28

(a) The C Synthesis result of unrolling inner loop of 8×98 structure

Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
10066	10066	10066	10066	none

Detail

Instance

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	10065	10065	61	-	-	165	no
+ Loop 1.1	32	32	25	1	1	8	yes
+ Loop 1.2	16	16	7	2	1	6	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	98	-	-	-
Expression	-	14	0	14089	-
FIFO	-	-	-	-	-
Instance	-	0	838	1203	-
Memory	99	-	64	14	-
Multiplexer	-	-	-	257	-
Register	0	-	4932	224	-
Total	99	112	5834	15787	0
Available	280	220	106400	53200	0
Utilization (%)	35	50	5	29	0

(b) The C Synthesis result of pipelining outer loop and unrolling inner loop of 8×98 structure

Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
7426	7426	7426	7426	none

Detail

Instance

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	7425	7425	45	-	-	165	no
+ Loop 1.1	15	15	8	1	1	8	yes
+ Loop 1.2	13	13	9	1	1	6	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	98	-	-
Expression	-	13	0	16991
FIFO	-	-	-	-
Instance	-	0	100	138
Memory	99	-	64	14
Multiplexer	-	-	-	309
Register	0	-	6496	96
Total	99	111	6660	17548
Available	280	220	106400	53200
Utilization (%)	35	50	6	32

(c) The C Synthesis result of solving UREM problem of 8×98 structure

Fig. 12. The C Synthesis results of optimization of the 8×98 structure

logic. Initially, we split the structure into 8 times 98. The results of it are shown in Figure 12:

It is remarkable that after using add tree structures and unrolling the inner loop, the latency decreases from 134476 to 37786. As shown in Figure 12(b), pipelining the outer loop can save the DSP resource, usage of DSP decreases from 95% down to 50%. Partition is also needed before we actually moving data between the PL. To improve the data access on programmable logic, the data should first be separated into 8 blocks, each of which has 98 elements. Besides, as discussed previously, the UREM operation is optimized by calculating the index in advance. The result is shown in Figure 12(c), The latency is again reduced to 7425 cycles. However, the resource is not increasing too much. This consequence shows that there is still a possibility that more parallelism can be employed. Therefore, we choose to split the loop into 4 times 196, where 196 operations are in parallel and the outer 4 loops are pipelined. partition should also changed to 4 block with 196 elements in each block, as shown in Figure 11(a). The results of 4 times 196 are shown in Figure 13.

Latency min	Latency max	Interval min	Interval max	Type
9406	9406	9406	9406	none

Detail

Instance

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	9405	9405	57	-	-	165	no
+ Loop 1.1	28	28	25	1	1	4	yes
+ Loop 1.2	16	16	7	2	1	6	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	185	-	-	-
Expression	-	14	0	25530	-
FIFO	-	-	-	-	-
Instance	-	0	2314	5028	-
Memory	349	-	104	124	-
Multiplexer	-	-	-	248	-
Register	0	-	14678	2496	-
Total	349	199	17096	33426	0
Available	280	220	106400	53200	0
Utilization (%)	124	90	16	62	0

(a) The C Synthesis result of 4×196 structure

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	11.062	1.25

Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
6766	6766	6766	6766	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	184	-	-
Expression	-	13	0	30353
FIFO	-	-	-	-
Instance	-	0	100	138
Memory	175	-	74	124
Multiplexer	-	-	-	309
Register	0	-	10278	96
Total	175	197	10452	31020
Available	280	220	106400	53200
Utilization (%)	62	89	9	58

(b) The C Synthesis result of 4×196 structure with array_partition

Fig. 13. The C Synthesis results of optimization of the 4×196 structure

In Figure 13 we make a comparison of whether to use array_partition to dispose of the data or not. Figure 13(b) is the design after partitioning the support vector and input data to 196 elements per block. The partition mode is configured to cyclic which is suitable for the BRAM to transfer data in PL. This attempt shows that better storage of parameters can give a better performance as well. After utilizing this directive, the usage amount of BRAM and FF reduce sharply. This 4×196 method has a relatively high utilization rate in DSP without exceeding the resource limitation. The DSP has 89% percent usage. It's hard to parallel more operations because the DSP is running out. That's the reason we decide to use a 4×196 structure.

VII. Export Customized IP Core to SDK

After finishing the above optimizations, we are going to export this customized IP core. Because there are 4 ports of the HP interface, we divide the input data into 4 groups and transfer it through 4 ports of the master interface. Then, using Vivado to integrate the self-customized IP core and generate the bit-streams. The integration of the IP core is shown in Figure 14.

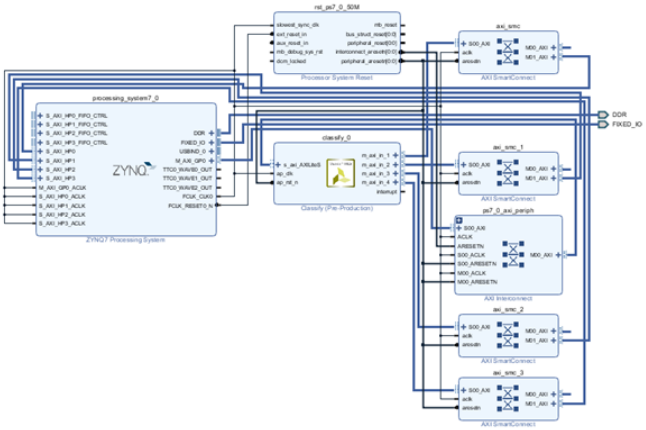


Fig. 14. The structure of the integration of IP core in Vivado

Finally, the firmware is downloaded to the development board and the program is created in SDK to calculate the time consumption of this module and the accuracy of the result. The part of the SDK code is shown in Figure 15.

In this program, the test_data is defined as 2601×784 data. For each image, the test data is divided into 4 parts and is read by 4 arrays. Then, these arrays will be transferred into hardware to run SVM classifier. The output of the hardware will be used to calculate the accuracy of the precision results.

VIII. Conclusion and Introspection

The final hardware time elapsed generated by SDK is shown in Figure 16. It takes 202.34 ms to finish the whole process, which is far more than expected. Additionally, the accuracy calculated in SDK turns out to be unexpected. After printing all the output in the terminal, we figure out that the values are completely positive which means the accuracy is the rate of the positive value in the ground truth table compare to

```
    XTime TimeEnd, TimeStart;
    XTime_GetTime(&TimeStart);

for (int i=0; i<NUM_IMGS; i++) //each picture
{
    // form input vector x

     for (int k=0; k<196; k++)
     {
            x1[k] = test_data[i*IMG_SIZE+k];
            x2[k] = test_data[i*IMG_SIZE + 196 + k];
            x3[k] = test_data[i*IMG_SIZE + 392 + k];
            x4[k] = test_data[i*IMG_SIZE + 588 + k];
     }


           // form input vector x
                  XClassify_Set_x1_V(&HlsClassify, (u32)x1);
                  XClassify_Set_x2_V(&HlsClassify, (u32)x2);
                  XClassify_Set_x3_V(&HlsClassify, (u32)x3);
                  XClassify_Set_x4_V(&HlsClassify, (u32)x4);
                  XClassify_Start(&HlsClassify); // start HW execution

           do {
                   scores[i] = XClassify_Get_output_V(&HlsClassify);
           } while (!XClassify_IsReady(&HlsClassify)); // wait until HW finishes
}

     XTime_GetTime(&TimeEnd);
```

Fig. 15. The code of part of the main.c file in SDK

```
HW Time (ns): 202335822

The end of the prediction

Classification Accuracy: 0.540177
```

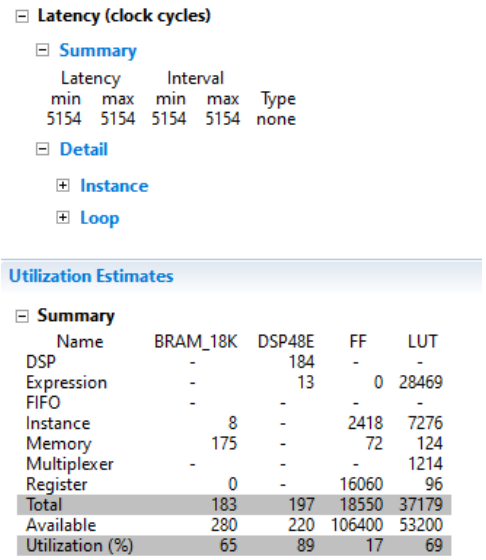
Fig. 16. The hardware time elapsed in nano second

the total number. One of the possible reasons why the output of our design is completely positive is that the SDK doesn't support the fixed point data. Therefore, the test data will have to be multiplied by 2 to adapt to the ap_fixed<8,7> fixed point data type where ap_fixed<8,7> means 8 bit long, 7 bit for the integer part and 1 bit for the fractional part. When it converts to int data type, the value is stored in the lower 8-bit and the value is 2 times the original value. We have tried to use long, int and signed char to store the value. Unfortunately, all the results are positive. Maybe there are other protocols for storing the data that we don't know. Another reason may be the timing issue generated in Vivado when generating the bit stream. There are several critical warnings in meeting the timing requirement. We don't know how to fix it. Therefore, the timing issue may affect our final result.

As for such long hardware processing time, the reason why it takes 203 ms to finish the process is that in our SDK code, we have an initialization step in which we put the input data into a specific pattern to feed into our design of each image. We actually measure the time of dividing and disposing of data in SDK before reading into hardware. This process takes a lot of time that we can't avoid.

Another problem we have met is that in Vivado HLS if we defined the data type of parameters as fixed point in the lookup table, the C Simulation will encounter a compilation error so that we can not obtain the accuracy. Therefore, we have to define these data as float type in the lookup table and use type casting to change float to fixed point in the source file, which, however, will increase the latency of the module. On the contrary, If we set data as fixed point in lookup table, the latency will achieve 5154 cycles which is the best. The

result of this solution is shown in Figure 17.



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5154	5154	5154	5154	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	184	-	-
Expression	-	13	0	28469
FIFO	-	-	-	-
Instance	8	-	2418	7276
Memory	175	-	72	124
Multiplexer	-	-	-	1214
Register	0	-	16060	96
Total	183	197	18550	37179
Available	280	220	106400	53200
Utilization (%)	65	89	17	69

Fig. 17. The C Synthesis result of the final version

Additionally, when we decide to combine our four 196 arrays into one 784 arrays in HLS, we think it may take more resources to combine them manually. Therefore, initially, we used the array_map directive to group them together. However, when we add the m_axi interface to control the data flow, the HLS didn't synthesize the master interface at all. Every port is s_axilite interface. We debugged it for a whole day and finally found that we had to combine it manually rather than using a directive. Otherwise, the interface will be kept using slower axilite.

If there are some drawbacks or defects in our design, it might be a failure of taking full advantage of the LUT which still can be improved to do the calculation simultaneously. In addition, our architecture may not be able to maximize performance. Maybe a different splitting scheme will have better performance in this task. Even though we used four HP ports for transferring the data, we didn't pack the data more tightly and only use 8 bit in 64 bit bandwidth. This may result in poor memory accessing time. This may be the reason why the actual latency becomes longer than we expected.

To conclude, although we did not get the aim latency in our prediction and the correct accuracy, we have gained and learned a lot during the process of the coursework and lecture. We have learned how to design a digital system using high level synthesis, and how to optimize the operation on hardware including using fixed point, parallelism structure, and the trade-off between resource and speed, which we think are beneficial to our future study and career.