Hardware-Software Co-Design for Accelerating the Image Searching Program "Find Face"

Chih-Yung Liang(梁智湧)
Department of Computer Science(資訊工程學系)
National Chiao Tung University, Taiwan(國立交通大學)
stu89424@gmail.com

***Abstract*—This study introduces several methods with hardware-software co-design for accelerating an image searching program "Find Face" running on the experiment board ZedBoard. "Find Face" is a program originally executed only by software that find 4 target faces in a group picture by using sum of absolute difference (SAD). These target pictures and group picture are initially stored in the SD card attached on the experiment board. The program can load these pictures from the SD card, doing median filtering, and find the positions of four target faces by calculating SAD. By behavior profiling, it is observed that the hotspot of this program lies on a sub-routine containing many loop iterations. The key idea to accelerate the program is to migrate this portion to hardware to parallelize it. Since there exists many overhead communicating between software and hardware, other mechanisms are applied to the acceleration to improve the overall performance. After applying the acceleration in this study, the program gains a speedup for approximately 400000 times.**

Keywords—co-design; block ram; pipeline; master IP; sum of absolute difference; behavior profiling

I. INTRODUCTION

"Find Face" [1] is an image searching program that find positions of four target faces in a group picture by finding the minimal sum of absolute difference (SAD) [2]. Witten in *C* language, it is originally processed only by software, and is slowed down by a portion of its implementation with many loop iterations. By performing behavior profiling to this program, it can be observed that the hotspot comes from one of its sub-procedure. In this study, the goal is to improve the performance of this program running on ZedBoard™ [3]. With hardware cooperation, this sub-procedure is accelerated by several mechanisms, making it run many times faster than the original version. In this report, the program "Find Face" along with the profiling methods and results are introduced. After that, some acceleration mechanisms taken on the program and the result analysis will be described in detail. In the end, a conclusion with respect to the acceleration are discussed.

II. THE ENVIRONMENT OF THE EXPERIMENT

- Experiment board: ZedBoard™ (Xilinx Zynq-7020 APD) [3]
  - 2 ARM Cortex A9's @ 667MHz
  - FPGA block @ 100MHz
  - 512MB DDR3 DRAM
  - Integrated UART, SD, USB, and Ethernet controller
  - AXI4 are used for IP integration
- Development tool: Xilinx EDA Tools [4]
  - Hardware design IDE: Xilinx Platform Studio (XPS)
  - Software development IDE: ISE Software Development Kit (SDK)

III. THE PROGRAM BEING ACCELERATED

"Find Face", as a course material, is originally a software-only program used to find a specified face from a picture with a group of faces. The program accomplishes this by looking for a region that have the minimal SAD, an algorithm for measuring the similarity between image blocks by summing absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison. There are five images, stored in a SD card installed on the device, to be given as inputs of this program, one for group picture (1920*1080 pixels) and the other four for target pictures (32*32 pixels). "Find Face" will do from reading these images from SD card to completing the searching and output the result.

For the steps in detail, the program starts by reading two images from the SD card, and after performing median filtering [5] for noise removal, do the computation to find the face.

A. Read Two Images

Completed by function *read_pnm_image*, two images are going to be read from the SD card. One for the bigger group picture that supposed to contain many faces; the other for the smaller target picture that contains only a face to look for. These images, in Portable GrayMap format (PGM), are loaded by using a library for Portable AnyMap format (PNM) [6], and after successful reading, the data of each image can be accessed as a 2-D byte-array in the memory of the device.

B. Perform Median Filtering

Median filter is a nonlinear digital filtering technique that is often used to remove noise. Completed by function *median3x3*, "Find Face" performs a 2-D median filtering using a 3x3 window to the image that contains a group of faces.

C. Find The Face

To determine which region in the group picture is the most similar one to the target picture, the algorithm is to iterate all the possible regions in the group picture to find out the one that has the smallest SAD compared with the target picture. A function

```
int32 compute_sad(  uint8 *image1, int w1,
                    uint8 *image2, int w2, int h2,
                    int row, int col)
{
    int  x, y;
    int32 sad = 0;

    for (y = 0; y < h2; y++)
    {
        for (x = 0; x < w2; x++)
        {
            /* compute the sum of absolute difference */
            sad += abs(image2[y*w2+x]
                    - image1[(row+y)*w1+(col+x)]);
        }
    }
    return sad;
}
```

Fig. 1. The original code of function *compute_sad*.

```
int32 match(CImage *group, CImage *face,
            int *posx, int *posy)
{
    int  row, col;
    int32  sad, min_sad;

    min_sad = 256*face->width*face->height;
    for (row = 0; row < group->height-face->height; row++)
    {
        for (col = 0; col < group->width-face->width; col++)
        {
            /* compute the matching cost at (col, row) */
            sad = compute_sad(group->pix, group->width,
                        face->pix, face->width, face->height,
                        row, col);
            /* if the matching cost is minimal, record it */
            if (sad <= min_sad)
            {
                min_sad = sad;
                *posx = col, *posy = row;
            }
        }
    }
    return min_sad;
}
```

Fig. 2. The original code of function *match*.

named *compute_sad* (Fig. 1) in the program iterates each pixel in a region of one image to compute the SAD of that region, and is used in every iteration of each region in the big picture by another function named *match* (Fig. 2). The program remembers the location and its SAD where the minimal SAD occurs, namely the most possible face being looking for, and reports it to the user as the completion of the execution.

IV. Profiling Methods and Results

As a program scaling up and executing slower, profiling it becomes important to understand the behavior of it and help us to find out the hotspots in the executing code. A hotspot is a region of computer program where a high proportion of executed instructions occur or where most time is sent during the program's execution [7] that programmers wish to speed up.

There are two common profiling methods, sampling-based and real-time [8], to be described below:

A. The Sampling-Based Profiling

A sampling-based profiler inserts a timer interrupt service routine into programs and uses interrupt of a fixed frequency to collect the content of the program counter (PC) as samples during the executions of those programs. For each sampled value, the profiler is able to know which subroutine the address contained in PC is in by looking up the Executable and Linkable Format (ELF) file. This makes the profiler have the ability to, by means of statistical methods, calculate the frequency and time spent in each subroutine, as a result that is wished to produce by performing profiling.

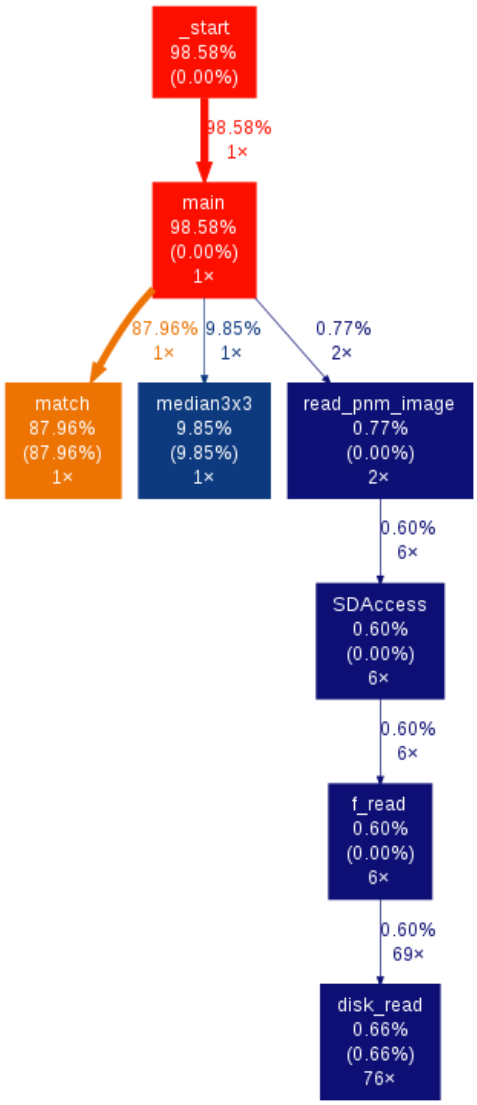

Fig. 3. The visualized behavior of "Find Face" profiled under release mode.

Fig. 3 is a visualized sampling-based profiling result of "Find Face". In a visualized result, each block represents a subroutine that had been sampled and shows the percentage of the time spent on that subroutine. Child subroutine is connected to its parent subroutine by an arrow. Notice that because of the optimization done by the compiler, some subroutines may be inline combined to its parent subroutine so these subroutines don't appear on the visualized result.

B. The Real-Time Profiling

The concept of real-time profiling is to use timer to measure the time spent in a block of code. For the development in detail,

```
1. Reading images, done in 149 msec.
2. Median filtering, done in 898 msec.
3. Face-matching, done in 8101 msec.
** Found the face at (881, 826) with cost 3080
```

Fig. 4. The terminal output of "Find Face", showing the results and times spent in each functions, under release mode.

```
1. Reading images, done in 149 msec.
2. Median filtering, done in 3175 msec.
3. Face-matching, done in 10305 msec.
--- compute_sad done in 9164 msec. ---
** Found the face at (881, 826) with cost 3080
```

Fig. 5. The terminal output of "Find Face" after inserting a timer to measure the time spent in the function *compute_sad*.

usually two timestamps are used to record the times entering and leaving a block of code, and the difference between them would be the time spent in the block. Fig. 4 is the real-time profiling result of "Find Face" after inserting code to measure time spent in each three major functions, *read_pnm_image*, *median3x3*, and *match*.

In order to have detail also about the function *compute_sad*, a timer is inserted into the function to measure the time spent in each time of executing. Because this function is being called repeatedly, it is desired that to accumulate the spending time. Fig. 5 is the result after inserting code to measure time spent on *compute_sad*.

Although the total time spent in *compute_sad* is calculated, the result is not as perfect as expected. For the function *match*, extra 2,204 microseconds are spent after inserting the timer to measure *compute_sad*. This is mainly because the timer has some negative effects that cannot be ignored and causes the execution slow down.

By the result of behavior profiling, the hotspot of "Find Face" comes from its sub-procedure, *match*. It is due to *match* processes a time consuming sub-procedure *compute_sad*, which loops 1024 times, for every position in the group picture. Taking advantage of parallel computing, the hardware may accomplish *compute_sad* in only few clock cycles, and make it the reason to migrate this portion from software to hardware.

V. The Hardware Acceleration Concept

Although it sounds reasonable to migrate only *compute_sad* to hardware, with the assumption that only *compute_sad* is parallelizable, leaving the remaining part of match in software is still not perfect enough because doing this would require lots of memory communication between hardware and software. For the reason that memory communication has huge costs on performance that flaw the acceleration, the design should have memory communication as less as possible. As a result, the principle of the acceleration starts with migrating the whole *match* function to hardware. The following describe the acceleration concept step by step:

*A. Implement **compute_sad** in hardware*

Instead of the original iterative style implemented in software, 1024 subtractors are utilized to calculate all 1024 differences between each target image pixel and corresponding group image pixel, and after obtaining absolute values of these difference, these absolute values are summed up as the result of SAD of this region. The summing task is done by an adder-tree, which is too resource consuming to utilize another. Because of the long propagation delay, the adder-tree has to take several cycles to accomplish.

As the calculation of SAD is migrated from software to hardware, the parallelized computation has make the loop iteration originally done in software finish in only few clock cycles, improving the performance significantly. However, due to hardware resource restriction, it is impossible to utilize another same adder-tree to work with more than one target picture simultaneously.

*B. Implement the whole **match** in hardware*

To minimize the memory communication between hardware and software, instead of controlling the flow in the software, it is better to implement the complete *match* in hardware. The software will not send any pixel data to the hardware; however, it gives the memory addresses of 2D-arrays all 5 pictures, including one group picture and four target pictures, to have the hardware get those pixel data by itself. After sending these addresses, all the software has to do is to trigger the hardware and wait for the finish response from the hardware.

To fulfill this concept, namely, to prevent from depending on the software, the hardware must have the ability to read pixels from a given address of memory, so *master IP* [9] is chosen to implement the hardware. On the other hand, in this implementation, due to the insufficient hardware resource to parallel compute SAD of four faces, these faces are searched separately and sequentially. For each face, the searching flow is same as the original one in the software, iterating every region of position, row by row and column by column, to get the minimal SAD. The minimal SAD, along with the position where it occurs, of each face is memorized, and all these values, accompanied with a finish signal, are sent back to the software when the *IP* finishes searching all the faces.

C. Improve the utilization of group data

Since the SAD computation requires all pixels of the face and the region of the group to be accessible asynchronously for performance, pixels are preloaded into distributed RAMs (dRAM[1]). Because distributed RAMs are constructed by lots of LUTs, it is resource consuming that only two RAMs of 1024 bytes of this kind are created for storing pixels of the target and the region of the group. Thus, there must be some data dropped from the group RAM and some new data inserted into it for each iteration computing SAD.

[1] Abbreviation for distributed RAM, not to be confused with Dynamic Random Access Memory (DRAM).

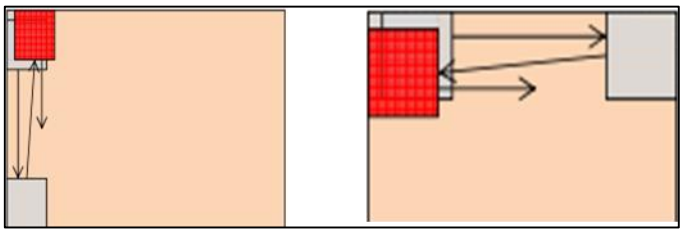

Fig. 6. The left one illustrates round-robin top-down scanning path. The right one illustrates round-robin left-right scanning path.

As the result, the iterating way, or the scanning path, affects the performance a lot. Taking the round-robin top-down scanning path, illustrated in the left of Fig. 6, for example, this scanning method drops and inserts a row of group pixels for each iteration in the same column. When the iteration jumps to next column, all group pixels of this column have to be loaded again, although most of these pixels have been loaded before. In general, most pixels of group picture are loaded from memory for 32 times.

In consideration with the time costs for memory reading, it would obtain a large improvement on performance if all pixels in the group picture are read from memory only once. To fulfill this concept, 32 32bit-width block RAMs (BRAM) are used in the *IP*, each storing a whole row of pixels of the group picture, and the scanning path is changed to the round-robin left-right style, illustrated in the right of Fig. 6. Before starting the iteration of a row, all 32 involved rows of pixels are loaded and each row is stored into a respective block RAMs; then, each iteration of a row can be simplified to loading pixels of a column from BRAMs into dRAMs and computing the SAD. Fig. 8 illustrate how pixels are loaded from memory to BRAMs and extracted to dRAMs. When the iteration continues into next row, these BRAMs are shift up by one row to reuse those 31 already loaded rows of pixels, and the new row of pixels is loaded into the unused BRAM. With help of BRAM and the change to the scanning path, none of the pixels in the group picture is loaded from memory more than once, and the performance gets an effective improvement.

D. Compute SAD in pipeline style

The SAD computation must spend a couple of cycles to finish to prevent a long propagation delay exceeds the timing constraint. In other words, the output result retrieved from the computation is related to the input that is fed few cycles ago. After the hardware is accelerated by those methods previous

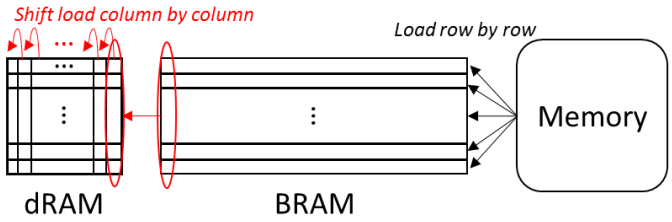

Fig. 8. The data flow illustration. Pixels are loaded from memory to BRAM row by row, and are shift-loaded from BRAM to dRAM one column each cycle.

mentioned, this cycle delay makes the SAD computation become the slowest portion of the execution, even slower than those involved with memory.

However, by taking advantage of pipeline implementation, these delay cycles can be completely eliminated. Because of the characteristic that the output retrieved after few cycles will not be influenced by any input not relating to it, the input pixels can be fed into the computation cycle by cycle, without immediately waiting for the output result, and the corresponding output SADs will be available after some amount of cycles also cycles by cycles. For the implementation, loading pixels from BRAMs and feeding pixels into the SAD computation are done simultaneously, meaning that the computing iteration of a row can be finished in only a little more than 1920 cycles.

E. Load next row while computing SAD

Because the SAD computing iteration of a row still consumes more cycles than loading a row of pixels into BRAMs, loading next row into BRAMs can be done at the same time SADs are computed. The second port and the remaining space of BRAM are used to accomplish this concept. Although this improvement will not greatly accelerate the program, as mentioned that the slowest part is still the SAD computation, it still makes difference on executing time when the program has already become very fast.

VI. THE OVERVIEW OF THE PROGRAM

The software of the program is actually doing not much now, while most part of the program, except for image reading and median filtering, are migrated to hardware, and the data communication between software and hardware is left with a trigger/complete signal and few addresses for pictures. To realize the behavior of the hardware of this program, Fig. 7 illustrates the finite state machine of it, and the behavior is introduced by the following brief description to each state.

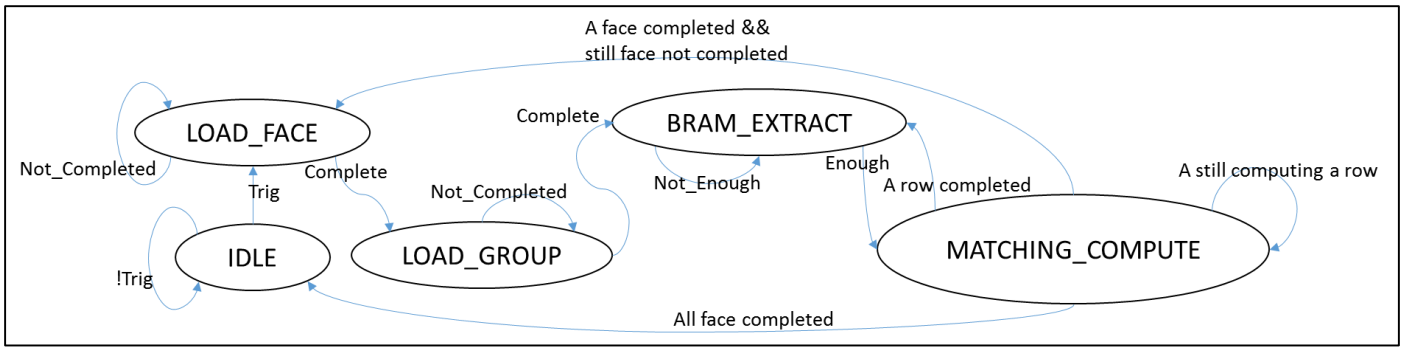

Fig. 7. The finite state machine of the hardware IP.

A. IDLE

Being in this state represents that the hardware *IP* has not received any request from the software since the previous execution completed, and the hardware *IP* will keep sending the complete signal as long as the state is not jumping away. The state jumps to *LOAD_FACE* when a trigger signal is set by the software.

B. LOAD_FACE

Pixel data of face are loaded from memory into the dRAM of the hardware *IP* in this state. As four different occasions entering this state, the corresponded face is chosen to be loaded. The state jumps to *LOAD_GROUP* when the face loading is completed.

C. LOAD_GROUP

In this state, the hardware *IP* loads pixel data of the first 32 rows of the group picture from memory to BRAMs. The state jumps to *BRAM_EXTRACT* when the loading is completed.

D. BRAM_EXTRACT

In this state, the first 32 pixels of each of the 32 rows to be computed are extracted from the corresponded BRAM to the dRAM in the *IP*. After that, the dRAM is ready for computation, and the state jumps to *MATCHING_COMPUTE*.

E. MATCHING_COMPUTE

This is the most complex state of the hardware *IP* where the pipeline computation for SAD occurs. In this state, the SAD results is generated by the computation for each clock cycle while BRAMs keep extracting pixels into the dRAM in order to have the result after a couple of cycle. Each generated SAD value is compared to the minimal one of the searched face recorded until that cycle, and replace it if the generated one is smaller. At the same time the replacement occurs, the position of that result is also recorded. Notice that the minimal SAD of different faces are recorded separately, and the comparisons only involve the one relative to the searched face.

At the same time the computation occurs, the *IP* loads pixels of next row to be computed into the BRAM responsible for that row. After the computation of the row completes, the state jumps to *BRAM_EXTRACT* if there is still row to compute, jumps to *LOAD_FACE* if there is still face to search, or jump to *IDLE* if everything is completed.

VII. The Execution and Performance Analysis

A. The execution

Fig. 9 is a snapshot of the executing result. As the snapshot shows, the found positions along with its *costs* are correct compared with the original version before acceleration, and the time spending on searching four faces is 80 ms.

B. The effect after applying each acceleration

Table 1 demonstrates the time spending on *match* after applying each acceleration. Some of these methods have the effect making the program run few times faster while some decrease the executing time for couples of microseconds. However, the combination of these acceleration methods makes the great performance.

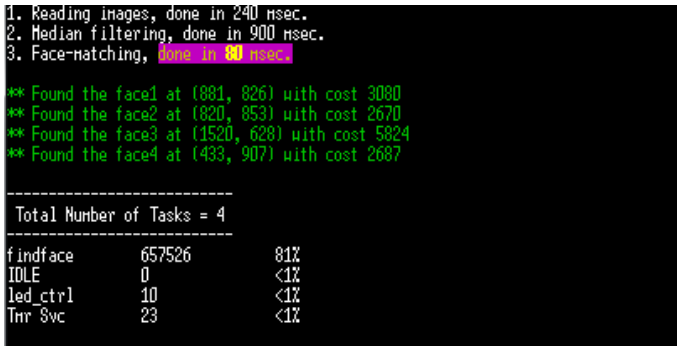

Fig. 9. The snapshot of the program execution.

Table 1 The executing time and acceleration effect of each method

Method	Executing time	Acceleration effect
Original	32 sec	X
A&B	1600 msec	95% reduced
C	770 msec	52% reduced
D	110 msec	6/7 reduced
E	80 msec	30 msec reduced

C. Performance analysis

The following roughly analyzes the reason for spending 80 ms to accomplish the *match*. Table 2 estimate the clock cycles required for each state.

Table 2 The estimated number of cycles required in each state

State	Reason	Cycles[2]
LOAD_FACE	load 256 words	400
LOAD_GROUP	load 15,360 words	23,000
BRAM_EXTRACT	extract 9 words	9
MATCH_COMPUTE	iterate 1,978,624 regions	1,978,624
TOTAL		2,002,033

The total number of cycles listed in Table 2 is the estimated one for one target face. Searching four faces requires 4 times of it, about 8,000,000 cycles, to accomplish. As a result, executing on the hardware with clock rate of 100MHz, 8,000,000 clock cycles spend 80ms to get the result of four faces searching, which seems to meet the real result executed on the machine.

[2] The numbers of cycles required for memory accessing are estimated by multiplying about 1.5 to the number of words (4 bytes) to be read. The factor 1.5 is realized by using ChipScope [10] to observe the number of cycles needed for memory reading.

	Met	Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
1	Yes	TS_clk_fpga_0 = PERIOD TIMEGRP "clk_fpga_0" 100 MHz HIGH 50%	SET...	0.015ns 0.000ns	9.985ns	0 0	0 0
2	Yes	PATH "TS_axi_interconnect_1_reset_resync_path" TIG	SET...		1.874ns		0
3	Yes	PATH "TS_axi4lite_0_reset_resync_path" TIG	SET...		1.680ns		0

Fig. 10. The timing report.

VIII. Hardware Resource Usage

Fig. 10 and Fig. 11 are parts of the report of the hardware generated by Xilinx Platform Studio. In additional to that, 32 block RAMs are also used in the design.

IX. Conclusion

Although the hardware has a lower clock rate of 100MHz, it still has the performance much better than the software, which runs on a processor with clock rate 667MHz. That is the reason to implement some parts of a program on hardware. As in the first acceleration, the ability of parallel computing by hardware gives the performance of "Find Face" a huge improvement.

However, this advantage cannot be taken if there is too much communication between hardware and software to have the costs on performance. That's why the second acceleration method, which implement the whole match in hardware, removes almost all communications and gains a great improvement on performance.

Either in software or hardware, tasks involved with I/O are always consuming the performance, including reading data from memory. In spite of that, some memory actions have to happen, at least once, in many situations. The only thing can be done for improve this is to reduce the repetitive access to same data. Caching is what helps doing this, but it is difficult to take benefit from it in the hardware; instead, it usually bothers the data accessing. As the result, the third acceleration method, which makes every pixel in the group picture loaded from memory only once for each face, achieves the best possible utilization of group pixels loaded from memory, and improves the performance the most.

To wait, to waste. As the fourth acceleration method, although the SAD computation cannot be completed in a single cycle, be good at using pipeline style computation can still make the best utilization of clock cycle that eliminate this disadvantage. On the other hand, the consideration of parallel computing is sometimes viewed as the reason that the hardware can beat the software a lot. In addition, not only parallel computing but also parallel doing computation and I/O can happen on the hardware. The fifth acceleration method take advantage of this, simultaneously computing SAD and reading pixels, and complete the last effort improving the performance.

As shown in the analysis previous made, after applying these acceleration, the time costs except for the computation is remaining only the time reading 4 1024-byte and 128 1920-byte memory regions, which is about 100,000 clock cycles or 1 microsecond. Since the utilization has reached the best, no more effective performance improvement can be made on the aspect relative with memory accessing, even discarding all the memory action. On the other hand, since the pipeline implementation has made the calculation for each position consume only amortized

Device Utilization Summary (actual values)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	32,582	106,400	30%
Number used as Flip Flops	30,112		
Number used as Latches	1		
Number used as Latch-thrus	0		
Number used as AND/OR logics	2,469		
Number of Slice LUTs	41,601	53,200	78%
Number used as logic	39,420	53,200	74%

Fig. 11. The hardware resource usage.

1 clock cycle, it is chosen to stop the acceleration of this program here, as the result has already achieve the expectation.

Either the software or the hardware has jobs that are more suitable than the other, and both have the advantages doing its jobs. For example, the hardware has a better performance while the software has a better flexibility and a better development efficiency. The goal of hardware-software co-design is to take these benefits on developing programs, and a better hardware-software cooperation may lead to a better result, even not only on performance.

X. Future Perspective

Besides to increase hardware resource, there are still some way to improve the performance without requiring additional resource. For example, if the maximum time required between any adjacent pipeline stages can be reduced, perhaps by increasing the number of pipeline stages, the clock rate of the hardware can be tuned higher than 100MHz to make the program finish earlier. For another example, if there are some way to reduce the hardware resource usage, perhaps by using a specially designed adder, to make the program utilize more than one adder-tree to compute more than one face each cycle, the performance of the program can get another significant improvement.

Acknowledgment

Thanks to the laboratory for providing the environment to develop this program, and thanks to teacher, teaching assistants, and classmates for inspiring me a lot.

References

[1] Find Face, given as a course lab material, provided by the course "Introduction to Hardware-Software Codesign and Implementation" given in NCTU.

[2] Sum of Absolute Difference, https://en.wikipedia.org/wiki/Sum_of_absolute_differences

[3] ZedBoard, http://zedboard.org/product/zedboard

[4] Xilinx EDA Tools, http://www.xilinx.com/products/design-tools.html

[5] Median filter, https://en.wikipedia.org/wiki/Median_filter

[6] PNM, https://en.wikipedia.org/wiki/Netpbm_format

[7] Hotspot, https://en.wikipedia.org/wiki/Hot_spot_(computer_programming)

[8] Profiling, https://en.wikipedia.org/wiki/Profiling_(computer_programming)

[9] Master IP, http://www.xilinx.com/support/documentation/ip_documentation/axi_master_burst/v1_00_a/ds844_axi_master_burst.pdf

[10] ChipScope, http://www.xilinx.com/products/design-tools/chipscopepro.html