Optimizations of FSBMA on Zedboard

吳赫倫 Wu Ho Lun 0316320

***Abstract*—In this lab, we are asked to optimize the given program *find_motion* by hardware-software codesign. To effectively accelerate to performance, we almost re-implement the total core design in hardware. In the report, we implement a number of optimized methods to decrease the computation time. Finally, the simulation results show that a reduction of over 99% in computations is achieved after integrating all approaches into full-search algorithm, while ensuring the optimal accuracy. In other words, our implementation reaches over 70 fps.**

Keywords—Full Block-Matching Algorithm; Pipeline; Burst Mode Transfer; Multiple-Core Programming; Finite-State Machine;

I. INTRODUCTION

The given *find_motion* program we need to optimize is to find the motion vector of each block whose size is 16*16 in a 32*32 searching window. Due to that, each block need to be compared with other 1024 blocks to get the minimum matching error, and the matching error function we use is defined as bellow,

$$SAD(p_x, p_y, c_x, c_y) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|f_{t-1}(p_x+i, p_y+j) - f_t(c_x+i, c_y+j)\right| \quad (1)$$

where $N = 16$, $f_t(x, y)$ is the value of current frame at (x, y), $f_{t-1}(x, y)$ is the value of previous frame at (x, y), and $c_x - N \le p_x < c_x + N$, $c_y - N \le p_y < c_y + N$. So that what we want to do is to obtain a pair of (p_x, p_y) such that the value of *SAD* is minimum, and (p_x, p_y) is called as motion vector of block (c_x, c_y).

Before further optimization, we can learn from the previous experiments (lab1) that finishing *find_motion* in *252 ms* is achieved by using software optimization consists of *PDSBMA*, *SEA*, *SSA* and *NEON* and median filter can be done in 49 *ms* through *smith median filter* in software.

In this report, on purpose to describe each optimization method more detailedly. We categorize the methods we implement into two main categories, *Software Optimization* and *Hardware Optimization*, discussing both of two categories in detail respectively.

First, in *Software Optimization*, we still apply the algorithms mentioned in lab1, *PDSBMA*, *SEA*, *SSA*, *NEON* and s*mith median filter*. What's more, we speed up the software through *Multiple-Core Programming* to scatter the computation over two CPUs averagely.

Next, in *Hardware Optimization*, we come up with several approaches to hasten *full_search*. *Pipeline* provides a faster procedure to calculate motion vector and *SAD* value of each block. *Data provider* offers more efficient method to obtain enough amount of data for each round of computation. *Burst mode transfer* of *AXI* uses a more efficient bus protocol to access data on DDR. *Data rearrangement* will rearrange the pixels so that data can be transferred much more rapidly.

II. SOFTWARE OPTIMIZATION

A. Multiple-core Programming

Since there are two CPUs in Zedboard, we are able to take advantage of them to the full in order to reduce the time consumption in software.

To make different CPUs do different tasks, we define CPU0 as master and CPU1 as salve, and master CPU dispatches tasks to CPU1 by using slave registers through *AXI* bus. The operations on salve registers can be seen as atomic operations, for there is only one CPU can access the salve registers at the same time owing to the protocol of *AXI* bus. Therefore, we are able to share data and communicate with each CPU safely without race condition.

Since there are two images need to be processed, we let both CPUs do *median3x3* and *preprocess* on one image respectively. Moreover, motion vectors and *SAD* values of odd rows and even rows are computed by different CPUs. By doing so, we scatter computation effort over two CPUs averagely, and only a half of the original time consumption will be taken in theory.

III. HARDWARE OPTIMIZATION

A. Pipeline

In order to take advantage resources to the full and reduce the idle time period of each functional modules in hardware, we implement *pipeline*. In *pipeline* model, it divides the computation into several stages, functional modules, and each stage does specific task ceaselessly and concurrently. At the positive edge of the clock signal, each stage will pass its results to the next stage. All we need to do is feed the input to the first stage, and the answers can be received from the last stage once all computation is done.

In our *pipeline* model, we split the whole computation into six main stages, and the six stages are the following respectively,

1. Init: in this stage, the next position of x and y that need to be computed are determined.
2. Fetch: this stage reads data, used in following computation, from the *data provider* (discussed later) and stores into the local registers.
3. Abs: the absolute value of the difference between two pixels from two images is calculated in this stage.
4. Adder tree: in fact, there are four adder tree stages in our model. In each adder tree stage, it adds up four values into one value. After four stages of adder tree, 256 values will be accumulated into a single value, the *SAD* value.
5. Min tree: due to the *multiple pipeline* (discussed later), we need to obtain the minimum *SAD* value among each *pipeline*. In each min tree stage, it compares between

two *SAD* values from different *pipeline*, passing the smaller one to the next stage. That is, the number of min tree stage depends on the number of the *pipelines*, for instance, four *pipelines* need two stages of min tree.

6. Compare: the final result of *pipeline* will be updated in this stage. If it receives a smaller *SAD* value than the current one, the smaller one replaces the final result; otherwise, the old value is reserved.

According to Fig. 1., we can understand more how *pipeline* works and the procedure of *pipeline* in our design.

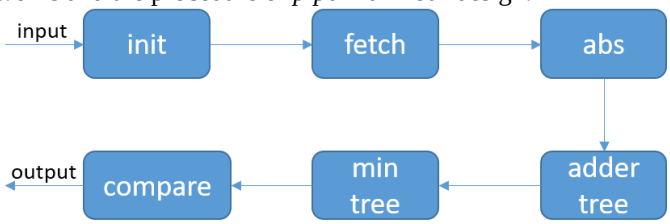

Fig. 1. *Pipeline* flow chart

Furthermore, the scanning sequence is vital as well. There are two common scanning sequences, *round-robin* and *zig-zag*, and the latter is implemented in our *pipeline*. The *zig-zag scanning* makes better use of the overlap between the two blocks of successive computation, for it can just shift images when it meets top or bottom. Nevertheless, *round-robin scanning* needs to read the whole blocks at the top or the bottom. The following figures illustrate the difference between *zig-zag* and *round-robin*.

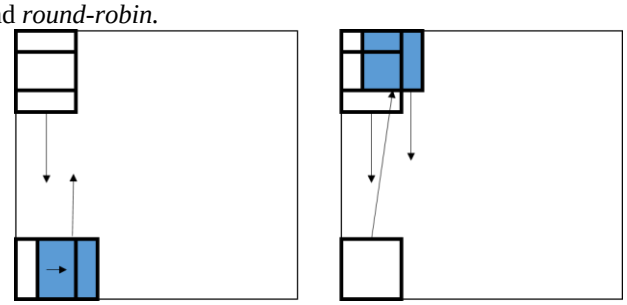

Fig. 2. The left is *zig-zag scanning*, and the right is *round-robin scanning*.

Finally, to speed up the computation even more, we setup *multiple pipelines*, calculating the *SAD* values of different positions concurrently. The following figures illustrate the difference between *single pipeline* and *multiple pipeline*.

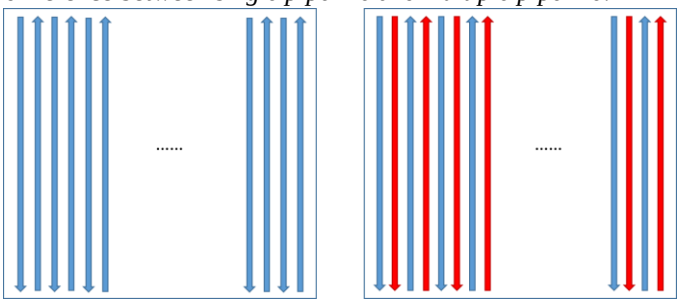

Fig. 3. The left is *single pipeline*, and the right is *multiple pipeline*.

The blue arrows and red arrows in the right figure in Fig. 3. represent the first and the second *pipeline* respectively, and both of the *pipeline* scan from the top to the bottom or from the bottom to the top concurrently and simultaneously.

B. Burst Mode Transfer

In *AXI* protocol, it provides two type of design to communicate with *AXI* bus, *salve mode* and *master mode*. Communication in the *salve mode* is more convenience, nonetheless, the handshaking is inevitable before each communication and you can only transfer four bytes at most after in each handshaking. In other words, it is quite expensive to transfer large amount of data in such mode.

To improve the performance of transferring large amount of data, *AXI* protocol provides *master mode* in which *Burst Mode Transfer* is implemented. Before each *Burst Mode Transfer*, handshaking is unavoidable as well, yet it can transfer a double word, four bytes, per clock cycle in average after handshaking. During handshaking, the length of the data, called beats as well, must be specified, and the length is restricted, only the power of two is allowed, for example, 1, 2, 4, 8, … up to 256 double word (dour bytes). In our design, we only pass the memory address to the hardware in the beginning through *salve mode*, after that, when communication is needed, hardware will make requests to DDR memory through *master mode* by itself *Burst Mode Transfer*. To completely transfer all extra data for next computation, 48 * 8 + 32 * 8 bytes, it takes 16-beats 8 times and 8-beats 8 times, for the data is not successive totally.

C. Data Provider

To obtain data from block ram more efficiently, a *Data Provider* is necessary. In our design, we implement two type of *Data Provider*, *Basic* and *Advance*.

The *Basic Data Provider* consists of several small block rams. Take the left of Fig. 4. as an example, if the address bit width is 4 bits, 16 entries in total, and it is split into 4 small block rams, then we are able to get or set 4 entries in each clock cycle, for the 4 entries scatter over 4 block rams and it satisfies the constraint of block ram.

Beside all features of *Basic Data Provider*, the data arrangement can be seen as a square, and the data access is circular, that is, if you meet the end of a row or a column, the next entry will be at the head of the row or the column. What's more, to reduce the complexity of *Pipeline* implementation, it supports offset of x and y. Take the right of Fig. 4 as an instance, an *Advance Data Provider*, split into 4 block rams, and offset of x and y are set to be 2 and 1 respectively. If reading from x=2 and y=3, the real address will be x=0 and y=0. If reading from x=1 and y=1, the real address will be x=3 and y=2 and the last two entries of the obtained data will be located at the head of the row.

In addition, we enable dual ports block ram in the *Advance Data Provider*, so that we allow *Pipeline* reading and *Burst Mode Transfer* writing meanwhile. It is beneficial to the *Finite-State Machine* (discussed later) design.

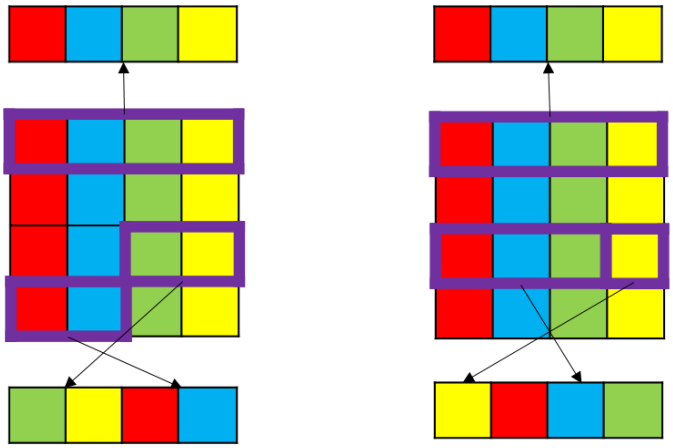

Fig. 4. The left is *Basic Data Provider*, and the right is *Advance Data Provider*. The different colors in the figures mean different block rams.

D. Finite-State Machine (FSM)

To wrap all functional modules into an integrated design, we need a *FSM* to control each module and make them work properly. The following are seven main states in our FSM,

1. Idle: the *FSM* will be in an idle loop to wait for an active signal.
2. Prefetch: in this state, the first block of two images for the first computation will be fetched, and it will stay in this state until the transferring is done.
3. Fetch: the next block of images for next computation will be fetched in this state, however, it goes to the next state directly instead of waiting for transferring. In other words, it only dispatches a signal to start transferring.
4. Pipeline: an active signal will be generated and be dispatched in this state to start the computation in *pipeline*. Moreover, it will stay in this state until both tasks dispatched in fetch state and pipeline state are finished.
5. Save result: the results of *Pipeline* will be stored into the answer storage in this stage. Then, it will determine the next state depending on whether it is the last position need to be computed or not.
6. Write back: if all computation is done, the *FSM* will go to this stage, writing the data in answer storage back to the DDR memory. Similarly, it will stay in this state until all data is transferred.
7. Done: the *FSM* will tell the software that all computation is done in this stage, and then returns to idle state.

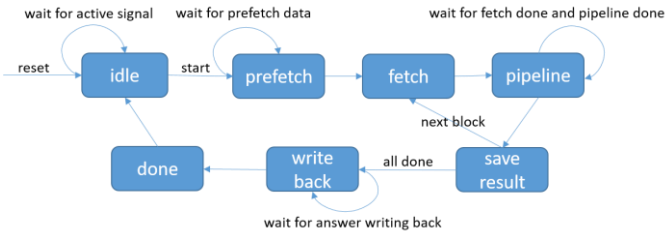

Fig. 5. The figure is the flow chart of FSM.

According to the above discussion, we can notice that fetching the next block of data and computation of *Pipeline* works concurrently. Since we find out that the tasks of two modules are totally independent, they don't need to wait for each other. We can just dispatch tasks to two modules and wait until both of them are done. By doing do, not only do we reduce the time consumption but also make a better use of hardware resources.

E. Data Rearrangement

Because the transferred data must be continuous in memory owing to the *Burst Mode Transfer* of *AXI* protocol, we need many times of *Burst Mode Transfer* to transfer all data completely. However, handshaking is quite expensive, so we need to cut down the number of handshaking to reduce the transferring time consumption. For easier manipulation, we define a structure in software,

```
struct {
    char prev[48][64];
    char curr[16][32];
} prefetch_package;
```

```
struct {
    struct {
        char prev[8][64];
        char curr[8][32];
        char padding[256]
    } data [60][128];
} package;
```

Fig. 6. The structure we define in software.

Although previous and current images only need 48 and 16 bytes in a row for each computation, due to *multiple pipeline*, *zig-zag scanning* and right shifting at the top and the bottom of images, extra data is required. And we align the size to 64 and 32 respectively, for the power of two is more easily to be dealt with. What's more, *char padding [256]* and *data [60] [128]* are also align to power of two owing to the same reason. After the rearrangement, we need to transfer 768 bytes, extra data for the next computation. In other words, one 128-beast and one 64-beats are needed.

IV. Final Model

In our final design, we setup four *Pipeline*, so it is up to 10 stages. *Advance Data Provider* is selected as image data provider, and *Basic Data Provider* is selected to be answer storage. The address bit width of image data provider is 12 bits, and it is split into 32 small block rams, for the design has four *pipeline* and the *zig-zag scanning* need to right shift the images. Fortunately, each *Pipeline* can share the data port of block ram, so only one port is required for *Multiple Pipeline*. Moreover, we enable *Burst Mode Transfer*, *Data Rearrangement* and *Multiple-Core Programming* on *median3x3*. In the end, *Finite-State Machine* (*FSM*) is used to wrap each functional module into an integrated design, and there are 13 states in the final design of *FSM*. The following is the diagram of the final model.

According to Fig. 7, we can learn that the hardware is controlled by software actually. After the software sends an active signal to the hardware by setting a slave register in hardware to 1, it starts a busy loop to wait for the hardware until the hardware finish its task.

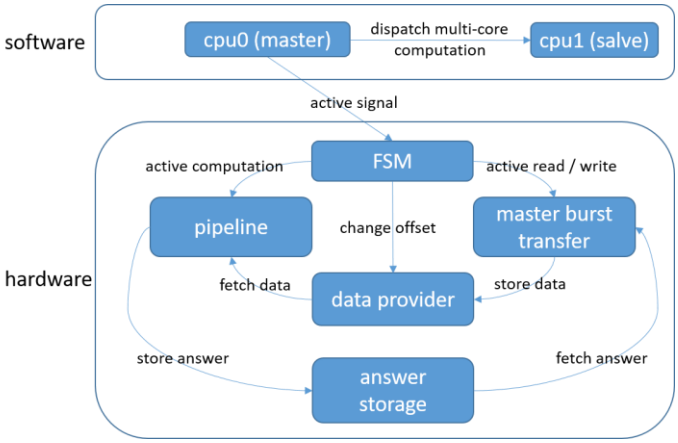

Fig. 7. The diagram of the final model.

V. Simulation Result

The efficiency of the proposed algorithm was tested by using two given consecutive frames of size 720*480 pixels. In the following paragraphs, we will show the result of each proposed optimization methods, and discuss the difference between each other in detail. In the beginning of the discussion, we focus on the software optimization.

A. Multiple-Core Programming

In this part, we make experiments on *median3x3*, *preprocess* and *find_motion*, observing the differences of time consumption between single-core and multiple-core (two-core) programming.

TABLE I. Software Optimization between Single-Core and Multiple-Core

function	Single-Core (*ms*)	Multiple-Core (*ms*)
median3x3	49	25
preprocess	16	9
find_motion	252	126

The above data is measured when the algorithms mentioned in lab1 are implemented. According to table I., we can figure out that only about a half of the original time consumption is taken when *Multiple-Core Programming* is used in each function that conforms to our expectation.

Then, we will focus on the hardware optimization in the following. In addition to the performance, the differences of resource utilization between each approaches are also discussed. And the hardware works on frequency 100 *MHz*.

B. Pipeline

In this part, we focus on the resource utilization and performance between different number of *Pipeline*, for we implement *pipeline* without other optimizations, *Burst Mode Transfer* and *FSM*. By doing so, the resource utilization which is not for *Pipeline* will decrease. Due to that, the data is transferred through the salve registers only. And we calculate the time consumption of *Pipeline* only by subtracting the time for data transferring from the total time consumption.

TABLE II. Performance and Utilization between Different Number of Pipeline

pipeline number	LUT	REG	Total time (*ms*)	*Pipeline* [a]
1	9406	9932	420	51
2	16518	17445	404	28
4	31801	33219	383	14

[a] *The time consumption of Pipeline only*

According to table II., the utilization of LUT and REG and the time consumption of *Pipeline* are almost proportional to the number of *Pipeline*. And then, we calculate the number of clock cycles it need to complete the computation.

$$clock\ \ cycles \approx \left(\frac{32*32}{Pipeline\ \ number}\right) * \left(\frac{720}{8} - 5\right) * \left(\frac{480}{8} - 5\right)$$

Take one *Pipeline* for an example,

$$clock\ \ cycles = 4787200 \approx 47\ \ (\text{ms})$$

it is very close to the experimental result above.

C. Brust Mode Transfer

In this part, we emphasize on the *Burst Mode Transfer*, the time consumption of *Burst Mode Transfer*. To using *Burst Mode Transfer*, *FSM* is necessary to handle the flow of computation. According to the previous discussion, we know that 16-beats 8 times and 8-beats 8 times are required to obtain all extra block for the next computation. Basing on the observation of the wave form (*Integrated Logic Analyzer*), it could be figured out that each handshake takes 32 clock cycles in average, and then the total clock cycles of each *Burst Mode Transfer* can be defined as

$$clock\ \ cycles \approx 8 * (32 + 8) + 8 * (32 + 16) = 704$$

As our discussion above, we know that *Pipeline* and *Burst Mode Transfer* do their own tasks concurrently, hence, the time consumption of each round (one *Pipeline* and one *Burst Mode Transfer*) is the minimum among two of them. Then, the total time consumption cab be defined as

$$clock\ \ cycles = \max(704, Pipeline) * \left(\frac{720}{8} - 5\right) * \left(\frac{480}{8} - 5\right)$$

Take 2 *Pipeline* as an example, and according to equation, we know that *Pipeline* consumes 512 clock cycles, therefore, the total time consumption will be

$$clock\ \ cycle = \max(704, 512) * \left(\frac{720}{8} - 5\right) * \left(\frac{480}{8} - 5\right) = 3291200 \approx 33\ \ (\text{ms})$$

According to table III., it is very close to the experimental results. Furthermore, the time consumption of 4 *Pipeline* and 2 *Pipeline* are the same, for while *Pipeline* works faster, *Burst Mode Transfer* still consume the same amount of time. Also, we know that the bottleneck of the total computation is still the data transferring.

TABLE III. BURST MODE TRANSFER

Pipeline number	LUT	REG	Time (*ms*)
1	16789	14302	50
2	24344	21822	35
4	39869	37649	34

D. The Final Model

In our final model, we integrate all optimization methods, including *Pipeline*, *Burst Mode Transfer*, *Data Provider*, *Finite-State Machine* and *Data Rearrangement*. Because of the support of *Data Rearrangement*, we cut down the time consumption of data transferring further. As we mentioned above, only 128-beats one time and 64-beats one time are required to transfer the extra data of next block. According to the observation of handshake, the time consumption of each *Burst Mode Transfer* is

$$clock\ \ cycles = (32 + 128) + (32 + 64) = 256$$

It is much faster than the one without *Data Rearrangement*. Nonetheless, extra time and memory consumption is required in software to rearrange the data. First, according to the structure we define in software, it costs

$$(8 * 64 + 8 * 32 + 256) * 60 * 128 = 7864320\ \ bytes$$

, and the original costs

$$720 * 480 * 2 = 691200\ \text{bytes}$$

Although it costs more than 11 times the original consumption, we own 512 *MB* DDR memory on our Zedboard. Secondly, to move the data from original address to the new structure, it cost some time. We implement the moving with *memcpy* which has been optimized, and it only costs 8 *ms* to finish complete rearrangement.

Due to the reduction of data transferring, we can find out that 4 *pipeline* is about two times faster than 2 *pipeline*, that conforms to our expectation.

TABLE IV. FINAL MODEL

Pipeline number	LUT	REG	Time (*ms*)
1	21007	14359	50
2	28562	21879	25
4	44088	37706	13

However, the utilization of LUT becomes a little bit more than the design without *Data Rearrangement*, because to deal with the special arrangement of data structure, extra design is required to fit *Data Rearrangement*.

VI. CONCLUSION

In this lab, we finally achieve over 70 *fps* performance, 13 *ms*, in the final model with four *Pipeline*. Nevertheless, it consumes almost 90% of LUT resource on the Zedboard. It is a trade-off between performance and resource utilization. High performance and lower resource utilization cannot coexist. If you pursue the high performance, the resource utilization must be high as well. In other words, low resource utilization must accompany with lower performance. So you have to make sure the aspect you concern more, so that you can choose the parameters properly.

REFERENCES

[1] Mohammed Golam Sarwer and Q.M. Jonathan Wu, “EFFICIENT PARTIAL DISTORTION SEARCH ALGORITHM FOR BLOCK BASED MOTION ESTIMATION”, Department of Electrical and Computer Engineering, University of Windsor, Windsor, ON, Canada.

[2] Yih-Chuan Lin and Shen-Chuan Tai, “Fast Full-Search Block-Matching Algorithm for Motion-Compensated Video Compression”, IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. 45, NO. 5, MAY 1997.

[3] Th. Zahariadis, D. Kalivas "A Spiral Search Algorithm For Fast Estimation Of Block Motion Vectors" Signal Processing VIII, theories and applications. Proceedings of the EUSIPCO 96. Eighth European Signal Processing Conference p.3 vol. lxiii + 2144, 1079-82,vol. 2.

[4] ARM® C Language Extensions Release 1.1., http://infocenter.arm.com/help/topic/com.arm.doc.ihi0053b/IHI0053B_arm_c_language_extensions_2013.pdf

[5] JOHN L. SMITH, “Implementing Median Filters in XC4000E FPGAs”, Univision Technologies Inc., Billerica, MA.